

# Ingeniería de Software Moderna: Principios para Construir Mejor y Más Rápido

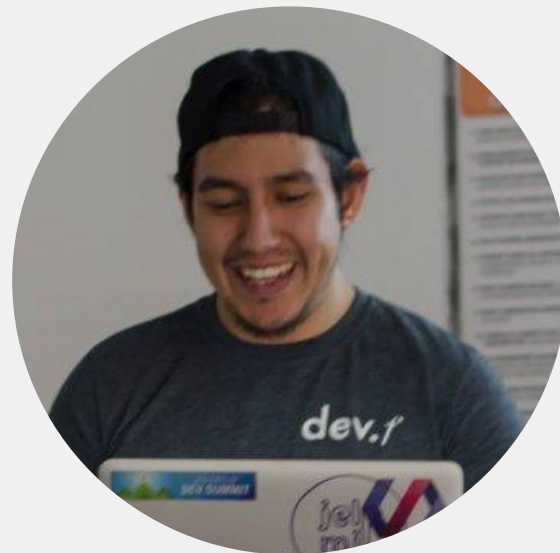


Software empresarial fácil y completo

# Kike Diaz

Co-founder & Machine Learning  
Engineer @ DEVF, CodeFlux.ai

GDG | TFUG CDMX co-organizer  
@cogitovsmachina






# Ingeniería de Software Moderna: Principios para Construir Mejor y Más Rápido

- 0. El Verdadero Enemigo:** La Complejidad. (Por qué necesitamos mejorar).
- 1. Nuestro Mayor Aliado:** El Aprendizaje. (El cambio de mentalidad fundamental).
- 2. Sostenibilidad en la Práctica:** Cultura, Procesos y Herramientas. (Cómo aplicamos esta mentalidad).
- 3. Preguntas y Respuestas.**
- 4. Recursos y Siguiendo Pasos.**

# 0. El Verdadero Enemigo: La Complejidad. (Por qué necesitamos mejorar).

# 0. El Verdadero Enemigo: La Complejidad.

## (Por qué necesitamos mejorar).

 **Objetivo:** Establecer un entendimiento común del problema principal que enfrentamos. La **complejidad** es el factor que **ralentiza** el desarrollo, **introduce bugs** y **causa frustración**.

Fuentes Principales: A Philosophy of Software Design (Ousterhout).

**"La complejidad es incremental. Tienes que preocuparte por las cosas pequeñas".**




# 0. ¿Qué es la complejidad?




No se trata del tamaño, sino de lo que hace que un sistema sea **difícil de entender y modificar.**

## **Síntomas de la Complejidad (Ousterhout):**


 Amplificación del Cambio: Un cambio simple requiere modificaciones en muchos lugares.


 Carga Cognitiva: Un desarrollador necesita saber demasiadas cosas para completar una tarea.

 "Desconocidos Desconocidos": No es obvio qué código o información se necesita para hacer un cambio.

# 0. Programación Táctica vs. Estratégica

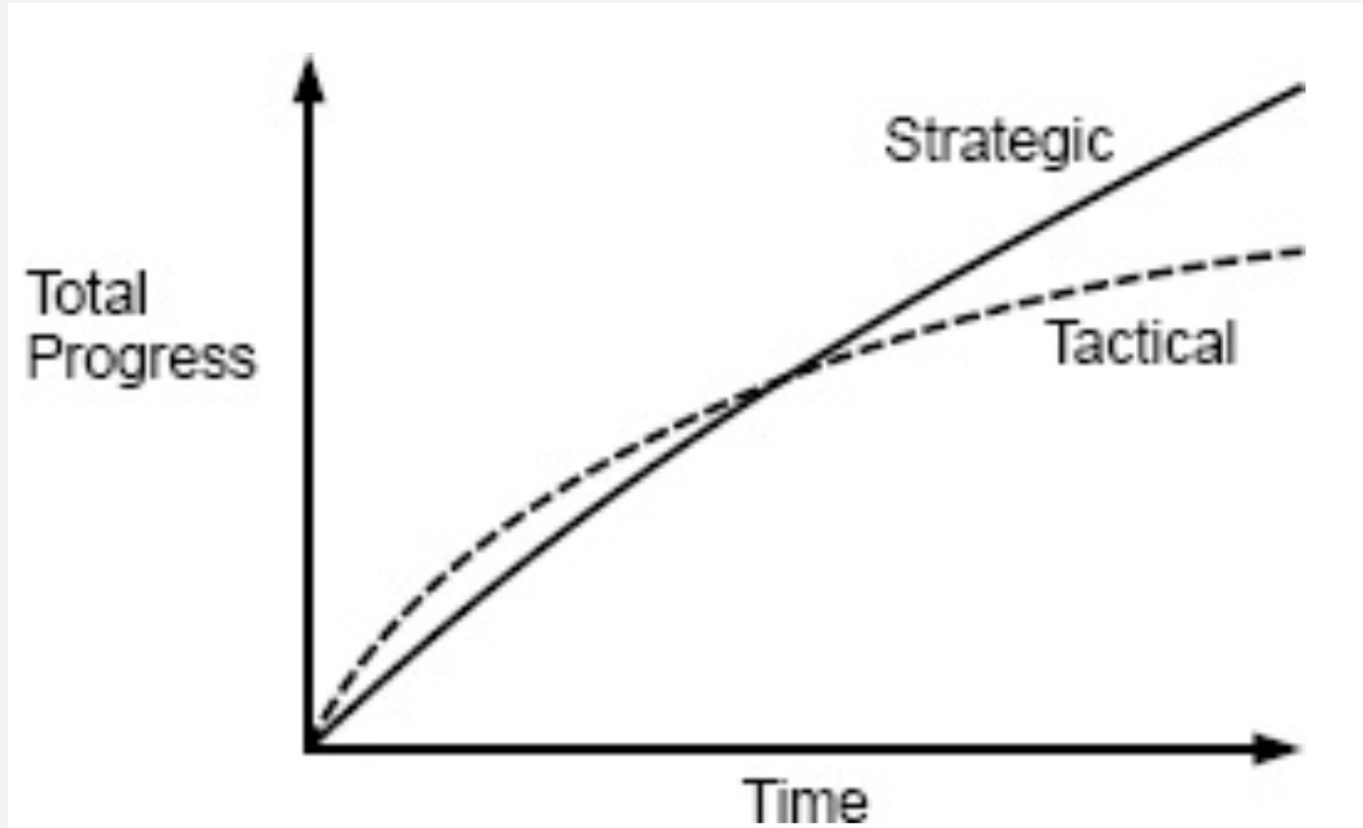


 **Táctica (El Tornado Táctico):** El objetivo es hacer que algo funcione lo más rápido posible, acumulando deuda técnica. ("Working code isn't enough").

 **Estratégica (La mentalidad de inversión):** El objetivo es **producir un gran diseño que también funcione**. Invertir un 10-20% de tiempo extra en diseño y refactorización para acelerar a largo plazo.





# 0. Programación Táctica vs. Estratégica



# 0. Nuestra Primera Herramienta: El Diseño de Módulos Profundos (Deep Modules).



 **Módulos Superficiales (Shallow):** La interfaz es casi tan compleja como la implementación (e.g., getters/setters simples). Aportan poco valor.

 **Módulos Profundos (Deep):** Interfaz simple que oculta una gran funcionalidad. (Ej: La API de I/O de Unix).

Principio clave: **"Es más importante que un módulo tenga una interfaz simple que una implementación simple".**

# 0. En resumen




 La complejidad es nuestro enemigo número uno. Se acumula con cada pequeña decisión "táctica".

 Adoptar una mentalidad estratégica no es un lujo, es una necesidad para la sostenibilidad.

 Nuestro objetivo de diseño debe ser crear abstracciones profundas que oculten la complejidad.

# 1. Nuestro Mayor Aliado: El Aprendizaje. (El cambio de mentalidad fundamental).

# 0. Nuestro Mayor Aliado: El Aprendizaje


 **Objetivo:** Introducir el cambio de mentalidad fundamental. La ingeniería de software no es un proceso de fabricación, sino un proceso de descubrimiento y aprendizaje continuo.

Fuente Principal: Modern Software Engineering (Farley).

# 1. La Ingeniería de Software es una Disciplina de Aprendizaje



Definición de Farley: **"La aplicación de un enfoque empírico y científico para encontrar soluciones eficientes y económicas a problemas prácticos en el software".**

 No se trata de planes rígidos, sino de navegar la incertidumbre.  
**Nuestro trabajo principal es aprender.**

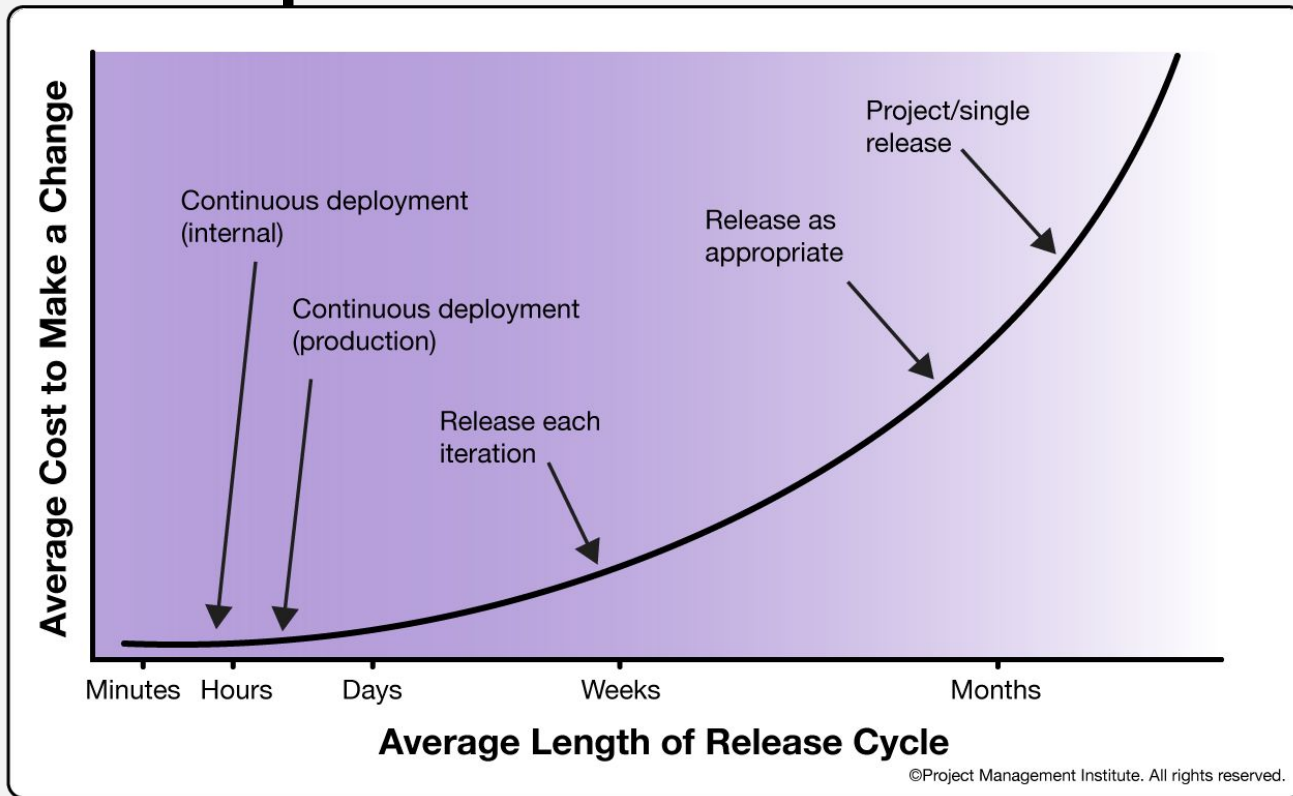
# 1. El Motor del Aprendizaje: Los Ciclos de Feedback Rápidos



 **Cuanto más tardamos en descubrir un error, más caro es corregirlo.**

 **El objetivo es "aplanar la curva del costo del cambio" trabajando de forma que el feedback sea constante y rápido.**


# 1. El Motor del Aprendizaje: Los Ciclos de Feedback Rápidos







# 1. ¿Cómo Obtenemos Feedback Rápido?



 **Iteración:** Trabajar en pequeños ciclos. No intentar construir todo de una vez.

 **Testing (TDD como herramienta de diseño):** Los tests no son solo para verificar, son para guiar el diseño. Un test difícil de escribir indica un mal diseño.


 **Integración Continua (CI):** La pulsación del equipo. Integrar el trabajo de todos constantemente (varias veces al día) para descubrir problemas de inmediato.

# 1. En resumen



 Debemos optimizar nuestros procesos para aprender lo más rápido posible.


 El feedback rápido no es opcional, es la herramienta más poderosa que tenemos.

 Prácticas como TDD y CI no son "extras", son la base de un proceso de aprendizaje eficiente.

## **2. Sostenibilidad en la Práctica:** **Cultura, Procesos y Herramientas.** **(Cómo aplicamos esta** **mentalidad).**

## 2. Sostenibilidad en la Práctica: Cultura, Procesos y Herramientas




 **Objetivo:** Mostrar cómo los principios de "lucha contra la complejidad" y "optimización del aprendizaje" se manifiestan en prácticas concretas y sostenibles a lo largo del tiempo.

Fuente Principal: Software Engineering at Google (Winters, Manshreck, & Wright).

## 2. Software Engineering is Programming


### Integrated Over Time


 El software no solo se escribe, se mantiene durante años o décadas.


 Los tres pilares para lograrlo: Cultura, Procesos y Herramientas.

## 2. Pilar 1: Cultura - El Equipo por Encima del Individuo



 **Seguridad Psicológica:** Es la base de todo. La gente debe sentirse segura para preguntar, equivocarse y dar feedback.

 **Humildad, Respeto y Confianza:** Los "tres pilares" de la interacción social. No hay lugar para el "programador genio solitario".

 **Code Reviews como Colaboración:** El objetivo principal no es solo encontrar bugs, sino compartir conocimiento, mantener la consistencia y mejorar el código juntos.


## 2. Pilar 1: Cultura - El Equipo por Encima del Individuo



## 2. Pilar 2: Procesos - Consistencia a Escala




 **Trunk-Based Development:** Explicar por qué las ramas de desarrollo de larga duración son perjudiciales (retrasan el feedback, complican la integración).


 **La Pirámide del Testing:** La mayoría de los tests deben ser unitarios (rápidos, estables). Menos tests de integración, y muy pocos tests End-to-End (lentos, frágiles).



## 2. Pilar 3: Herramientas - Automatizar para Escalar






 **Sistemas de Build Modernos (e.g., Bazel):** Deben ser rápidos, correctos y reproducibles. La base para una CI/CD eficaz.

 **Análisis Estático: Automatizar la detección de errores comunes y la aplicación de guías de estilo.** Liberar a los humanos para que se centren en problemas más complejos.

## 2. En resumen



-  La sostenibilidad a largo plazo depende de una cultura de equipo saludable.
-  Los procesos deben favorecer la integración continua y un testing inteligente.
-  Las herramientas deben automatizar las tareas repetitivas para liberar el potencial humano.

A close-up portrait of a man with dark, curly hair and a light beard, wearing black-rimmed glasses. He is looking upwards and to the left with a thoughtful expression, his hand resting on his chin. The background is a dark, textured grey.

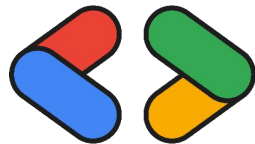
# Q&A

# 4. Recursos y Siguietes Pasos.

# Ingeniería de Software Moderna: Principios para Construir Mejor y Más Rápido



Software empresarial fácil y completo



Google Developer Group  
**Ciudad de México**

