

UNIVERSIDAD NACIONAL DE INGENIERÍA

Ingeniería de Software

Ingeniería de software

Programar a lo largo del tiempo

Gerson Garrido

Semana 1

Hoy

Sesión A

1. El curso en una página
2. Programar frente a ingeniería de software
3. La ley de Hyrum, en Kallpa
4. Escala, políticas y shift left
5. Decisiones con costos

Sesión B y laboratorio

6. Qué hace bien y mal la IA
7. Código que pasa sus propias pruebas
8. Evidencia: METR, DORA, CMU
9. Responsabilidad y política del curso
10. Git en equipo: ramas, PR y conflictos

Lectura

SWEG, cap. 1 (complementaria: cap. 2) · CMU News, *CMU Students Put AI Coding Tools to the Test* (2025) · METR, *Measuring the Impact of Early-2025 AI on…* (2025), con su actualización.

SECCIÓN 01

El curso

01

Cuatro partes

Proceso y requisitos

procesos ágiles, requisitos, especificaciones

sem. 1–4

Diseño y arquitectura

complejidad, módulos, APIs, ADR

sem. 5–8

Calidad

pruebas, revisión, análisis estático, CI

sem. 9–12

Producción y evolución

despliegue, agentes, seguridad, mantenimiento

sem. 13–16

Las herramientas de IA son parte del tema en todo el curso, no un atajo ni un tabú.

Cómo funciona el curso

- **Kallpa**: las reservas de un complejo deportivo comunal de Lima. TypeScript, Node.js 26, SQLite. Casi todos los laboratorios.
- **Proyecto**: un producto para un cliente real, en tres sprints.
- **P0** (individual): un issue de Kallpa resuelto en una rama, con pruebas y un PR revisado. Entrega en la semana 3.

Entrega	Semana
P0 (individual)	3
P1 a P4 (equipo)	5, 8, 11, 14
P5: producto final	16
Examen parcial	8
Examen final	tras la 16

IA en el curso

Permitida en proyecto y laboratorios, con **registro de uso** y **defensa oral**. Exámenes sin IA.

El entorno: el mismo para todos

- Un contenedor **Docker** con Node.js 26, TypeScript, Git, `gh` y las herramientas de calidad.
- Igual en Linux, Windows (WSL2) y macOS.
- Editan en su computadora; ejecutan en el contenedor.
- Se instala en el laboratorio de hoy.



SECCIÓN 02

Programación e ingeniería de software

02

Tres diferencias

Tiempo

El código tiene que poder cambiar mientras viva.

Escala

Más personas y más código: ¿más eficiente o más lento?

Decisiones

Con más en juego y estimaciones imprecisas.

Ingeniería de software

Programación integrada en el tiempo (así lo dicen dentro de Google): políticas, prácticas y herramientas para que el código siga siendo útil mientras haga falta, y un equipo pueda colaborar en él.

No es “mejor” que programar: son dos dominios con prácticas distintas.

¿Cuánto tiempo va a vivir este código?

	Script de una tarde	**Kallpa**
Ejemplo	Contar las reservas de marzo en un archivo	Las reservas de los vecinos
Vida esperada	Una tarde	Años
Quién lo cambia	Nadie	Muchas personas
Qué cambia debajo	Nada que importe	Node.js, el navegador, los precios, la ley
¿Pruebas, revisión, CI?	Probablemente no	Sí



Sostenibilidad

Proyecto sostenible

Durante toda la vida esperada del software, es **capaz** de reaccionar a cualquier cambio **que valga la pena**, por razones técnicas o de negocio.

- **Capacidad**, no obligación: se puede decidir no actualizar. Lo riesgoso es no poder.
- Cambiar no es bueno en sí mismo: solo los cambios que valen la pena.
- En Kallpa: una vulnerabilidad en una dependencia, Node.js nuevo, comprobantes electrónicos, “los domingos atendemos de 8:00 a 20:00”.

La primera actualización grande duele

Supuestos escondidos, poca experiencia, años de cambios juntos. Mejor invertir en que actualizar sea barato.

Escala: muchas personas, muchas versiones

- “The multiperson development of multiversion programs”. (Atribución incierta, dice SWEG: Randell, Hamilton o Parnas.)
- El término se popularizó con las conferencias de la OTAN de 1968 y 1969.
- **El mito del genio** (SWEG cap. 2): Linux no lo escribió una persona; Torvalds escribió el comienzo y coordinó a miles.

Bus factor

Cuántas personas tendrían que ser atropelladas por un bus para que el proyecto quede condenado.

En su proyecto

Una parte que solo entiende una persona (o nadie) es un riesgo para el equipo y la defensa oral.

SECCIÓN 03

La ley de Hyrum

03

La ley de Hyrum

Ley de Hyrum

Con una cantidad suficiente de usuarios de una API, no importa lo que se prometa en el contrato: todo comportamiento observable del sistema va a ser usado por alguien.

- **Contrato**: lo que se promete (documentación, tipos, especificación).
- **Comportamiento observable**: todo lo que alguien puede ver. Orden, mensajes, tiempos, formatos.
- Como la entropía: no se elimina, se administra.



En Kallpa: el contrato y lo que se ve

Contrato (README): `GET /api/courts` lista las canchas, `200 { courts: [{ id, name, sport }] }`. Nada sobre el orden.

comportamiento observable

```
$ curl -s localhost:3000/api/courts | jq -c '.courts[]'
{"id":4,"name":"Básquet","sport":"basquet"}
{"id":1,"name":"Fulbito 1","sport":"fulbito"}
{"id":2,"name":"Fulbito 2","sport":"fulbito"}
{"id":3,"name":"Vóley","sport":"voley"}
```

src/store/court-repository.ts

```
1  return this.#db.prepare('SELECT id, name, sport FROM courts ORDER BY name').all() as
       Court[];
```

Ordenadas por nombre. Nadie lo prometió; cualquiera lo puede ver.

Alguien depende de ese orden

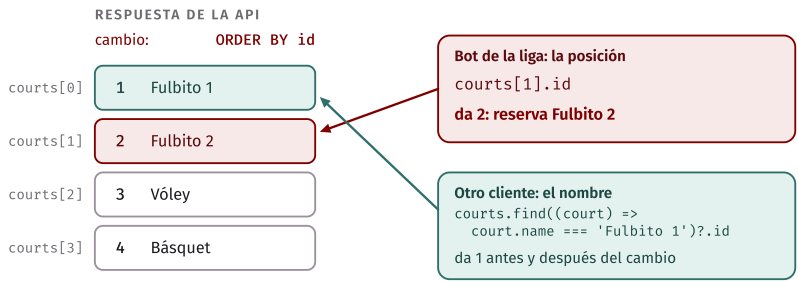
hyrum-clients.ts

```ts
// El bot de WhatsApp de una liga: "la segunda cancha es Fulbito 1".
export function fulbitoOneFragile(courts: CourtJson[]): number {
  return courts[1].id;
}
// Busca por lo que la API sí da: el nombre.
export function fulbitoOne(courts: CourtJson[]): number | undefined {
  return courts.find((court) => court.name === 'Fulbito 1')?.id;
}
```

- Hoy los dos devuelven `1`.
- Alguien cambia la consulta a `ORDER BY id`: respeta el README, las pruebas de Kallpa pasan…
- …y el bot reserva Fulbito 2. La versión que busca por nombre sigue en `1`. (comprobado en `hyrum.test.ts`)

El mismo cambio, dos clientes

Contrato (README): `200 { courts: [{ id, name, sport }] }` *del orden no dice nada*



El bot reserva Fulbito 2 sin que nadie se entere, hasta que dos equipos llegan a la misma cancha.

Más usuarios del mismo orden

- La **página** de Kallpa arma la lista de canchas en el orden de la API: abre con Básquet elegida.
- SQLite ordena el texto **byte a byte** (intercalación `BINARY`), no como el diccionario. En UTF-8, la `Á` va después de la `V`:

code/week-01

```
$ node courts-order.ts
Hoy:                    4 Básquet | 1 Fulbito 1 | 2 Fulbito 2 | 3 Vóley
Con una cancha nueva:   4 Básquet | 1 Fulbito 1 | 2 Fulbito 2 | 3 Vóley | 5 Área techada
```

Casi cierto

“Vienen en orden alfabético” es casi cierto. El “casi” es donde alguien se va a equivocar.

Códigos para las máquinas, mensajes para las personas

El README promete un `code` estable “que los clientes HTTP pueden comparar sin leer el mensaje”.

una fecha que ya pasó

```
$ curl -s -X POST localhost:3000/api/bookings -H "content-type: application/
    json" -d '{"courtId":1,"memberId":2,"date":"2026-11-01","startHour":7,"
    hours":1,"teamName":"Real Barranco"}' | jq
{
  "error": {
    "code": "IN_THE_PAST",
    "message": "Esa hora ya pasó."
  }
}
```

El `code` es para los programas; el `message`, para las personas (la página lo muestra tal cual).

Un cliente que compara el mensaje

hyrum-clients.ts

```ts
export function isPastHourFragile(body: ErrorJson): boolean {
  return body.error.message === 'Esa hora ya pasó.';   // se rompe si alguien mejora el texto
}
export function isPastHour(body: ErrorJson): boolean {
  return body.error.code === 'IN_THE_PAST';            // lo que el README promete
}
```

En el laboratorio de hoy van a cambiar justo ese mensaje. Las pruebas de Kallpa comparan el código: siguen pasando.

Qué puede hacer quien mantiene una API

1. **Decir qué se promete y qué no**: “el orden no está garantizado”.
2. **Prometerlo de verdad** si es útil, con una prueba que lo verifique.
3. **Dar algo estable para las máquinas**: `code` frente a `message`.
4. **Hacer inestable lo no prometido**: lenguajes que cambian el orden de sus tablas hash entre ejecuciones.
5. **Medir antes de cambiar**: quién depende, y avisar con tiempo.

Error común

“Si no está en la documentación, lo puedo cambiar.” La documentación decide de quién es la culpa, no si se rompe.

Con IA

Un agente que lee `ORDER BY name` escribe `courts[1]` sin dudar: depende de todo lo que observa.

“Funciona” frente a “es correcto”

- El ejemplo de SWEG: el orden de recorrido de una tabla hash.
- Para un script de una tarde, depender de él **funciona**.
- Para un sistema que vive años, es un riesgo: algo va a hacer valioso cambiarlo (eficiencia, hash flooding, evolución).
- Hasta el orden *aleatorio* tuvo usuarios: programas que lo usaban como generador de números aleatorios.

> **SWEG, cap. 1**
>
> Es programación si “ingenioso” es un elogio, y es ingeniería de software si “ingenioso” es una acusación.

SECCIÓN 04

Escala y políticas

04

Lo que se hace muchas veces tiene que escalar

La prueba del ×10: multipliquen la organización por 10. ¿Quien cumple la política tiene 10 veces más trabajo?

No escala	Escala
"El 15 de agosto borramos el componente viejo; migren"	**Churn Rule**: quien cambia la infraestructura migra a sus usuarios
Ramas de desarrollo de larga vida, merges grandes	Cambios chicos, con merge frecuente en la rama principal
Pruebas que cada equipo corre a su manera	**Beyoncé Rule**: "if you liked it, you should have put a CI test on it"

El compilador de 2006

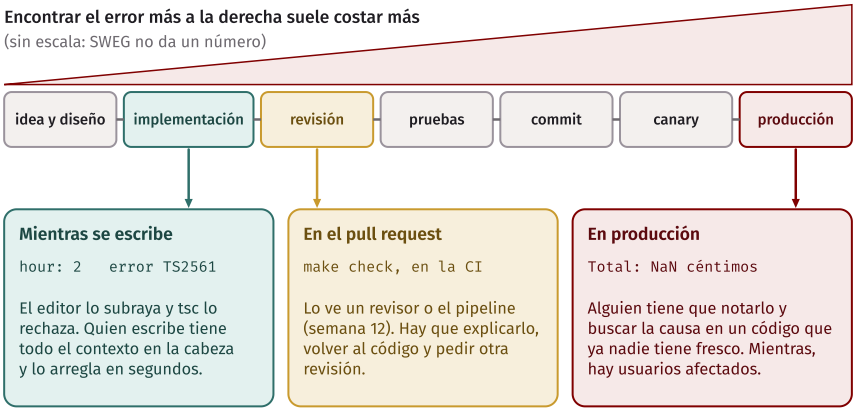
- Varios miles de ingenieros, cinco años con el mismo compilador.
- Casi nadie sabía hacer la actualización; el código dependía de detalles de esa versión (ley de Hyrum).
- Sin Beyoncé Rule ni CI extendida: imposible saber el impacto de antemano.
- Después: **automatización**, **consistencia** y **especialistas**. Hoy, para algunos lenguajes, una actualización cada una o dos semanas.

La lección

Cuanto más seguido se cambia la infraestructura, más fácil se vuelve cambiarla.

SWEG, cap. 1, *Example: Compiler Upgrade*.

Shift left: encontrar el error antes



Shift left: por qué conviene

- Quien escribe el código tiene todo el contexto; en producción, nadie lo tiene fresco y hay usuarios afectados.
- SWEG no da un número: dice que *usualmente* reduce costos. Varias capas: análisis estático, revisión, pruebas.
- En Kallpa, `make check` empieza con `tsc`: Node no revisa los tipos, solo los quita.

Un error de tipeo: Node no lo ve, `tsc` sí

shift-left/price-typo.ts

```
1  const cents = quotePriceCents({ sport: 'voley', date: '2026-03-10', startHour: 9, hour: 2, plan: '
       regular' });
2  console.log(`Total: ${cents} céntimos`);
```

shift-left

```
$ node price-typo.ts
Total: NaN céntimos
$ tsc -p .
price-typo.ts(7,83): error TS2561: Object literal may only specify known
    properties, but 'hour' does not exist in type 'PriceQuote'. Did you mean to
     write 'hours'?
```

SECCIÓN 05

Decisiones con costos

05

Qué es un costo

Costo	Ejemplo
Financiero	Licencias, servidores
De recursos	CPU, memoria
De personal	Horas de ingeniería
De transacción	Coordinar, migrar, capacitar
De oportunidad	Lo que cuesta *no* actuar
Social	Accesibilidad, equidad, abuso

Los plumones

Armarios abiertos llenos de plumones: una lluvia de ideas sin cortes vale más que unos plumones perdidos.

Nunca

“Porque lo digo yo.” Sí: porque hay que hacerlo, o es la mejor opción con la evidencia de hoy.

Decidir con datos, y revisar la decisión

- Lo medible (horas frente a CPU) se mide. Lo difícil de medir (una API mal diseñada) se decide con experiencia, precedentes y más cuidado.
- **Builds distribuidos** en Google: compilar dejó de doler… y nadie cuidó las dependencias. Los builds se inflaron. Hubo que replantear el sistema.

Idea clave

La evidencia cambia; las decisiones se revisan. Quien decide necesita el derecho a admitir que se equivocó.

Error común

Medir un proyecto por cuánto código se escribió: cada línea hay que leerla, probarla y mantenerla.

Decisiones con costos en Kallpa

Decisión	Cuesta	Gana
Dinero en **céntimos enteros**	Convertir al mostrar y al leer	Sumas exactas; la caja cuadra
Fecha como **texto**, hora como **entero**	No sirve para instantes	Sin zonas horarias que confundir
El reloj como **dependencia**	Un parámetro más	Pruebas que siempre dan lo mismo
Sin dependencias de npm	Escribir el router a mano	Nada que actualizar ni auditar

SECCIÓN 06

La IA en el ciclo de desarrollo

06

Qué es una herramienta de IA para programar

- Un **modelo de lenguaje** (LLM) predice cómo sigue un texto, el **prompt**. “Probable” no es “correcto”.
- Solo sabe de su proyecto lo que cabe en su **context window**.

Tipo	Qué hace	Qué ve
Autocompletado	Propone cómo sigue la línea	El archivo y algunos cercanos
Chat	Responde y propone código que ustedes copian	Lo que le pegan
Agente	Edita archivos, corre comandos, mira el resultado, reintenta	Lo que decide leer y ejecutar

Qué hace bien y qué hace mal

Suele hacer bien

- Código repetitivo
- Primeros borradores de algo que se sabe revisar
- Explicar código ajeno
- Traducir entre lenguajes o versiones
- Proponer pruebas y casos borde

Suele hacer mal

- Código que compila, pasa sus propias pruebas y está mal
- Inventar funciones, opciones, paquetes
- Seguir instrucciones escondidas en lo que lee
- Saber lo que el cliente quiso decir

Lo que hace bien tiene algo en común: quien la usa sabe juzgar el resultado.

Un cambio como los que suele proponer un agente

Pedido: “convertir a céntimos montos como `'S/ 112.50'`”. (Ilustración escrita para la clase.)

agent-change/parse-soles.ts

```ts
export function parseSoles(text: string): number {
  const amount = parseFloat(text.replace('S/', '').trim());
  return Math.floor(amount * 100);
}
```

agent-change/parse-soles.test.ts (las pruebas que trae)

```ts
test('convierte una tarifa de fulbito en hora punta', () => {
  assert.equal(parseSoles('S/ 112.50'), 11250);
});
test('convierte un monto sin céntimos', () => {
  assert.equal(parseSoles('S/ 90'), 9000);
});
// ... y dos más: 'S/ 62.50' y 'S/ 81.00'
```

Las pruebas pasan

```
node --test agent-change/parse-soles.test.ts

✓ convierte una tarifa de fulbito en hora punta (0.47725ms)
✓ convierte un monto sin céntimos (0.082333ms)
✓ convierte una tarifa de vóley en hora punta (0.06875ms)
✓ acepta el monto sin el símbolo de soles (0.065209ms)
ℹ tests 4
ℹ suites 0
ℹ pass 4
ℹ fail 0
ℹ cancelled 0
ℹ skipped 0
ℹ todo 0
ℹ duration_ms 76.41875
```

Compila, se lee bien y las cuatro pruebas pasan. ¿Lo aceptarían?

La regla estaba en el README

Antes de responder, vuelvan al README:

> **README de Kallpa**
>
> Todo el dinero se maneja en **céntimos, como enteros**. Con números decimales, `0.1 + 0.2` no da `0.3` y los totales no cuadran con la caja.

- `0.29` no existe exacto en binario: `0.29 * 100` da `28.999999999999996`; `Math.floor` lo baja a 28.
- Las cuatro pruebas usan 112.50, 90, 62.50 y 81: montos exactos en binario. Salieron de la implementación, no de la regla.

Una prueba que sale de la regla

Todo monto que Kallpa muestra con `formatSoles` (11250 se muestra `'S/ 112.50'`) tiene que volver a leerse igual. Para todos los montos de S/ 0.00 a S/ 1000.00:

```
node --test agent-change/roundtrip.check.ts

✗ todo monto que muestra Kallpa se vuelve a leer igual (16.208208ms)
ℹ tests 1
ℹ suites 0
ℹ pass 0
ℹ fail 1
...
✗ failing tests:

test at agent-change/roundtrip.check.ts:11:1
✗ todo monto que muestra Kallpa se vuelve a leer igual (16.208208ms)
  AssertionError [ERR_ASSERTION]: 4586 montos no vuelven: 'S/ 0.29' -> 28, 'S/ 0.57' -> 56, 'S/ 0.58' -> 57...
  ...
```

4586 de 100 001 montos se leen con un céntimo de menos.

La corrección, y la lección

agent-change/parse-soles-fixed.ts

```ts
const AMOUNT = /^(?:S\/ ?)?(\d{1,7})(?:\.(\d{2}))?$/;

export function parseSoles(text: string): number | null {
  const match = AMOUNT.exec(text.trim());
  if (match === null) return null;
  const soles = Number(match[1]);
  const cents = match[2] === undefined ? 0 : Number(match[2]);
  return soles * 100 + cents;
}
```

Error común

Aceptar código que pasa pruebas que generó la misma herramienta. Esta versión pasa los 4 ejemplos y la ida y vuelta; ¿de dónde salió cada valor esperado?

Evidencia: METR (primera mitad de 2025)

El diseño: experimento controlado aleatorio.

- 16 desarrolladores con experiencia, en **sus propios** repositorios grandes de código abierto.
- 246 issues reales, de unas 2 horas.
- Cada issue, **al azar**: con IA o sin IA.
- Herramientas de frontera de ese momento.

Tiempo con IA	
Esperaban antes	−24 %
Creían después	−20 %
Se midió	+19 %

La lección

La sensación de productividad no es una medición.

No muestra (según METR): que la IA no acelere a la mayoría, ni en otros dominios, ni en el futuro cercano, ni que no haya mejores formas de usarla.

La actualización de 2026

- La página del estudio hoy dice: resultados desactualizados.
- Segundo experimento (desde agosto de 2025): 57 desarrolladores, más de 800 tareas.
- Cambio estimado en el tiempo: -18 % para los que repetían (intervalo -38 % a $+9$ %); -4 % para los nuevos (-15 % a $+9$ %). **Los dos intervalos incluyen el cero.**
- METR dice que la señal es **poco confiable**: muchos ya no querían trabajar sin IA, y el estudio pierde justo a quienes más se benefician.
- Cree que hoy la aceleración es probablemente mayor que en 2025, pero sus datos son evidencia muy débil de cuánto.

Dos conclusiones equivocadas

"La IA te hace 19 % más lento" (herramientas de 2025). "La IA te hace 18 % más rápido" (datos que sus autores llaman poco confiables).

DORA 2025 y un curso de CMU

DORA, *State of AI-assisted Software Development*

- Una **encuesta** a profesionales: describe y asocia, no prueba causas.
- La IA es un **amplificador** de las fortalezas y debilidades que el equipo ya tiene.

CMU, *AI Tools for Software Development*

- Un curso experimental (otoño 2025) donde los estudiantes no escriben código a mano.
- Vibe coding: “programar con la esperanza de que funcione sin probarlo”.
- El objetivo: una mentalidad de ingeniería para que la IA sea confiable.
- Es una noticia, no un estudio.

Cómo leer un estudio sobre IA y productividad

1. **¿Qué se midió?** Tiempo, líneas, PRs, satisfacción, o la opinión sobre todo eso.
2. **¿Cómo?** Experimento aleatorio, datos observados o encuesta.
3. **¿A quiénes, con qué tareas?** Expertos en su código, estudiantes, una empresa.
4. **¿Cuándo, con qué herramientas?** Un año es mucho en este tema.
5. **¿Qué dicen los autores que *no* muestra?**

> **Anécdota frente a medición**
>
> "A mí me hizo el doble de rápido" dice algo de una persona en una situación. No dice qué va a pasar en la de ustedes.

La responsabilidad no se delega

- **Quien hace commit responde por el código**, lo haya escrito quien lo haya escrito.
- Si producir código se vuelve barato, lo caro es saber si está bien: especificaciones (sem. 3–4), pruebas (9–10), revisión (11), CI (12) importan más.

Con IA: antes de hacer commit

1. ¿Qué hace este código, línea por línea?
2. ¿Contra qué **regla** lo comprobé (no contra qué prueba de la misma herramienta)?
3. ¿Qué caso borde probé yo?
4. ¿Lo podría explicar en la defensa oral sin mirar la conversación?

La política del curso: el registro de uso de IA

- Permitida en proyecto y laboratorios.
- Se registra: qué se pidió, a qué herramienta, qué se aceptó, qué se rechazó y por qué.
- Lo copiado casi tal cual se cita.
- Exámenes sin IA.

Campo	Ejemplo (el de hoy)
Qué se pidió	Convertir `'S/ 112.50'` a céntimos, con pruebas
Qué se aceptó	Aceptar el monto con o sin `S/`; los 4 casos como ejemplos
Qué se rechazó	`parseFloat` y `Math.floor`: viola la regla de enteros; lee mal 4586 montos
Cómo se verificó	Ida y vuelta con `formatSoles` hasta S/ 1000.00; inválidos dan `null`

El registro obliga a escribir **cómo se verificó**: justo lo que nadie hace cuando acepta código sin leerlo.

SECCIÓN 07

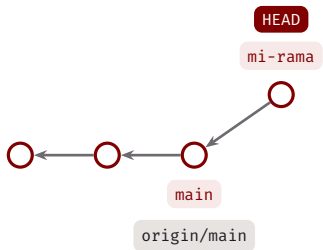
Git en equipo

07



Git en cinco ideas

1. **Repositorio**: la carpeta con toda su historia (`.git`).
2. **Commit**: una foto completa + autor, fecha, mensaje y **padres**; se identifica por su **hash**.
3. **Rama**: un nombre que apunta a un commit y avanza con cada commit nuevo.
4. **HEAD**: dónde están parados.
5. **Remoto** (`origin`): otra copia. `push` sube, `fetch` trae.



Lo que guarda un commit

git cat-file -p

```
ana$ git cat-file -p HEAD
tree 06ff0c6bc1a05169b294be8637971eb4b570d565
parent e5ba27cec5dc6e56a89ab46e9c78cadb122851b0
parent 2d1b2a97187d6d171839b68bc0463f9c930ff307
author Ana Quispe <ana@uni.example> 1774452600 -0500
committer Ana Quispe <ana@uni.example> 1774452600 -0500

Merge pull request #1 from ana/mensaje-hora-pasada
```

- `tree`: la foto de los archivos. Dos `parent`: es un **merge**.
- Si cambia cualquier cosa (un byte, la fecha, el autor), cambia el hash.

Commits con mensajes útiles

- Una línea de resumen de hasta ∼60 caracteres, en **imperativo**: “si se aplica, este commit…”
- Si hace falta, una línea en blanco y el **porqué**. El qué ya está en el diff.
- Un commit, un cambio.

En lugar de	Escriban
`cambios`	`Sugiere elegir otra hora cuando la hora ya pasó`
`arreglé el mensaje`	`Aclara que tampoco se reserva una hora ya empezada`
`Actualiza handlers.ts`	Lo que cambió en el comportamiento

Ramas y pull requests



Un buen PR

Chico. La descripción dice **qué** cambia, **por qué** y **cómo se probó**. `Closes #3` cierra el issue 3 al hacer merge en la rama principal.

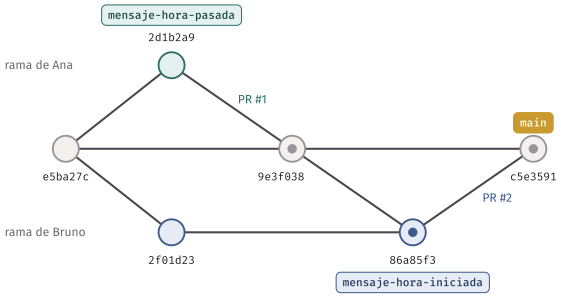
Revisar en GitHub

Comentar en una línea (*Files changed*), sugerir un cambio (bloque `suggestion`), y *Approve* o *Request changes*.

El laboratorio, con Ana y Bruno

`team-git.sh`: un repositorio bare hace de GitHub; el botón *Merge pull request* es `git merge --no-ff`.

Cada línea une un commit con su padre; lo más viejo está a la izquierda. Un punto al centro marca un merge.



Merge del PR #2: c5e3591 tiene como padres 9e3f038 y 86a85f3. main queda ahí.

Ana y Bruno cambian la misma línea

ana

```
ana$ git switch -c mensaje-hora-pasada
Switched to a new branch 'mensaje-hora-pasada'
ana$ git diff
...
@@ -63,7 +63,7 @@ export function checkBookingRules(slot: TimeSlot, now: LimaTime): BookingRuleVio
...
-    return { code: 'IN_THE_PAST', message: 'Esa hora ya pasó.' };
+    return { code: 'IN_THE_PAST', message: 'Esa hora ya pasó: elijan una hora libre.' };
...
ana$ git commit -am "Sugiere elegir otra hora cuando la hora ya pasó"
[mensaje-hora-pasada 2d1b2a9] Sugiere elegir otra hora cuando la hora ya pasó
 1 file changed, 1 insertion(+), 1 deletion(-)
```

bruno

```
bruno$ git commit -am "Aclara que tampoco se reserva una hora ya empezada"
[mensaje-hora-iniciada 2f01d23] Aclara que tampoco se reserva una hora ya empezada
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Merge del PR #1; Bruno trae `main`

bruno

```
bruno$ git log --oneline --graph --decorate --all
*   9e3f038 (origin/main, origin/HEAD) Merge pull request #1 from ana/mensaje-hora-pasada
|\
| * 2d1b2a9 (origin/mensaje-hora-pasada) Sugiere elegir otra hora cuando la hora ya pasó
|/
| * 2f01d23 (HEAD -> mensaje-hora-iniciada, origin/mensaje-hora-iniciada) Aclara que tampoco se reserva una hora ya empezada
|/
* e5ba27c (main) Versión inicial de Kallpa
bruno$ git merge origin/main
Auto-merging src/domain/booking.ts
CONFLICT (content): Merge conflict in src/domain/booking.ts
Automatic merge failed; fix conflicts and then commit the result.
bruno$ git status
...
Unmerged paths:
  (use "git add <file>..." to mark resolution)
    both modified:   src/domain/booking.ts
```

`e5ba27c` es la **base del merge**. Los dos lados cambiaron la misma zona: Git no puede decidir solo.

Las marcas de un conflicto

src/domain/booking.ts

```
  if (daysAhead < 0 || (daysAhead === 0 && slot.startHour <= now.hour)) {
<<<<<<< HEAD
    return { code: 'IN_THE_PAST', message: 'No se puede reservar una hora que ya empezó.' };
=======
    return { code: 'IN_THE_PAST', message: 'Esa hora ya pasó: elijan una hora libre.' };
>>>>>>> origin/main
```

- Arriba: `HEAD`, la rama donde están parados (Bruno).
- Abajo: lo que están trayendo (`origin/main`, con el cambio de Ana).
- Resolver **no** es elegir un lado al azar ni dejar las dos líneas: es decidir qué tiene que decir el código ahora.

Resolver, probar, hacer commit

bruno

```
bruno$ sed -n 64,66p src/domain/booking.ts
  const daysAhead = daysBetween(now.date, slot.date);
  if (daysAhead < 0 || (daysAhead === 0 && slot.startHour <= now.hour)) {
    return { code: 'IN_THE_PAST', message: 'Esa hora ya empezó: elijan una hora libre.' };
bruno$ git add src/domain/booking.ts
bruno$ git commit --no-edit
[mensaje-hora-iniciada 86a85f3] Merge remote-tracking branch 'origin/main' into mensaje-hora-iniciada
bruno$ git push origin mensaje-hora-iniciada
```

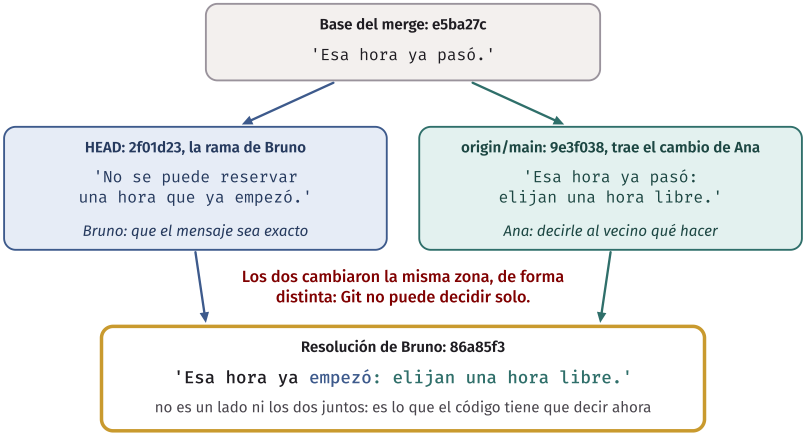
Error común

“Aceptar lo mío” sin leer lo otro.

Con IA

Si resuelve una herramienta: lean y prueben.

Tres versiones de la misma línea



Leer `git log --graph`

ana

```
ana$ git log --oneline --graph --decorate
*   c5e3591 (HEAD -> main, origin/main) Merge pull request #2 from ana/mensaje-hora-iniciada
|\
| *   86a85f3 (origin/mensaje-hora-iniciada) Merge remote-tracking branch 'origin/main' into mensaje-hora-iniciada
| |\
| |/
|/|
* |   9e3f038 Merge pull request #1 from ana/mensaje-hora-pasada
|\ \
| * | 2d1b2a9 (origin/mensaje-hora-pasada, mensaje-hora-pasada) Sugiere elegir otra hora cuando la hora ya pasó
|/ /
| * 2f01d23 Aclara que tampoco se reserva una hora ya empezada
|/
* e5ba27c Versión inicial de Kallpa
```

- Arriba lo más nuevo. Cada `*` es un commit; `|\` es un merge con dos padres.
- Tres merges para dos cambios de una línea: por eso hay equipos que usan squash o `rebase`. Es una decisión con costos.

En el laboratorio

1. Docker y Git; `user.name` y `user.email` **en su computadora** (el contenedor monta `~/.gitconfig` de solo lectura).
2. Su copia de Kallpa como dice P0 (paso 1). `make check` tiene que pasar; `make start` y una reserva.
3. En parejas: la **misma línea**, dos ramas, dos PR. Quien hace el segundo merge resuelve el conflicto y corre `make check`.
4. Revisión del PR de su pareja: comentario, sugerencia, aprobación.
5. `git log --oneline --graph`: expliquen cada `*` y cada merge.
6. Lean P0 y calculen su issue.

> **Push y pull**
>
> Desde la terminal de su computadora, donde están sus credenciales de GitHub.

Para la próxima clase

- Leer SWEG, cap. 1, la noticia de CMU y el resumen de METR con su actualización. Complementarias: SWEG, cap. 2; DORA, *State of AI-assisted Software Development 2025*; el artículo de METR en arXiv.
- Terminar el laboratorio: conflicto resuelto, PR revisado, `git log --graph` explicado.
- Calcular su issue de **P0** (entrega en la semana 3) y leerlo.
- Código de la semana: `code/week-01`, `make test`.

Pregunta

Una herramienta les genera una función y sus pruebas, y todas pasan. ¿Qué tienen que hacer antes de hacer commit? ¿Qué escribirían en el registro de uso de IA?