> *El software no madura cuando se completa.*
> *Madura cuando aprende a evolucionar.*

Prefacio:

Antes de construir, aprender a escuchar el cambio

La mayoría del software nace desde una premisa silenciosamente equivocada: creemos que estamos construyendo una solución, cuando en realidad estamos capturando una fotografía temporal de una organización en movimiento.

Los negocios cambian, los usuarios aprenden, los procesos se tensionan y los datos envejecen. La operación descubre excepciones que nadie imaginó en el Sprint 1. Sin embargo, gran parte de la industria sigue diseñando sistemas como si el presente pudiera congelarse. Ese es el origen de gran parte de la deuda que no aparece en el código, pero sí en la frustración de los equipos y en la pérdida de confianza de los usuarios.

Este libro nace desde una convicción práctica: **el problema del software no comienza en la implementación, sino en la interpretación de la necesidad.**

Cuando una organización pide una funcionalidad, rara vez describe la necesidad real; verbaliza la mejor solución que conoce desde su experiencia previa y desde la forma en que ha sobrevivido operativamente hasta hoy. Por eso, este marco propone un cambio de mirada. No se trata solo de desarrollar por etapas o de aplicar UX; se trata de construir sistemas capaces de evolucionar sin perder la verdad de lo vivido.

En este modelo, donde Adaptive Software Development (ASD) deja de ser solo una metodología incremental para transformarse en una disciplina de lectura continua del cambio. El objetivo es uno: diseñar sistemas que acompañen la evolución del negocio sin borrar la historia que explica por qué evolucionaron.

Este libro propone una evolución del pensamiento ASD integrando UX, trazabilidad semántica y transición progresiva de modelos, para diseñar software capaz de aprender del negocio sin degradar su verdad histórica.

Porque el software maduro no es el que más funcionalidades tiene. Es el que puede seguir cambiando sin perder el sentido. Y para lograrlo, todo comienza mucho antes del código: en la forma en que aprendemos a interpretar la necesidad y escuchar el cambio.

Antes de comenzar

Conceptos base para navegar esta obra

ASD
Adaptive Software Development.

Metodología de desarrollo propuesta por Jim Highsmith a inicios de los años 2000, como evolución de enfoques iterativos orientados al cambio. Su eje central reemplaza la planificación rígida por un ciclo continuo de Especular, Colaborar y Aprender, entendiendo que el software existe dentro de negocios que cambian constantemente.

En esta obra, ASD se expande desde metodología ágil hacia una filosofía de evolución continua, donde no solo se gestiona incertidumbre, sino también significado, memoria histórica y trazabilidad de decisiones.

UX
User Experience (Experiencia de Usuario).

Disciplina enfocada en cómo una persona comprende, usa y confía en un sistema.

Sus raíces modernas provienen de HCI (Human–Computer Interaction), área que estudia la relación entre seres humanos y tecnología desde la psicología cognitiva, la ergonomía y el diseño de interacción.

En este libro, UX no se limita a pantallas o estética. Se redefine como una capa de protección cognitiva de decisiones, cuyo propósito es ayudar al usuario a interpretar contexto, riesgo y significado antes de actuar.

HCI
Human–Computer Interaction.

Campo interdisciplinario nacido entre la informática, la psicología cognitiva y el diseño industrial, orientado a estudiar cómo las personas interactúan con sistemas computacionales.

HCI es la raíz teórica de la UX moderna y aporta principios como: carga cognitiva, memoria de trabajo, affordance, feedback, prevención de errores.

Backlog
Lista priorizada de trabajo pendiente.

Tradicionalmente contiene historias de usuario, bugs o tareas técnicas. En esta obra, el backlog evoluciona desde una lista funcional hacia una brújula de decisiones orientadas a outcome, donde cada sprint debe mejorar la capacidad del negocio para decidir.

Outcome
Resultado observable que cambia positivamente una capacidad del negocio o la calidad de una decisión.

A diferencia de una funcionalidad, un outcome no describe qué se construye, sino qué mejora ocurre en la realidad.

Framework NRTOV

Necesidad → Restricción → Tarea → Outcome → Valor

Es el marco de traducción estratégica que se propone para convertir pedidos literales en capacidades sostenibles de negocio.

Trazabilidad
Capacidad del sistema para reconstruir: qué cambió, por qué cambió, quién lo cambió, bajo qué contexto cambió

Deuda semántica
Es la pérdida progresiva de significado entre:

necesidad real → modelo → dato → decisión → interfaz

Aparece cuando el sistema sigue funcionando técnicamente, pero deja de representar correctamente la verdad operacional del negocio.

Override
Intervención humana sobre una sugerencia, regla o automatización del sistema.

Feature creep
Crecimiento descontrolado de funcionalidades sin una mejora real del outcome.

Zero Hard Migration
Principio arquitectónico donde un modelo nuevo convive progresivamente con el antiguo, evitando reemplazos bruscos.

Este lenguaje no busca complejizar el desarrollo, sino volver visible aquello que normalmente se pierde entre requerimientos, código y operación.

A partir de este punto, cada capítulo utilizará estos conceptos como piezas de una misma disciplina: diseñar software capaz de evolucionar sin perder verdad.

> *El software no falla primero por bugs. Falla por significado*

Capítulo 1:

Por qué el software falla antes del primer sprint

Mucho antes de la primera historia de usuario, del primer wireframe o de la primera línea de código, existe una etapa silenciosa donde se define el destino real del producto: la traducción entre una necesidad operativa y la representación que el sistema hará de ella.

El sesgo del especialista y la ilusión de solución

En la mayoría de los desarrollos, quienes solicitan el software son especialistas profundos en su área. Dominan la operación, conocen las excepciones y han construido mecanismos personales para sobrevivir al caos diario. Este conocimiento es invaluable, pero introduce un **sesgo estructural**: el especialista privilegia la forma que ya conoce por sobre una representación más abstracta o eficiente.

- El supervisor que hoy usa Excel pedirá el mismo Excel, solo que "más rápido".
- El administrativo que limpia datos pedirá filtros para evitar su trabajo manual actual.
- El usuario final buscará que los botones estén donde su memoria muscular ya sabe encontrarlos.

El peligro real aparece cuando el equipo de desarrollo acepta estas soluciones literales como la arquitectura del producto. En ese momento, la conversación deja de girar sobre el modelo de negocio y empieza a orbitar sobre gustos personales. Cuando la interfaz gobierna al modelo de datos y a la trazabilidad, nace la primera deuda semántica del software.

1.1 La falsa sensación de claridad

Uno de los riesgos más críticos en la ingeniería de software no es la falta de información, sino la **ilusión de claridad**. La narrativa tradicional suele culpar a los retrasos o a la mala implementación, pero el fracaso temprano es semántico: ocurre cuando el equipo cree haber entendido el problema solo porque el usuario describió una solución familiar.

Esta "falsa claridad" se manifiesta cuando:

- El cliente describe una salida (un reporte) en lugar de una necesidad.
- Negocio confunde la urgencia del usuario con la prioridad estratégica.
- UX valida pantallas estéticas sin cuestionar las decisiones de fondo.
- Desarrollo asume que este proyecto es idéntico a uno anterior.

Lo costoso no es la ambigüedad evidente, sino la **ambigüedad disfrazada de certeza**. El Sprint 1 puede iniciar con una velocidad envidiable, pero si la dirección es incorrecta, solo estamos llegando más rápido al desastre.

1.2 El costo de escuchar deseos literales

Escuchar de manera literal produce software que automatiza costumbres, no resultados. Cuando un usuario dice "quiero el mismo informe de hoy", no está diseñando el futuro; está describiendo cómo aprendió a reducir su fricción diaria.

Si implementamos deseos literales, el sistema se llena de síntomas de vejez prematura: campos que nadie consulta, pantallas diseñadas para

excepciones que ocurren una vez al año y automatizaciones que solo aceleran procesos ineficientes.

Para evitarlo, el **Discovery de Necesidad Real** debe desplazar al simple levantamiento de requerimientos. Antes de tocar el backlog, debemos preguntarnos:

- ¿Qué decisión está intentando tomar el usuario con este dato?
- ¿Qué riesgo específico intenta mitigar?
- ¿Qué parte de su flujo actual es solo una adaptación temporal a una herramienta limitada?
- ¿Qué lógica debería sobrevivir si mañana cambiamos toda la interfaz?

1.3 Backlog por pantallas vs. Backlog por decisiones

Aquí reside el núcleo del cambio de paradigma. Un backlog tradicional organiza entregables visuales: "Pantalla de Clientes", "Modal de Edición", "Reporte de Ventas". Sin embargo, **una pantalla no es una unidad de valor**; es solo el contenedor visible de una decisión.

Propongo, en cambio, un **Backlog basado en Decisiones**:

- Decidir si el nivel de riesgo bloquea un crédito.
- Decidir si un producto nuevo debe asociarse a una categoría existente.
- Decidir si un evento requiere ser auditado en la trazabilidad.

Mientras que las pantallas son efímeras y cambian con la moda o el feedback de UX, las decisiones de negocio son persistentes. Si el backlog se construye sobre decisiones, el modelo de dominio y la base de datos se vuelven resistentes al cambio y verdaderamente evolutivos.

1.4 La deuda UX temprana

La deuda de experiencia de usuario no empieza con una interfaz "fea". Empieza cuando el modelo lógico obliga al usuario a realizar un esfuerzo cognitivo innecesario. Esta deuda nace en la fase de modelado cuando:

- El sistema obliga al usuario a recordar información que debería estar presente.
- El contexto se reparte en múltiples vistas sin una lógica de flujo.
- Los estados del sistema son ambiguos (¿se guardó o se procesó?).

Esta deuda no es superficial: queda incrustada en tablas rígidas y relaciones de base de datos pobres. La UX no empieza en el diseño visual; empieza en la forma en que decidimos representar la realidad en el código.

1.5 Microcaso: Cuando una pantalla rompe el negocio

Imagina que solicitan una pantalla para clasificar productos en categorías "A, B, C o D". La respuesta técnica estándar es añadir una columna categoria_abcd y un dropdown en la interfaz.

Pero, ¿qué pasa si esa clasificación depende de la rotación, el margen o la temporada? Si la implementamos como un atributo estático, hemos creado deuda técnica. Si la entendemos como una **decisión evolutiva**, la solución cambia: creamos un motor de eventos, un histórico por períodos y una relación lógica desacoplada.

La solución clásica entrega una pantalla rápido. La solución basada en decisiones entrega **flexibilidad competitiva**.

El fracaso temprano es un problema de traducción. Antes del Sprint 1, el trabajo real no es programar; es asegurar que no estamos confundiendo un pedido con una necesidad.