Modern Software Engineering - David Farley

Traducción realizada por Santiago Langer, Maximo Palopoli y Tomás Szejnfeld.
Algoritmos y Programación III - Leveroni

Capítulo 1

Ingeniería - La puesta en práctica de la ciencia

El desarrollo de software es un proceso de descubrimiento y exploración; entonces, para tener éxito, los ingenieros de software deben volverse expertos en **aprender**.

La ciencia es la mejor forma de aprender que tiene la humanidad, por lo que deberíamos adoptar los métodos y estrategias de la ciencia y aplicarlos a nuestros problemas. Regularmente, esto se confunde con la idea de que debemos convertirnos en físicos que midan hasta el último decimal, con muchísimo detalle.
La ingeniería es más pragmática que eso.

Implementar los métodos y estrategias de la ciencia es simplemente aplicar una serie de ideas muy simples, pero extremadamente importantes.

El método científico, como nos lo enseñan en la escuela, está descrito en Wikipedia de la siguiente manera:

- Caracterizar: Hacer una observación y descripción del estado actual.
- Hipotetizar: Crear una teoría que explique la observación.
- Predecir: Hacer una predicción de qué sucederá en función de la hipótesis formada.
- Experimentar: Probar la predicción, y verificar si se cumplió o no.

Una vez que comencemos a organizar nuestro pensamiento de esta manera y progresemos en base a muchos pequeños experimentos informales, limitaremos el riesgo de saltar a conclusiones equivocadas y terminaremos haciendo un mejor trabajo.

Si comenzamos a pensar en términos de **controlar las variables** de nuestros experimentos apuntando a obtener más consistencia y confiabilidad, llegaremos a sistemas y código deterministas. Además, si somos escépticos con nuestras ideas y exploramos cómo refutarlas, podremos identificar (y luego eliminar) malas ideas rápidamente y progresar mejor.

Este libro está basado en un acercamiento práctico y pragmático para resolver problemas en software, con una adaptación informal de principios científicos básicos. O en otras palabras **¡ingeniería!**

¿Qué es la ingeniería del software?

Mi definición operativa de la ingeniería del software que ancla las ideas de este libro es la siguiente:

La ingeniería del software es la aplicación de un método empírico y científico de encontrar maneras eficientes y económicas de solucionar problemas en software.

La adopción de un método ingenieril al desarrollo de software es importante por dos razones principales. Primero, el desarrollo de software siempre es un ejercicio de descubrimiento y aprendizaje; y segundo, si nuestro objetivo es ser “eficientes” y “económicos” entonces nuestra habilidad de aprender debe ser sostenible.
Esto significa que debemos manejar la complejidad de los sistemas que creamos de forma que podamos sostener nuestra habilidad de seguir aprendiendo cosas nuevas y adaptarnos a ellas.

Sintetizando, debemos convertirnos en **expertos del aprendizaje**, y **expertos en administrar la complejidad del software.**

Para centrarnos en el aprendizaje, hay cinco técnicas principales. Particularmente, para ser **expertos del aprendizaje** necesitamos lo siguiente:

- Iteración
- Feedback
- Incrementalismo
- Experimentación
- Empirismo

Este es un método con una mentalidad evolutiva hacia la creación de sistemas complejos. Los sistemas complejos no surgen completos en nuestra imaginación sino que son el resultado de miles de pequeños pasos en los que probamos el éxito o fracaso de cada una de nuestras ideas. Estas son las herramientas que nos permiten explorar y descubrir.

Trabajar de esta manera limita cómo podemos avanzar de forma segura. Necesitamos facilitar el camino de exploración, que es el corazón de todos y cada uno de los proyectos de software.

Además de estar muy concentrados en el aprendizaje, también necesitamos trabajar en maneras de poder progresar cuando las respuestas, o incluso la dirección del proyecto, son inciertas.

Es por esto que debemos convertirnos en expertos en manejar complejidad. Sin importar la naturaleza de los problemas que enfrentamos o las herramientas que usemos, enfrentar la complejidad de los problemas que enfrentamos y las soluciones que les aplicamos es el factor diferencial entre buenos y malos sistemas.

Para convertirnos en expertos de manejar complejidad, necesitamos lo siguiente:

- Modularidad
- Cohesión
- Separación de responsabilidades
- Abstracción
- Bajo acoplamiento

Es fácil ver estas ideas y creer que son conocidas. Es por esto, que la idea de este libro es organizarlas bajo un método coherente de desarrollo de software que mejor nos permita aprovechar su potencial.

Este libro describe diez ideas que serán herramientas para dirigir/orientar el desarrollo de software. Algunas de estas ideas son:

- Testeabilidad
- Facilidad de despliegue (deployability)
- Velocidad
- Control de variables
- Entrega continua de software

Cuando aplicamos esta mentalidad, los resultados son notables. Podemos crear software de mayor calidad, más rápido. Además, las personas que trabajan bajo este paradigma reportan que disfrutan más de su trabajo, están menos estresados y tienen un mejor balance entre su vida y su trabajo.[1]

Estas últimas pueden parecer grandes afirmaciones, pero están respaldadas por datos y estadísticas.

Recuperar la “Ingeniería del software”

Fue muy difícil definir el título de este libro, pero no porque no supiese cómo nombrarlo, sino porque en nuestra industria se ha utilizado tanto el término *ingeniería* que ha perdido su valor.

En el desarrollo de software se cree que ingeniería es simplemente un sinónimo de código, o bien algo burocrático y mecánico que disuade a las personas de interesarse en la materia. Pero la verdadera *ingeniería* no podría ser más distinta.

En otras disciplinas, *ingeniería* significa “cosas que funcionan”. Es el proceso que uno aplica para incrementar las chances de hacer un buen trabajo.

Si nuestras prácticas de “desarrollo de software” no nos permiten desarrollar mejor software y más rápido, entonces no son prácticas ingenieriles ¡Cambiemos esas malas prácticas!

Esta es la idea fundamental de este libro. Su objetivo es describir un modelo intelectualmente consistente que amalgame todos los principios fundamentales del desarrollo de software.

Cómo progresar

El desarrollo de software es una actividad compleja y sofisticada. Es tal vez una de las actividades más complejas que desarrolla la humanidad. Es ridículo pensar que cada persona o equipo puede (o debe) reinventar todo de cero cada vez que comienza un nuevo proyecto.

[1] Basado en reportes de “State of DevOps”, Microsoft y Google.

Hemos aprendido (y continuamos aprendiendo) qué funciona y qué no ¿Pero cómo podemos, como industria, continuar avanzando sobre hombros de gigantes si cualquier persona puede vetar cualquier cosa? Como industria, necesitamos pactar ciertos principios y reglas para definir nuestra actividad.

También debemos ser cuidadosos con esta forma de pensamiento ya que, si se aplica erróneamente, puede llevar a una mentalidad dependiente de las decisiones de nuestros superiores. Recaeremos en malas ideas, donde el trabajo de los managers y los líderes será definir constantemente qué debe hacer cada persona y cómo.

El gran problema de ser proscriptivo y mandón es qué sucede cuando una de nuestras ideas es errónea. Inevitablemente esto sucederá en algún momento, así que debemos comprender cómo enfrentaremos y refutamos las viejas (pero establecidas) malas ideas para darle lugar a la evaluación de nuevas ideas.

Afortunadamente, ya existe una disciplina que nos ayuda a resolver estos problemas. Es una disciplina que nos da la libertad de enfrentar el dogma y diferenciar las buenas de las malas ideas, sin importar de dónde vengan. Fundamentalmente necesitamos una estructura que nos permita crecer nuestras estrategias, procesos, tecnologías y soluciones. Y podemos agradecer que esta estructura ya existe ¡se llama *ciencia*! ¡Y cuando aplicamos este estilo de pensamiento a la realidad, lo llamamos *ingeniería*!

Este libro explica qué significa aplicar el razonamiento científico a nuestra disciplina. Además, ayudará a implementar algo que podamos orgullosamente definir como *ingeniería del software*.

El nacimiento de la ingeniería del software

El concepto de “ingeniería del software” fue creado a finales de los sesenta. El término fue utilizado por primera vez por Margaret Hamilton (quien luego se convertiría en la directora de la división de ingeniería del software del Laboratorio Instrumental del MIT). En esa época, Margaret estaba dirigiendo el desarrollo del software de control de vuelo para el programa espacial Apollo.

En el mismo periodo de tiempo, la Organización del Tratado del Atlántico Norte (OTAN) tuvo una conferencia en Garmisch-Partenkirchen, Alemania, para definir el término. Esta fue la primera conferencia sobre **ingeniería del software**.

Al principio de nuestra disciplina, las primeras computadoras se utilizaban accionando interruptores, o incluso estaban completamente hardcodeadas. Rápidamente los pioneros en la materia notaron que esto era muy inflexible, por lo que la idea del programa almacenado surgió. Es a partir de este momento que nace la diferenciación entre software y hardware.

Para finales de los sesenta, los programas de computadoras se habían vuelto lo suficientemente complejos como para comenzar a ser difíciles de mantener. Se estaban

comenzando a usar para resolver problemas cada vez más complejos y para enfrentar siquiera cierta clase de problemas antes imposibles.

Se percibía que había una gran brecha entre el avance del progreso que se realizaba en el hardware en comparación con el software. A esto se lo llamó, en su momento, como la *crisis del software*. La conferencia de la OTAN se organizó, en parte, como respuesta a esta crisis.

Leyendo las actas de la conferencia hoy en día, es notable cómo muchas ideas siguen persistiendo hasta hoy. Han resistido el paso del tiempo y son tan reales ahora como en 1968. Esto debería ser interesante para nosotros, si buscamos identificar características fundamentales que definen nuestras disciplina

Unos años más tarde, Fred Brooks (ganador del premio Turing en 1999) comparó el progreso del software con el del hardware:

> *No existe ningún otro desarrollo en la próxima década que prometa grandes saltos de mejora tanto en la productividad, la confiabilidad o la simplicidad [como el hardware].*[2]

Brooks hacía referencia a la Ley de Moore, que en el caso del hardware ya venía siguiendo el desarrollo del hardware de los últimos años.

Brooks afirmaba, además, que no era un estancamiento del desarrollo de software, sino una observación del crecimiento notable del rendimiento del hardware.

> *Debemos observar que la anomalía no es que el desarrollo de software sea lento, sino que el avance del hardware es tán rápido. No existe otra tecnología, desde el comienzo de la humanidad, que haya visto mejorar en 6 órdenes de magnitud la relación rendimiento-precio en tan solo 30 años.*

Él escribió esto en 1986, lo que ahora consideramos el amanecer de la era de la información. El avance del hardware ha mantenido su ritmo vertiginoso (convirtiendo a las computadoras de la época de Brooks ahora en simples juguetes), pero aún así su observación sobre la velocidad de desarrollo de software se mantiene vigente.

Cambiando de paradigma

La idea de *cambio de paradigma* fue creada por el físico Thomas Kuhn.

La mayoría del aprendizaje es un proceso de construcción aditiva. Formamos capas de comprensión, basando cada idea en conceptos vistos anteriormente.

Sin embargo, el aprendizaje no es siempre así. A veces, debemos cambiar fundamentalmente nuestra perspectiva al respecto de un tópico, lo que nos permite aprender cosas nuevas a cambio de descartar las viejas.

---

[2] Fuente: “No Silver Bullet”, paper de Fred Brooks de 1986.

En el siglo XVIII, los biólogos creían que los animales se generaban espontáneamente. Pero Darwin en el siglo XIX describió el proceso de selección natural, desarmando así la teoría de la generación espontánea. Ese cambio nos ha llevado a nuestro actual entendimiento de la genética y nuestra habilidad para comprender la vida en un nivel fundamental, permitiéndonos manipular genes y crear vacunas contra el COVID-19.

Similarmente Kepler, Copérnico y Galileo enfrentaron la teoría contemporánea de que la Tierra es el centro del universo, proponiendo en cambio un modelo heliocéntrico (con el Sol como centro). Este cambio permitió que Newton formulara las leyes de gravitación, que Einstein modelara la relatividad general, desarrollo del viaje espacial y la invención de nuevas tecnologías como el GPS.

El concepto de *cambio de paradigma* implícitamente incluye la idea de que cuando hacemos un gran cambio, descartamos las ideas que ahora sabemos que no son correctas.

Las implicancias de tratar al desarrollo del software como una disciplina ingenieril, basada en la filosofía del método científico y el racionalismo científico, son profundas. Es profundo no solo por su impacto y efectividad, sino por la necesidad esencial de descartar las ideas que los nuevos enfoques reemplazan.

Esto nos da un método para aprender de manera efectiva y descartar las malas ideas.

Creo que el método de desarrollo de software que describo en este libro representa un cambio de paradigma. Nos provee con una nueva perspectiva sobre qué hacemos y cómo lo hacemos.

Resumen

Aplicar este pensamiento ingenieril al software no necesita ser pesado e inmensamente complejo. El cambio de paradigma sobre cómo pensamos nos permitirá ser más simples, eficientes y confiables.

Esto no es más burocracia, sino que es mejorar nuestra habilidad de crear software de alta calidad de forma sostenible y confiable.

Capítulo 2

¿Qué es la ingeniería?

Converso sobre ingeniería de software desde hace ya unos años. Y regularmente termino en conversaciones sobre construcción de puentes. Normalmente comienzan con la frase “Si, pero el software no es construcción de puentes” como si fuese una gran revelación.

Por supuesto, la ingeniería de software no es lo mismo que la construcción de puentes, pero además lo que la mayoría de los ingenieros de software consideran como construcción de puentes tampoco lo es. Esta conversación, en realidad, es una confusión entre la ingeniería de producción y la ingeniería de diseño.

La ingeniería de producción es un complejo problema donde la disciplina involucrada tiene que lidiar con el mundo material. Esas cosas materiales deben ser creadas con un cierto grado de precisión y calidad.

La ingeniería de producción necesita entregar artilugios a algún lugar, en algún momento, bajo un cierto presupuesto…etc. La disciplina necesita entonces, adaptar sus ideas teóricas a la realidad práctica cuando los modelos y diseños preexistentes resultan insuficientes.

Los artilugios digitales son completamente diferentes. Si bien puede haber algunas similitudes menores entre los productos físicos y los digitales, el costo de producción de estos últimos es esencialmente gratis.

La producción no es nuestro problema

Para la mayoría de los proyectos humanos, la producción es la parte difícil. Requiere esfuerzo e ingeniosidad diseñar un auto, un avión o un teléfono, pero llevar ese prototipo inicial a producción en masa es el desafío más caro y complicado.

Esto es particularmente real si apuntamos a hacerlo bajo alguna clase de eficiencia. Como resultado de esta dificultad, la mayoría de los productos de la era industrial centran sus preocupaciones en la producción sobre cualquier otro aspecto.

En cambio, para la informática, la producción no es nuestro problema, es muy fácil. Esto hace a nuestra disciplina bastante inusual, pero también la expone a prácticas mal aplicadas para nuestro caso.

Ingeniería de diseño, no ingeniería de producción

Lo que la mayoría de las personas piensan de la construcción de puentes varía mucho según si es un puente tradicional o un nuevo tipo de puente innovador (el primero de su clase). En este último tipo de circunstancias se tienen dos problemas, uno relevante a la ingeniería del software y otro no.

Primero, el aspecto que no nos concierne: Cuando se construye por primera vez un nuevo tipo de puente (físico) se tendrán por primera vez todos los problemas de producción. Desde un punto de vista del software, estos problemas de producción no son un factor.

El segundo aspecto, que afecta tanto al desarrollo de nuevos puentes como al desarrollo de software, es el desafío del **diseño** del nuevo producto (ya sea software o un puente).

Esto es difícil porque no se puede iterar rápidamente cuando el producto existe en el plano físico. Durante la construcción de elementos físicos, estos son difíciles de cambiar. Como resultado de esta dificultad, los ingenieros de otras disciplinas adoptan técnicas de modelado y eligen modelar pequeñas maquetas (modelos) de sus proyectos. Por ejemplo, hoy en día construyen simulaciones en computadora de sus diseños.

En este aspecto, los desarrolladores de software tenemos una gran ventaja. El constructor de puentes puede hacer un modelo de su diseño, pero este será solo una aproximación del producto real. La simulación, su modelo, siempre será impreciso. En cambio, los modelos de software que creamos **son el producto.**

No necesitamos preocuparnos si nuestros modelos coinciden con la realidad, nuestros son la realidad de nuestro sistema, y por ende podemos verificarlos fácilmente. No necesitamos preocuparnos por el costo de cambiarlos. Son software, y por ende son dramáticamente más fáciles de cambiar (al menos si lo comparamos con un puente).

La nuestra es una disciplina técnica. La mayoría de las personas que se identifican como desarrolladores profesionales de software han aprendido sobre ciencia en su educación.

A pesar de esto, muy poco desarrollo de software se desarrolla con el racionalismo científico en mente. En parte, esto es producto de haber tomado algunos pasos equivocados a lo largo de la historia de nuestra disciplina. Asumimos que la ciencia es difícil, cara e imposible de aplicar en los plazos de tiempo del desarrollo de software.

Parte del error es intentar alcanzar un nivel de precisión ideal que es imposible de lograr en cualquier campo, incluido el software ¡Hemos cometido el error de buscar la precisión matemática, cuando en realidad estamos haciendo ingeniería!

Ingeniería como matemática

A finales de los 80’ y principios de los 90’, se hablaba mucho de programación estructurada. La ingeniería del software pasó a examinar cómo generamos código. Específicamente, cómo podíamos trabajar mejor para identificar y eliminar problemas en nuestros diseños e implementaciones.

Los *métodos formales* comenzaron a ganar popularidad. La mayoría de los cursos universitarios enseñaban métodos formales. Un método formal es un estilo de construir software que incluye una validación matemática del código escrito. La idea era demostrar que el código estaba demostrado que era correcto.

El gran problema es que si ya es difícil escribir código para un sistema complejo, es incluso más difícil escribir código que define el comportamiento de un sistema complejo y que además demuestre que sí mismo es correcto.

Los métodos formales son una idea atractiva, pero no han logrado la adopción masiva en la industria del desarrollo del software porque, a la hora de la producción, hacen más difícil producir código.

Aún así, existe un argumento filosófico distinto: El software es una cosa extraña, que claramente atrae a personas que disfrutan el pensamiento matemático. Así, el atractivo de darle una mirada matemática al software es grande, pero limitante.

Consideremos esta analogía del mundo real: Los ingenieros usarán todas las herramientas a su disposición para desarrollar un nuevo sistema, ellos crearán modelos y simulaciones para definir si el sistema funcionará. Su trabajo depende en gran medida de la matemática, pero los ingenieros son quienes realmente lo pondrán en práctica en el mundo real, no los matemáticos.

En otras disciplinas ingenieriles, la matemática es definitivamente una herramienta central, pero no reemplaza la necesidad de probar y aprender empíricamente con experiencia del mundo real. Hay demasiadas variables en el mundo real como para predecir con exactitud el resultado final. Si solo la matemática alcanzara para diseñar un avión, las compañías aeroespaciales no construirían grandes prototipos ni los probarían en costosos túneles de viento, pero en la realidad si lo hacen. Basan sus conocimientos en la matemática, pero luego ponen en práctica lo ideado. Por supuesto, el software no es lo mismo que un avión o un cohete.

El software es digital y corre en máquinas (que son casi siempre) determinísticas llamadas *computadoras*. En problemas simples, contenidos, determinísticos y con suficientes pocas variables los métodos formales pueden aplicarse.
La duda es hasta qué punto el sistema es realmente determinístico. Si el sistema es concurrente, interactúa con el mundo real y las personas, o está en un contexto suficientemente complejo; demostrar todo se vuelve impráctico.

Por esto tomamos el mismo camino que nuestros colegas aeroespaciales y aplicamos el pensamiento matemático donde podemos; mientras que tomamos un método pragmático, empírico y experimental al aprendizaje permitiéndonos adaptar nuestro sistema mientras lo hacemos crecer.

Mientras escribo este libro, SpaceX está ocupada explotando cohetes mientras perfecciona Starship. Seguramente ha construido modelos matemáticos para casi cualquier aspecto del diseño de sus cohetes, sus motores, sus sistemas de combustible, su infraestructura de lanzamiento y todo lo demás, pero aún así los testea.

Incluso algo aparentemente simple como pasar la piel del cohete de placas de acero de 4 milímetros a placas de 3 milímetros parecería un cambio controlado. SpaceX tiene acceso a

detallada información sobre la fuerza tensil del metal. Tiene experiencia e información sobre qué tan fuertes son los cohetes de 4 mm. Pero aún así, luego de hacer los cálculos, SpaceX construyó prototipos experimentales para evaluar la diferencia. Presuriza las piezas de prueba hasta destruirlas para comprobar que sus cálculos fuesen correctos, recolecta información y valida sus modelos porque inevitablemente van a tener algún error difícil de encontrar o prevenir.

La ventaja notable que tenemos sobre todo el resto de disciplinas es que los modelos que creamos son un producto directo de nuestro trabajo. Cuando los testeamos, estamos probando nuestro producto, no una aproximación del mismo. Si aislamos la parte del sistema que queremos probar, podemos evaluarla exactamente en el mismo ambiente en el que estará en producción. Nuestra simulación experimental representará casi exactamente cómo será en producción.

En su excelente charla llamada “Verdadera ingeniería del software” Glenn Vandenburg dice que en las otras disciplinas “ingeniería significa cosas que funcionan” mientras que en el software casi lo contrario se ha convertido en la norma.

Vanderburg explora por qué sucede esto. Describe un método académico a la ingeniería de software tan oneroso que casi nadie que lo haya aplicado alguna vez lo recomendaría para proyectos futuros.

Era pesado y no agregaba ningún valor al proceso de desarrollo de software. En una contundente frase, Vanderburg decía:

[El desarrollo de software académico] solo funcionaba porque personas inteligentes, con voluntad, estaban dispuestas a evitar el proceso.

Esto no es ingeniería bajo ningún término.

La definición de Vanderburg de “ingeniería significa cosas que funcionan” es importante. Si las prácticas que elegimos definir como ingeniería no nos permiten hacer mejor software más rápido ¡entonces no son ingeniería!

El desarrollo de software, a diferencia de cualquier proceso de producción física, es completamente un ejercicio de descubrimiento, aprendizaje y diseño. Nuestro desafío es uno de exploración, por lo que deberíamos (incluso más que los ingenieros aeroespaciales) aplicar técnicas de la exploración sobre las técnicas de producción. La nuestra es una disciplina de ingeniería de diseño.

Entonces, si nuestro entendimiento de la ingeniería es frecuentemente erroneo ¿de qué trata realmente la ingeniería?

La primera ingeniera de software

Mientras Margaret Hamilton dirigía el desarrollo de los sistemas de control de vuelo del programa Apollo, no había “reglas de juego” a seguir. Ella dice:

“Nosotros evolucionamos nuestras reglas de ‘ingeniería de software’ con cada nuevo descubrimiento, mientras que las reglas desde top management de NASA pasaron de ser ‘libertad total’ a ‘aplastamiento burocrático’ ”.

Existía poca experiencia previa sobre sistemas tan complejos, por lo que el equipo frecuentemente innovaba para toda la disciplina. Los desafíos que enfrentaba Hamilton y su equipo eran complejos, y no había forma de mirar la respuesta en Stack Overflow en 1960.

Hamilton describe algunos de los desafíos:

La misión espacial tenía que estar certificada para tripulaciones humanas. No solo tenía que funcionar, tenía que funcionar a la primera. El software no solo tenía que ser ultra-confiable, sino que tenía que detectar errores y solucionarlos en tiempo real. Nuestro lenguaje, además, nos dejaba expuestos a los más sutiles errores. Teníamos la libertad de formar nuestras propias reglas del desarrollo de software. Lo que aprendimos de nuestros errores estaba lleno de sorpresas.

En esa misma época, el software era menospreciado en comparación con otras formas de ingeniería más desarrolladas. Una de las razones por las que Hamilton acuñó el término *ingeniería de software* fue para lograr que personas de otras disciplinas comenzasen a tomarse el software seriamente.

Una de las fuerzas troncales tras el método de Hamilton era entender cómo las cosas fallan, la manera en la que cometemos errores.

Yo tenía una fascinación con los errores, uno de mis pasatiempos era entender cómo surgía un error, o una serie de errores, y cómo prevenirlos en el futuro.

Esta obsesión estaba basada en el método científico y racional para la solución de problemas. La suposición no era que encontrarías la solución perfecta a la primera, sino que uno debía tratar todas las ideas con escepticismo hasta que no pudiese encontrar más maneras de romperlas. Ocasionalmente, la realidad tendrá alguna sorpresa para darte, pero esto es simplemente el empirismo ingenieril haciendo su parte.

El otro principio ingenieril en el trabajo de Hamilton fue la idea de “fallar de forma segura”. Si uno supone que nunca podremos escribir código para cubrir cualquier escenario ¿entonces cómo podemos codear de forma que nuestros sistemas resistan lo inesperado y aún así progresen? Famosamente, fue la aplicación (sin ser solicitado) de esta idea lo que salvó la misión del Apollo 11 y permitió el aterrizaje del módulo lunar luego que la computadora se abrumara durante el descenso hacia la Luna:

Mientras Neil Armstrong y Buzz Aldrin descendían hacia la Luna, la computadora presentó los errores 1201 y 1202 a los astronautas. Ellos preguntaron entonces si debían seguir con el alunizaje o cancelar la misión.
NASA dudó hasta que uno de los ingenieros gritó “¡Siga!” porque entendió qué sucedió con el software.

En el Apollo 11, cada vez que aparecían los errores 1201 y 1202 la computadora se reiniciaba, recargaba todo lo importante como la dirección del motor de descenso y la interfaz, avisando a la tripulación en el proceso. Los ingenieros de NASA en la sala de control de misión sabían, ya que MIT había testeado extensamente la capacidad de reinicio de la computadora, que la misión podría seguir.

Este comportamiento a prueba de fallos estaba implementado en el sistema, sin saber cuándo o cómo podría ser útil.

Así, Hamilton y su equipo introdujeron dos conceptos a este nuevo estilo de pensamiento ingenieril, con aprendizaje empírico y el hábito de imaginar cómo las cosas podrían salir mal.

Una definición operativa de Ingeniería

La mayoría de las definiciones de la palabra *ingeniería* incluyen palabras y frases como “aplicación de la matemática”, “evidencia empírica”, “razonamiento científico”, “limitaciones económicas”.

Yo propongo la siguiente definición operativa:

Ingeniería es la aplicación de un método científico y empírico para encontrar soluciones eficientes y económicas a problemas prácticos.

Todas las palabras elegidas importan. La ingeniería es ciencia aplicada, es práctica. Usar la palabra “empírico” significa aprender y avanzar nuestro entendimiento y sus soluciones hacia la resolución del problema. Las soluciones que la ingeniería propone no son ideas abstractas inalcanzables; sino que son prácticas y aplicables al problema y su contexto. Son eficientes, son creadas con un entendimiento (y limitadas por) las condiciones económicas del problema.

Capítulo 3

Elementos fundamentales del método ingenieril

La ingeniería varía según la disciplina. La construcción de puentes no es lo mismo que la ingeniería aeroespacial, ni es lo mismo que la ingeniería eléctrica o química. Sin embargo, todas estas disciplinas tienen ideas en común. Están basadas en el racionalismo científico y toman un enfoque pragmático y empírico para lograr avances.

Si vamos a lograr nuestra meta de intentar definir una colección de pensamientos, ideas, prácticas y comportamientos duraderos que podríamos agrupar colectivamente bajo el nombre de ingeniería de software, estas ideas deberán ser fundamentales a la realidad del desarrollo de software y robustas frente al cambio y el paso del tiempo.

¿Una Industria de cambio?

Hablamos mucho sobre el cambio en nuestra industria. Nos emocionamos por nuevas tecnologías y nuevos productos pero ¿hacen estos cambios una gran diferencia en el desarrollo de software? Muchos de estos cambios que nos conmueven no parecen hacer un cambio tan grande como el que creemos que harían.

Mi ejemplo favorito sobre este caso fue demostrado en una maravillosa conferencia presentada por Christin Gorman[3]. En ella, Christin demuestra que usando Hibernate (la entonces popular biblioteca de mapeo relacional de objetos de código abierto) debía escribir más código que usando el comportamiento equivalente escrito en SQL. Además, subjetivamente, el código de SQL era más fácil de entender. Christin continúa haciendo un divertido contraste entre el desarrollo de software y hacer tortas: ¿Haces tu torta con una mezcla para torta o eliges ingredientes frescos y la haces desde cero?

Mucho del cambio en nuestra industria es efímero y no cambia realmente las cosas. Algunos, como en el ejemplo de Hibernate, incluso las empeoran.

Mi impresión es que nuestra industria tiene problemas para aprender y progresar. Esta relativa falta de avances ha estado enmascarada por el increíble progreso que se ha hecho en el hardware en el cual nuestro código se ejecuta.

No estoy diciendo que no haya habido progreso alguno en el software - todo lo contrario - pero creo que el ritmo de avance es mucho más lento de lo que varios de nosotros creemos. Piense, por un momento, qué cambios han tenido un impacto significativo en la forma en que piensa y practica el desarrollo de software ¿Qué ideas hicieron una diferencia en la calidad, escala o complejidad de los problemas que puede resolver?

La lista es más corta de lo que solemos suponer.

[3] Fuente: “Gordon Ramsay Doesn’t Use Cake Mixes” por Christin Gorman

Por ejemplo, he trabajado con 15 o 20 lenguajes de programación a lo largo de mi carrera profesional. Aunque tenga preferencias, solo dos lenguajes han cambiado radicalmente mi forma de pensar sobre el software y el diseño.

Esos grandes saltos fueron el paso de Assembler a C y el paso de programación procedural a la orientada a objetos. En mi opinión, los lenguajes por sí solos tienen menos importancia que el paradigma de programación. Esos pasos representaron cambios significativos en el nivel de abstracción que podía manejar a la hora de escribir código. Cada uno representó un gran cambio en la complejidad de los sistemas que podíamos construir.

Cuando Fred Brooks escribió que no había ganancias de orden de magnitud, se perdió de algó. Tal vez no se pueda decuplicar la ganancia, pero ciertamente se pueden decuplicar las pérdidas.

He visto organizaciones que estaban paralizadas por su enfoque respecto del desarrollo de software, a veces por tecnologías, pero más a menudo por su proceso. Una vez consulté en una gran organización que no había lanzado ningún software a producción en más de cinco años.

No solo nos es difícil aprender nuevas ideas, sino que también se nos hace casi imposible descartar ideas viejas, por más desacreditadas que estén.

La importancia de la medición

Una de las razones por las que nos es difícil descartar viejas ideas es porque no acostumbramos a medir nuestro rendimiento de desarrollo del software.

La mayoría de las métricas aplicadas al desarrollo de software son irrelevantes (por ejemplo la velocidad) o dañinas (por ejemplo la cantidad de líneas de código escritas).

En los círculos del desarrollo ágil, desde hace tiempo, se cree que es imposible medir el rendimiento de un equipo de software y sus proyectos. Martin Fowler escribió sobre un aspecto de esto en una nota en 2003.[4]

El argumento de Fowler es correcto; no tenemos ninguna métrica sólida para la productividad, pero esto no significa que no podamos medir nada ni que hacerlo sea completamente inutil.

El valioso trabajo de Nicole Forsgren, Jez Humble y Gen Kim en el libro *Accelerate: The Science of Lean Software & DevOps*[5] representa un importante paso hacia adelante en el ámbito de tomar decisiones más fuertes basadas en evidencias sólidas. En su trabajo, ellos presentan un modelo interesante de medición del rendimiento de los equipos de software.

Interesantemente, no intentan medir la productividad, sino que evalúan la efectividad de un equipo de software basándose en sólo dos atributos. Estos resultados son entonces usados

[4] Fuente: “Cannot Measure Productivity” por Martin Fowler,
[5] Y en los reportes de “State of DevOps”

en un modelo predictivo. El equipo no puede demostrar que haya una relación causal entre lo medido y el rendimiento, pero si pueden demostrar una correlación estadística.

Las medidas son **estabilidad** (stability) y **flujo** (throughput). Equipos con ambos valores altos son determinados de *alto rendimiento*, mientras que aquellos con ambos valores bajos tienen *bajo rendimiento*.

Lo interesante es que si se comparan las actividades de múltiples equipos, aquellos con puntajes similares tenderán a tener comportamientos similares. De la misma manera, si estudiamos cómo se comportan ciertos equipos, podremos predecir sus puntajes.

Por ejemplo, si un equipo usa testeo automático, automatización de despliegue y otras diez prácticas en particular, el modelo predice que practican *entrega continua*. Si un equipo practica entrega continua, el modelo predice que tendrá un alto rendimiento en términos de entrega de software y organización.

Alternativamente, si vemos organizaciones consideradas muy performantes, estas tendrán comportamientos en común, como entrega continua y la organización en equipos pequeños.

La medición de estabilidad y flujo, entonces, nos dan un modelo que nos permite calcular el rendimiento de un equipo.

Ambas métricas son calculadas mediante dos métricas cada una:

Estabilidad tiene las siguientes:

- Tasa de fallo en cambios: La tasa en la que un cambio desencadena alguna falla en todo el proceso.
- Tiempo de recuperación: Cuánto tiempo toma al equipo recuperarse de un error en algún punto del proceso.

Medir la estabilidad es valioso para cuantificar la calidad del trabajo realizado. No dice nada si el equipo está trabajando en las soluciones correctas, pero si mide su capacidad para implementar correctamente esas soluciones ideadas.

El **flujo** es medido de la siguiente manera:

- Tiempo de ejecución (lead-time): Cuánto tiempo se tarda desde que algo es una idea hasta que está implementado como código funcional.
- Frecuencia: Cada cuanto tiempo los cambios son desplegados en producción.

El flujo mide la capacidad del equipo para desplegar ideas (como software funcional). Mide cuánto tiempo toma en llevar cambios al cliente y con qué frecuencia estos se hacen. Esto además permite medir cuántas oportunidades de aprendizaje se le presentan al equipo. El equipo podría no aprovecharlas, pero sin un buen flujo estas no aparecerán.

Estas dos son medidas técnicas para medir nuestro rendimiento. Responden las preguntas “¿Cuál es la calidad de nuestro trabajo?” y “¿Qué tan frecuentemente podemos producir productos de esa calidad?”

Estos son dos aportes muy útiles, pero no deben ser los únicos. No miden si se está construyendo el software correcto, sino si este se está construyendo correctamente (de todas formas es útil recordar que el software imperfecto no necesariamente es inutil).

Interesantemente, el modelo correlativo que describí va más allá de predecir el tamaño del equipo y si están aplicando entrega continua. Los autores de Accelerate tienen información sobre correlaciones con otras cosas mucho más importantes.

Por ejemplo, organizaciones conformadas por equipos muy performantes (según este modelo) hacen más dinero que organizaciones con malos equipos. Este análisis afirma que hay una correlación entre la calidad de los equipos y el rendimiento de sus empresas. También refuta la idea de que es imposible tener buen software con frecuencia. La calidad y la velocidad son atributos correlacionados, según la información de este estudio. El camino a la velocidad es software de buena calidad, y el camino a software de buena calidad es la velocidad del feedback.
Así, el camino hacia ambos es una buena implementación de la **ingeniería**.

Aplicando la estabilidad y el flujo

La correlación entre buenos puntajes y resultados de buena calidad es importante. Nos ofrecen una oportunidad de evaluar cambios a nuestro proceso, organización, cultura y tecnología.

Imagine, por ejemplo, que nos preocupa la calidad del software ¿Cómo podemos mejorarlo? Podríamos decidir hacer un cambio en nuestro proceso, nuestra forma de trabajar. Formemos, entonces, un comité de aprobación de cambios.

Claramente la adición de una revisión extra afectará negativamente nuestro flujo y la velocidad de nuestro proceso pero ¿mejorarán nuestra estabilidad?

Sorprendentemente, estos comités no mejoran la estabilidad. Pero si, al ralentizar todo el proceso, hieren a la estabilidad de la producción.

> *Encontramos que las aprobaciones externas afectaban negativamente el tiempo de ejecución, la frecuencia de despliegue y el tiempo de restauración pero no afectaban la tasa de fallo. Resumiendo, la aprobación de un cuerpo externo simplemente no funciona para mejorar la estabilidad de los sistemas de producción (midiéndose a través del tiempo de restauración y la tasa de fallo). Con certeza ralentiza las cosas y es peor que no tener un proceso de verificación de cambios en absoluto.*

Mi objetivo aquí no es hacer gracia de los comités de aprobación de cambios, sino mostrar la importancia de tomar decisiones basadas en evidencia en vez de conjeturas.

No es obvio que estos comités no funcionan, todo lo contrario, son muy atractivos (y muchas organizaciones los incluyen en sus estructuras. El problema, simplemente, es que no funcionan. Sin mediciones efectivas no nos podemos dar cuenta que no funcionan, solo podemos adivinar.

Si comenzamos a aplicar un método racional, científico y basado en evidencia no será necesario creer ciegamente en el autor de este libro ni en la palabra de Forsgren y sus coautores. En cambio podrías medir esto vos mismo, con tu equipo. Mide el flujo y la estabilidad con tu método de trabajo actual y luego haz un cambio en el método ¿Se nota algún cambio en las métricas estudiadas?

Puedes leer más sobre este modelo correlativo en el excelente libro Accelerate. Describe el método para medir y cómo el modelo evoluciona mientras la investigación continúa. **Finalmente deberíamos tener con qué medir nuestro rendimiento.**

Los cimientos de la disciplina del desarrollo del software

¿Cuáles son algunas ideas fundamentales? ¿Cuáles son las ideas que esperamos que se mantengan vigentes en 100 años y aplicables bajo cualquier problema o tecnología?

Existen dos categorías: la primera es proceso y la segunda es técnica (o diseño). Nuestra disciplina debe concentrarse en dos competencias centrales.

Debemos convertirnos en maestros del aprendizaje. Debemos reconocer y aceptar que nuestra disciplina es sobre diseño creativo y que no tiene ninguna relación con la ingeniería de producción. En cambio, debemos dominar las habilidades de exploración, descubrimiento y aprendizaje. Esta es una puesta en práctica del método científico.

También necesitamos concentrarnos en mejorar nuestra capacidad de manejar complejidad. Construimos sistemas que no entran en nuestras mentes sino que son a gran escala y tienen a grandes grupos trabajando en ellos. Para esto, necesitamos convertirnos en expertos en manejar complejidad (tanto para resistir a nivel técnico como a nivel organizativo).

Expertos en aprender

La ciencia es la mejor forma de aprender que tiene la humanidad. Si queremos volvernos expertos en aprender necesitamos dominar el método práctico para resolver problemas, que es la esencia de otras ingenierías.

Este método debe estar tallado a nuestros problemas. El proceso de la ingeniería del software será distinto al resto de las disciplinas como la ingeniería aeroespacial es distinta de la ingeniería química. Necesita ser práctica, ligera y penetrante para nuestro proceso de resolver problemas.

Existe un consenso considerable entre los líderes de este tópico sobre el tema. Pero estas ideas no son aplicadas, a gran escala, como las fundaciones del desarrollo de software.

Existen cinco comportamientos interconectados en esta categoría:

- Trabajar iterativamente.
- Usar feedback rápidamente y frecuentemente.
- Trabajar incrementalmente.

- Ser experimental.
- Ser empírico.

Si nunca antes las has evaluado, estas ideas pueden parecer abstractas y lejanas al día-día del desarrollo de software, y más aún de la ingeniería del software.

Pero el desarrollo de software es un ejercicio de exploración y descubrimiento. Siempre estamos intentando aprender más sobre qué desean nuestros clientes del sistema, cómo resolver mejor los problemas que se nos presentan y cómo aplicar óptimamente las técnicas y herramientas a nuestra disposición

Así aprendemos que hemos obviado algo y debemos solucionarlo. Aprendemos a organizarnos mejor y a comprender profundamente los problemas en los que estamos trabajando.

Aprender está en el corazón de todo lo que hacemos. Estas prácticas son los cimientos de cualquier método efectivo de desarrollo de software, y también nos ayudan a descartar métodos menos efectivos.

Los métodos de cascada, por ejemplo, no exhiben estas cinco propiedades. De todas formas estos comportamientos están correlacionados con un alto rendimiento en equipos de desarrollo de software y han sido las hallmarks de equipos exitosos durante décadas.

La parte II explora cada una de estas ideas en profundidad desde una perspectiva práctica ¿Cómo nos convertimos en expertos en aprender y cómo aplicamos esto a nuestro trabajo diario?

Expertos en manejar complejidad

Como un desarrollador de software, veo el mundo con los ojos de un desarrollador de software. Como resultado de esto, mi percepción de los fracasos del desarrollo de software y su cultura puede ser pensada en términos de 2 ideas de la ciencia de la información: la concurrencia y el acoplamiento.

Estas son complejas en general, no solo en el diseño de software. Estas ideas, que aplicamos en el diseño de nuestros sistemas, comienzan a replicarse y aplicarse en las organizaciones en las que operamos. Esto se puede explicar con ideas como la ley de Conway[6].

Esto se puede pensar en términos más técnicos: Una organización humana es un sistema de información como lo es cualquier sistema informático. Es definitivamente más complejo, pero los mismos principios se aplican.

Algunos elementos son fundamentalmente difíciles en ambos casos, como la concurrencia y el acoplamiento.

[6] En 1967, Mervin Conway observó que “Cualquier organización que diseñe un sistema producirá un diseño cuya estructura sea una copia de la organización de comunicación de su estructura”

Si queremos construir sistemas más complejos que un simple juego de práctica de programación, necesitamos tomarnos estas ideas con seriedad.
Necesitamos manejar la complejidad de los sistemas que creamos mientras los creamos; y si queremos hacer un sistema que trabaje más que solo un pequeño equipo, necesitamos manejar la complejidad de la organización en la que trabajamos, además de la complejidad técnica del sistema que diseñamos.

Como industria, tengo la impresión que prestamos muy poca atención a estas ideas. Tanto es así que aquellos que hemos trabajado suficiente tiempo en estos proyectos estamos familiarizados con las consecuencias: sistemas y código spaghetti, deuda técnica descontrolada, listados de errores enormes y organizaciones aterradas de implementar cambios en su propio sistema.

Percibo que todos estos son síntomas de organizaciones que han perdido el control de la complejidad de los sistemas que trabajan.

Si se está trabajando en un sistema simple y descartable no importa la calidad de su diseño. Pero si deseamos diseñar un sistema más complejo debemos fraccionar sus partes para evitar abrumarnos con su complejidad total.

Donde se hace la diferencia entre un sistema y otro depende de muchas variables, pero lo importante es entender que en algún momento se debe fraccionar el sistema para poder resolver problemas más complejos. Esto tiene un gran impacto en términos del diseño y la arquitectura de los sistemas que creamos.

Una de los muchos aprendizajes que podemos tomar de nuestro uso informal de la ciencia es que tendemos a sobreestimar nuestra capacidad de codear para resolver problemas. Es por esto que es útil asumir que todas nuestras soluciones son erróneas. Si nos preocupamos por esto, estaremos atentos a contener la explosión de complejidad del sistema que vamos creando.

Existen cinco ideas en esta categoría también. Estas ideas están correlacionadas entre sí y con el proceso de convertirse en **expertos del aprendizaje**.

Estas son las cinco ideas que vale la pena tener en cuenta si tenemos que manejar la complejidad de cualquier sistema de información:

- Modularidad.
- Cohesión.
- Separación de preocupaciones.
- Abstracción y escondimiento de información.
- Acoplamiento.

Exploraremos más cada una de estas ideas en la parte III.

Resumen

Las verdaderas herramientas de nuestro rubro no son las que normalmente tenemos en mente. Los lenguajes, herramientas y frameworks cambian con el paso del tiempo y los cambios de proyecto, mientras que las ideas que facilitan nuestro aprendizaje y nos ayudan a manejar la complejidad de nuestros sistemas son las verdaderas herramientas de nuestro rubro. Al centrarnos en estas, elegiremos mejor los lenguajes que usamos, manejaremos mejor nuestras herramientas y aplicaremos los frameworks de la mejor manera que nos ayude a resolver problemas de software.

Tener con qué medir estos elementos es una enorme ventaja si queremos tomar decisiones basadas en evidencia e información, a diferencia de moda o conjeturas. Cuando tomemos una decisión debemos preguntarnos ¿esto mejora la calidad del software que estamos creando? (medido en términos de estabilidad). O ¿mejora esto la eficiencia con la que creamos software de esa calidad? (medido en términos de flujo).
Si no empeora a ninguno de esos dos sentidos, podemos hacer la elección que queramos; pero si no es el caso ¿por qué elegiríamos algo que empeora cualquiera de esos dos aspectos?