Maksym Stepanenko



Technische Schulden: Wie ein Start-up sich mit dem eigenen Code ruiniert

Was Tech Debt ist, wie Sie ihn messen und wann Sie ihn zurückzahlen sollten

Maksym Stepanenko · 2026

Technische Schulden: Wie ein Start-up sich mit dem eigenen Code ruiniert

Was Tech Debt ist, wie Sie ihn messen und wann Sie ihn zurückzahlen sollten

Maksym Stepanenko

Innova Publishing · 2026

Buchseite, aktuelle Fassung und die weiteren Leitfäden der Reihe: guides.mmix.biz/de/tech-debt

KAPITEL 1

Was technische Schulden sind und woher sie kommen

Technische Schulden sind eine Metapher, die Ward Cunningham 1992 vorgeschlagen hat. Eine schnelle Lösung jetzt bedeutet, einen Kredit bei der Zukunft aufzunehmen. Das ist nicht zwangsläufig schlecht – manchmal ist es strategisch richtig. Das Problem beginnt, wenn sich die Schulden anhäufen und die Entwicklungsgeschwindigkeit bremsen. Sie verursachen „Zinszahlungen“ in Form langsamer Delivery und vieler Bugs.

Woher Tech Debt kommt: Time Pressure (MVP-Rush): „Wir müssen in 2 Wochen starten.“ Entscheidungen fallen schnell, Tests werden nicht geschrieben, die Architektur ist nicht durchdacht. Das ist normal und in der Pre-Product-Phase oft richtig. Wachsendes Team: Verschiedene Entwickler haben unterschiedliche Standards. Ohne Code-Review-Kultur schreibt jeder auf seine Weise. Es sammeln sich Inkonsistenzen an. Changing Requirements: Die Architektur wurde für einen Use Case entworfen. Die Requirements ändern sich, und statt eines Redesigns kommen Hacks und Workarounds dazu. Knowledge Silos: „Nur Wasja weiß, wie das Autorisierungsmodul funktioniert.“ Wasja kündigt. Niemand versteht den Code. Outsourced Development: Externe Auftragnehmer optimieren auf Fertigstellung, nicht auf Wartbarkeit. Die Architektur ist dahin.

Arten von Tech Debt nach dem Quadranten von Martin Fowler (Technical Debt Quadrant, martinfowler.com, 2009; die Schuldenmetapher selbst stammt von Ward Cunningham): Reckless Deliberate: „Keine Zeit für Architektur – wir gehen live.“ Prudent Deliberate: „Wir wissen, dass es nicht ideal ist, aber jetzt müssen wir starten – wir korrigieren es später.“ Reckless Inadvertent: „Wir wussten nicht, dass wir es falsch machen.“ Prudent Inadvertent: „Jetzt verstehen wir, wie es richtig gewesen wäre.“ Reckless Deliberate ist die giftigste Form. Prudent

Deliberate ist bei richtigem Management normale Geschäftspraxis.

Wann Tech Debt Start-ups tötet: Das Start-up wächst. Es stellt mehr Entwickler ein. Die Velocity sinkt (sie steigt nicht!) mit jeder Neueinstellung. Jedes neue Feature erfordert Änderungen an 10 Stellen im Code. Ein Bugfix erzeugt drei neue Bugs. Der CEO: „Wir haben 5 weitere Entwickler eingestellt, warum deployen wir langsamer?“ Das ist Tech-Debt-Vergiftung im fortgeschrittenen Stadium. Anzeichen: Die Cycle Time eines Features von der Idee bis zur Produktion liegt bei > 4 Wochen. Mehr als 30 % der Teamzeit gehen für Bugfixes und Incidents drauf. Entwickler scheuen sich, bestimmte Teile des Codes anzufassen. Das Onboarding eines neuen Entwicklers dauert 2+ Monate.

KAPITEL 2

Tech Debt objektiv messen

„Wir haben viel Tech Debt“ ist ein subjektives Gefühl. Ohne Daten kauft der CEO es Ihnen nicht ab. So messen Sie objektiv.

DORA-Metriken (DevOps Research and Assessment): vier Metriken, die mit der Business Performance korrelieren: Deployment Frequency: Wie oft deployen Sie in die Production? Elite: mehrmals pro Tag. Low: einmal pro Monat oder seltener. Lead Time for Changes: vom Commit bis zur Production. Elite: < 1 Stunde. Low: 1–6 Monate. Change Failure Rate: Anteil der Deployments, die einen Incident auslösen. Elite: 0–15 %. Low: 46–60 %. Time to Restore Service: Wie schnell Sie sich nach einem Incident erholen. Elite: < 1 Stunde. Low: 1–6 Monate. Hoher Tech Debt bedeutet schlechte DORA-Metriken. Das ist ein objektiver Beleg für den CEO.

Cycle Time: von „Start“ (der Entwickler übernimmt das Ticket) bis „Done“ (Feature in der Production). Messen Sie in Jira, Linear oder Ihrem Projektmanagement-Tool. Der Trend ist wichtiger als die absolute Zahl. Wenn die Cycle Time bei gleichem Team steigt, häuft sich Tech Debt an.

Defect Rate: Anteil der gemeldeten Bugs pro ausgeliefertem Feature. Wenn sie steigt, sinkt die Qualität. Code Coverage: Anteil des Codes, der durch automatisierte Tests abgedeckt ist. Das ist keine Goldmetrik (70 % Coverage können nutzlos sein, wenn die Tests schlecht sind). Aber das völlige Fehlen von Tests ist ein Risikoindikator.

Developer Experience Survey: Fragen Sie das Team einmal im Quartal: „Wie leicht fiel es Ihnen auf einer Skala von 1–10, in den letzten 2 Wochen Änderungen vorzunehmen?“ „Was hat Ihre Arbeit am meisten behindert?“ „Gibt es Teile des Codes, die Sie nicht anzufassen wagen?“ Qualitative und quantitative Daten zusammen ergeben das vollständige Bild.

SonarQube und CodeClimate: Werkzeuge für statische Codeanalyse. Sie zählen: Code Duplication (Verstöße gegen DRY). Complexity (Cyclomatic Complexity). Code Smells (Muster, die mit Problemen verbunden sind). Technical Debt in Stunden (Schätzung, wie viel für die Behebung nötig ist). SonarQube: kostenlos, Self-Hosted. CodeClimate: ab 50 $ im Monat für Teams.

KAPITEL 3

Priorisierung: Wann Sie Schulden zurückzahlen und wann Sie sie ignorieren

Nicht jeder Tech Debt lohnt die Rückzahlung. Die zentrale Frage: Welche Schulden bremsen das Wachstum am stärksten?

Tech-Debt-ROI-Framework: Bewerten Sie für jede Schuld: Impact auf die Velocity (1–5): Wie stark bremst diese Schuld die Delivery? Impact auf die Reliability (1–5): Wie oft verursacht diese Schuld Bugs und Incidents? Effort to Fix (1–5, wobei 5 = hoher Aufwand). ROI Score = (Velocity + Reliability) / Effort. Priorisieren Sie nach absteigendem ROI Score.

Kategorien von Tech Debt nach Dringlichkeit: P0 – Fix Now: verursacht Production-Incidents. Sicherheitslücken. Blockiert zentrale Kundenanfragen. P1 – Fix This Sprint: bremst die Entwicklung spürbar. Wird in Standups oft als Friction genannt. P2 – Schedule for Next Quarter: verringert die Lesbarkeit des Codes. Wird künftige Features bremsen. P3 – Track but Don't Fix Yet: Code Smell ohne praktischen Impact. Wird vielleicht nie angefasst.

„Boy Scout Rule“ (schrittweise Verbesserung): Jedes Mal, wenn Sie ein Stück Code anfassen, hinterlassen Sie es etwas besser, als Sie es vorgefunden haben. Refaktorieren Sie nicht alles. Einfach: Fügen Sie einen Test für die Funktion hinzu, die Sie geändert haben. Verbessern Sie den Namen einer Variablen, die Sie verwirrt hat. Entfernen Sie toten Code, auf den Sie gestoßen sind. Das ist organischer Schuldenabbau ohne spezielle Sprints.

20 % Time: Reservieren Sie in jedem Sprint 20 % der Entwicklerzeit für Tech Debt. Das ist kein „Nice to have“, sondern eine Investition in die künftige Velocity. Formulieren Sie es für den CEO so: „Wenn wir jetzt nicht 20 % investieren, brauchen wir im nächsten Jahr für dasselbe 2-mal so lange.“ Debt Sprints: einmal

im Quartal ein ganzer Sprint nur für Tech Debt. Das passt, wenn sich die Schulden kritisch angehäuft haben. Nachteil: Die Entwickler verlieren den Kontext zum Produkt.

Was Sie nie tun sollten: einen vollständigen Rewrite „von Grund auf“. Ein Big Bang Rewrite ist fast immer eine Strategie, die scheitert. Er dauert 3–5-mal länger als erwartet. Das Produkt steht still, solange der Rewrite läuft. Der „neue“ Code häuft eigene Schulden an. Die Alternative: das Strangler-Fig-Pattern. Schrittweiser Austausch in Teilen.

KAPITEL 4

Wie Sie mit CEO und Investoren über Tech Debt sprechen

Technische Schulden sind für nichttechnische Stakeholder unsichtbar. Die Aufgabe des CTO oder Tech Leads ist es, sie in der Sprache des Business sichtbar zu machen.

Übersetzen Sie technische Metriken in Business-Metriken: Statt „wir haben eine hohe Cyclomatic Complexity“ → „jedes neue Feature braucht wegen der Komplexität des Codes 2-mal so viel Zeit“. Statt „es gibt keine Unit Tests“ → „wir entdecken 40 % der Bugs in der Production statt in der Entwicklung, und das kostet uns N Tage Wiederherstellung“. Statt „Tech Debt“ → „Engineering Debt“ oder „Infrastrukturinvestition“. Worte haben Gewicht.

Business Case für Investitionen in Tech Debt: ROI-Rechnung: aktuelle Velocity: 5 Features pro Monat. Bei dem derzeitigen Schuldenwachstum sinkt die Velocity in 12 Monaten auf 2 Features pro Monat. Kosten der Entwickler: 50.000 $ im Monat. 50 % der Entwicklungsleistung gehen in Bugfixes statt in neue Features: Das sind 25.000 $ im Monat „Verlust“. Investition in 2 Monate Schuldenabbau: 100.000 $. Ergebnis: Die Velocity kehrt auf 5+ Features pro Monat zurück. ROI: Die Investition amortisiert sich in 4 Monaten.

Visualisierung von Tech Debt: ein Burn-up-Chart mit Velocity-Trend. Er zeigt den Rückgang der Velocity im Zeitverlauf. Bug Rate über die Zeit. Incident-Häufigkeit. Diese Diagramme versteht jeder CEO.

Technische Due Diligence vor einer Finanzierung: Investoren und Käufer führen eine technische Due Diligence durch. Red Flags: ein Monolith ohne Dokumentation. Keine Tests. Kein CI/CD. Sicherheitslücken (werden über SAST-Tools gefunden). Key-Person-Abhängigkeit (nur eine Person versteht den Code). Bereiten Sie sich vorab vor: grundlegendes CI/CD. Dokumentation

der wichtigsten Architekturentscheidungen. Secret Scanning im Repository. Dependency Audit (veraltete Bibliotheken mit bekannten CVEs).

Qualitätskultur: Tech Debt ist ein Symptom, nicht die Ursache. Die Ursache ist eine Kultur, die Codequalität nicht wertschätzt. So schaffen Sie eine Qualitätskultur: Code Reviews sind Pflicht (auch wenn sie langsam sind). Die Definition of Done schließt Tests ein. „Broken Window“-Policy: Sie lassen keinen kaputten Code liegen. Blameless Postmortems nach Incidents. Feiern Sie Refactoring genauso wie neue Features.

KAPITEL 5

Werkzeuge und Praktiken für das Schuldenmanagement

Die richtigen Werkzeuge machen das Management von Tech Debt systematisch statt reaktiv.

CI/CD als Grundlage: Continuous Integration: automatischer Build und automatische Tests bei jedem PR. GitHub Actions, GitLab CI, CircleCI sind für kleine Teams kostenlos. Continuous Delivery: automatisches Deployment ins Staging nach dem Merge. Continuous Deployment: automatisches Deployment in die Production. Ohne CI/CD: manuell, langsam, fehleranfällig. Mit CI/CD: schnell, zuverlässig, der menschliche Faktor verschwindet aus dem Deployment.

Automatisierte Tests: Unit Tests: testen einzelne Funktionen. Sie sind schnell. Integration Tests: testen das Zusammenspiel von Komponenten. E2E Tests: testen User Flows im Browser (Playwright, Cypress). Jagen Sie nicht 100 % Coverage hinterher. Beginnen Sie mit: Tests für geschäftskritische Flows. Tests für Stellen, die oft kaputtgehen. Tests für jeden neuen Bug (Regression Tests).

Monitoring und Observability: Datadog, New Relic oder selbst gehostet Grafana+Prometheus. Die wichtigsten Metriken: Application Performance (p95, p99 Latency). Error Rate. Performance von Datenbankabfragen. Zustand der Infrastruktur. Sentry oder Bugsnag: automatisches Error Tracking. Jede unbehandelte Exception → ein Ticket in Ihrem Tracker. PagerDuty oder OpsGenie: Incident Management und On-Call-Rotation.

Dokumentation: Architecture Decision Records (ADR): kurze Dokumente zu den wichtigsten Architekturentscheidungen. Warum haben wir PostgreSQL statt MongoDB gewählt? Warum Microservices statt Monolith? Das ist nach 12 Monaten unbezahlbar. README.md in jedem Repository: wie man es lokal

startet. Wie man die Tests ausführt. Wie man deployt. Ein Onboarding Guide für neue Entwickler.

Dependency Management: Dependabot (GitHub) oder Renovate erstellen automatisch PRs zur Aktualisierung von Abhängigkeiten. OWASP Dependency Check prüft Abhängigkeiten auf bekannte CVEs. Veraltete Abhängigkeiten sind Tech Debt plus Sicherheitsrisiko. Automatisieren Sie ihre Aktualisierung.

Wie geht es weiter?

Danke, dass Sie „Technische Schulden: Wie ein Start-up sich mit dem eigenen Code ruiniert“ gelesen haben.

Aktualisierungen dieses Buches, EPUB- und PDF-Dateien und alle Leitfäden der Reihe finden Sie auf der Buchseite: guides.mmix.biz/de/tech-debt

Maksym Stepanenko, Innova Publishing.