

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Repaso para el examen parcial

Del pedido de un cliente a una arquitectura documentada

Gerson Garrido

Semana 8

Hoy

Sesión A: repaso

1. Cómo es el examen
2. Ingeniería de software y procesos
3. Requisitos
4. Especificaciones
5. Diseño
6. APIs y patrones
7. Arquitectura
8. El proyecto: P2 y P3

Sesión B: examen parcial.

Lecturas

Las *Preguntas de repaso* de las guías de las semanas 1 a 7 y `exams/README.md`.

Complementarias: las lecturas principales que no hayan alcanzado a leer. Pesan más SWEG, caps. 1 y 10; la Guía de Scrum; *Design Docs at Google*; el ADR de Nygard.

SECCIÓN 01

Cómo es el examen

01

Qué entra: semanas 1 a 7, sin herramientas de IA

Procesos	Hyrum, trade-offs, Scrum, Kanban; qué se puede concluir de un estudio con IA
Requisitos	Historias con criterios verificables; lo ambiguo y lo contradictorio
Especificaciones	Pre, post, invariantes; tipos que impiden un estado inválido
Diseño	En un fragmento: módulos superficiales, filtración, acoplamiento; una interfaz mejor
APIs	Rutas, códigos, errores; idempotencia
Arquitectura	Elegir entre alternativas; ADR; atributo de calidad medible

Error común

Memorizar definiciones. El examen da un caso que no han visto: hay que reconocer un módulo superficial en código ajeno y decir cómo cambiar su interfaz.

Cómo estudiar

1. **Preguntas de repaso de las semanas 1 a 7**, por escrito y sin mirar las lecturas.

Cómo estudiar

1. **Preguntas de repaso de las semanas 1 a 7**, por escrito y sin mirar las lecturas.
2. **Su propio P2.** Expliquen a otra persona cada decisión y por qué se descartaron las alternativas.

Cómo estudiar

1. **Preguntas de repaso de las semanas 1 a 7**, por escrito y sin mirar las lecturas.
2. **Su propio P2.** Expliquen a otra persona cada decisión y por qué se descartaron las alternativas.
3. **Rehacer en papel los laboratorios:** historias, refactorización, OpenAPI, ADR.

Cómo estudiar

1. **Preguntas de repaso de las semanas 1 a 7**, por escrito y sin mirar las lecturas.
2. **Su propio P2.** Expliquen a otra persona cada decisión y por qué se descartaron las alternativas.
3. **Rehacer en papel los laboratorios:** historias, refactorización, OpenAPI, ADR.
4. **Parcial de práctica,** con el reloj corriendo.

Cómo estudiar

1. **Preguntas de repaso de las semanas 1 a 7**, por escrito y sin mirar las lecturas.
2. **Su propio P2.** Expliquen a otra persona cada decisión y por qué se descartaron las alternativas.
3. **Rehacer en papel los laboratorios:** historias, refactorización, OpenAPI, ADR.
4. **Parcial de práctica,** con el reloj corriendo.

> **Lo que hace buena una respuesta**
>
> Usen los datos del caso, digan qué se pierde y sean concretos: un criterio lleva un monto y una hora; un problema de diseño, una línea de código y el cambio que duele.

Un caso nuevo: la lavandería de Hilda

Lo que dijo Hilda

“El cliente deja su ropa, la peso y le doy un ticket de papel. Cobro S/ 7 por kilo. El servicio normal sale en dos días; el exprés sale el mismo día y cuesta el doble. Se cobra al entregar, siempre. A los clientes frecuentes les hago un descuento y ellos me pagan por adelantado. La gente llama todo el día a preguntar si su ropa ya está, y a veces se me pierden los tickets. Quiero que cuando la ropa esté lista les llegue un aviso rápido por WhatsApp. Un pedido se puede anular, pero no cuando ya se está lavando.”

Lo seguimos de punta a punta: historias, especificación, módulos, API y ADR.

SECCIÓN 02

Ingeniería de software y procesos

02

Tiempo, Hyrum y trade-offs

- **Ingeniería de software:** programación integrada en el tiempo. ¿Cuánto va a vivir este código?
- **Ley de Hyrum:** con suficientes usuarios, todo lo observable termina siendo usado, prometa lo que prometa el contrato.
- **Shifting left:** encontrar el error antes cuesta menos.
- **Trade-off:** toda decisión tiene costos, se toma con datos y se revisa.

Tipo examen

La API siempre devolvió el historial del más reciente al más antiguo, sin prometerlo, y el orden cambia.

Tiempo, Hyrum y trade-offs

- **Ingeniería de software:** programación integrada en el tiempo. ¿Cuánto va a vivir este código?
- **Ley de Hyrum:** con suficientes usuarios, todo lo observable termina siendo usado, prometa lo que prometa el contrato.
- **Shifting left:** encontrar el error antes cuesta menos.
- **Trade-off:** toda decisión tiene costos, se toma con datos y se revisa.

Tipo examen

La API siempre devolvió el historial del más reciente al más antiguo, sin prometerlo, y el orden cambia. Hyrum: una pantalla que usa `orders[0]` se rompe, aunque nadie lo prometió. Se promete y se prueba, o se avisa antes de cambiarlo.

Scrum, según la guía de 2020

Tres responsabilidades	Product owner, Scrum master, developers
Cinco eventos	El sprint (un mes o menos), que contiene a la sprint planning, la daily, la sprint review y la retrospectiva
Tres artefactos	Product backlog → product goal Sprint backlog → sprint goal Incremento → Definition of Done

Tipo examen

Una historia con código y sin la revisión que pide la Definition of Done, al terminar el sprint.

Scrum, según la guía de 2020

Tres responsabilidades	Product owner, Scrum master, developers
Cinco eventos	El sprint (un mes o menos), que contiene a la sprint planning, la daily, la sprint review y la retrospectiva
Tres artefactos	Product backlog → product goal Sprint backlog → sprint goal Incremento → Definition of Done

Tipo examen

Una historia con código y sin la revisión que pide la Definition of Done, al terminar el sprint. No se libera ni se presenta en la sprint review: vuelve al product backlog, y sus puntos no cuentan en la velocity.

Kanban, Little y velocity

Ley de Little

En promedio, en un sistema estable:

$$\text{cycle time} = \text{WIP} / \text{throughput}$$

10 tareas en curso, 2 terminadas por día:

Un ítem detenido se ve con la **work item age**.

Estimar

- Estimación $\neq$ compromiso $\neq$ objetivo.
- Story points: tamaño relativo, no horas.
- Velocity: para planificar al mismo equipo, no para comparar.

Kanban, Little y velocity

Ley de Little

En promedio, en un sistema estable:

$$\text{cycle time} = \text{WIP} / \text{throughput}$$

10 tareas en curso, 2 terminadas por día: 5 días. Con WIP 6: 3 días, si el throughput no cambia.

Un ítem detenido se ve con la **work item age**.

Estimar

- Estimación $\neq$ compromiso $\neq$ objetivo.
- Story points: tamaño relativo, no horas.
- Velocity: para planificar al mismo equipo, no para comparar.

Kanban, Little y velocity

Ley de Little

En promedio, en un sistema estable:

$$\text{cycle time} = \text{WIP} / \text{throughput}$$

10 tareas en curso, 2 terminadas por día: 5 días. Con WIP 6: 3 días, si el throughput no cambia.

Un ítem detenido se ve con la **work item age**.

Estimar

- Estimación $\neq$ compromiso $\neq$ objetivo.
- Story points: tamaño relativo, no horas.
- Velocity: para planificar al mismo equipo, no para comparar.
- **Goodhart:** si se vuelve meta, los puntos se inflan.

Kanban, Little y velocity

Ley de Little

En promedio, en un sistema estable:

cycle time = WIP / throughput

10 tareas en curso, 2 terminadas por día: 5 días. Con WIP 6: 3 días, si el throughput no cambia.

Un ítem detenido se ve con la **work item age**.

Estimar

- Estimación $\neq$ compromiso $\neq$ objetivo.
- Story points: tamaño relativo, no horas.
- Velocity: para planificar al mismo equipo, no para comparar.
- **Goodhart:** si se vuelve meta, los puntos se inflan.

737 MAX: no fue "un bug"

El MCAS hizo lo especificado, con el dato de un solo sensor. Fallaron requisitos, análisis de riesgo, revisión independiente y honestidad.

Leer un estudio: qué se midió, a quiénes, cuándo

METR 2025: experimento aleatorio

16 desarrolladores con experiencia, 246 issues de sus propios repositorios. Con IA tardaron 19 % más; antes esperaban ir 24 % más rápido y después creían haber ido 20 % más rápido. METR advierte que está desactualizado.

DORA 2025: encuesta

Lo que reportan los profesionales y las asociaciones entre sus respuestas: no permite concluir causa. La IA como amplificador de lo que la organización ya tiene.

La lección vigente

La sensación de productividad no es una medición. Un “19 % más lentos” no autoriza a prohibir ni un “dice que va más rápido” a adoptar: hay que decir hasta dónde llega el estudio.

SECCIÓN 03

Requisitos

03

De lo que dijo Hilda: lo contradictorio y lo ambiguo

Lo que dijo	Problema	Qué preguntarían
“Al entregar, siempre” y “pagan por adelantado”	Se contradicen	¿Cuándo le cobró al último cliente frecuente?
“S/ 7 por kilo”	¿Y 3,2 kg?	¿Cuánto cobró al último que trajo un poco más de 3 kilos?
“Frecuentes”, “un descuento”	Quién, cuánto	¿Cómo sabe hoy quién es frecuente?
“Sale el mismo día”	¿Si llega de noche?	¿Hasta qué hora recibe un exprés?
“Un aviso rápido”	Sin número	¿Cuándo sería demasiado tarde?

Problema frente a solución

El WhatsApp es una solución; las llamadas y los tickets perdidos, el problema. Preguntas abiertas y sobre el pasado: “¿cómo fue la última vez…?”.

Una historia, dos criterios

HU-02

Como cliente, quiero recibir un aviso cuando mi ropa esté lista, para no tener que llamar a preguntar.

1. **Dado** que el pedido 12 de Ana está *lavando*, **cuando** la encargada lo marca como listo, **entonces** a Ana le llega por WhatsApp un mensaje con el número del pedido y el monto.
2. **Dado** que el pedido 12 está *recibido*, **cuando** la encargada intenta marcarlo como listo, **entonces** se rechaza, sigue *recibido* y no sale ningún aviso.

Lo que debilita una respuesta

"Entonces funciona bien" (no se comprueba). "Se agrega una fila a `orders`" (implementación). "Como sistema, quiero..." (falla la V). "Administrar mi lavandería" (épica: se divide por reglas o caminos).

SECCIÓN 04

Especificaciones

04

Contrato y tipos: lo que hay que tener preciso

- **Precondición:** lo que garantiza quien llama. **Postcondición:** lo que garantiza la función. **Invariante:** lo que vale siempre entre operaciones.

Contrato y tipos: lo que hay que tener preciso

- **Precondición:** lo que garantiza quien llama. **Postcondición:** lo que garantiza la función. **Invariante:** lo que vale siempre entre operaciones.
- **Parcial** (exige y no revisa) o **total** (responde algo definido para toda entrada): la especificación dice cuál.

Contrato y tipos: lo que hay que tener preciso

- **Precondición:** lo que garantiza quien llama. **Postcondición:** lo que garantiza la función. **Invariante:** lo que vale siempre entre operaciones.
- **Parcial** (exige y no revisa) o **total** (responde algo definido para toda entrada): la especificación dice cuál.
- **Postcondición completa:** “devuelve horas libres” la cumple `[]`.

Contrato y tipos: lo que hay que tener preciso

- **Precondición:** lo que garantiza quien llama. **Postcondición:** lo que garantiza la función. **Invariante:** lo que vale siempre entre operaciones.
- **Parcial** (exige y no revisa) o **total** (responde algo definido para toda entrada): la especificación dice cuál.
- **Postcondición completa:** “devuelve horas libres” la cumple `[]`.
- **Parse, don’t validate:** en la frontera, de `unknown` a un tipo que lleva la garantía. `x as Order` no revisa nada.

Contrato y tipos: lo que hay que tener preciso

- **Precondición:** lo que garantiza quien llama. **Postcondición:** lo que garantiza la función. **Invariante:** lo que vale siempre entre operaciones.
- **Parcial** (exige y no revisa) o **total** (responde algo definido para toda entrada): la especificación dice cuál.
- **Postcondición completa:** "devuelve horas libres" la cumple `[]`.
- **Parse, don't validate:** en la frontera, de `unknown` a un tipo que lleva la garantía. `x as Order` no revisa nada.
- **Design doc:** antes de construir, con no objetivos y alternativas.

Tipos que impiden el estado inválido

code/week-08/order.ts

```ts
export type ReadyOrder = OrderBase & { status: 'ready'; readyAt: string };
export type DeliveredOrder = OrderBase & { status: 'delivered'; readyAt: string; deliveredAt:
    string };

export function markDelivered(order: ReadyOrder, now: string): DeliveredOrder {
  return { ...order, status: 'delivered', deliveredAt: now };
}
```

make errores

```
bad/deliver-washing.ts(6,24): error TS2345: Argument of type 'WashingOrder' is not assignable to parameter of type 'ReadyOrder'.
...
bad/missing-case.ts(17,26): error TS2345: Argument of type 'CancelledOrder' is not assignable to parameter of type 'never'.
```

Un `switch` sin un caso no compila. Los tipos no pueden decir que el monto pagado es igual al precio: eso se revisa al ejecutar.

La especificación de `deliver(order, paidCents, now)`

Total sobre el estado y el monto

Precondición: `now` es un instante en ISO 8601 (a cargo de quien llama).

Postcondición. Si `order` está *listo* y `paidCents` es su precio: `ok: true` y un pedido *entregado* con `deliveredAt = now` y los demás datos iguales. Si no: `ok: false` con `NOT_READY`, `ALREADY_DELIVERED`, `CANCELLED` o `WRONG_AMOUNT`. `order` no se modifica.

Invariantes. `priceCents` es un entero positivo. El estado solo avanza *recibido* → *lavando* → *listo* → *entregado*, o *recibido* → *anulado*.

Error común

Olvidar lo que **no** cambia; escribir “funciona correctamente”; copiar la implementación: “hace un `switch`” dice cómo, no qué.

SECCIÓN 05

Diseño

05

El vocabulario, para señalar con el dedo

- **Complejidad:** amplificación de cambios, carga cognitiva, incógnitas desconocidas.
- **Causas:** dependencias y oscuridad.
- **Profundo:** da mucho con una interfaz pequeña. **Superficial:** la interfaz es casi lo que hace.
- **Parnas:** cada módulo esconde una decisión que puede cambiar.
- **Filtración:** la misma decisión en varios módulos.
- **Acoplamiento,** de más a menos dañino: detalles internos, estado global, banderas, estructuras enteras, solo lo necesario.

Un buen comentario de diseño

Un lugar en el código y un cambio concreto: “si se agrega el estado *anulado*, hay que revisar `readyForPickup` y `revenue`, y nada lo avisa”.

Tipo examen: señalen cuatro problemas

code/week-08/shallow.ts

```ts
// est: 1 recibido, 2 lavando, 3 listo, 4 entregado.
export type OrderRow = { id: number; cust_phone: string; kg: number; est: number; amount: number
      };

export class OrderManager {
  #table: OrderTable;
  // ...
  all(): OrderRow[] {
    return this.#table.all();
  }
  setEst(id: number, est: number): void {
    this.#table.setEst(id, est);
  }
}
export function readyForPickup(manager: OrderManager, phone: string): OrderRow[] {
  return manager.all().filter((r) => r.cust_phone === phone && r.est === 3);
}
```

Los problemas, con su nombre y su evidencia

1. **Módulo superficial.** `OrderManager` solo reenvía a `OrderTable`: quitándolo, nadie pierde nada.

Los problemas, con su nombre y su evidencia

1. **Módulo superficial.** `OrderManager` solo reenvía a `OrderTable`: quitándolo, nadie pierde nada.
2. **Información que se filtra.** Los códigos de `est` y las columnas los conocen el mostrador y la caja.

Los problemas, con su nombre y su evidencia

1. **Módulo superficial.** `OrderManager` solo reenvía a `OrderTable`: quitándolo, nadie pierde nada.
2. **Información que se filtra.** Los códigos de `est` y las columnas los conocen el mostrador y la caja.
3. **Acoplamiento a detalles internos.** `all()` entrega la lista privada: `m.all().length = 0` la vacía.

Los problemas, con su nombre y su evidencia

1. **Módulo superficial.** `OrderManager` solo reenvía a `OrderTable`: quitándolo, nadie pierde nada.
2. **Información que se filtra.** Los códigos de `est` y las columnas los conocen el mostrador y la caja.
3. **Acoplamiento a detalles internos.** `all()` entrega la lista privada: `m.all().length = 0` la vacía.
4. **Reglas sin owner.** `setEst(1, 99)` compila y funciona; de *entregado* se puede volver a *recibido*.

Los problemas, con su nombre y su evidencia

1. **Módulo superficial.** `OrderManager` solo reenvía a `OrderTable`: quitándolo, nadie pierde nada.
2. **Información que se filtra.** Los códigos de `est` y las columnas los conocen el mostrador y la caja.
3. **Acoplamiento a detalles internos.** `all()` entrega la lista privada: `m.all().length = 0` la vacía.
4. **Reglas sin owner.** `setEst(1, 99)` compila y funciona; de *entregado* se puede volver a *recibido*.

Error común

"Falta separar en más clases." Separar `OrderTable` y `OrderManager` es justo lo que produjo el módulo superficial.

Una interfaz mejor, pensada desde quien la usa

firmas de OrderDesk, code/week-08/desk.ts

```ts
receive(phone: string, weightGrams: number, service: Service): Order;
startWashing(id: number): StepResult;
markReady(id: number): StepResult;
deliver(id: number, paidCents: number): DeliverResult | { ok: false; code: 'NOT_FOUND' };
readyFor(phone: string): ReadyOrder[];
revenueCents(): number;
```

- **Esconde:** cómo se guardan los pedidos, qué transiciones existen, cómo se calcula el precio, qué códigos de estado hay.
- **Tiene más métodos y es más profunda:** no se cuentan métodos, se cuenta lo que hay que saber. `setEst(id, 3)` obliga a saber qué es el 3; `markReady(id)`, no.
- **Con IA:** un agente agregaría un `OrderManagerService` que reenvía, y sus pruebas pasarían. Revisen el diseño, no solo el verde.

SECCIÓN 06

APIs y patrones

06



Seguro e idempotente, según RFC 9110

> **Idempotente**
> El efecto sobre el servidor de N solicitudes iguales es el mismo que el de una.

Seguros (no piden cambios)	`GET`, `HEAD`, `OPTIONS` (y `TRACE`)
Idempotentes	Los seguros, `PUT` y `DELETE`
No definidos como idempotentes	`POST`, `PATCH`

Seguro e idempotente, según RFC 9110

> **Idempotente**
> El efecto sobre el servidor de N solicitudes iguales es el mismo que el de una.

Seguros (no piden cambios)	`GET`, `HEAD`, `OPTIONS` (y `TRACE`)
Idempotentes	Los seguros, `PUT` y `DELETE`
No definidos como idempotentes	`POST`, `PATCH`

- La respuesta puede cambiar: un segundo `DELETE` puede dar `404`.

Seguro e idempotente, según RFC 9110

> **Idempotente**
> El efecto sobre el servidor de N solicitudes iguales es el mismo que el de una.

Seguros (no piden cambios)	`GET`, `HEAD`, `OPTIONS` (y `TRACE`)
Idempotentes	Los seguros, `PUT` y `DELETE`
No definidos como idempotentes	`POST`, `PATCH`

- La respuesta puede cambiar: un segundo `DELETE` puede dar `404`.
- El método y la operación son dos cosas: un `POST` concreto puede comportarse así, si el contrato lo promete. Nadie intermedio lo puede suponer.

La API de los pedidos

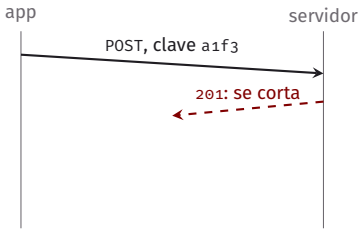
Regla escrita: `400` es forma; `422`, una regla sobre un dato bien formado; `409`, un choque con el estado actual.

Ruta	Éxito	Errores	
`POST /api/orders`	`201` y `Location`	`400 VALIDATION_ERROR`, `422 INVALID_WEIGHT`, `IDEMPOTENCY_KEY_REUSED`	`422`
`POST` `/api/orders/:id/cancel`	`200`	`404 ORDER_NOT_FOUND`, `ALREADY_CANCELLED`, `ORDER_IN_PROCESS`	`409` `409`

Compatible o incompatible

Compatible: un endpoint, un campo opcional en la solicitud, un campo en la respuesta. **Incompatible:** quitar o renombrar un campo, cambiar su tipo o significado, volver obligatorio algo opcional, cambiar un código de estado.

El reintento y la clave de idempotencia

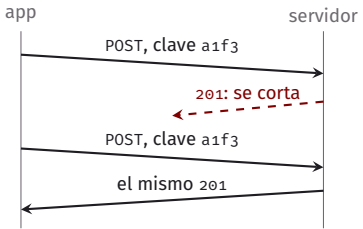


- El pedido y la clave se guardan en la misma transacción.
- Otro cuerpo con la misma clave: `422` (borrador de la IETF, todavía no es un RFC).
- “Ver si ya hay uno igual” no basta: Ana puede traer dos pedidos iguales en un día.

```
node --test api.test.ts: las tres primeras pruebas

✓ sin clave, repetir el POST crea dos pedidos (18.457084ms)
✓ con clave, el reintento devuelve la misma respuesta y crea un solo pedido (3.566625ms)
✓ la misma clave con otro cuerpo se rechaza con 422 (3.580167ms)
```

El reintento y la clave de idempotencia



- El pedido y la clave se guardan en la misma transacción.
- Otro cuerpo con la misma clave: `422` (borrador de la IETF, todavía no es un RFC).
- “Ver si ya hay uno igual” no basta: Ana puede traer dos pedidos iguales en un día.

```
node --test api.test.ts: las tres primeras pruebas

✓ sin clave, repetir el POST crea dos pedidos (18.457084ms)
✓ con clave, el reintento devuelve la misma respuesta y crea un solo pedido (3.566625ms)
✓ la misma clave con otro cuerpo se rechaza con 422 (3.580167ms)
```

Patrones: la pregunta es siempre la misma

- **Strategy:** cambiar un algoritmo sin tocar a quien lo usa. En TypeScript, suele ser una función.
- **Dependency injection:** recibir las dependencias, sin crearlas.
- **Adapter:** encajar una interfaz con la que se espera.
- **Repository, decorator, observer.**

Error común

Aplicar patrones porque existen. Una fábrica con una sola implementación agrega interfaces sin esconder nada.

En una revisión de código

¿Qué cambio futuro abarata este patrón, y qué tan probable es?

SECCIÓN 07

Arquitectura

07

Arquitectura: las decisiones caras de cambiar

	Se gana	Se paga
Capas	Se entiende y se revisa	Vale lo que vale su cumplimiento
Hexagonal	Probar sin base de datos, cambiar de proveedor	Más interfaces
Monolito	Llamadas locales, transacciones simples, un despliegue	Todo se despliega junto
Modular	Lo anterior, y se puede partir después	Cuidar las fronteras
Microservicios	Despliegues independientes, escalar por partes	Fallas parciales, latencia, consistencia eventual, operación

Dos leyes

Conway: el sistema tiende a copiar la comunicación de quien lo construye. **Monolith first:** partir antes de conocer las fronteras produce fronteras equivocadas.

Un atributo de calidad, con sus cuatro partes

Estímulo	El proveedor de WhatsApp deja de responder
Entorno	En horario de atención, un sábado, con la encargada atendiendo
Respuesta	Marcar como listo sigue funcionando; el aviso queda pendiente y se reintenta
Medida	Marcar como listo responde en menos de 1 s; el 99 % de los avisos pendientes sale dentro de los 5 minutos siguientes a que el proveedor vuelva

No es un requisito

“La aplicación debe ser rápida y estar siempre disponible.” Sin estímulo, sin entorno y sin número no se puede verificar.

Un ADR, con las secciones de P2

ADR-0001: Los avisos se envían desde una tabla de pendientes

Contexto. El proveedor a veces falla o tarda. Hilda no puede dejar de atender. Dos personas, 30 pedidos al día, un servidor.

Decisión. Marcar como listo guarda, en la misma transacción, el estado y un aviso pendiente. Un proceso del mismo monolito los envía cada minuto.

Alternativas. Enviar en la misma solicitud: simple, pero la encargada espera y el aviso se puede perder. Un servicio con una message queue: separa la falla, pero es mucha operación para 30 avisos al día.

Consecuencias. Ganamos: el mostrador no depende del proveedor. Perdemos: hasta un minuto de demora, y un aviso puede salir dos veces.

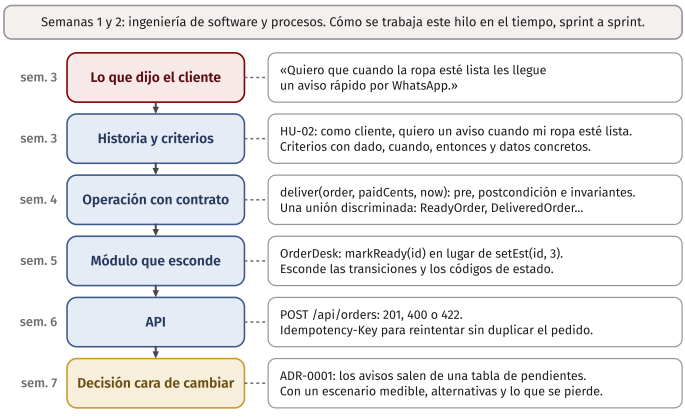
Lo que hace buena la respuesta

Alternativas de verdad; los números que deciden; lo que se pierde.

No se edita

Si la decisión cambia, se escribe un ADR nuevo que lo reemplaza.

El hilo de la primera mitad



El parcial cubre las semanas 1 a 7, sobre un caso que no han visto: pide aplicar y justificar.

SECCIÓN 08

El proyecto

08

Cerrar el sprint 1 y entregar P2

Lean su P2 como si fuera de otro equipo

- ¿Cada criterio se comprueba con un sí o un no?
- ¿Qué decisión esconde cada módulo?
- ¿Está escrita la frontera entre `400` y `422`?
- ¿Qué pasa si la solicitud más importante se repite?
- ¿Cada ADR dice lo que se pierde?

Demo y retrospectiva

- Solo lo que cumple la Definition of Done.
- Una conversación sobre qué cambia en el backlog.
- Acciones concretas, con responsable.

repositorio del equipo

```
make -f /work/projects/p2-diseno/Makefile check
```

Para la próxima clase

- **Sesión B:** examen parcial, sin herramientas de IA. Practiquen con el parcial de práctica.
- **P2:** entrega y demo del sprint 1.
- **P3:** publicado. Segundo sprint, calidad; entrega en la semana 11.
- Semana 9, pruebas. Lecturas: SWEG, caps. 11 (*Testing Overview*), 12 (*Unit Testing*) y 13 (*Test Doubles*).

Pregunta

¿Qué decisión de su P2 no podrían defender con sus alternativas? Empiecen a estudiar por ahí.