

LEGACY UNTER KONTROLLE

DAS DIAGNOSE-HANDBUCH FÜR LEGACY-SYSTEME

Legacy unter Kontrolle

Wenn jede Änderung zum Risiko wird und jedes Release zum Glücksspiel.

Oliver Fries

Legacy Systems Specialist · oliverfries.com

Die Diagnose, die niemand stellt

Was Legacy wirklich bedeutet und warum „das System ist stabil“ oft nur Stillstand beschreibt

DIAGNOSEPRINZIP

Freitag, 22:47 Uhr. Die Produktionslinie steht. Das MES wirft einen Fehler, den niemand eindeutig zuordnen kann. Der einzige Entwickler, der das System wirklich verstanden hat, hatte am Mittwoch seinen letzten Arbeitstag. Was geschieht jetzt?

Eine Plattform unter akutem Änderungsdruck

Dieser Zustand ist in vielen gewachsenen Systemen anzutreffen. Im Alltag scheint die Plattform weiterhin zu funktionieren. Bestellungen werden verarbeitet, Umsätze werden erzielt, die Fachbereiche arbeiten mit dem System. Gleichzeitig erzeugt jede Änderung spürbare Unsicherheit, und jedes Release wird zu einem Ereignis, das nicht mehr als kontrollierter technischer Vorgang wahrgenommen wird, sondern als potenzielle Störung des laufenden

Geschäfts. Im Team ist längst bekannt, dass die Plattform auf einem fragilen Fundament steht. Ebenso klar ist, dass die Bereinigung dieses Fundaments im laufenden Betrieb schwer zu rechtfertigen ist, solange Zeit, Budget und operative Prioritäten dagegenstehen.

Der aktuelle Zustand zeigt sich an mehreren wiederkehrenden Mustern:

- Releases werden nur noch nachts oder zu festgelegten Zeitfenstern mit geringer Systemlast durchgeführt.
- Jede Änderung am Bestell-Workflow gilt als Risiko, das nur durch umfangreiches manuelles Testing halbwegs kontrollierbar erscheint.
- Kritische Fragen landen regelmäßig bei denselben zwei Personen, weil das notwendige Kontextwissen nicht breit im Team verteilt ist.
- Änderungen an einzelnen Teilsystemen werden wiederholt verschoben, weil ihre Seiteneffekte schwer abzuschätzen sind und niemand verlässlich sagen kann, welche Abhängigkeiten dadurch berührt werden.
- Deployments sind nicht automatisiert sondern sind nur Bausteine mit denen die fertige Version manuell zusammengebaut werden muss.

Mit diesem Ausgangspunkt arbeitet dieses Buch weiter. Die folgenden Kapitel zeigen, wie aus einem diffusen Problembild belastbare Daten entstehen, wie Risiken sichtbar gemacht werden und wie sich daraus ein nachvollziehbarer Weg aus diesem Zustand ableiten lässt.

Was „Legacy“ tatsächlich beschreibt

Legacy-Systeme sind nicht einfach alte Systeme. Dieser Irrtum führt häufig dazu, dass das falsche Problem gelöst wird. Eine Anwendung kann fünfzehn Jahre alt und dennoch gut wartbar sein. Ebenso kann ein System nach drei Jahren bereits in einem Zustand sein, in dem jede Änderung unverhältnismäßig teuer, riskant oder schwer nachvollziehbar wird. Das Alter eines Systems entscheidet nicht über seinen Zustand. Entscheidend ist, wie sicher, schnell und verständlich es verändert werden kann.

Auch die klassische Definition greift nur teilweise. Michael Feathers beschreibt Legacy Code als Code ohne Tests¹. Diese Definition ist für viele technische Situationen hilfreich, weil sie den Blick auf Veränderbarkeit und Absicherung lenkt. Sie reicht jedoch nicht aus. Auch Code mit Tests kann unwartbar sein, wenn niemand mehr versteht, was diese Tests absichern, wie sie aufgebaut sind oder wie sie sinnvoll erweitert werden können. Eine Code-Coverage²

¹Feathers, Michael C.: *Working Effectively with Legacy Code*, Upper Saddle River, NJ: Prentice Hall, 2004.

²Code-Coverage berechnet sich daraus, wie viel Prozent der Codebasis über automatisierte Tests aufgerufen werden. Sie gibt keinen verlässlichen Hinweis auf Korrektheit, Aussagekraft oder Vollständigkeit der Tests.

von 90% wirkt auf dem Papier beruhigend, erzeugt aber keinen praktischen Wert, wenn jedes Release trotzdem ein Glücksspiel bleibt. In diesem Fall ist auch ein getestetes System weiterhin Legacy, weil die Tests keine verlässliche Änderungsfähigkeit herstellen.

Auch neuere Definitionen aus Wissenschaft und Industrie lösen das Problem nur teilweise. Sommerville beschreibt Legacy-Systeme als Systeme, die über lange Zeit entwickelt wurden, auf veralteten Technologien basieren und für das Unternehmen weiterhin essenziell sind³. Diese Beschreibung ist zutreffend, bleibt jedoch stärker phänomenologisch als analytisch. Sie legt nahe, dass das Alter der Technologie ein zentrales Merkmal sei. Damit entsteht leicht der Eindruck, ein System höre auf, problematisch zu sein, sobald es auf einen modernen Stack migriert wird. Wer jedoch eine schlecht strukturierte Anwendung von .NET Framework auf .NET 8 aktualisiert und danach dieselben Abhängigkeiten, dieselben Verantwortungsunklarheiten und dieselben Release-Risiken in neuer technischer Verpackung vorgefunden hat, erkennt den Unterschied: Technologiegeneration und Wartbarkeit sind zwei verschiedene Dimensionen.

Gartner nähert sich dem Begriff stärker aus betriebswirtschaftlicher Perspektive und beschreibt Legacy-Systeme als Systeme mit hohem Wartungsaufwand, geringer Agilität und akkumulierten technischen Schulden.⁴ Diese Sicht liegt näher an der operativen Praxis, bleibt aber ebenfalls symptombezogen. Hoher Wartungsaufwand ist eine Beobachtung, noch kein belastbares Kriterium. Es bleibt offen, ab wann Aufwand als hoch gilt, gegenüber welchem Referenzpunkt er gemessen wird und welche Ursache ihn tatsächlich treibt. Ohne diese Einordnung eignet sich die Definition nur begrenzt als Entscheidungsgrundlage.

Der IEEE-Standard 1219 zur Software-Wartung⁵ liefert zumindest eine belastbare Größenordnung. Wartungsaktivitäten beanspruchen typischerweise 40 bis 80 Prozent der gesamten Softwarekosten, im Mittel rund 60 Prozent. Allein das Verstehen bestehenden Codes nimmt dabei etwa 30 Prozent der Wartungszeit in Anspruch. Diese Zahlen definieren Legacy nicht, sie erklären aber, warum gewachsene Systeme im Alltag so teuer werden. Der größte Kostentreiber ist nicht das Schreiben neuen Codes, sondern das Verstehen des vorhandenen Systems. Offen bleibt jedoch, warum dieses Verstehen so aufwendig wird und ab welchem Punkt ein System in einen Zustand kippt, in dem Weiterentwicklung wirtschaftlich kaum noch vertretbar ist.

Allen genannten Definitionen ist gemeinsam, dass sie vor allem das System selbst beschreiben: seine Technologie, seinen Wartungsaufwand, seine Kosten oder den Umfang seiner Tests. Was dabei zu wenig sichtbar wird, ist die Organisation hinter dem System. Ein System ist nicht Legacy, weil es alt ist, weil es viel kostet oder weil es auf einer bestimmten Technologie

³Sommerville, Ian: *Software Engineering*. 10. Aufl. Harlow: Pearson, 2015, S. 444,ff.

⁴Gartner Glossary: *Legacy Application or System*. <https://www.gartner.com/en/documents/6435507>.

⁵IEEE Std 1219-1998: *IEEE Standard for Software Maintenance*. New York: IEEE, 1998.

basiert. Es wird Legacy in dem Moment, in dem das Verhältnis aus Änderungsrisiko, Änderungsaufwand und organisatorischem Wissensverlust die Fähigkeit des Teams übersteigt, es sicher weiterzuentwickeln.

Legacy ist damit kein absolutes Attribut des Codes, sondern ein relationaler Zustand zwischen System, Team und Anforderung. Dieser Zustand entsteht selten durch eine einzelne falsche Entscheidung. Meist ist er das Ergebnis vieler nachvollziehbarer Entscheidungen im Projektalltag. Unter Zeitdruck werden Abkürzungen gewählt, Verantwortlichkeiten bleiben implizit, Wissen geht verloren, Tests werden nachgezogen statt mitentwickelt, Dokumentation entsteht nur dort, wo gerade akuter Bedarf besteht. Jede einzelne Entscheidung ist erklärbar. In ihrer Summe können sie ein System jedoch in einen Zustand führen, in dem es zunächst übermäßig komplex, später fragil und schließlich kaum noch wirtschaftlich veränderbar wird. Overengineering⁶ ist dabei häufig kein bewusster Entwurf, sondern die Folge lokaler Optimierungen ohne gemeinsame Architekturdisziplin.

Daher muss die Definition von Legacy-Systemen erweitert werden. Sie darf nicht nur Code, Technologie und Wartungskosten betrachten, sondern muss auch die Fähigkeit einer Organisation einbeziehen, ein System unter realen Bedingungen sicher zu verstehen, zu verändern und zu betreiben. Eine tragfähige Definition lautet deshalb:

DEFINITION

Ein **Legacy-System** ist ein Softwaresystem, in dem akkumulierte Entropie durch nicht getroffene Maßnahmen, implizites Wissen und fehlende strukturelle Gegenkräfte dazu geführt hat, dass Änderungsrisiko und Änderungsaufwand die Evolutionsfähigkeit der Organisation übersteigen. Legacy ist dabei kein Attribut des Codes allein, sondern ein Zustand im Verhältnis zwischen System, Team und Geschäftsanforderungen.

Was zunächst akademisch klingt, lässt sich in wenige konkrete Prüffragen übersetzen. Sie machen sichtbar, in welchem Zustand sich ein System befindet und ob es noch kontrolliert weiterentwickelt werden kann.

1. **Änderungsrisiko:** Was passiert, wenn Modul X verändert wird? Existieren automatisierte Sicherheitsnetze? Gibt es bekannte Hotspots, bei denen Änderungen regelmäßig unerwartete Seiteneffekte erzeugen?
2. **Änderungsaufwand:** Wie lange braucht ein neuer Entwickler, um eine einfache Änderung sicher umzusetzen? Wenn die ehrliche Antwort mehrere Wochen lautet, ist das ein deutliches Signal für strukturelle Reibung.
3. **Wissenskapazität:** Wer kann das System erklären? Welche Teile sind nur einzelnen Personen bekannt? Was passiert, wenn diese Personen morgen ausfallen?

⁶**Overengineered** bezeichnet eine unnötig komplexe Lösung, deren Aufwand, Funktionen oder Struktur über den eigentlichen Bedarf hinausgehen und dadurch Wartung sowie Weiterentwicklung erschweren.

Aus diesen Fragen ergibt sich eine unbequemere Anschlussfrage. Wenn in der Softwareentwicklung seit Jahrzehnten bekannt ist, welche Muster Systeme fragil machen, warum wird dieser Zustand trotzdem immer wieder zugelassen? Weshalb wird nicht früher gegengesteuert, bevor Änderungsrisiken, Wissensverlust und strukturelle Schulden die tägliche Arbeit bestimmen?

Wie Legacy-Systeme im Projektalltag entstehen

Kein Team setzt sich zusammen und beschließt: „**Wir bauen jetzt ein Legacy-System.**“ Legacy entsteht in Situationen, die zunächst vernünftig, harmlos oder sogar notwendig wirken.

Im Beispiel des Bestellsystems könnte sich dieser Prozess rückblickend so abgespielt haben:

Die Architektur aus Sprint 1 passte in Sprint 20 längst nicht mehr, doch niemand hatte Zeit, die ursprünglichen Entscheidungen erneut zu prüfen. Der Code wurde Schicht für Schicht erweitert. Die Entwicklerin, die die kritische Kernkomponente entworfen hatte, arbeitet seit zwei Jahren in einem anderen Unternehmen. Der Architekt, der wusste, warum das Datenmodell so aussieht, wie es aussieht, verließ das Unternehmen in einem Quartal, in dem niemand Zeit für eine ordentliche Übergabe hatte. Zurück bleibt Code, der funktioniert, dessen Begründung jedoch nur noch bruchstückhaft nachvollziehbar ist.

Der schnelle Fix wurde undokumentiert und ungetestet deployt, weil das Produktivsystem stillstand. Nicht, weil dem Team das Verständnis fehlte, sondern weil in diesem Moment Stabilisierung wichtiger war als strukturelle Sauberkeit. Der Ticket-Eintrag „muss noch sauber gemacht werden“ blieb offen, und die nächste Dringlichkeit kam früher als der geplante Aufräum-Sprint. Die saubere Schichtenarchitektur funktionierte, bis das Drittsystem keinen verlässlichen API-Vertrag lieferte und die Abkürzung durch alle Schichten kurzfristig die einzige praktikable Lösung war. Wissen blieb in Köpfen statt in Dokumenten, weil Wissensmanagement im Alltag immer wieder gegen Features, Incidents und Deadlines verlor.

Tests wurden für die Features geschrieben, die im Sprint lagen. Die Integrationspfade, die über drei Quartale hinweg gewachsen waren, blieben ungesichert. Der nachträgliche Testaufwand für bereits ausgerollte Funktionen wurde als zu hoch bewertet, weil diese Teile scheinbar liefen und neue Anforderungen aus der Fachabteilung dringlicher wirkten. So entsteht ein System, das äußerlich stabil erscheint, intern aber immer weniger überprüfbare Annahmen enthält.

Das ist nicht die Geschichte von Menschen, die schlecht gearbeitet haben. Die meisten Legacy-Systeme enthalten sorgfältig durchdachte Lösungen von Entwicklern, Architekten und Product Ownern, die Verantwortung übernommen und unter realen Zwängen entschieden haben. Der Code ist das ehrliche Abbild dieser Zwänge. Er ist kein Beweis individuellen Versagens, sondern ein Hinweis darauf, dass Software unter Druck entsteht und Zeit eine der unerbittlichsten Ressourcen in der Softwareentwicklung ist.

KERNAUSSAGE

Legacy entsteht durch den normalen Lauf der Dinge: Fluktuation, Zeitdruck, verschobene Anforderungen, fehlendes Wissensmanagement. Die meisten Teams sitzen vor einem System, das ihre Geschichte abbildet, nicht ihre Fehler.

Fünf Signale für strukturelle Legacy-Risiken

Für eine erste Einordnung ist keine aufwändige Analyse erforderlich. Bestimmte Muster treten in nahezu jedem System auf, das seine sichere Veränderbarkeit teilweise verloren hat. Sie zeigen nicht nur technische Schwächen, sondern machen sichtbar, wo Vertrauen in den eigenen Entwicklungs-, Test- und Betriebsprozess bereits beschädigt ist.

□ Releases sind Stressereignisse, keine Routinen.

Vor dem Release finden zusätzliche Abstimmungen statt. Am Wochenende bleibt jemand erreichbar, falls etwas schiefgeht. Das Team befindet sich in erhöhter Alarmbereitschaft, obwohl ein Deployment ein kontrollierter technischer Vorgang sein sollte. Wenn ein Release nicht mehr als Routine behandelt werden kann, zeigt das System deutlich, dass dem eigenen Prozess nicht ausreichend vertraut wird.

□ Der Build ist langsam oder instabil.

Ein Build dauert zwanzig Minuten oder länger. Gelegentlich schlägt er fehl, ohne dass sich der Grund eindeutig erklären lässt, und beim nächsten Versuch läuft er wieder durch. Das ist kein nebensächliches technisches Detail, sondern ein Symptom für fehlende Kontrolle über Abhängigkeiten, Umgebung und Zustand.

□ Kritisches Wissen hängt an wenigen Personen.

Weniger als zwei Personen können das System eigenständig deployen oder zentrale Teile der Software sicher weiterentwickeln. Eine Kündigung löst sofort die Frage aus, welches Wissen mit dieser Person verschwindet. Das ist kein reines Kulturproblem, sondern ein Systemrisiko mit messbaren Konsequenzen, spätestens in dem Moment, in dem diese Person tatsächlich ausfällt.

□ **Auswirkungen von Änderungen lassen sich nicht verlässlich vorhersagen.**

„Ich weiß nicht, was passiert, wenn wir das anfassen“ ist einer der ehrlichsten Sätze, die über ein Legacy-System gesagt werden können. Wenn Änderungen an einer Stelle unerwartete Effekte an anderer Stelle erzeugen, fehlt nicht einem einzelnen Entwickler das Verständnis. Dem Team fehlt ein gemeinsames mentales Modell des Systems.

□ **Dokumentation spiegelt die Realität nicht mehr wider.**

Veraltete Dokumentation ist häufig gefährlicher als fehlende Dokumentation, weil sie falsches Vertrauen erzeugt. Wenn Architekturdiagramme noch aus dem Jahr des ersten Deployments stammen, seither aber mehrere größere Umbauten stattgefunden haben, beschreiben sie nicht mehr das bestehende System, sondern einen früheren Zustand.

Wenn drei oder mehr dieser Signale gleichzeitig auftreten, liegt in der Regel kein isoliertes technisches Problem mehr vor. Dann hat das System einen Zustand erreicht, in dem Veränderung nicht mehr zuverlässig aus dem normalen Entwicklungsprozess heraus beherrscht wird. Viele produktive Unternehmenssysteme zeigen mindestens zwei dieser Muster. Entscheidend ist deshalb nicht, ob Legacy vorhanden ist, sondern wie weit der Verlust an Veränderbarkeit bereits fortgeschritten ist.

Die Stabilitätsbehauptung

Die häufigste Antwort auf die Frage nach dem Zustand eines gewachsenen Systems lautet: „Das läuft seit Jahren unverändert.“ Gemeint ist damit Stabilität. Tatsächlich beschreibt dieser Satz oft nur Stillstand.

Ein System, das niemand mehr verändert, weil alle Beteiligten die Konsequenzen fürchten, ist nicht stabil, sondern eingefroren. Stabilität bedeutet, dass ein Team Vertrauen in Änderungen hat. Eingefrorene Komplexität bedeutet, dass jede Änderung als Risiko behandelt wird.

Die entscheidende Frage lautet daher nicht nur, warum diese Aussage fachlich falsch ist. Wichtiger ist, warum Teams sie über Jahre wiederholen: gegenüber dem Management, gegenüber neuen Kolleginnen und Kollegen und mitunter auch gegenüber sich selbst.

Der Grund liegt selten in Unehrlichkeit. Häufiger liegt er in einem Anreizsystem, das genau dieses Narrativ belohnt. „Stabil“ ist das Wort, das im Quartalsreport als grüner Punkt erscheint. Wer „stabil“ liefert, muss nichts erklären. Wer dagegen beschreibt, was das System tatsächlich kostet, etwa Wartungsprämien, Bus-Faktor 1 oder manuelle Releases an einem Samstagabend, eröffnet eine Budgetdiskussion. Diese Diskussion kann das Team in vielen Organisationen kaum gewinnen. Für Modernisierung gibt es selten den einfachen Business-Case auf einer Folie, weil der Nutzen nicht als neue Funktion sichtbar wird, sondern als

vermiedener zukünftiger Schaden. Gegen ein neues Feature verliert dieses Argument fast immer.

So entsteht ein rationales Gleichgewicht: Das System wird als stabil bezeichnet, der operative Schmerz wird ausgehalten, und alle warten auf den nächsten Blocker-Bug⁷. Ein Blocker-Bug ist in vielen Organisationen die einzige Währung, die im Management-Reporting für Legacy-Systeme zuverlässig zählt. Bis dahin ist Schweigen günstiger als eine Diskussion, die mit hoher Wahrscheinlichkeit verloren geht.

Genau deshalb ist „das System ist stabil“ keine neutrale Einschätzung, sondern eine Positionierung. Wenn du den Zustand eines Systems verstehen willst, musst du diesen Mechanismus zuerst erkennen und bewusst unterbrechen. Dafür braucht es Evidenz, die nicht erst nach einem Incident sichtbar wird. Genau an dieser Stelle setzen die vier Metriken und die Hotspot-Analyse der folgenden Kapitel an.

Warum typische Gegenmaßnahmen scheitern

Wenn der Schmerz groß genug wird, reagieren Organisationen häufig nach einem vertrauten Muster. Drei Antworten tauchen besonders zuverlässig auf. Sie wirken zunächst entschlossen, scheitern aber aus strukturellen Gründen.

Der Big-Bang-Rewrite. „Wir schreiben das komplett neu.“ Dieser Satz klingt befreiend, weil er einen klaren Schnitt verspricht. In der Praxis scheitert dieser Ansatz jedoch häufig nicht an einem schlechten Plan, sondern daran, dass das entscheidende Wissen nicht vollständig im Code, in Tickets oder in Dokumentation steht. Es liegt in Ausnahmen, Sonderfällen, stillschweigenden Annahmen und in den Köpfen der Menschen, die das System über Jahre begleitet haben. Das neue System übernimmt daher selten die verborgenen Lösungen des alten. Es ersetzt ein bekanntes Risiko durch ein unbekanntes und erzeugt häufig andere Fehler, ohne die betrieblich erprobten Korrekturen des gewachsenen Systems bereits zu besitzen.

Die Tool-Entscheidung. „Wir machen jetzt Microservices, Kubernetes oder KI.“ Werkzeuge lösen keine Architekturprobleme, wenn die zugrunde liegenden Abhängigkeiten, Verantwortlichkeiten und Datenflüsse nicht verstanden sind. Ein schlecht strukturiertes System, das auf Microservices verteilt wird, bleibt schlecht strukturiert. Es ist dann lediglich ein schlecht strukturiertes verteiltes System mit zusätzlicher Infrastrukturkomplexität, mehr Schnittstellen, höherem Betriebsaufwand und neuen Fehlerklassen. Ähnlich verhält es sich mit der Einführung von KI. Sie kann Analyse, Entwicklung und Dokumentation beschleunigen, aber sie beseitigt keine unklaren fachlichen Grenzen, keine inkonsistenten Datenflüsse und keine impliziten Architekturentscheidungen. Ohne Diagnose erhöht sie vor allem die Geschwindigkeit, mit der bestehende strukturelle Probleme weitergetragen werden.

⁷Damit sind Bugs gemeint, die Nutzer daran hindern, bestimmte Funktionen im System auszuführen.

Der Refactoring-Sprint. „Wir nehmen uns einen Sprint und räumen auf.“ Auch dieser Impuls ist nachvollziehbar, weil er handhabbar wirkt. Technische Schulden, die über Jahre entstanden sind, lassen sich jedoch nicht in zwei Wochen beseitigen. Ein einzelner Sprint kann Symptome glätten, Namen verbessern oder lokale Unordnung reduzieren. Er verändert aber weder die Ursachen noch die Entscheidungslogik, durch die die Schulden entstanden sind. Nach dem Sprint läuft der Alltag weiter, dieselben Prioritäten greifen erneut, und nach wenigen Monaten sieht vieles wieder aus wie zuvor.

WAS STATTDESSEN FUNKTIONIERT

Erst verstehen, dann handeln. Die Diagnose kommt vor der Maßnahme. Genau dafür bekommst du in den nächsten Kapiteln ein Verfahren an die Hand. In der Praxis überspringen viele Teams genau diesen Schritt, weil Handeln produktiver wirkt als Verstehen und weil sichtbare Aktivität leichter zu rechtfertigen ist als sorgfältige Analyse. Dieses Handbuch ist eine Anleitung für diese Diagnosephase. Was danach geschieht, hängt davon ab, was diese Diagnose sichtbar macht.