

Repaso - Final - Ingeniería en Software II

Semana 01: Presentación, Configuración, Nivelación

• Pair Programming

"Pair programming" es una práctica colaborativa en la que 2 desarrolladores trabajan juntos en un lugar para diseñar, codear & resolver problemas en tiempo real. Aunque al principio puede parecer ineficiente, sus beneficios a largo plazo (Como mayor calidad del código, Compartir conocimientos & menos errores) la hacen esencial para el trabajo en equipo efectivo.

Estilos de Pair Programming

- **Driver & Navigator:** Un dev escribe (Driver) & otro mira, analiza & piensa en la estrategia (Navigator). Los roles se deben rotar con frecuencia para mantener el compromiso.
- **Ping Pong:** Guiado por TDD. Uno escribe una prueba que falla, el otro implementa la solución, seguido de un refactor. Luego se swapean los roles (en algún momento).
- **Strong-Style:** Ideal para la mentoría, el conocimiento debe "pasar por las manos de otro" asegurando que los juniors aprendan haciendo bajo guía.
- **Pair Development:** Extiende la práctica a tareas no relacionadas con código (planificación, investigación, documentación), para evitar brechas de contexto.

Beneficios

- **Calidad & Precisión:** Menos errores gracias a la revisión continua, diseños más robustos mediante crítica colaborativa.
- **Compartir Conocimiento:** Permite compartir mejor conocimiento de aspectos técnicos del proyecto.
- **Dinámica de Equipo:** Genera confianza, mejora la comunicación & reduce el tiempo de adaptación de nuevos miembros.

Gestión Práctica

- **Manejo del Tiempo (Pomodoro!)**
- **Rotación de Parejas**
- **Herramientas para hacer "pp" de forma remota**
(Vs Code Live Share ó gestión de commits x persona)

Desafíos Comunes & Soluciones

- **Fatiga** ^{→ Solución} ⇒ Descansar & Limitar tiempo para programar
- **Diferencias de Habilidad** ⇒ En pares de júnior, priorizar el diálogo & refactorizar el código para claridad & aprendizaje
- **Resistencia** ⇒ Arrancar de a poco

Mitos Desmentidos

- **Perdida de Productividad**
- **Solo para Código Complejo**

Semana 02 & 03: El Proceso de Construcción

Growing Object Orientated Software, Guided by tests por Steve Freeman & Nat Pryce

• What is the point of Test-Driven-Development?

Los proyectos de software suelen enfrentar lo desconocido entonces los equipos deben aprender durante el proceso, adaptándose a cambios inesperados

La importancia de Feedback

- **Ciclos de feedback empírico** (ej: unit-tests, daily's, releases) ayudan a validar suposiciones & corregir errores
 - **Bucles internos:** Enfocados en detalles técnicos
 - **Bucles externos:** Enfocados en necesidades del usuario & eficiencia del equipo
- **INCREMENTAL:** Construir FEATURE BY FEATURE (no por capas) → *Si feature se implementa de forma END2END "slice" por todas las partes relevantes del sistema. El sistema siempre está integrada & lista para ser deployada.*
- **Desarrollo incremental e iterativo**
 ITERATIVO: Refinar características basándose en feedback & se continua refinando hasta ser aceptado

Bases Técnicas para manejar Cambios

- **Automatización de Pruebas:** Para detectar errores de regresión
- **Código simple & Refactorización:** KISS!
- **Test-Driven-Development (TDD):**



Refactoring. Think Local, Act Local

- Aplique cambios pequeños & seguros verificando con pruebas automatizadas
- Mejores INCREMENTALES & de Pequeña Escala

Beneficios

- Clarifica criterios de aceptación
- Promueve el diseño modular (*Acoplamiento: Cohesión*)
- Genera una suite de regresión Automatizada

(Conjunto de pruebas que se ejecutan automáticamente para verificar que los cambios recientes no han roto nada)

REGLA DORADA de TDD: Nunca escribas nueva funcionalidad sin hacer fallar un test

Jerarquía de Pruebas

- **Acceptance/End2end:** Valida el sistema completo (perspectiva del usuario)
- **Integration:** Verifica interacciones con código externo (API's, etc)
- **Unit:** Evalua funcionalidades de objetos individuales

Calidad Externa vs. Interna

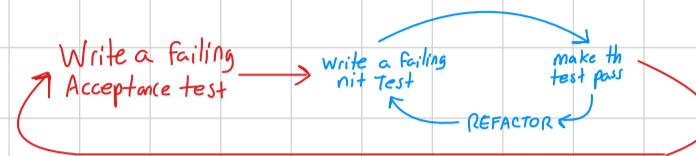
- **Externa:** Cumple necesidades del usuario (features, rendimiento)
- **Internas:** Facilita mantenimiento & cambios (código legible, bien estructurado)

End 2 End Vs. Edge 2 Edge:

End2End: Pruebas que validan todo el sistema & su entorno real, simulando el flujo completo que un user seguiría

Edge2Edge: Validan solo el sistema bajo prueba, pero sin incluir su entorno real

- No involucra infra externa {se stubean endpoints}
- Se mockean API's & BDD
- Se enfoca en la comunicación entre componentes internos



Behaviour Driven Development (BDD) + TDD

Kick-Starting the Test-Driven Cycle

> build, deploy, test en sistemas inexistentes suena raro, pero es Esencial. El riesgo de dejarlo a último momento, es demasiado alto.

First, Test a WALKING SKELETON

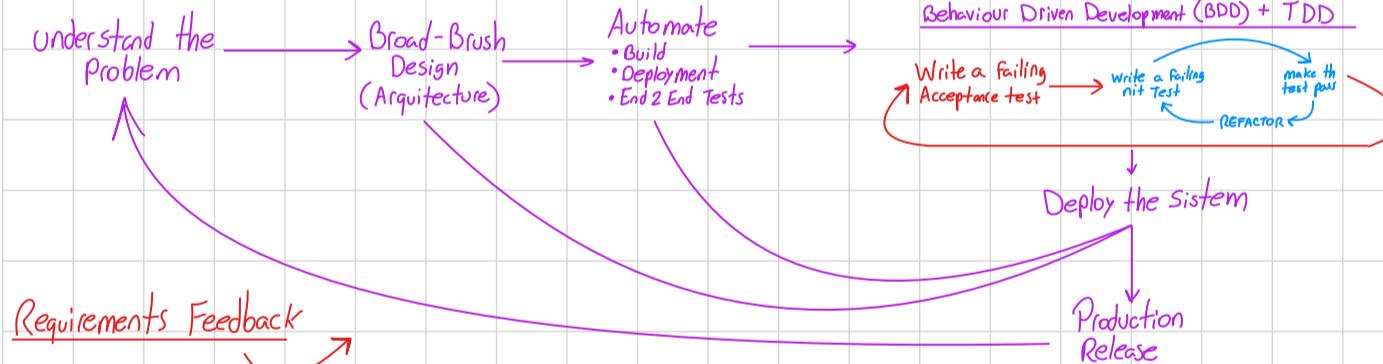
- Es una implementación de la porción más pequeña posible de funcionalidad real que podemos construir, implementar & probar con pruebas END2END
- Debe incluir automatización, los componentes principales & los mecanismos de comunicación necesarios para que podamos empezar a trabajar en el 1^{er} feature
- Mantenemos la funcionalidad del "Walking Skeleton" simple para permitir mayor foco en la infraestructura.
- **Problema:** Antes de implementar features, se necesita una **infra base** para ejecutar el 1^{er} test de aceptación
- **Riesgo:** Postergar esto puede llevar a fallos críticos en etapas avanzadas
- **Objetivo del "WS":**
 - { Validar la infra técnica (CI/CD, comunicación entre componentes)
 - { Identificar riesgos tempranos (ej: dependencias externas, procesos organizacionales, etc)

Beneficios Claves

- **Feedback Inmediato:** Detecta problemas de integración/deploy desde el inicio
- **Reducción de Incertidumbre:** Expone "unknown unknowns"
- **Base para TDD:** Permite agregar features posteriores con tests integrados

Diseño del Walking Skeleton:

- **Enfoque Minimalista:** Solo decisiones arquitectónicas esenciales (ej: Componentes principales, protocolos de comunicación, etc)
- **No es "BIG DESIGN UP FRONT" (BDUF):** Evitar "over-engineering", el diseño evoluciona con el feedback



Arquitectura Hexagonal

Es un patrón de diseño que aisla el núcleo de negocio (domain) de los detalles técnicos & externos (UI, BDD, API's, etc)

Objetivo: { Evitar que la lógica de negocio dependa de tech específica
Facilitar cambios en infra sin modificar el núcleo

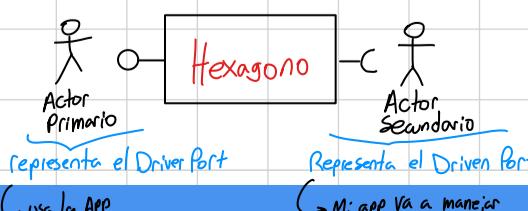
Componentes Claves:

Puertos (Ports)

Son interfaces abstractas que definen cómo el sistema interactúa con el exterior

Ejemplos: { Payment Gateway: Para procesar pagos
User Repository: Para guardar/recuperar users

*OJO!: Están escritos en TERMINOS DE DOMINIO (ej: guardar pedido)
NO TECH (ej: FROM & SELECT ALL...)



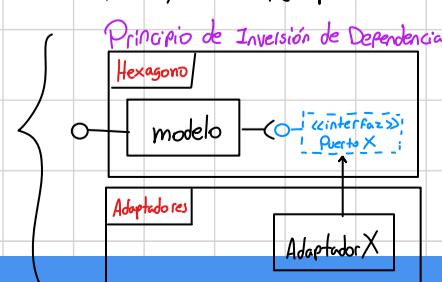
Adaptadores (Adapters)

Son implementaciones CONCRETAS de los Puertos, que traducen entre el núcleo & tech externa

Tipos:

- **Primarios (Driving Adapters):** Reciben Inputs (ej: API, REST, CLI)
- **Secundarios (Driven Adapters):** Gestionan Outputs (ej: BDD, mensajería)

Ejemplo: Un adapter [MySQLUser Repo] implementa la interfaz [UserRepo] usando consultas SQL



Beneficios

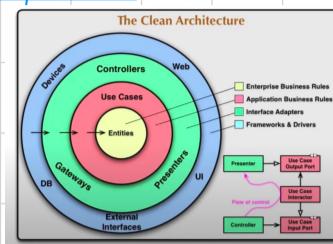
- **Desacoplamiento:** Protege la lógica de negocio de cambios de tech externas
- **Testabilidad:** Test unitarios sin dependencias reales. Los puertos permiten usar mocks o stubs en pruebas
- **Flexibilidad:** Intercambiar tech sin impactar el núcleo
- **Enfoque en el dom:** Agregar nuevas funcionalidades es más fácil

Arquitectura Domain-Driven-Design (DDD)

Breve Descripción de Clean Architecture

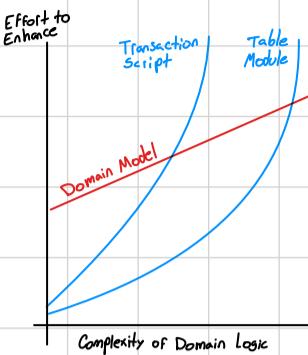
Es un enfoque de diseño de software que organiza el código en capas concéntricas donde el núcleo del negocio es independiente de frameworks, BDD ó interfaces externas.

Su objetivo principal es crear sistemas mantenibles, escalables & fáciles de probar

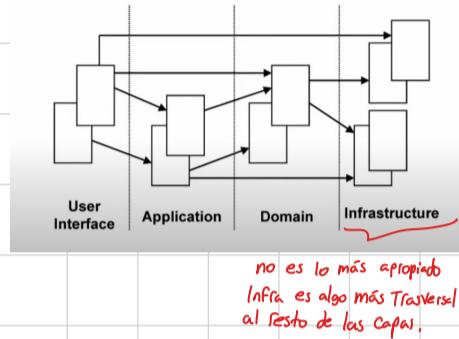


Organización de Lógica de negocio (Patrones de diseño)

Patrón	Descripción Breve	Cuando Usarlo	Ejemplo
Transaction Script	Organiza lógica en SCRIPTS Lineales que manejan una Transacción Completa.	ideal para apps simples Con foco lógica compleja ó Prototipos rápidos	Un método <code>ProcesarPedido()</code> que valida datos, aplica descuento & guarda en la DB
Table Module	Agrupa la lógica de negocio alrededor de tablas de BDD	Útil en apps centradas en operaciones CRUD ó sistemas muy acoplado a la DB	Clase <code>ProductoModule()</code> con métodos <code>guardar()</code> , <code>actualizar()</code> asociados a la tabla <code>Productos</code>
Domain Model	Modela el dominio como objetos con comportamientos y relaciones	sist. comp. con reglas de negocio dinámicas & alta mantenibilidad requerida	Clase <code>Pedido</code> con métodos como <code>calcularTotal()</code> que encapsula lógica de negocio.



DDD Layered Arquitectura



Servicios en DDD:

- Servicios de Domain:** Objetos que tienen lógica de negocio (no infra)
- Servicios de Aplicación:** Lógica de Alto nivel de mi app en particular
- Servicios de Infraestructura:** Librerías que usa nuestra app

Repositorios:

- No tienen lógica de negocio, sino lo que hacen es manejar el ciclo de vida de nuestros objetos que fueron creados
- Interface del repo es Parte del Dominio
- Implementación del repo es parte de la Infraestructura

DDD Definición

Es un enfoque de desarrollo de software que enfoca el diseño en el dominio del negocio, organizando el código alrededor de los conceptos, reglas & procesos clave del negocio. Su objetivo es crear un modelo de software que refleje fielmente el dominio real.

Relación de MVC con DDD

Modelo: Puede incluir entidades, value objects & servicios de dominio (capa de negocio)

Vista: Muestra datos pero no debe Contener lógica de negocio ← (esta definición NO ES ÚNICA)

Controller: Delega lógica compleja a servicios de dominio ó apps

Semana 05: Scrum & Extream Programming

El manifesto Ágil:



Algunos Principios Relevantes

- #9) Build projects around motivated individuals. Give them the environment & support they need, and trust to get the job done.
- #9) Continuous attention to technical excellence & good design enhances agility
- #11) The best architectures, requirements, and designs emerge from self-organizing teams.
- #12) At regular intervals, the team reflects on how to become more effective then tunes & adjusts its behavior accordingly

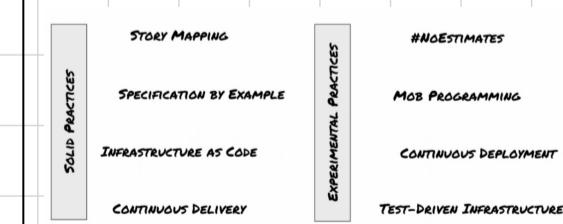
Algunas Prácticas Generales

- Proceso iterativo & time-boxed
- Retrospectivas
- Entrega Continua
- Propiedad Compartida
- Auto organización
- Diseño emergente
- Prueba automatizada
- Integración Continua
- TDD
- Pair Programming

Premisas para el uso de métodos ágiles

- Alto involucramiento del cliente/usuario
- Bajo Costo de Iteración
- Gente preparada / "Bajo Juniority"

Extream Programming Moderno (según Paez)



Extream Programming (xp)

Es una metodología ágil de desarrollo de software que prioriza la calidad, flexibilidad & colaboración, enfocándose en entregar valor al cliente de forma rápida & adaptativa. Se basa en prácticas técnicas & valores diseñados para proyectos con requisitos cambiantes ó alta incertidumbre.

Principales Características

- 1) Iteraciones Cortas Con entregas frecuentes
- 2) Feedback constante con el cliente (User-stories & acceptance tests)
- 3) Enfasis en calidad técnica mediante

TDD
Refactorización Continua
Integración Continua

Valores

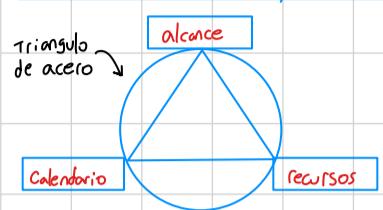
- Comunicación
- Simplicidad
- Feedback
- Coraje
- Respeto

Semana 06: Introducción a la Gestión de Proyectos; Estimación; Planificación & gestión de Backlog

Introducción a la gestión de Proyectos

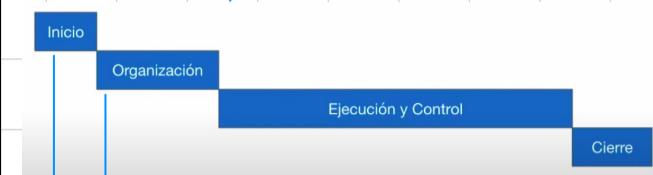
Proyecto: Emprendimiento temporal para la provisión de un servicio/producto particular bajo ciertas restricciones

VARIABLES DEL PROYECTO



- { alcance: lo determina el cliente
- recursos: lo determina el proveedor
- calendario: herramienta de AJUSTE
- Es siempre importante tener una variable AJUSTABLE
- Todo FIJO \Rightarrow RIESGO IMPORTANTE

Fases de un Proyecto:



- El armado del equipo, compra de licencias, capacitaciones, etc.
- Empezamos a dar forma al proyecto
 - definimos la visión (objetivos, riesgos, presupuesto)
 - se espera la APROBACIÓN

Roles de los intervinientes:

- Sponsor
- Usuario
- Equipo de especialistas
- Stakeholders

Forma de Contratación/Venta

-Llave en mano

Fix Price / Alcance & precios fijos / Esfuerzo Variable

-Tiempo & materiales (Precio Fijo x hora)

Times & Materials / Alcance Variable / Precio fijo x hora

Planificación

Elementos para la planificación

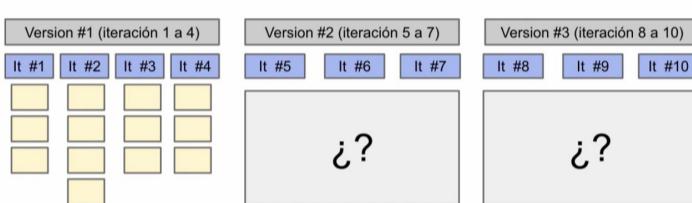
- Trabajo a trabajar (WBS, backlog ó similar)
- Disponibilidad de Recursos/personas/capacidad/velocidad
- Restricciones de tiempo/fechas

→ suficiente gente
Para alimentar 2
Pizzas grandes

Planificar Implica

- Estimar
- Establecer fechas tentativas de release
- Establecer el tamaño de iteración
- Identificar riesgos

Release Plan



Estimación

VARIABLES DE ESTIMACIÓN

- Tamaño/Complejidad
- Esfuerzo
- Tiempo
- Costo

MÉTODOS DE ESTIMACIÓN

- Paramétricos
 - COCOMO
 - COSYSMO
 - COCOMO II
- Basados en opiniones
 - Delphi
 - Wideband Delphi
 - Planning Poker

ESTIMACIÓN RELATIVA

- suele ser más simple
- Buscamos el ítem más chico & establecemos como unidad (1sp)
- Luego, ponemos el resto de los ítems en relación a la unidad

Velocidad \Rightarrow Dependiente del Equipo & el tamaño de la iteración

Yesterday's Weather: Para planificación la iteración N+1, podemos hacer tantos puntos (sp) como los que hayamos completado en la iteración N

Cuando no puedo estimar una tarea...
SPIKE

- ítem de backlog con el agregado para despejar la incertidumbre sobre otro ítem de backlog
- Debe generar como output la estimación del ítem original
- Suelen incluir la realización de una prueba y/o investigación

SLACK

- Estrategia para manejo de imprevistos
- Inicialmente planificado para tareas de baja prioridad
- Deuda técnica/refactor/investigación

Gestión de Backlog

Backlog

- Artefacto de Scrum
- Herramienta de gestión del proyecto
- Work Break Structure
- Conjunto de ítems

TIPOS DE BACKLOGS

- Features
- Tareas
- Defectos
- User Stories

RELEASE PLAN

- Definición de tamaño de iteración
- Ubica tentativamente los ítems de backlog en iteraciones
- Calendariza iteraciones
- Identifica Versiones/fechas de release

Semana 09: Escalamiento Agile & Arquitectura de Software

Escalamiento Agile

Se refiere a la adaptación & aplicación de metodologías ágiles en **proyectos grandes/complicados**, donde múltiples equipos deben

Colaborar de forma coordinada sin perder los principios ágiles originales

Por qué es necesario?

Los marcos ágiles tradicionales están diseñados para equipos chicos. Cuando un proyecto involucra muchísima gente, surgen desafíos como:

- Coordinación
- Dependencias/Burocracia
- Competencia
- Entregas
- Procesos

Escalamiento Agile brinda estructuras & prácticas para resolver (Nunca hay que perder el mindset agile del manifesto!)

Estos problemas SIN PERDER AGILIDAD!

FRAMEWORKS: Indican responsabilidades & roles para mejorar equipos de trabajo

Framework: Nexus

- 3 a 9 integrantes trabajando en el mismo proyecto
- **Nexus Integration Team** (elimina dependencias & asegura integración continua)
- L Integación/Coordinación de equipos
- Mismo backlog de Producto
- Refinamiento Continuo entre equipos

Framework: LeSS (Large Scale Scrum)

- 2 a 8 integrantes trabajando en el mismo proyecto
- **L + 8 equipos → LeSS Huge** (concepto de "áreas")
- Mismo backlog de Producto
- **MORE WITH LESS:**
G no agregar más roles, artefactos ó ceremonias
- **TEORÍA de ENCOLAMIENTO**
↳ Manejar COLAS DE TAREAS, PRIORIDADES, WIP LIMIT

Framework: SAFe (Scaled Agile Framework)

- Decenas a cientos de personas & equipos
- Integra Conceptos de **Scrum + KANBAN + LEAN + DEVOPS**
- Foco de Entrega e innovación continua.
- Empoderamiento de equipos (90/10)
- Métricas → **Valor, Velocidad, Calidad**

Arquitectura de Software ⇒ 12 FACTOR APP

12 Factor app es una metodología para construir aplicaciones web modernas, escalables & mantenibles. Establece 12 principios que ayudan a desarrollar software como servicio (SaaS) de manera limpia & portatil, especialmente en entornos de deploys en la nube.

1) CODE BASE

- Una única base de Código bajo control de versiones, muchas deploys
- Todo el Código en un solo repositorio
- Puede haber múltiples entornos (prod, staging), pero todos usan la misma base de Código

2) DEPENDENCIAS

Declarar explicitamente & aislar las dependencias

- No asumir que el sistema tiene instalada librerías ó herramientas.
- Usar archivos como requirements.txt, package.json ó Cargo.toml

3) Config

Almacena la configuración en Variables de entorno

- Todo lo que cambie en entornos (API keys, URLs, flags), se define como variables externas, no dentro del código fuerte

4) Backing Services

Trata servicios externos (BDD, cachés, etc) como recursos adjuntos

- Sin importar si es una base local ó remoto, el código accede al recurso mediante su config.

5) Build, release, run

Separar claramente los pasos de BUILD, RELEASE & RUN

- BUILD: Compilar el Código / empaquetar dependencias
- RELEASE: Combinar BUILD + CONFIG
- RUN: Ejecutar la app como un RELEASE ESPECÍFICO

6) Processes

Ejecuta la app como 1 o más procesos sin estado

- No guardar nada en memoria local, la app debe ser STATELESS
- Los estados se almacenan en BDD ó otros servicios externos

Arquitectura de Software ⇒ Extra definiciones

Contexto de software:

- Restricciones técnicas
- Requisitos de Negocio
- Factores humanos
- Objetivos de Calidad

¿Qué hace que una decisión SEA ARQUITECTÓNICA?

- Tiene Consecuencias NO LOCALES
- Afecta a los **DRIVERS** del sistema
- Guía otras actividades del desarrollo

Niveles de Diseño:

- **Arquitectura:** Se decide la estructura general del sistema
- **Interacción:** Se define como interactúan las partes del sistema entre sí
- **Interno:** Se refiere al diseño interno de un módulo.

Beneficios de buena arquitectura

- Reducir riesgo
- Permitir agregar nuevas funcionalidades más rápido
- Adaptable a cambios

¿Qué es un Driver?

Factor que influye DIRECTAMENTE en el diseño de Arq.

IMPORTANTE: No todos los requerimientos están documentados.
Hay que negociar ciertas cuestiones & estar atentos al contexto

En que consiste los Drivers:

- { 1) Propósito del sistema 4) Aspectos Arquitectónicos
- 2) Funcionalidad Principal 5) Restricciones
- 3) Atributos de calidad

CONCEPTOS de DISEÑO

- Recursos, principios & estructuras REUTILIZABLES
- bloques que el arquitecto puede usar para armar una solución robusta
- NO REINVENTAR LA RUEDA

Toma de Decisiones

- **Experiencia:** ¿Puedo aplicar un patrón? { ya hemos resuelto algo similar? }
- **Pruebas:** Verificar viabilidad con pruebas de concepto
- **Documentar:** ¿Cuál es el costo de NO DOCUMENTAR?

Consumo de Servicios Externos & Estrategia de Testing

Tests Dobles

Término general que se refiere a objetos SIMULADOS/SUSTITUIDOS usados en tests para reemplazar dependencias reales

Veamos algunas estrategias relevantes...

Tipo	¿Qué hace?	Ejemplo
Dummy	No hace nada, solo existe porque se necesita un objeto.	Un objeto vacío pasado como parámetro.
Stub	Devuelve respuestas predefinidas cuando se le llama.	Un stub que devuelve siempre "200 OK".
Mock	Verifica si se llamaron ciertos métodos con ciertos datos (verificación de comportamiento).	Un mock que verifica que se llamó send_email("hola").
Spy	Igual que el mock, pero graba lo que pasó sin que se lo programes antes.	Un spy que registra llamadas para checarlas luego.
Fake	Implementación más realista, pero simplificada.	Una base de datos en memoria, en vez de una real.

Stubbing

Técnica donde un objeto reemplaza el comportamiento de otro con una respuesta controlada para aislar la unidad bajo prueba (util para unit tests que requieren información de una API externa, por ejemplo)

Mocking

El mocking no solo reemplaza, sino que verifica interacciones: Cuantas veces fue llamado un método, args, etc

Ejemplo (usando unittest.mock en Python):

```
python
mock_service = Mock()
mock_service.send_email("hello@example.com")

mock_service.send_email.assert_called_with("hello@example.com")
```

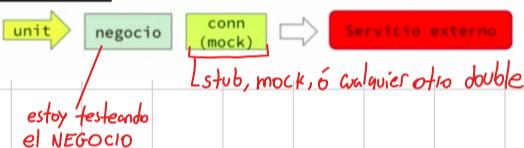
Test Desiderata (Desiderata = Deseables)

Características que debería tener un BUEN TEST, como:

Criterio	Significado
Rápido	Deben ejecutarse en milisegundos o segundos.
Aislado	No debe depender de internet, bases de datos reales, etc.
Determinista	Siempre debe dar el mismo resultado con los mismos inputs.
Legible	Fácil de entender y mantener.
Expresivo	Que el fallo indique claramente qué parte del sistema está mal.
Automatizable	Debe poder ejecutarse automáticamente en CI/CD.

Estrategia de Testing:

Pruebas Unitaria

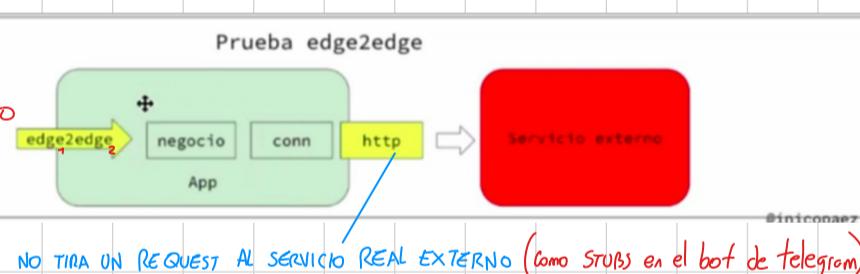
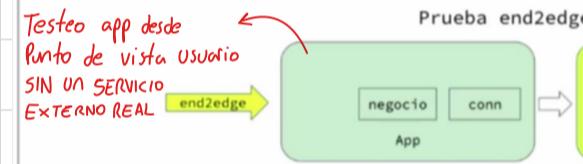


Prueba de Integración



me comporto como si fuera el usuario Utilizando el servicio externo de Verdad
(se realiza en un ambiente previo a Prod)

Esto lo hacíamos en los ambientes de TEST, previo a PROD & testeábamos a mano el BOT con el API (también en TEST)



REST APIs

API:

Significa Application Programming Interface.

Es un Conjunto de reglas que permite que 2 programas se comuniquen entre sí.

RESTful API

- Es una API que sigue los principios de REST (REpresentational State Transfer)
- Rest se construye en torno a un Concepto Central: RECURSOS
- Se basa en 6 restricciones de arquitectura que se deben cumplir para que una API sea RESTFUL

REST: Uniform interface - Tácticas

- Identification of resources: los recursos deben tener un identificador.
- Access methods have the same semantics. Ej: GET, POST, etc. (en HTTP)
- Manipulation of resources through representations: los recursos se manipulan utilizando su representación.
- Self-descriptive messages: un mensaje contiene toda la información necesaria para comprenderlo.
- HATEOAS (hypermedia as the engine of application state): las respuestas de los recursos deberían incluir links para obtener información relacionada.

RECURSOS REST:

Abstracción de info a la que le podemos poner un nombre. Tiene una representación, que muestran el ESTADO del recurso. Se puede acceder a esos recursos & modificarlo mediante métodos

RESTRICIONES de Arquitectura

Client-Server:

Mantener la interfaz de usuario (client) separada de los datos (server). Los usuarios solo deben conocer las URL de los recursos

Stateless:

Todas las requests de los clientes deben contener toda la info para que sea comprendida sin tomar ventaja de requests anteriores

Cacheable

Los clientes pueden CACHEAR respuestas las cuales deben identificarse como Cacheables

Layered System:

Distribuir el sistema en capas, de manera que una capa no pueda ver más allá de la capa inmediatamente debajo

Code on Demand:

Opcional. Permite al server enviar CÓDIGO EJECUTABLES a los clientes (ej: script tag ejecuta js)

Uniform Interface

Mantener una interfaz uniforme a lo largo de todo el sistema para el acceso a recursos.

REST APIs: Recursos - Métodos

- Los métodos son los que nos permiten acceder a recursos & actualizarlos para modificar su estado

Hay 2 Partes:

- URL: ruta en la que se accede al recurso. Se deben usar sustitutivos para describir rutas. Ej: /clientes/{id}/cuentas

- MÉTODO HTTP: REST no define una relación entre métodos HTTP & los que se deberían usar en REST (pero se usan convenciones)

REST APIs: Routes

Nombre	Ruta (path)	Verbo HTTP	Propósito
Index	/dogs	GET	Listar todos los perros
New	/dogs/new	GET	Mostrar formulario (html) para crear perro
Create	/dogs	POST	Crear un perro
Show	/dogs/:id	GET	Mostrar información de un perro en particular
Edit	/dogs/:id/edit	GET	Mostrar formulario (html) para crear perro
Update	/dogs/:id	PUT	Actualizar un perro en particular
Destroy	/dogs/:id	DELETE	Eliminar un perro en particular

Semana 15: OperacionesService Level Objectives¿Qué son los SLOs?

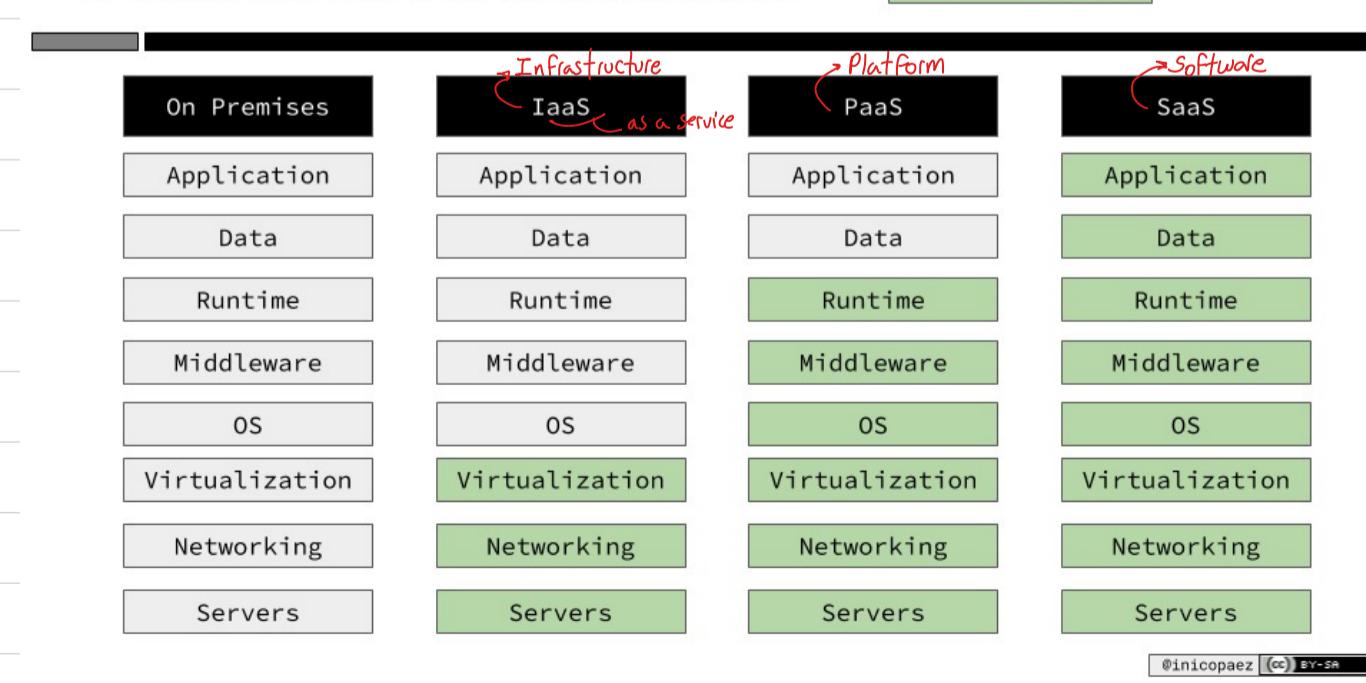
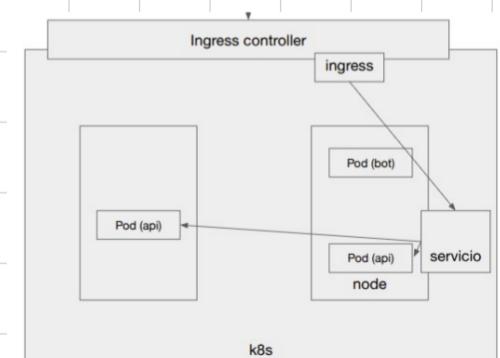
- Son metas cuantificables que definen el nivel de confiabilidad que un servicio debe ofrecer, basada en métricas específicas como DISPONIBILIDAD, LATENCIA ó TASA de ERRORES
- Son distintos de los SLAs (Service level agreements), que son acuerdos contractuales con los users, mientras que los SLOs son objetivos internos para guiar decisiones operativas
- Son una herramienta clave para equilibrar innovación & estabilidad
- Una buena implementación evita problemas de Confiability & mejora la experiencia del user

Definición de Buenas Métricas para SLOs

- Deben basarse en lo que realmente importa para el usuario (ej: tiempo de respuesta)
- Ejemplo: Un servicio web podría definir un SLO como "99.9% de las solicitudes deben responder en <500 ms"

Implementación & Usos de SLOs

- SLI (Service Level Indicators): Métricas que miden el cumplimiento del SLO
- Presupuesto de Error: Define cuánto margen de fallo se permite antes de tomar acciones correctivas
- Si un servicio supera un Presupuesto de Error, se debe priorizar la mejora de Confiability sobre nuevas características

Cosas extras para saber...Modelos de InfraestructuraNuestro modelo de Turneo de Consultas médicas

1. Kubernetes en IaaS

- Si despliegas Kubernetes en máquinas virtuales (VMs) de un proveedor cloud (ej: GKE en Google Cloud, EKS en AWS, AKS en Azure), estás usando IaaS porque:
 - Tú gestionas el clúster (o usas un servicio gestionado parcialmente).
 - Controlas la infraestructura subyacente (nodos, redes, almacenamiento).

2. Kubernetes como PaaS (enfoque moderno)

- Si usas servicios serverless o PaaS que abstractan Kubernetes (ej: Google Cloud Run, AWS App Runner, o plataformas como Heroku con soporte para contenedores), te acercas al modelo PaaS, porque:
 - Solo te preocupas por el código y la configuración de los contenedores.
 - El proveedor maneja el escalado, balanceo de carga y la infraestructura.

¿Dónde cae tu caso?

- Si administras tú mismo el clúster de k8s (incluso con herramientas como Helm o Terraform) → IaaS.
- Si usas un servicio totalmente gestionado (ej: despliegues en Vercel o Fly.io con integración k8s transparente) → PaaS.

User Stories & Example Mapping

This Document has been modified with Flexcil app (Android) <https://www.flexcil.com>

User Stories

Es una descripción simple & concisa de una funcionalidad/requisito de software, escrita desde la perspectiva del usuario final.

Su objetivo es definir qué necesita el usuario & por qué, SIN ENTRAR A DETALLES TÉCNICOS.

Estructura Básica

es: COMO, en: AS,
QUIERO, : I WANT,
PARA SO

Ejemplos del TP:

Definition of Ready:

Necesitamos:

- Criterios de aceptación.
- Casos de error, mensajes de error.
- Si requiere modificación de UI: conocer dónde tiene que estar ubicado, en qué páginas, prototipo/layout.

Definition of Done:

Se cumplieron los siguientes puntos:

- Implementación completa con tests de los Gherkins.
- Validación del PO en el ambiente de test para ponerlo en producción.
- Corrección de posibles modificaciones requeridas por el PO sobre el trabajo hecho.

US-3.0: Como paciente, quiero registrarme en el sistema usando mi handle de Telegram para gestionar mis turnos (sin validaciones)

(closed) Issue created 1 month ago by Gabriel Diem

Un paciente tiene:

- Username: string
- DNI: string
- Email: string

Criterios de aceptación:

- Username: requerido, provisto por Telegram
- DNI: requerido, no puede ya estar registrado, podría ser de otro tipo (libre civila, pasaporte), único
- Email: requerido, no necesita ser único, formato de mail (algo@dominio)

Assigees
Santiago

Labels
None

Parent
None

Weight
3

Características Claves:

- Enfocado en el user: Describe el QUÉ & el PORQUÉ, no el cómo
- Corta & Específica: Idealmente, entra en un Post-IT
- Parte de un Backlog: Se usa en metodologías ágiles (ej: scrum)
- Acceptación: Incluye CRITERIOS de ACEPTACIÓN

CRITERIOS de ACEPTACIÓN:

Condiciones específicas & Verificables que definen cuando una User Story Cumple con lo que el usuario espera.

Ejemplo (para una US de login)

- El user puede ingresar con mail & Contraseña
- Si las credenciales son incorrectas, se muestra un msj de error
- Al ingresar exitosamente, se redirige al dashboard

Definition of Done:

Una lista de requisitos que una User Story debe cumplir para considerarse Completada & entregable al Usuario. Se debe garantizar la Calidad & Consistencia en lo que se entrega.

Definition of READY:

Una lista de requisitos que una USER STORY debe cumplir antes de ser incluida en un sprint

Hay que asegurar que la historia está clara, bien definida & viable antes de entrar en desarrollo

Verificación Vs. Validación

Verificación

Confirmar que el producto se construyó correctamente

Enfoque: ¿Cumple con los requisitos formales?

Validación

Confirmar que el producto satisface las necesidades reales del user

Enfoque: ¿Resuelve el problema correctamente?

Verificación	Validación
"¿Lo hicimos bien?"	"¿Hicimos lo correcto?"
Pruebas técnicas (unitarias, integración).	Pruebas con usuarios finales (tests UX, feedback).

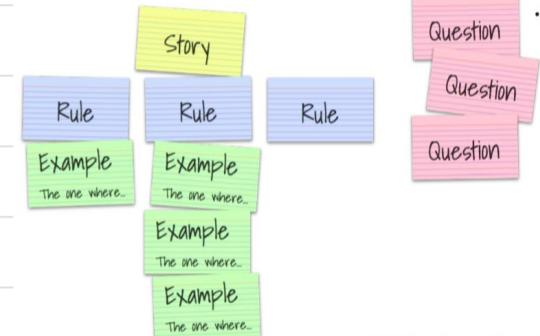
Cowboy Coding / Desarrollo Ad-hoc

Es un enfoque desestructurado & caótico para desarrollar software donde:

- No hay procesos definidos (no plans, documentation or tests)
- El código se escribe sobre la marcha, basándose en decisiones improvisadas

Example Mapping

- Taller para clarificar & confirmar los criterios de aceptación
- Ayuda a descubrir reglas, ejemplos & preguntas para las User Stories



Question
Question
Question

Exitos!

~migas !!

