Capítulo 1 del documento sobre Test-Driven Development

Desarrollo de Software como Proceso de Aprendizaje

- Los proyectos de software casi siempre intentan hacer algo que nadie ha hecho antes
- Todos los involucrados (desarrolladores, clientes, usuarios) deben aprender durante el proceso
- Los proyectos interesantes suelen tener muchas sorpresas e incertidumbre

La Retroalimentación como Herramienta Fundamental

- El mejor enfoque es usar retroalimentación empírica para aprender sobre el sistema
- Se necesitan ciclos repetidos de actividad con retroalimentación constante
- Los ciclos de retroalimentación anidados van desde segundos hasta meses (programación en parejas, pruebas unitarias, pruebas de aceptación, reuniones diarias, iteraciones, lanzamientos)

Desarrollo Incremental e Iterativo

- **Incremental**: construye el sistema característica por característica, no por capas
- **Iterativo**: refina progresivamente la implementación en respuesta a la retroalimentación
- El sistema siempre está integrado y listo para despliegue

Test-Driven Development (TDD)

Regla de Oro del TDD

"Nunca escribas nueva funcionalidad sin una prueba que falle"

Ciclo Fundamental del TDD

1. Escribir una prueba
2. Escribir código para que funcione
3. Refactorizar el código para que sea lo más simple posible
4. Repetir

Beneficios de Escribir Pruebas Primero

- **Diseño**: Clarifica criterios de aceptación y fuerza componentes débilmente acoplados
- **Implementación**: Detecta errores temprano y crea una suite de regresión completa

Estructura de Pruebas (Bucles Anidados)

- **Bucle Externo**: Pruebas de aceptación (¿funciona todo el sistema?)
- **Bucle Interno**: Pruebas unitarias (apoyan a los desarrolladores)

Niveles de Pruebas

1. **Aceptación**: ¿Funciona todo el sistema?
2. **Integración**: ¿Funciona nuestro código con código que no podemos cambiar?
3. **Unitarias**: ¿Hacen nuestros objetos lo correcto y son convenientes de usar?

Pruebas Extremo a Extremo

- Deben ejercitar el sistema desde el exterior (interfaz de usuario, servicios web, etc.)
- Incluyen tanto el sistema como el proceso de construcción y despliegue
- Son críticas para detectar problemas de integración

Calidad Externa vs Interna

- **Externa**: Qué tan bien satisface las necesidades de clientes y usuarios
- **Interna**: Qué tan bien satisface las necesidades de desarrolladores (facilidad de entendimiento y cambio)

Refactorización

- Cambiar la estructura interna del código sin cambiar su comportamiento
- Técnica disciplinada de pequeñas transformaciones "seguras"
- Mejora la representación del código y lo hace más mantenible

Acoplamiento y Cohesión

- **Acoplamiento**: Elementos están acoplados si un cambio en uno fuerza un cambio en el otro (deseable: bajo acoplamiento)
- **Cohesión**: Medida de si las responsabilidades forman una unidad significativa (deseable: alta cohesión)

El TDD convierte las pruebas de una actividad de verificación en una actividad de diseño, proporcionando retroalimentación rápida sobre la calidad del diseño y creando confianza para hacer cambios.

Capítulo 4 sobre "Kick-Starting the Test-Driven Cycle"

El Problema del Primer Test

- El proceso TDD asume una infraestructura existente, pero ¿qué pasa con la primera característica?
- La primera prueba de aceptación debe ejecutarse extremo a extremo, requiriendo todo un ciclo automatizado de construcción, despliegue y pruebas
- Esto es mucho trabajo antes de poder ver fallar el primer test

La Importancia del Despliegue Temprano

- Fuerza al equipo a entender cómo su sistema encaja en el mundo
- Expone riesgos técnicos y organizacionales "desconocidos desconocidos"
- Ayuda a identificar con quién deben colaborar (administradores de sistemas, proveedores externos)
- Los riesgos de dejarlo para después son demasiado altos

"Walking Skeleton" (Esqueleto Caminante)

Definición

Una implementación de la **porción más delgada posible de funcionalidad real** que se puede construir, desplegar y probar automáticamente de extremo a extremo.

Características

- Incluye solo la automatización, componentes principales y mecanismos de comunicación mínimos necesarios
- La funcionalidad de aplicación se mantiene tan simple que es obvia y sin interés
- Permite concentrarse en la infraestructura
- Ejemplo: para una aplicación web con base de datos, mostrar una página plana con campos de la base de datos

Propósito

- Resolver la "paradoja de la primera característica"
- Dividir el problema en dos partes más pequeñas:
    1. Construir, desplegar y probar el "esqueleto caminante"
    2. Usar esa infraestructura para escribir pruebas de aceptación de la primera característica significativa

Extremo a Extremo Verdadero

- "Extremo" se refiere tanto al **proceso** como al **sistema**
- El test debe comenzar desde cero, construir un sistema desplegable, desplegarlo en un entorno similar a producción y ejecutar pruebas
- Incluir el paso de despliegue es crítico por dos razones:
    1. Es una actividad propensa a errores que no debe hacerse manualmente
    2. Es donde el equipo de desarrollo se encuentra con el resto de la organización

Decidiendo la Forma del Esqueleto Caminante

- Momento de hacer elecciones sobre la estructura de alto nivel de la aplicación
- **Regla práctica**: debería poder dibujarse el diseño en pocos minutos en una pizarra
- No es "Big Design Up Front" (BDUF) - solo las decisiones mínimas necesarias para iniciar el ciclo TDD
- Se necesita una vista de alto nivel de los requerimientos funcionales y no funcionales

Construcción de Fuentes de Retroalimentación

Situación Ideal

- El equipo libera regularmente a un sistema de producción real
- Permite a los stakeholders responder qué tan bien el sistema satisface sus necesidades
- Permite juzgar la implementación

Beneficios de la Automatización

- Retroalimentación sobre cualidades del sistema (facilidad de versionar y desplegar, qué tan bien funciona el diseño)
- Despliegue automatizado ayuda a liberar frecuentemente a usuarios reales
- Conjunto completo de pruebas de regresión permite hacer cambios mayores de forma segura

Exponiendo la Incertidumbre Temprano

Desarrollo Incremental vs Tardío

- **Desarrollo incremental**: "front-loads" el estrés en el proyecto
- **Integración tardía**: comienza calmadamente pero se vuelve difícil al final

Patrones de Desarrollo

- **Desarrollo incremental bien ejecutado**: comienza inestable pero se asienta en una rutina
- **Integración tardía**: impredecible porque debe ensamblar muchas partes móviles con tiempo y presupuesto limitados

Beneficios del Esqueleto Caminante

- Expone problemas temprano cuando aún hay tiempo, presupuesto y buena voluntad para abordarlos
- Automatiza tareas mundanas pero frágiles (despliegue, actualizaciones)

Desarrollo Brownfield

- Trabajar con sistemas existentes que deben extenderse, adaptarse o reemplazarse
- Proceso similar pero potencialmente más difícil debido al "bagaje técnico"
- **Enfoque más seguro**:
    1. Automatizar el proceso de construcción y despliegue
    2. Agregar pruebas extremo a extremo que cubran las áreas que necesitan cambio
    3. Con esa protección, abordar problemas de calidad interna con más confianza

La clave es comenzar con el **camino más simple** a través del sistema, similar a un "esqueleto caminante", para construir infraestructura de soporte antes de abordar funcionalidades más complicadas.

Capítulo 5: Manteniendo el Ciclo de TDD

Proceso de Desarrollo TDD

Comenzar cada funcionalidad con una prueba de aceptación

- Escribir pruebas de aceptación que fallen antes de implementar la funcionalidad
- Usar únicamente terminología del dominio de la aplicación, no de tecnologías subyacentes
- Las pruebas de aceptación protegen contra cambios en la infraestructura técnica
- Ayuda a clarificar qué se quiere lograr y descubre suposiciones implícitas

Separar pruebas que miden progreso de las que detectan regresiones

- Nuevas pruebas de aceptación: representan trabajo por hacer (pueden fallar)
- Pruebas de funcionalidades completadas: detectan regresiones (deben pasar siempre)
- Pruebas unitarias y de integración: apoyan al equipo de desarrollo (deben ejecutarse rápido y pasar siempre)

Mejores Prácticas de Testing

Comenzar con el caso de éxito más simple

- No empezar con casos degenerados o de falla
- Los casos de éxito dan mejor retroalimentación sobre la validez de las ideas
- Mejora la moral del equipo
- Mantener una lista de casos de falla y refactorizaciones para abordar después

Escribir la prueba que te gustaría leer

- Concentrarse en la claridad del texto de la prueba
- Ignorar inicialmente que la prueba no compile o ejecute
- Construir la infraestructura de soporte después
- Hacer que la prueba falle de la manera esperada antes de escribir el código

Observar la prueba fallar

- Siempre ver la prueba fallar antes de escribir el código
- Verificar que el mensaje de diagnóstico sea claro y útil
- Ajustar el código de prueba hasta que los mensajes de error guíen al problema

Desarrollar desde las entradas hacia las salidas

- Comenzar con los eventos que llegan al sistema
- Trabajar desde objetos que reciben eventos externos hasta el modelo de dominio
- Continuar hasta objetos que generan respuestas visibles externamente
- Evita problemas de integración tardíos

Principios de Testing

Probar comportamiento, no métodos

- Enfocarse en las funcionalidades que el objeto debe proveer
- Una prueba como `testBidAccepted()` dice qué hace, pero no para qué sirve
- Pensar en cómo usar la clase para lograr un objetivo
- Elegir nombres de pruebas que describan cómo se comporta el objeto

Escuchar a las pruebas

- Cuando el código es difícil de probar, probablemente el diseño necesita mejoras
- La dificultad para escribir pruebas es una señal de alerta temprana
- Refactorizar el código de producción cuando las pruebas son difíciles de escribir

MOCKS

Los **mock objects** son objetos simulados que imitan el comportamiento de objetos reales en tests. Se usan para **aislar la unidad de código bajo prueba** y **verificar interacciones**, sin depender de implementaciones reales (como bases de datos o APIs).

Tipos de Mock Objects (según Martin Fowler):

1. **Dummy**:
   - Solo están para completar parámetros.
   - No se usan en el test.
2. **Fake**:
   - Tienen lógica simple que funciona, pero no es la real.
   - Ej: un repositorio en memoria.
3. **Stub**:
   - Devuelven respuestas fijas para métodos.
   - No registran interacciones.
   - Ej: `userRepo.get(id)` devuelve un `User` fijo.
4. **Spy**:
   - Como un stub, pero **registra llamadas** (cuántas veces, con qué argumentos).
   - Se usa para verificar comportamiento.
5. **Mock (estricto)**:
   - Se le definen **expectativas** de llamadas.
   - Falla si no se cumplen (cuándo y cómo se llamaron los métodos).

Test End-to-End (E2E)

- Prueba **todo el sistema como un todo**: desde el inicio (por ejemplo, la UI) hasta el final (como la base de datos o backend).
- Simula el uso real del sistema.
- Ej: un test que hace login, navega, hace una compra y verifica el resultado final.

Test Edge-to-Edge

- Prueba **interacciones entre dos o más módulos importantes** sin pasar por todo el sistema.
- No testea el sistema completo, pero sí una **parte crítica de extremo a extremo dentro de un subsistema**.
- Ej: testea una API REST desde el controlador hasta la base de datos, sin pasar por la interfaz gráfica.

Test End-to-Edge

- Comienza en un **extremo real** (ej: interfaz de usuario o API pública), pero **simula el resto** del sistema con mocks/stubs.
- Verifica que la entrada real funciona, pero **no llega hasta el fondo** del sistema.
- Ej: testea el frontend interactuando con una API mockeada.

¿Qué dice Friedman?

Michael Friedman, al escribir sobre testing, sugiere que:

- **E2E tests son valiosos pero lentos y frágiles**.
- **Edge-to-Edge tests son más rápidos, más confiables** y deberían usarse más, ya que prueban partes críticas sin depender de todo el stack.
- Los **End-to-Edge tests** ayudan a testear desde interfaces reales, pero sin el costo de los E2E.

Capítulo 6: Estilo Orientado a Objetos

Diseñando para Mantenibilidad

Principios fundamentales:

Separación de responsabilidades

- Cambiar el menor código posible cuando se modifica el comportamiento
- Agrupar código que cambiará por las mismas razones
- Ejemplo: código de protocolos separado del código de negocio

Niveles más altos de abstracción

- Programar combinando componentes de funcionalidad útil
- Como ordenar comida de un menú en términos de platos, no recetas detalladas

Arquitectura "Puertos y Adaptadores"

- Aislar el código del dominio de negocio de sus dependencias técnicas
- Escribir interfaces que describan relaciones con el mundo exterior
- Crear puentes (adaptadores) entre el núcleo de aplicación y dominios técnicos

Encapsulación vs. Ocultación de Información

Encapsulación

- Asegura que el comportamiento de un objeto solo se pueda afectar a través de su API
- Controla el impacto de cambios entre componentes no relacionados

Ocultación de información

- Oculta cómo un objeto implementa su funcionalidad detrás de su API
- Permite trabajar con abstracciones más altas ignorando detalles de bajo nivel

Estructura de Objetos

Internos vs. Pares

- Decidir qué está dentro y fuera de cada objeto
- Los objetos se comunican enviando y recibiendo mensajes
- Evitar exponer demasiados detalles internos a través del API

Estereotipos de Objetos Par:

1. **Dependencias**: Servicios requeridos para funcionar (pasados en constructor)
2. **Notificaciones**: Pares que necesitan mantenerse actualizados ("disparar y olvidar")
3. **Ajustes**: Pares que ajustan el comportamiento (patrones Strategy, componentes)

Principios de Diseño

"Sin Y's, O's, o Pero's"

- Cada objeto debe tener una responsabilidad única y claramente definida
- Poder describir lo que hace un objeto sin usar conjunciones
- Si se necesitan cláusulas adicionales, probablemente debe dividirse

"Crear o no crear. No hay intentar"

- Siempre crear objetos válidos
- Pasar dependencias a través del constructor
- Notificaciones y ajustes pueden tener valores por defecto seguros

Compuesto más simple que la suma de sus partes

- El API del objeto compuesto debe ser más simple que sus componentes
- Ocultar la existencia e interacciones de las partes componentes
- Exponer una abstracción más simple a sus pares

Independencia del contexto

- Los objetos no deben tener conocimiento incorporado del sistema donde ejecutan

- Lo que necesiten saber del entorno debe pasarse como parámetro
- Facilita reutilizar objetos en nuevas situaciones
- Hace las relaciones explícitas y manejables

Un Vocabulario de Dominio

- Una clase que usa términos de múltiples dominios podría violar la independencia del contexto
- Excepción: capas de conexión entre dominios

Ocultando la Información Correcta

- La encapsulación es buena, pero la información puede ocultarse en el lugar equivocado
- Ser claro sobre la diferencia entre encapsulación y ocultación de información
- La independencia del contexto ayuda a identificar cuándo se ocultan detalles del nivel incorrecto

Arquitectura Hexagonal

La arquitectura hexagonal resuelve un problema muy común: cuando desarrollamos aplicaciones, mezclamos la lógica de negocio con detalles técnicos como bases de datos, APIs REST o interfaces de usuario. Esto hace que cambiar cualquier tecnología sea un dolor de cabeza porque todo está entrelazado.

Alistair Cockburn propuso una solución elegante: imagina tu aplicación como un hexágono donde el centro contiene únicamente las reglas de negocio, y los bordes se conectan con el mundo exterior a través de "puertos" y "adaptadores". Es como si tu aplicación fuera una caja negra que solo sabe hacer su trabajo, sin importarle si los datos vienen de una base de datos MySQL o PostgreSQL, o si la interfaz es web o móvil.

Cómo Funciona en la Práctica

El núcleo de tu aplicación contiene todo lo importante: las entidades de dominio (como Usuario, Producto, Pedido), los casos de uso (como "registrar usuario" o "procesar pago"), y las reglas de negocio. Este núcleo es completamente ignorante del mundo exterior. No sabe qué es una base de datos, no conoce HTTP, no tiene idea de qué es Spring Boot.

Para comunicarse con el exterior, el núcleo define "puertos", que son simplemente interfaces. Por ejemplo, si necesita guardar un usuario, define una interfaz `UserRepository` con métodos como `save(user)` y `findById(id)`. Si necesita enviar emails, define `EmailService` con un método `send(email, message)`. Estos puertos son contratos que dicen "necesito esto, pero no me importa cómo lo hagas".

Los "adaptadores" son implementaciones concretas de estos puertos. Tendrás un `MySQLUserRepository` que implementa `UserRepository` usando JPA, y un `SMTPEmailService` que implementa `EmailService` usando JavaMail. Si mañana quieres cambiar a MongoDB, simplemente creas un `MongoUserRepository` y cambias la configuración. El núcleo ni se entera.

Un Ejemplo Concreto

Supongamos que estás construyendo un sistema de biblioteca. El núcleo contiene la lógica: "un libro puede prestarse si está disponible y el usuario no tiene multas pendientes". Esta regla nunca cambia, independientemente de si usas MySQL o PostgreSQL, si tienes una app web o móvil.

El caso de uso "prestar libro" sería algo así: recibe un ID de libro y un ID de usuario, verifica las condiciones de negocio, actualiza el estado del libro, registra el préstamo, y opcionalmente envía una confirmación por email. Para hacer esto, necesita algunos servicios externos que declara como interfaces: `BookRepository`, `UserRepository`, `LoanRepository`, y `NotificationService`.

Luego tienes adaptadores concretos: `JpaBookRepository` para la persistencia, `RestBookController` para recibir peticiones HTTP, `EmailNotificationService` para enviar emails. Si decides migrar de JPA a MongoDB, solo cambias el adaptador de persistencia. Si quieres agregar una API GraphQL además de REST, solo agregas un nuevo adaptador de entrada.

La Magia de la Inversión de Control

El truco está en que las dependencias fluyen hacia adentro. El núcleo define qué necesita mediante interfaces, y los adaptadores se adaptan a esas interfaces. Es como si el núcleo dijera "yo defino las reglas del juego" y los adaptadores responden "ok, nosotros nos adaptamos a tus reglas".

En código tradicional, tu lógica de negocio dependería directamente de la base de datos o del framework web. En arquitectura hexagonal, estos detalles técnicos dependen de tu lógica de negocio. Esto significa que puedes cambiar tecnologías sin tocar el núcleo, y puedes probar el núcleo sin levantar una base de datos.

Los Beneficios Reales

La testabilidad se vuelve trivial porque puedes crear implementaciones falsas (mocks) de los puertos para probar el núcleo en aislamiento. Quieres probar que un usuario no puede pedir prestado más de 5 libros? Creas un `FakeUserRepository` que simula un usuario con 5 libros prestados y verificas que el caso de uso rechaza la operación.

El desarrollo paralelo se simplifica enormemente. Un equipo puede trabajar en la API REST mientras otro desarrolla la app móvil, porque ambos usan el mismo núcleo. Un tercer equipo puede estar migrando la base de datos sin afectar a nadie más.

Los cambios tecnológicos dejan de ser proyectos de meses. Migrar de Spring MVC a Spring WebFlux, o de MySQL a PostgreSQL, se convierte en cambiar algunos adaptadores y actualizar la configuración.

Cuándo Vale la Pena

Esta arquitectura brilla en aplicaciones complejas donde la lógica de negocio es sustancial y las tecnologías cambian frecuentemente. Si tu aplicación es básicamente un CRUD simple, probablemente es excesivo. Pero si tienes reglas de negocio complejas, múltiples interfaces de entrada, integración con sistemas externos, o un equipo grande, la inversión inicial en complejidad se paga rápidamente.

La clave está en reconocer que no todos los proyectos necesitan esta estructura. Para un prototipo rápido o una aplicación interna simple, puede ser overkill. Pero para sistemas que van a evolucionar y crecer, es una inversión en flexibilidad futura que vale la pena considerar.

Modelos Conceptual, de Objetos y Relacional

Modelo Conceptual: Representa la realidad del negocio de forma abstracta, identificando entidades, atributos y relaciones sin consideraciones técnicas.

Modelo de Objetos: Traduce el modelo conceptual a clases, objetos, herencia y polimorfismo del paradigma orientado a objetos.

Modelo Relacional: Estructura los datos en tablas con filas y columnas, usando claves primarias y foráneas para establecer relaciones.

Persistencia de Objetos en BD Relacionales

Se refiere al desafío de mapear objetos (con comportamiento y estado) a estructuras relacionales (tablas). Esto implica resolver diferencias como la herencia de objetos vs. estructura tabular, y mantener la integridad referencial entre objetos relacionados.

Implementación de Repositorios

El patrón Repository encapsula la lógica de acceso a datos, proporcionando una interfaz uniforme para operaciones CRUD. Actúa como una capa de abstracción entre la lógica de negocio y la persistencia, facilitando el testing y el cambio de tecnologías de almacenamiento.

Manejo de Evolución de BD (Migrations con Flyway)

Las migrations permiten versionar y evolucionar el esquema de base de datos de forma controlada. Flyway es una herramienta que ejecuta scripts SQL en orden secuencial, manteniendo un historial de cambios y asegurando que todos los entornos tengan la misma estructura de base de datos.

Introducción a los Métodos Ágiles

Los métodos ágiles son enfoques de desarrollo de software que priorizan la flexibilidad, colaboración y entrega continua de valor. Se basan en el Manifiesto Ágil con principios como individuos sobre procesos, software funcionando sobre documentación exhaustiva, colaboración con el cliente sobre negociación de contratos, y respuesta al cambio sobre seguir un plan rígido. Incluyen marcos como Scrum, Kanban y XP, enfocándose en iteraciones cortas, feedback constante y adaptación rápida a los cambios.

Introducción a Extreme Programming (XP)

XP es una metodología ágil que enfatiza las buenas prácticas técnicas y la comunicación constante. Sus valores fundamentales son simplicidad, comunicación, feedback, coraje y respeto. Las prácticas clave incluyen programación en parejas (pair programming), desarrollo dirigido por pruebas (TDD), integración continua, refactoring constante, releases pequeñas y frecuentes, cliente en sitio, y planificación

incremental. XP busca mejorar la calidad del código mientras mantiene la flexibilidad para adaptarse a cambios en los requisitos.

Introducción a la Gestión de Proyectos

Qué es un Proyecto

Un proyecto es un esfuerzo temporal emprendido para crear un producto, servicio o resultado único. Tiene un inicio y fin definidos, recursos limitados, y objetivos específicos que se deben alcanzar dentro de restricciones de tiempo, costo y alcance.

Triángulo de Acero

También conocido como "triple restricción", representa las tres variables fundamentales que determinan el éxito de un proyecto: **Tiempo** (cronograma), **Costo** (presupuesto)
y **Alcance** (funcionalidades/requisitos). Estas variables están interrelacionadas: cambiar una afecta a las otras. A veces se incluye una cuarta dimensión: la **Calidad**.

Definición del Éxito

El éxito de un proyecto se mide tradicionalmente por cumplir con el triángulo de acero, pero también incluye la satisfacción del cliente, el valor entregado al negocio, y el cumplimiento de los objetivos estratégicos. El éxito puede ser técnico (cumplir especificaciones) y comercial (generar valor).

Intervinientes (Roles) de un Proyecto

- **Sponsor/Patrocinador**: Proporciona recursos y apoyo ejecutivo
- **Project Manager**: Planifica, ejecuta y controla el proyecto
- **Equipo de Proyecto**: Realiza el trabajo técnico
- **Stakeholders**: Personas afectadas o interesadas en el proyecto
- **Cliente/Usuario Final**: Quien recibe el producto o servicio

Formas de Contratación

- **Precio Fijo**: Costo total acordado previamente
- **Tiempo y Materiales**: Se factura por horas trabajadas y recursos utilizados
- **Costo Plus**: Se reembolsan costos reales más una ganancia acordada
- **Contratos Híbridos**: Combinan elementos de los anteriores según las fases del proyecto

Estimación en Métodos Ágiles

Conceptos Clave

Pronosticar no es predecir: La estimación tiene incertidumbre inherente que disminuye con el tiempo (Cono de Incertidumbre de Boehm).

Estimación ≠ Planificación: Las estimaciones son aproximaciones, no compromisos. Sirven para planificar pero no representan fechas fijas.

Cuándo y Quién Estima

- **Dos momentos**: Estimación inicial (orden de magnitud) y estimaciones por iteración (más precisas)
- **Quién**: El equipo que desarrollará el trabajo, no terceros

Unidades de Estimación Ágil

- **Story Points**: Medida relativa de tamaño (usando escala Fibonacci: 1, 2, 3, 5, 8, 13...)
- **Días Ideales**: Tiempo sin interrupciones ni impedimentos
- **Escalas abstractas**: "Talles de camiseta" (S, M, L, XL) para estimaciones iniciales

Planning Poker

Técnica grupal donde cada estimador elige una carta privadamente, luego se revelan simultáneamente. Si hay diferencias, discuten los extremos hasta consensuar. Si persiste divergencia, se toma el valor más alto o el de quien asumirá el compromiso.

Spikes

Tareas de investigación time-boxed para reducir incertidumbre cuando no se puede estimar por falta de conocimiento. Su objetivo es obtener información suficiente para estimar la tarea original.

Planificación Ágil

"Lo importante no es el plan sino la planificación": Los métodos ágiles planifican continuamente, adaptándose a cambios. No significa ausencia de planificación, sino planificación adaptativa vs. seguir un plan rígido.

Conceptos Clave

Último Momento Prudente: Posponer decisiones hasta el último momento seguro para minimizar el costo de cambio y tomar mejores decisiones con más información.

Velocidad: Cantidad de trabajo que un equipo específico completa por iteración (en story points o días ideales). Es única por equipo y tamaño de iteración.

Slack: Tiempo/esfuerzo planificado para tareas no comprometidas con el cliente (pagar deuda técnica, investigación, mejoras). Diferente al "colchón" tradicional porque se planifica para actividades específicas.

Dos Niveles de Planificación

Planificación de Versiones (Alto Nivel):

- Horizonte de 3-6 iteraciones
- Determina cuántas iteraciones necesita el proyecto (total story points ÷ velocidad)

- Identifica MMFs (Minimum Marketable Features): conjuntos mínimos de funcionalidades con valor de mercado
- Solo detalla la primera versión

Planificación de Iteración (Bajo Nivel):

- Al inicio de cada iteración
- Equipo se compromete con user stories completas hasta alcanzar su velocidad
- Partidiona stories en tareas y estiman nuevamente
- Story points sobrantes se asignan a slack con tareas específicas
- Resultado: tablero con columnas (Pendiente, En Desarrollo, En Prueba, Completo)

Estimación de Velocidad Inicial

Para equipos nuevos: usar antecedentes organizacionales, convertir story points a horas/días, o consultar confianza del equipo para comprometerse con stories específicas.

Planificación

- **Estimar:**
  Implica calcular el esfuerzo necesario para completar una tarea o historia. No es un valor exacto, sino una aproximación basada en experiencia y contexto.
- **Establecer el tamaño de la iteración:**
  Se define cuánto durará cada ciclo de trabajo (por ejemplo, una sprint de 2 semanas). Ayuda a organizar la entrega incremental del producto.
- **Fechas tentativas de release:**
  Se fijan fechas posibles de entrega para que los stakeholders tengan una idea general de cuándo se podrán usar las funcionalidades.
- **Identificar riesgos:**
  Se detectan elementos que pueden afectar el desarrollo (tecnologías nuevas, dependencias externas, falta de conocimiento) y se los tiene en cuenta en la planificación.

Planificación (Parte 2)

- **Variables de estimación:**
  Tiempo, esfuerzo, complejidad, riesgo e incertidumbre afectan la precisión de las estimaciones.
- **Métodos de estimación:**
  - **Estimar por analogía:** Comparar con tareas similares del pasado.
  - **Descomposición:** Dividir tareas grandes en subtareas más manejables.
  - **Puntos de historia:** Medida abstracta del esfuerzo.
- **Estimación relativa:**
  Se compara una tarea con otras para estimar su tamaño en lugar de asignar horas. Ejemplo: Planning Poker.
- **Velocidad:**
  Cantidad de trabajo (puntos de historia) que un equipo puede completar por iteración. Sirve para predecir fechas de entrega.
- **Yesterday's Weather:**
  Técnica empírica que se basa en la velocidad real del equipo en iteraciones anteriores para planificar las futuras.

- **Spike:**
  Tarea de investigación o exploración que se hace cuando hay mucha incertidumbre. No busca entregar funcionalidad, sino reducir el desconocimiento.

Gestión de Backlog

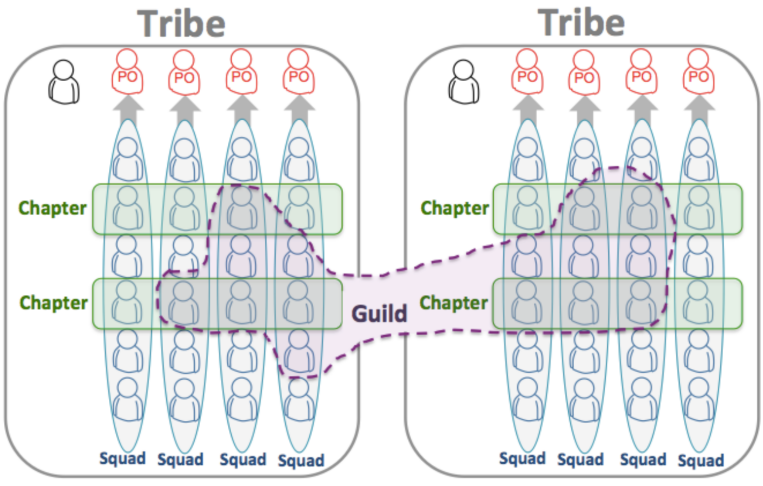
- **Artefacto Scrum:**
  El **Product Backlog** es el principal artefacto que contiene todos los ítems del producto que podrían desarrollarse.
- **Herramienta de gestión del trabajo:**
  Puede ser física (post-its, pizarras) o digital (Jira, Trello). Sirve para organizar y visualizar el trabajo pendiente.
- **Work Breakdown Structure (WBS):**
  Descomposición jerárquica del trabajo total del proyecto. Ayuda a estructurar el backlog en entregables pequeños y manejables.
- **Conjunto de ítems:**
  Cada ítem representa una funcionalidad, requerimiento o tarea técnica. Pueden incluir historias de usuario, bugs, tareas técnicas, etc.
- **Tipificación:**
  Clasificación de los ítems según su tipo (historia, bug, tarea técnica, spike, etc.), lo cual facilita su priorización y gestión.
- **Workflow:**
  Flujo de estados por los que pasan los ítems (Ej: "To Do" → "In Progress" → "Testing" → "Done"). Define el proceso de trabajo del equipo.
- **Release plans:**
  Planes de entrega de funcionalidades. Ayudan a organizar cuándo y qué se va a lanzar, alineando al equipo con los objetivos del negocio.

modelo de escalamiento de Spotify:

- **Squads (Escuadras)**: Son la unidad básica de desarrollo en Spotify, operan como mini-startups autónomas, autoorganizadas y con todas las habilidades necesarias para diseñar, desarrollar, probar y lanzar productos. Cada escuadra tiene una misión a largo plazo y decide su propia forma de trabajar, ya sea Scrum, Kanban o una mezcla. Fomentan los principios Lean Startup como MVP y aprendizaje validado. Se les anima a dedicar aproximadamente el 10% de su tiempo a "hack days" para la innovación. Cuentan con un Product Owner para la priorización del trabajo y un coach ágil para mejorar sus métodos.
- **Tribes (Tribus)**: Una tribu es una colección de escuadras que trabajan en áreas relacionadas, como el reproductor de música o la infraestructura de backend. Funcionan como "incubadoras" para las mini-startups de las escuadras, con un alto grado de libertad y autonomía. Cada tribu tiene un líder que es responsable de proporcionar el mejor "hábitat" para las escuadras. Las tribus están diseñadas para tener menos de 100 personas para evitar la burocracia. Realizan reuniones periódicas para compartir lo que están haciendo y lo que han aprendido.
- **Dependencias entre Squads**: Aunque se busca que las escuadras sean lo más autónomas posible, existen dependencias. Spotify busca minimizar las dependencias que bloquean o ralentizan el trabajo. Regularmente se encuestan a las escuadras sobre sus dependencias y se buscan formas de eliminarlas, lo que puede llevar a repriorización, reorganización, cambios arquitectónicos o

soluciones técnicas. La colaboración entre desarrollo y operaciones en Spotify es informal pero efectiva; el equipo de operaciones se enfoca en dar soporte para que las escuadras puedan liberar su propio código.

- **Chapters (Capítulos)**: Son una "pequeña familia" de personas con habilidades similares y que trabajan en la misma área de competencia general, dentro de la misma tribu. Se reúnen regularmente para discutir su área de especialización y desafíos. El líder de capítulo es el gerente de línea de sus miembros, con responsabilidades tradicionales como el desarrollo de personas y la fijación de salarios, pero también es parte de una escuadra y participa en el trabajo diario.
- **Guilds (Gremios)**: Son una "comunidad de interés" más orgánica y de mayor alcance, un grupo de personas que desean compartir conocimientos, herramientas, código y prácticas. A diferencia de los capítulos, que son locales a una tribu, los gremios suelen abarcar toda la organización (por ejemplo, el gremio de tecnología web o el gremio de testers). Cualquier persona interesada puede unirse a un gremio. Cada gremio tiene un coordinador.
- **Modelo de Matriz**: Spotify utiliza un modelo de matriz que prioriza la entrega. Las personas se agrupan en escuadras estables y co-ubicadas (dimensión vertical, el "qué" se construye), donde colaboran y se autoorganizan para entregar un gran producto. La dimensión horizontal (el "cómo" se construye bien) es para compartir conocimientos, herramientas y código, facilitado por los líderes de capítulo. El Product Owner representa al "emprendedor" (enfocado en el producto), y el líder de capítulo al "profesor" (enfocado en la excelencia técnica), creando una "tensión saludable" entre ambos roles.
- **Arquitectura y Propiedad del Sistema**: La tecnología de Spotify es altamente orientada a servicios, con más de 100 sistemas distintos que pueden ser mantenidos y desplegados por separado. Existe el rol de "System Owner" (Propietario del Sistema), quien es el punto de contacto para problemas técnicos o arquitectónicos de un sistema específico, coordinando y guiando a los desarrolladores para asegurar la integridad. El Propietario del Sistema no es un cuello de botella, sino que se enfoca en la calidad, documentación, deuda técnica, estabilidad y escalabilidad. También hay un "Chief Architect" (Arquitecto Principal) que coordina los problemas arquitectónicos de alto nivel que atraviesan múltiples sistemas, ofreciendo sugerencias y aportes a las escuadras.
- **Evolución Constante**: El modelo de Spotify está en constante evolución y es un "viaje en progreso", no un destino final. Ha sido introducido gradualmente y se ha adaptado al rápido crecimiento de la compañía



introducción a LeSS (Large-Scale Scrum):

- **LeSS (Large-Scale Scrum)**: LeSS es un marco para escalar Scrum en el desarrollo de productos grandes. Se diferencia de otros marcos de escalado al no agregar nuevas "capas", roles o artefactos a Scrum, sino que mantiene la estructura básica de Scrum, aplicando los principios de Scrum a una escala mayor. El objetivo es desescalar el desarrollo de productos mediante la simplificación de la organización, la eliminación de elementos innecesarios y la entrega de más valor con menos gente.
- **Principios de LeSS**: LeSS se basa en los principios de Scrum, como el empirismo, el control de procesos, la transparencia, la inspección y adaptación, el desarrollo basado en el valor, y el pensamiento sistémico. También enfatiza la gestión *lean* (que incluye principios como "pensamiento sistémico", "eliminar el desperdicio", "respeto por la gente" y "entrega rápida") y el pensamiento de producto, donde todo el equipo se centra en el producto completo en lugar de solo en sus propias partes.
- **Estructuras de LeSS**:
    - **LeSS (básico)**: Es para dos a ocho equipos, cada uno funcionando como un equipo de Scrum regular y multifuncional. Comparten un Product Owner (Propietario de Producto) y un Product Backlog (Pila de Producto). Todos los equipos participan en un Sprint Planning de dos partes: una planificación de alto nivel con todos los equipos y luego planificaciones específicas de equipo para las tareas. También comparten una Revisión de Sprint y una Retrospectiva de Sprint, con una retrospectiva general para la mejora de todo el proceso.
    - **LeSS Huge**: Diseñado para organizaciones más grandes, de cientos o miles de personas. En LeSS Huge, el Product Backlog se divide en áreas de requisitos (Requirement Areas), cada una con su propio Product Owner de Área (Area Product Owner) y un grupo de equipos. El Product Owner general se encarga del Product Backlog general y de las prioridades. Los ciclos de Sprint, la planificación, las revisiones y las retrospectivas son similares a LeSS básico, pero con la adición de una retrospectiva general de LeSS Huge.
- **Roles en LeSS**:
    - **Product Owner**: Es el único responsable del Product Backlog y maximiza el valor del producto.
    - **Product Owner de Área (LeSS Huge)**: Coordina las áreas de requisitos y los equipos dentro de ellas.
    - **Scrum Master**: Cada equipo tiene un Scrum Master dedicado que se enfoca en la mejora del equipo y del proceso. Puede servir a uno o dos equipos simultáneamente, o a tres en entornos avanzados.
- **Artefactos en LeSS**:
    - **Product Backlog**: Central para el producto, único y compartido por todos los equipos.
    - **Definition of Done (DoD)**: Una definición compartida de "terminado" para todo el producto, garantizando la calidad y la entrega. Se espera que esta definición sea robusta y fomente la entrega de un producto potencialmente utilizable al final de cada Sprint.
- **Prácticas y Enfoques**:
    - **Trabajo en Múltiples Equipos**: LeSS enfatiza que múltiples equipos Scrum deben trabajar juntos en un único Product Backlog para construir un único producto, evitando múltiples Product Backlogs o equipos aislados.
    - **Entrega Temprana y Frecuente**: El objetivo es liberar un incremento de producto potencialmente usable al final de cada Sprint, lo que LeSS llama "Done".
- **Cambio Organizacional**: La implementación de LeSS implica un cambio en la estructura organizacional, moviéndose de silos a equipos multifuncionales que se enfocan en el producto completo. Esto a menudo significa que los gerentes de línea deben tomar un nuevo rol de liderazgo centrado en la educación y la mejora continua, no en la dirección de la entrega.

- **Entrenamiento y Práctica**: Se recomienda un entrenamiento LeSS formal y la práctica continua para comprender y aplicar el marco eficazmente.

La metodología **12 Factor App** es un conjunto de buenas prácticas para desarrollar aplicaciones modernas, especialmente aquellas que se despliegan en la nube. Fue definida por desarrolladores de Heroku y busca que las apps sean portables, escalables y mantenibles. Aquí va un resumen de cada uno de los 12 factores:

1. Código base
   Una sola base de código bajo control de versiones, y muchas implementaciones (deploys). Mismo código, distintas configuraciones como producción, staging o testing.
2. Dependencias
   Declarar todas las dependencias de manera explícita. No depender de paquetes instalados globalmente en el sistema.
3. Configuración
   Separar la configuración del código. Usar variables de entorno para todo lo que varíe entre entornos.
4. Servicios de apoyo
   Tratar bases de datos, colas, almacenamiento y otros servicios externos como recursos que se pueden cambiar fácilmente.
5. Construcción, publicación, ejecución
   Separar claramente las etapas: build (compilar e instalar dependencias), release (combinar build y configuración), y run (ejecutar la app).
6. Procesos
   Ejecutar la app como uno o más procesos sin estado. Guardar archivos y sesiones fuera del proceso, por ejemplo en bases de datos o almacenamiento externo.
7. Asignación de puertos
   La app debe exponer un puerto y escuchar en él para atender solicitudes. No depender de servidores web integrados al entorno.
8. Concurrencia
   Escalar usando procesos múltiples (por ejemplo, web y workers). Separar responsabilidades por tipo de proceso.
9. Disposiciones efímeras
   Los procesos deben poder ser detenidos y reiniciados sin afectar el sistema. No guardar datos en el disco local del proceso.
10. Paridad entre entornos
    Mantener los entornos de desarrollo, testing y producción lo más similares posible para evitar errores por diferencias.
11. Logs
    Tratar los logs como flujos de eventos. Imprimirlos en la salida estándar y dejar que el entorno los capture o redireccione.
12. Procesos de administración
    Ejecutar tareas administrativas como comandos puntuales que se lanzan en el entorno de ejecución de la app, por ejemplo para hacer una migración.

Trunk-Based Development (Desarrollo Basado en Tronco)

- Es una estrategia de desarrollo donde todos los desarrolladores trabajan directamente sobre la rama principal (generalmente `main` o `trunk`).
- Se evitan ramas largas o de vida prolongada.
- Los cambios son pequeños, frecuentes y se integran continuamente.
- Se usan *feature toggles* (banderas de funcionalidades) para ocultar código incompleto sin necesidad de ramas.

Ventajas:

- Menos conflictos de merge.
- Menor riesgo de integración.
- Facilita CI y CD.

CI – Integración Continua (Continuous Integration)

- Práctica de integrar código nuevo frecuentemente (al menos una vez por día).
- Cada integración dispara un proceso automático que:
  - Compila el código
  - Corre tests automatizados
  - Verifica calidad del código (lint, cobertura, etc.)
- Se detectan errores temprano, antes de llegar a producción.

Beneficios:

- Detección rápida de errores
- Mejora la calidad del software
- Evita "merge hell"

CD – Entrega/Despliegue Continuo

1. **Entrega Continua (Continuous Delivery)**
   - La aplicación se puede desplegar en cualquier momento.
   - Después del CI, el sistema queda listo para ser liberado con un clic.
   - Aún requiere aprobación manual o acción del equipo.
2. **Despliegue Continuo (Continuous Deployment)**
   - Todo cambio que pasa los tests se **despliega automáticamente a producción**, sin intervención humana.
   - Requiere una pipeline muy robusta y tests confiables.

Beneficios:

- Reducción de tiempos entre desarrollo y producción
- Feedback rápido de usuarios
- Mayor confianza en cada release

Relación entre ellos

- **Trunk-Based Development** es la base para aplicar bien **CI/CD**.
- **CI** asegura que lo que está en `main` siempre es funcional.
- **CD** permite que eso se despliegue de forma segura, rápida y frecuente.

¿Por qué son importantes los SLOs?

- No se puede gestionar bien un servicio si no se entiende qué comportamientos importan y cómo se miden.
- Los **SLI, SLO y SLA** permiten modelar, medir y alinear expectativas entre usuarios, operadores y negocios.

Términos clave

SLI (Service Level Indicator)

- Métrica cuantitativa sobre el nivel de servicio: latencia, disponibilidad, errores, throughput, etc.
- Se mide con monitoreo y análisis de logs, tanto del lado servidor como cliente.

SLO (Service Level Objective)

- Meta o rango esperado para un SLI, por ejemplo: “el 99% de las búsquedas debe responder en <100ms”.
- No debe ser 100%: se define un **error budget** (presupuesto de errores) aceptable.
- SLOs bien definidos permiten saber cuándo intervenir en el sistema.

SLA (Service Level Agreement)

- Acuerdo legal/formal con consecuencias (ej: penalidades, descuentos).
- SRE no suele definirlos, pero colabora para evitar violarlos.

Buenas prácticas para definir SLOs

1. **Partir de lo que le importa al usuario**, no solo de lo que es fácil de medir.
2. Usar **percentiles** (p99, p99.9) en lugar de promedios, que pueden ocultar problemas graves.
3. No exigir perfección: definir objetivos alcanzables que permitan innovar y desplegar sin miedo.
4. Tener **pocos SLOs**, pero útiles: si no ayudan a priorizar trabajo o tomar decisiones, no sirven.
5. Publicar SLOs (internos o externos) para **establecer expectativas claras**.

Indicadores comunes por tipo de sistema

- **Servicios web**: disponibilidad, latencia, throughput

- **Sistemas de almacenamiento**: latencia, disponibilidad, durabilidad
- **Sistemas de datos (ETL)**: throughput, latencia de extremo a extremo
- **Todos los sistemas**: **correctitud** (que el resultado sea el correcto)

Ciclo de control con SLOs

1. Medir SLIs
2. Comparar con SLOs
3. Tomar decisiones si se violan
4. Actuar (ej: escalar recursos, mejorar código)

Evitar dependencias peligrosas

- Si un sistema es *demasiado confiable*, otros pueden depender de él sin plan B.
- Google forzaba fallas programadas en Chubby para evitar sobredependencia.

Recomendaciones finales

- Empezar con objetivos amplios e ir refinándolos.
- No sobrepasar SLOs constantemente: genera falsas expectativas.
- Usar **márgenes internos** más estrictos que los SLOs públicos.
- Alinear los SLOs con decisiones de producto y negocio.