Maksym Stepanenko



Deuda técnica: cómo una startup se mata con su propio código

Qué es la tech debt, cómo medirla y cuándo pagarla

Deuda técnica: cómo una startup se mata con su propio código

Qué es la tech debt, cómo medirla y cuándo pagarla

Maksym Stepanenko

Innova Publishing · 2026

Página del libro, versión actualizada y las demás guías de la serie:
guides.mmix.biz/es/tech-debt

CAPÍTULO 1

Qué es la deuda técnica y de dónde sale

La deuda técnica es una metáfora propuesta por Ward Cunningham en 1992. Una solución rápida ahora = pedirle un crédito al futuro. No tiene por qué ser malo: a veces es la decisión estratégicamente correcta. El problema empieza cuando la deuda se acumula y empieza a frenar la velocidad de desarrollo, con «pagos de intereses» en forma de entregas lentas (delivery) y un gran número de bugs.

De dónde sale la tech debt: Time pressure (prisa del MVP): «Tenemos que lanzar en 2 semanas». Las decisiones se toman rápido, no se escriben pruebas, la arquitectura no se piensa. Es normal y a menudo correcto en la etapa previa al producto (pre-product stage). Growing team: distintos desarrolladores tienen distintos estándares. Sin una cultura de code review, cada uno escribe a su manera. Se acumula la incoherencia (inconsistency). Changing requirements: la arquitectura se diseñó para un solo caso de uso. Los requisitos cambian y se añaden hacks y workarounds en lugar de rediseñarla. Knowledge silos: «solo Vasia sabe cómo funciona el módulo de autenticación». Vasia se marcha. Nadie entiende el código. Outsourced development: los contratistas externos optimizan para terminar (completion), no para la mantenibilidad (maintainability). Adiós a la arquitectura.

Tipos de deuda técnica según el cuadrante de Martin Fowler (Technical Debt Quadrant, martinfowler.com, 2009; la propia metáfora de la deuda la propuso Ward Cunningham): Reckless deliberate: «No hay tiempo para pensar en la arquitectura, lanzamos». Prudent deliberate: «Sabemos que no es lo ideal, pero ahora hay que lanzar; lo arreglaremos después». Reckless inadvertent: «No sabíamos que lo estábamos haciendo mal». Prudent inadvertent: «Ahora entendemos cómo había que hacerlo». Reckless deliberate es la más tóxica. Prudent deliberate es una práctica de negocio normal si se gestiona bien.

Cuándo la tech debt mata a las startups: la startup crece. Contrata a más desarrolladores. La velocity cae (¡no crece!) con cada nueva contratación. Cada nueva feature obliga a tocar 10 sitios del código. Arreglar un bug crea tres nuevos. El CEO: «Hemos contratado a 5 desarrolladores más, ¿por qué desplegamos más despacio?» Es el envenenamiento por tech debt en fase avanzada (advanced stage tech debt poisoning). Señales: el cycle time de una feature, de la idea a producción, > 4 semanas. Más del 30% del tiempo del equipo se va en corregir bugs y atender incidentes. Los desarrolladores temen tocar ciertas partes del código. El onboarding de un nuevo desarrollador lleva más de 2 meses.

CAPÍTULO 2

Cómo medir la tech debt de forma objetiva

«Tenemos mucha tech debt» es una sensación subjetiva. El CEO no la comprará sin datos. Así se mide de forma objetiva.

DORA Metrics (DevOps Research and Assessment): cuatro métricas que se correlacionan con el rendimiento del negocio (business performance): Deployment Frequency: ¿con qué frecuencia despliega a producción? Elite: varias veces al día. Low: una vez al mes o menos. Lead Time for Changes: del commit a producción. Elite: < 1 hora. Low: de 1 a 6 meses. Change Failure Rate: % de deployments que provocan un incidente. Elite: 0–15%. Low: 46–60%. Time to Restore Service: con qué rapidez se recupera tras un incidente. Elite: < 1 hora. Low: de 1 a 6 meses. Mucha tech debt = malas DORA metrics. Es una prueba objetiva para el CEO.

Cycle Time: de «start» (el desarrollador toma el ticket) a «done» (la feature en producción). Mídalo en Jira, Linear o en su herramienta de gestión de proyectos. La tendencia importa más que la cifra absoluta. Si el cycle time crece con el mismo equipo, la tech debt se está acumulando.

Defect Rate: % de bugs reportados por feature entregada. Si crece, la calidad cae. Code Coverage: % del código cubierto por pruebas automatizadas. No es una métrica de oro (un 70% de coverage puede ser inútil si las pruebas son malas). Pero la ausencia total de pruebas = un indicador de riesgo.

Developer Experience Survey: una vez por trimestre, pregunte al equipo: «En una escala de 1 a 10, ¿hasta qué punto le resultó fácil introducir cambios en las últimas 2 semanas?» «¿Qué fue lo que más estorbó en su trabajo?» «¿Hay partes del código que le dé miedo tocar?» Datos cualitativos + cuantitativos = el cuadro completo.

SonarQube y CodeClimate: herramientas de análisis estático de código. Calculan: code duplication (violaciones de DRY). Complexity (cyclomatic complexity). Code smells (patrones asociados con problemas). Technical Debt en horas (una estimación de cuánto tiempo hace falta para corregirla). SonarQube: gratuito, self-hosted. CodeClimate: $50+ al mes para equipos.

CAPÍTULO 3

Priorización: cuándo pagar la deuda y cuándo ignorarla

No toda la tech debt merece la pena devolverla. La pregunta principal: ¿qué deuda frena más el crecimiento?

Tech Debt ROI framework: para cada deuda, evalúe: Impacto en la velocity (1–5): ¿cuánto frena esta deuda la delivery? Impacto en la fiabilidad (reliability) (1–5): ¿con qué frecuencia esta deuda provoca bugs o incidentes? Effort to fix (1–5, donde 5 = mucho esfuerzo). ROI score = (Velocity + Reliability) / Effort. Priorice por ROI score descendente.

Categorías de tech debt por urgencia: P0 — Fix Now: provoca incidentes en producción. Vulnerabilidades de seguridad. Bloquea peticiones clave de clientes (customer requests). P1 — Fix This Sprint: frena de forma considerable el desarrollo. Se comenta a menudo en los standups como fricción. P2 — Schedule for Next Quarter: reduce la legibilidad del código. Frenará futuras features. P3 — Track but Don't Fix Yet: un code smell sin impacto práctico. Puede que nunca se toque.

«Boy Scout Rule» (mejora gradual): cada vez que toque un fragmento de código, déjelo un poco mejor de como lo encontró. No refactorice todo. Simplemente: añada una prueba para la función que cambió. Mejore el nombre de la variable que le confundió. Elimine el dead code con el que se tope. Es una reducción orgánica de la deuda, sin sprints especiales.

20% time: dedique el 20% del tiempo de los desarrolladores a la tech debt en cada sprint. No es un «nice to have»: es una inversión en la velocity futura. Plantéeselo así al CEO: «Si no invertimos el 20% ahora, el año que viene haremos lo mismo en el doble de tiempo (2x)». Debt Sprints: una vez por trimestre, un sprint completo solo para la tech debt. Es adecuado si la deuda se ha acumulado hasta un nivel crítico. Inconveniente: los

desarrolladores pierden el contacto con el producto.

Lo que nunca debe hacer: un rewrite completo «desde cero». El Big Bang Rewrite es casi siempre una estrategia fallida. Lleva 3–5x más tiempo de lo esperado. El producto se queda parado mientras dura el rewrite. El código «nuevo» acumula su propia deuda. La alternativa: el Strangler Fig pattern. Sustitución gradual, por partes.

CAPÍTULO 4

Cómo hablar de tech debt con el CEO y los inversores

La deuda técnica es invisible para los stakeholders no técnicos. La tarea del CTO o Tech Lead es hacerla visible con lenguaje de negocio.

Traduzca las métricas técnicas a métricas de negocio: En lugar de «tenemos una cyclomatic complexity alta» → «cada nueva feature requiere el doble de tiempo (2x) por la complejidad del código». En lugar de «no hay unit tests» → «detectamos el 40% de los bugs en producción en lugar de en desarrollo, lo que nos cuesta N días de recuperación». En lugar de «tech debt» → «engineering debt» o «infrastructure investment». Las palabras importan.

Business case para invertir en tech debt: cálculo de ROI: velocity actual: 5 features al mes. Con el crecimiento actual de la deuda: dentro de 12 meses la velocity caerá a 2 features al mes. Coste de los desarrolladores: $50.000 al mes. 50% del trabajo de desarrollo dedicado a arreglar bugs en lugar de a nuevas features: son $25.000 al mes de «pérdidas». Inversión en 2 meses de reducción de deuda: $100.000. Resultado: la velocity vuelve a más de 5 features al mes. ROI: se amortiza en 4 meses.

Visualización de la tech debt: un burn-up chart con la tendencia de la velocity. Muestra la caída de la velocity con el paso del tiempo. La tasa de bugs a lo largo del tiempo. La frecuencia de incidentes. Estos gráficos los entiende cualquier CEO.

Technical due diligence antes de una ronda (pre-funding): los inversores y los compradores hacen una technical due diligence. Red flags: un monolito sin documentación. Sin pruebas. Sin CI/CD. Vulnerabilidades de seguridad (las encontrarán con herramientas SAST). Key-person dependency (el código solo lo lee una persona). Prepárese con antelación: un CI/CD básico. Documentación de las decisiones arquitectónicas clave. Secret scanning en el repositorio.

Dependency audit (bibliotecas obsoletas con CVEs conocidos).

Cultura de calidad: la tech debt es un síntoma, no la causa. La causa: una cultura que no valora la calidad del código. Cómo crear una cultura de calidad: code reviews obligatorios (aunque ralenticen el proceso). La Definition of Done incluye pruebas. Política de «broken window»: no deje código roto. Blameless postmortems después de los incidentes. Celebre el refactoring igual que las nuevas features.

CAPÍTULO 5

Herramientas y prácticas para gestionar la deuda

Las herramientas adecuadas hacen que la gestión de la tech debt sea sistemática y no reactiva.

CI/CD como base: Continuous Integration: build y pruebas automáticos en cada PR. GitHub Actions, GitLab CI y CircleCI son gratuitos para equipos pequeños. Continuous Delivery: deploy automático a staging tras el merge. Continuous Deployment: deploy automático a producción. Sin CI/CD: manual, lento y propenso a errores. Con CI/CD: rápido, fiable, elimina el factor humano del deployment.

Pruebas automatizadas: Unit tests: prueban funciones individuales. Son rápidas. Integration tests: prueban la interacción entre componentes. E2E tests: prueban los user flows en el navegador (Playwright, Cypress). No persiga un 100% de coverage. Empiece con: tests para los business-critical flows. Tests para los puntos que se rompen a menudo. Tests para cada nuevo bug (regression tests).

Monitorización y observabilidad: Datadog, New Relic o Grafana+Prometheus self-hosted. Métricas principales: application performance (latencia p95, p99). Error rate. Database query performance. Infrastructure health. Sentry o Bugsnag: error tracking automático. Cada unhandled exception → un ticket en su tracker. PagerDuty u OpsGenie: incident management y on-call rotation.

Documentación: Architecture Decision Records (ADR): documentos breves sobre las decisiones arquitectónicas clave. ¿Por qué elegimos PostgreSQL y no MongoDB? ¿Por qué microservicios y no un monolito? Esto no tiene precio dentro de 12 meses. README.md en cada repositorio: cómo arrancarlo en local. Cómo ejecutar las pruebas. Cómo hacer el deploy. Una guía de

onboarding para los nuevos desarrolladores.

Gestión de dependencias (dependency management): Dependabot (GitHub) o Renovate crean automáticamente PRs para actualizar las dependencias. OWASP Dependency Check comprueba los CVEs conocidos en las dependencias. Las dependencias obsoletas = tech debt + riesgo de seguridad. Automatice su actualización.

¿Y ahora qué?

Gracias por leer «Deuda técnica: cómo una startup se mata con su propio código».

Las actualizaciones de este libro, los archivos EPUB y PDF y todas las guías de la serie están en la página del libro: guides.mmix.biz/es/tech-debt

Maksym Stepanenko, Innova Publishing.