
Software-Engineering

Software-Engineering beginnt bei der Programmierung und endet mit Vorgehensweisen, die eine langfristige Wartbarkeit und Erweiterbarkeit des erstellten Systems gewährleisten. In diesem Kapitel geht es um Bereiche, in denen vielseitige (*well-rounded*) Engineers über umfassende Kenntnisse verfügen:

1. Sprachen, Plattformen und Fachgebiete
2. Debugging
3. Technische Schulden
4. Dokumentation
5. Best Practices auf das gesamte Team ausdehnen

Sprachen, Plattformen und Fachgebiete

Von einem vielseitigen Senior-Engineer wird erwartet, dass er solide Fachkenntnisse in mehreren Programmiersprachen und Plattformen hat, wie Frontend, Backend, iOS, Android, nativer Desktop, Embedded und so weiter, von denen er mindestens eine wirklich sicher beherrscht. Mehr zum »Meistern« einer Sprache finden Sie in Kapitel 9 (Teil II), »Softwareentwicklung«.

Allerdings reicht es für effektive Engineers nicht aus, nur ein paar Technologien zu beherrschen. Sie arbeiten beständig daran, ihr Wissen zu Frameworks, Sprachen und Plattformen zu erweitern.

Wenn Sie schon eine Programmiersprache kennen, ist das Lernen einer zweiten deutlich einfacher. Das liegt daran, dass sich die meisten Programmiersprachen stark ähneln – zumindest oberflächlich. Wenn Sie sich mit JavaScript auskennen, ist es nicht schwer, TypeScript zu lernen, jedenfalls am Anfang. Das Gleiche gilt, wenn Sie Swift kennen. Entsprechend bedeutet die Kenntnis von Swift, dass Sie große Teile von Java, Kotlin oder C# verstehen können, einfach indem Sie den Code lesen.

Natürlich hat jede Sprache ihre eigene Syntax sowie eigene Eigenarten, Stärken und Schwächen. Alle diese Details entdecken Sie, indem Sie die Sprache benutzen und mit anderen Sprachen vergleichen, die Sie bereits gut genug kennen.

Lernen Sie eine imperative, eine deklarative und eine funktionale Sprache in ihrer ganzen Bandbreite

Es gibt drei Arten von Sprachen:

1. **Imperativ:** Diese Programmiersprachen kommen am häufigsten vor. Hierbei erhält der Computer Befehle, die ihm Schritt für Schritt sagen, was er tun soll, beispielsweise: »Wenn X, dann mache dies. Ansonsten mache das.« C, C++, Go, Java, JavaScript, Swift, PHP, Python, Ruby, Rust, TypeScript und die meisten objektorientierten Sprachen sind imperativ.
2. Die **deklarative Programmierung** definiert die erwarteten Ergebnisse des Programms, gibt aber nicht an, wie diese erreicht werden sollen. Beispiele hierfür sind SQL, HTML und Prolog.
3. **Funktionale** Sprachen bilden eine Untermenge der deklarativen Sprachen, die sich stark genug davon unterscheiden, um eine eigene Kategorie zu rechtfertigen. Hierbei stehen die Funktionen im Vordergrund. Das heißt, Funktionen können als Argumente an andere Funktionen übergeben oder als Werte zurückgegeben werden. Beispiele hierfür sind Haskell, Lisp, Erlang, Elixir und F#. Funktionale Sprachen haben die Tendenz, immutable Zustände und reine Funktionen ohne Nebeneffekte zur Verfügung zu stellen.

Ihre erste – vielleicht sogar Ihre zweite – Programmiersprache ist sehr wahrscheinlich imperativ. Das Lernen weiterer imperativer Sprachen kann nützlich sein, allerdings kann das Lernen einer anderen Art von Sprache Ihnen helfen, beruflich voranzukommen.

Für imperative, deklarative und funktionale Sprachen ist jeweils eine eigene Denkweise nötig. Der Wechsel von einer imperativen zu einer funktionalen oder deklarativen Sprache ist nicht immer einfach, aber Sie erweitern hierdurch Ihr Verständnis und Ihren »Werkzeugkasten«.

So kommt die funktionale Programmierung oft in der imperativen Programmierung zum Einsatz, da die Befolgung eines funktionalen Modells einen immutablen Zustand garantiert. Ein gutes Beispiel ist das Reactive-Programmiersmuster¹, das Ideen aus der funktionalen Programmierung aufgreift und für Sprachen wie Java (RxJava), Swift (RxSwift), C# (Rx.NET), Scala (RxScala) und andere ein funktionaleres Muster zur Verfügung stellt.²

Sobald Sie eine Sprache aus jeder Kategorie beherrschen, werden Sie keine Schwierigkeiten mehr haben, sich weitere Sprachen anzueignen. Der Grund ist, dass es größere Unterschiede zwischen einer imperativen und einer funktionalen Sprache gibt, wie etwa Go und Elixir, als zwischen zwei Sprachen der gleichen Kategorie wie Go und Ruby oder Elixir und Haskell.

1 <https://reactivex.io/>

2 <https://reactivex.io/languages.html>

Machen Sie sich mit verschiedenen Plattformen vertraut

Üblicherweise spezialisiert sich ein Software-Engineer auf Plattformen wie:

- Backend
- Frontend
- Mobile
- Embedded-Plattformen

Wenn Ihr Team ein neues Feature erstellt oder ein Problem löst, stehen die Chancen gut, dass dies plattformübergreifend stattfindet. So bedeutet die Bereitstellung eines neuen Zahlungsablaufs mit Sicherheit Änderungen am Backend, am Frontend und sogar in mobilen Apps. Um eine solche App zu debuggen, müssen Sie herausfinden, ob der Ursprung des Problems in der mobilen Geschäftslogik, dem Backend oder vielleicht an der Stelle zu finden ist, an der sich Backend-APIs mit der mobilen Geschäftslogik überschneiden, um die API-Antwort zu parsen.

Sollten Sie keine Ahnung haben, was in den benachbarten Technologie-Stacks vor sich geht, wird es schwierig, komplexere Full-Stack-Probleme zu debuggen und Projekte zu leiten, die Full-Stack-Features erstellen und bereitstellen.

Steigern Sie Ihre Full-Stack-Fähigkeiten

Full-Stack-Engineering wird immer mehr zu einer grundsätzlichen Erwartung an Senior-Engineers in der Technologiebranche. Der Grund ist, dass Produktverantwortlichen und geschäftlichen Entscheidungsträgern die Unterscheidung zwischen Embedded, Backend und Frontend/Web oft einfach egal ist. Aus ihrer Sicht ist diese Unterscheidung eine Engineering-Entscheidung.

Ein vielseitiger Engineer kann sich jedem Problem stellen und herausfinden, wie es über die verschiedenen Plattformen verteilt ist. Hierfür ist Fachwissen in Ihrem Bereich nötig, aber auch genügend Kompetenz auf anderen Gebieten.

Wie erarbeiten Sie sich dieses Wissen? Hier ein paar mögliche Ansätze:

- **Verschaffen Sie sich Zugang zu den Codebasen für andere Plattformen.** Wenn Ihr Team beispielsweise für die Mobile-, Web- und Backend-Codebasen zuständig ist, sollten Sie versuchen, Zugang zu den Codebasen zu bekommen, die nicht Ihre »primäre« Plattform betreffen. Werfen Sie als Backend-Engineer einen Blick auf die Web- und Mobile-Codebasen und richten Sie auf Ihrem Rechner eine Möglichkeit ein, sie lokal zu kompilieren, zu testen und bereitzustellen.
- **Lesen Sie Reviews für Code von anderen Plattformen.** Verfolgen Sie die Code-Reviews mit, indem Sie auch diejenigen für andere Plattformen lesen oder indem Sie darum bitten, als nicht blockierender Reviewer dabei zu sein. Es ist viel einfacher, Code zu lesen, als ihn zu schreiben. Dabei geht es bei den meisten Codeänderungen um die Geschäftslogik. Sie sollten also kaum Schwierigkeiten haben, den Zweck der Änderungen zu verstehen. Vielleicht fallen Ihnen sogar Probleme in der Geschäftslogik oder fehlende Testfälle auf!

- **Melden Sie sich für kleinere Aufgaben auf anderen Plattformen freiwillig.** Die beste Möglichkeit, sich mit anderen Plattformen vertraut zu machen, besteht darin, mit ihnen zu arbeiten. Suchen Sie sich eine nicht dringende, unwichtige Aufgabe, die Sie in Ihrem eigenen Tempo erledigen können. Bitten Sie bei anderen Engineers im Team um Rat.
- **Bilden Sie ein Tandem mit einem Engineer, der auf einem anderen Stack arbeitet.** Um einen neuen Stack zu lernen, kann die paarweise Programmierung ein effizientes Mittel sein. Bitten Sie jemanden mit mehr Erfahrung in dem Stack, den Sie lernen möchten, sich mit Ihnen zusammenzutun. Das wird Ihren Lernprozess beschleunigen. Sie könnten beginnen, indem Sie bei der Person »Mäuschen spielen«. Sobald Sie mehr Praxiserfahrung haben, fragen Sie, ob Sie die Session leiten können. Bitten Sie auch die andere Person um Feedback zu Ihrer Vorgehensweise.
- **Starten Sie einen »Austauschmonat«, in dem Sie an einer anderen Plattform arbeiten.** Noch besser kann man intensiv lernen, indem man die Plattform für eine bestimmte Zeit wechselt. Das kann für ein paar Wochen oder auch Monate sein. Der Nachteil liegt darin, dass Ihr Arbeitstempo kurzfristig sinkt, weil Sie die Grundlagen der neuen Plattform erst lernen müssen. Mittelfristig wird Ihre Geschwindigkeit aber wieder zunehmen, da Sie über das nötige Fachwissen und die Werkzeuge verfügen, mögliche Hindernisse selbst zu überwinden.

KI-Assistenten können den Wechsel beschleunigen

KI-Assistenten können beim Wechsel zwischen Sprachen und Plattformen hilfreich sein. Mit Werkzeugen wie GitHub Copilot, ChatGPT, Sourcegraph Cody, Google Bard und anderen KI-Helfern ist es deutlich einfacher, neue Programmiersprachen zu lernen oder die Plattform zu wechseln. Diese Assistenten können folgende Aufgaben übernehmen:

- Ein Stück Code von einer Sprache in eine andere übersetzen.
- Eine Zusammenfassung davon geben, wie Funktionen und Variablen in einer Sprache deklariert werden.
- Die Unterschiede zwischen zwei Sprachen zusammenfassen.

Vergessen Sie aber nicht, dass viele KI-Assistenten unter Halluzinationen leiden: Manchmal erfinden sie Dinge, die nicht real sind. Sie müssen die Ausgaben also unbedingt überprüfen. Um sich aber mit einer Sprache oder Plattform vertraut zu machen, können KI-Assistenten hilfreich sein und den Lernprozess beschleunigen.

Debugging

Der Unterschied zwischen einem Senior- und einem anderen Engineer zeigt sich deutlich beim Debugging und dem Aufspüren schwieriger Fehler. Erfahrenere Engineers sind beim Debugging meist schneller und ebenso beim Finden der Hauptursache von schwierigeren Problemen – es scheint ihnen geradezu leichtzufallen. Sie

haben auch ein besseres Gespür für die Quelle eines Problems und dafür, wo man mit dem Debugging anfangen sollte, um die Schwierigkeit zu beseitigen. Wie machen sie das?

Ein Teil ist Übung und Fachwissen. Je länger Sie Code schreiben, desto öfter werden Ihnen unerwartete Grenzfälle und Bugs begegnen. Mit der Zeit schaffen Sie sich eine »Werkzeugkiste« möglicher Hauptursachen von Problemen.

Das Gleiche gilt für Ihre Debugging-Werkzeuge. In Kapitel 9 (Teil II), »Softwareentwicklung«, gehen wir darauf ein, wie man seine Debugging-Fähigkeiten verbessern kann:

- Machen Sie sich mit Ihren Debugging-Werkzeugen vertraut.
- Lernen Sie, auch ohne Werkzeuge zu debuggen.
- Machen Sie sich mit fortgeschrittenen Debugging-Werkzeugen vertraut.

Die Fähigkeit, effizient zu debuggen, ist offenbar ein Unterscheidungsmerkmal zwischen erfahrenen und weniger erfahrenen Engineers. Unten finden Sie eine Reihe von Ansätzen, im Debugging besser zu werden.

Wissen, welche Dashboards und Logging-Systeme man sich ansehen muss

Besonders in großen Unternehmen müssen Sie wissen, wo die Produktions-Logs und die Produktionsmetriken zu finden sind, um Probleme im produktiven Betrieb debuggen zu können. Und selbst dann kann es Monate dauern, bis ein Senior-Engineer die Wichtigkeit, diese Orte zu kennen, wirklich zu schätzen weiß.

Das Auffinden der richtigen Dashboards und Logging-Portale kann in Unternehmen besonders schwierig sein, in denen Teams für eine große Zahl von Diensten zuständig sind, die jeweils ihre eigenen Logging-Mechanismen verwenden, Informationen in verschiedenen Systemen protokollieren und verschiedene Log-Formate einsetzen.

Wenn Sie neu in einem Unternehmen anfangen, sollten Sie es sich zur Priorität machen, herauszufinden, wo die Produktions-Logs gespeichert werden und wo die Dashboards für den Systemstatus zu finden sind. Diese könnten in einem System wie Datadog, Sentry, Splunk, New Relic oder Sumo Logic zu Hause sein. Sie könnten aber auch in hauseigenen Systemen liegen, die auf Basis von Prometheus, Clickhouse, Grafana speziell für das Unternehmen erstellt wurden. Oder sie befinden sich an mehreren verschiedenen Orten. Finden Sie heraus, wo die Logs und Dashboards sind, wie man darauf zugreift und wie man sie abfragt. Machen Sie das für Systeme, für die Ihr Team zuständig ist, aber auch für verwandte Systeme, mit denen Sie interagieren.

Erleichtern Sie anderen das Debugging

Als Senior-Engineer sollten Sie wissen, welche Logging-Systeme und Dashboards wichtig sind. Wenn diese sich an unterschiedlichen Orten befinden, können Sie das

vielleicht ändern, indem Sie sie an einem Ort zusammenfassen und so ihre Benutzung erleichtern.

Mehr zu diesem Thema finden Sie in Kapitel 24 (Teil V), »Zuverlässige Softwaresysteme«.

Die Codebasis verstehen

Kleinere Codebasen sollten Sie in- und auswendig kennen. Bei der Arbeit mit einer überschaubaren Codebasis – normalerweise nicht mehr als 100.000 Zeilen, die von maximal 20 Personen geschrieben wurden – gibt es keine Ausrede, nicht *präzise* zu wissen, wo sich alles befindet. Sehen Sie sich die Struktur der Codebasis an, lesen Sie eine Menge Code und machen Sie sich klar, wie die einzelnen Codebestandteile miteinander verbunden sind.

Erstellen Sie sich auf Basis des gelesenen Codes Architekturdiagramme und bitten Sie Teamkollegen um Bestätigung, dass Sie die Struktur richtig verstehen. Steigen Sie so tief ein, dass Sie wissen, welcher Teil des Codes für welche Funktionalität zuständig ist.

Bei großen Codebasen ist es sinnvoll, ihre Struktur zu verstehen und wie die relevanten Teile zu finden sind. In größeren Unternehmen haben die Codebasen häufig einen Umfang von deutlich über einer Million Zeilen, die von Hunderten von Engineers geschrieben wurden. Es ist kaum möglich, eine Codebasis dieser Größe in allen Einzelheiten zu kennen. Dennoch ist es angemessen, auf ein *breites* Verständnis hinzuarbeiten, um bei den Teilen, an denen Sie arbeiten müssen, in die Tiefe gehen zu können.

In Firmen, die Monorepos einsetzen, sollten Sie sich einen Überblick über die Teile verschaffen, für die Sie verantwortlich sind. Wie sind die verschiedenen Teile des Systems aufgebaut? Wie werden die Tests durchgeführt?

In Unternehmen mit eigenständigen (*standalone*) Repositories sollten Sie sich um Zugang bemühen. Versuchen Sie, zu verstehen, wie die Systeme grundsätzlich funktionieren. Es ist eine gute Übung, einige von ihnen auszuchecken, zu kompilieren, Tests durchzuführen und den Dienst oder das Feature lokal auszuführen.

Finden Sie heraus, wie Sie die gesamte Codebasis durchsuchen können, und lernen Sie nützliche Abkürzungen. In den meisten Firmen gibt es eine Art »globale Codesuche«. Das könnte eine hausinterne Lösung sein, oder die Lösung könnte durch einen externen Dienst wie GitHubs Codesuche oder Sourcegraph bereitgestellt werden. Lernen Sie, das Werkzeug für die globale Codesuche zu anzuwenden, und finden Sie heraus, welche Features es unterstützt. Können Sie beispielsweise in einem bestimmten Ordner suchen? Wie können Sie die Testfälle finden? Wie steht es mit einer Suche nur in der Codebasis, für die Ihr Team zuständig ist?

Selbst in großen Unternehmen, in denen die Engineers auf die meisten Teile der Codebasis zugreifen können, ist vielleicht nicht alles freigegeben. Das geschieht meistens aus Gründen der Compliance, aufgrund gesetzlicher Vorschriften oder we-

gen der Vertraulichkeit. Meistens sollte dies für Ihre tägliche Arbeit aber keinen Unterschied bedeuten. Aber wenn Sie dadurch langsamer vorankommen, bitten Sie um Zugang.

Eignen Sie sich genug Wissen zur Infrastruktur an

Manche Probleme im Produktivbetrieb haben ihren Ursprung in der Infrastruktur. Finden Sie heraus, welche Dienste für den produktiven Betrieb bereitgestellt wurden, wie Geheimnisse gespeichert und wie Zertifikate eingerichtet werden. Sehen Sie nach, wie die Infrastruktur verwaltet wird und wo die Konfigurationen gespeichert sind.

Wenn es in Ihrem Unternehmen ein eigenes Infrastrukturteam gibt, kann es verführerisch sein, den Lernprozess zu überspringen und sich direkt an das Team zu wenden, wenn Sie ein Problem mit der Infrastruktur vermuten. Auf lange Sicht bremsst dieser Ansatz Sie aber aus. Davon abgesehen ist es nicht nur interessant, zu lernen, wie die Infrastruktur im Inneren funktioniert, es ist auch eine Grundvoraussetzung, um ein vielseitiger Engineer zu sein.

Aus Störungen lernen

Debuggen Sie Störungen, sobald sie auftreten, und lesen Sie alte Nachuntersuchungen (*Postmortems*) hierzu. Eine großartige Möglichkeit, Ihre Debugging-Fähigkeiten zu verbessern, besteht darin, dann zu debuggen, wenn es *wirklich* darauf ankommt – wenn die Probleme auftreten. Gibt es in Ihrem Team eine Störung, bieten Sie an, bei der Suche nach der Ursache zu helfen, um sie in Zukunft zu vermeiden.

Um Störungen zu debuggen, müssen Sie lernen, auf Produktions-Logs zuzugreifen, sie zu analysieren, den für die Störung verantwortlichen Code zu finden, Änderungen daran vorzunehmen und die Änderungen vor dem Einspielen ins Produktivsystem zu überprüfen.

Um Ihre Debugging-Fähigkeiten zu verbessern, müssen Sie nicht darauf warten, dass der nächste Bug Ihr System trifft. Lesen Sie die Nachuntersuchungen früherer Störungen, sofern Ihr Unternehmen diese veröffentlicht. Versuchen Sie, beim Lesen zu »debuggen«, indem Sie die Logs ermitteln, in denen die Probleme protokolliert sind, und indem Sie sich den Code hinter der Störung ansehen. Die Recherche von vergangenen Störungen stellt eine hervorragende Möglichkeit dar, neue Dashboards und Systeme zu erlernen, die Sie noch nicht so gut kennen, und neue Möglichkeiten zu entdecken, Störungen zu vermeiden.

Technische Schulden

»Technische Schulden« (*Tech Debt*) ist ein Begriff, den Software-Engineers nur zu gut kennen. Er beschreibt die steigenden Kosten, die die Softwareentwicklung von Systemen im Laufe der Zeit verursacht. Technische Schulden sammeln sich an, wenn der Code umfangreicher wird und die Komplexität steigt.

Die Merkmale technischer Schulden haben Ähnlichkeit mit der Aufnahme eines Kredits. Geschickt eingesetzt, können Schulden den Fortschritt voranbringen. Wenn man sie dagegen unsachgemäß verwendet, kann es teuer werden, sie zu bedienen. Bankrott durch technische Schulden ist real. Dieser Punkt ist erreicht, wenn es kostengünstiger ist, die gesamte Codebasis zu löschen und neu zu schreiben, als sie weiter zu pflegen und zu reparieren.

Hier ein paar typische Phasen der Beziehung zwischen technischen Schulden und Software-Engineers bzw. Engineering-Abteilungen:

- Wenn Software-Engineers mit der Erstellung einer Software beginnen, gibt es eine – meist kurze – Zeit, in der ihnen nicht klar ist, dass technische Schulden existieren.
- Leugnen. Als Engineer am Anfang seiner Karriere könnten Sie leicht glauben, dass Sie niemals die Quelle für technische Schulden sein können oder dass technische Schulden, die Sie bemerken, schon nicht so schlimm sind und es also nicht wehtut, sie zu ignorieren.
- Akzeptanz. Schon bald geht den meisten Engineers aber auf, dass das Schreiben von Code und technische Schulden oft Hand in Hand gehen, besonders wenn nicht genug Zeit ist, die Dinge »ordentlich« zu erledigen. Je besser die Engineering-Kultur eines Unternehmens, desto besser sind Engineers und ihre Führungskräfte darin, technische Schulden anzuerkennen.

Auch wenn sich technische Schulden kaum vermeiden lassen, kann man ihr Wachstum deutlich verlangsamen, wenn man sich im Engineering an vernünftige Regeln für die Wartbarkeit und die Veränderungen des Codes hält. Hierzu gehören das Schreiben von lesbarem Code, Testing, Code-Reviews, CI/CD, Dokumentation, gesunde Architekturentscheidungen und mehr.

Technische Schulden abbauen

Kleinere technische Schulden sollten Sie möglichst sofort »abbezahlen«. Befolgen Sie hierbei die Pfadfinderregel, einen Ort – hier die Codebasis anstelle eines Lagerplatzes – sauberer zu verlassen, als Sie ihn vorgefunden haben.

Bei größeren technischen Schulden machen Sie eine Bestandsaufnahme und beziffern die Auswirkungen und den für die Beseitigung nötigen Aufwand. Sind die technischen Schulden besonders hoch, werden Sie sie nicht alle tilgen können.

Ohne Daten zu höheren technischen Schulden können Sie nur schlecht über das nötige Vorgehen entscheiden. Geht es um Probleme, deren Lösung Wochen oder Monate in Anspruch nehmen würde, muss ein Team Prioritäten setzen. Welchen Wert hat die Begleichung technischer Schulden im Vergleich mit Arbeiten, die sich auf das Geschäft auswirken?

Um technische Schulden zielgerichtet zu beseitigen, sollten Sie Projekte mit klar definierten Ergebnissen vorschlagen. Gibt es technische Schulden, die geradezu darum betteln, abgebaut zu werden, weil ihre Auswirkungen so offensichtlich sind?

Zuverlässigkeit, Kostenersparnis, schnellere Entwicklungszyklen und weniger Bugs sind gängige Argumente, die Menschen anführen, um den Abbau größerer technischer Schulden zu rechtfertigen oder Migrationsprojekte umzusetzen.

Stellen Sie sich duplizierte Logik als eine Art von technischen Schulden vor, bei der Codeteile kopiert und an verschiedenen Stellen wieder eingefügt wurden. Welche Auswirkungen hätte es, die duplizierten Teile in eine gemeinsam genutzte Bibliothek auszulagern, und was würde das kosten? Die Auswirkungen sind deutlich größer, wenn es um eine häufig genutzte Codebasis geht. Andererseits kann eine veraltete Codebasis, die ohnehin bald außer Dienst gestellt wird, einen großen Aufwand für einen kleinen Gewinn bedeuten.

Oder nehmen Sie das Beispiel langsamer Build-Zeiten. Wird ein Build von vielen Engineers sehr oft durchgeführt, können die Auswirkungen abgezahlter technischer Schulden ziemlich groß sein. Hierfür müssen Sie nur die pro Build verschwendete Zeit mit der Anzahl der täglichen Build-Vorgänge pro Engineer und dies wiederum mit der Anzahl der beteiligten Engineers multiplizieren.

Koppeln Sie den Abbau technischer Schulden mit sehr einflussreichen Projekten.

Hier kommt das Geheimnis, als Engineer angesehen zu werden, der produktiv ist und gleichzeitig technische Schulden abbaut: Nicht um Erlaubnis fragen! Anstatt sich vor der Entfernung technischer Schulden rückzuversichern, bauen produktive Engineers die Schulden einfach als Teil eines besonders einflussreichen Projekts ab.

Die Projekte mit der höchsten Priorität sind meist ehrgeizig und haben eine hohe Sichtbarkeit. Um sie umzusetzen, müssen immer wieder Systeme mit hohen technischen Schulden verändert werden. Wenn Sie ein solches System anpassen müssen, verlangsamt das logischerweise Ihre Arbeit. Daher ist es sinnvoll, sich dafür einzusetzen, die technischen Schulden direkt mit abzubauen, wenn Sie ohnehin schon an dem System arbeiten.

Die Anhäufung technischer Schulden verringern

Anstatt Zeit und Energie mit dem Abbau technischer Schulden zu verschwenden, ist es sinnvoller, mehr Zeit und Energie dafür aufzuwenden, dass technische Schulden langsamer oder gar nicht erst entstehen. Hier ein paar Möglichkeiten:

- **Schreiben Sie gut lesbaren Code.** Gut lesbarer Code ist leichter zu verstehen. Und wenn verständlicher Code das Ziel ist, ist es weniger wahrscheinlich, dass Engineers, die später damit arbeiten, »Hacks« einbauen müssen, um Teile zu umgehen, die sie nicht verstehen.
- **Nehmen Sie sich Zeit, Ihren Code aufzuräumen.** Ein paar technischen Schulden schleichen sich in den Code, weil redundante Dinge nicht aufgeräumt wurden. Entfernen Sie abgeschlossene Experimente, unbenutzte Feature-Flags, nicht länger ausgeführte Codepfade und nie fertiggestellte Codeerweiterungen.
- **Erstellen Sie Systeme, die erweiterbar sind.** Oft treten technische Schulden auf, wenn ein System erweitert werden muss – etwa bei einem neuen Anwendungs-

fall – und die verfügbare Zeit sehr knapp ist. Versuchen Sie bei der Erstellung eines Systems, zukünftige Anwendungsfälle vorzusehen, und schaffen Sie Möglichkeiten, Ihre Lösung zu erweitern. Hierbei können bekannte Entwurfsmuster wie Strategy, Decorator oder das Factory-Muster helfen. Das Gleiche gilt für Dinge wie Konfigurationsdateien, um ein bestimmtes Verhalten zu definieren.

- **Nehmen Sie technische Schulden wahr und fragen Sie nicht um Erlaubnis, kleinere Teile zu entfernen.** Bei der Arbeit an der Codebasis werden Ihnen Dinge auffallen, die Sie ausbremsen. Machen Sie sich eine Notiz und arbeiten Sie daran, wenn Sie Zeit haben.

»Gerade genug« technische Schulden

Gibt es so etwas wie »zu wenige technische Schulden«? Wenn Sie genug technische Schulden abgezahlt haben, werden Sie irgendwann feststellen, dass es so etwas wirklich gibt. Der Name hierfür lautet »verfrühte Optimierung« und kann Teams oder, wenn es hart auf hart kommt, auch ganze Unternehmen ausbremsen.

Nehmen Sie beispielsweise ein Start-up. Bei Markteintritt sind Geschwindigkeit und schnelle Iterationen besonders wichtig, um zu überleben und zu gewinnen. Machen Sie sich in solchen Zeiten Gedanken über saubere APIs und schöne Datenmodelle, oder hauen Sie einfach alles in eine unstrukturierte JSON-Datei, die von jedem Entwickler verändert werden kann? Die Start-ups, in denen ich gearbeitet habe und die erfolgreich waren, verfolgten alle einen Ansatz, der in der Anfangszeit mit hohen technischen Schulden verbunden war.

Die Mitfahr-App Uber war eines dieser Start-ups. Als ich dort anfang, gab es eine Menge übrig gebliebener früher technischer Schulden, und Teile der Codebasis wurden von Folgen kurzfristiger Entscheidungen heimgesucht. Aber die technischen Schulden hatten einen Sinn, denn Sie erlaubten Uber ein zügiges Vorankommen, als es darum ging, das Produkt möglichst schnell an den Markt anzupassen. Danach investierte Uber in die Bereinigung des Codes.

Technische Schulden haben ihre Berechtigung bei Projekten in der Frühphase, bei Wegwerfprototypen, bei Produkten, die nur Mindestanforderungen erfüllen müssen (*Minimum Viable Products*, MVP), und wenn es darum geht, das Geschäftsmodell eines Start-ups zu überprüfen. Technische Schulden können abgebaut werden, indem man, wie Uber, später Zeit und Entwickler dafür bereitstellt. Die meisten schnell wachsenden Start-ups in der Spätphase sind normalerweise damit beschäftigt, nebenbei technische Schulden abzubauen, denn in dieser späteren Phase verfügen sie über mehr Personal – und Zeit! –, um sich diesem Problem zu widmen. Wenn ein Team, das für ein ausgereiftes Produkt zuständig ist, die technischen Schulden nicht immer wieder reduziert, ist wahrscheinlich etwas nicht in Ordnung.

Pragmatische Engineers sehen technische Schulden nicht zwingend als etwas Schlechtes. Sie betrachten sie als Kompromiss zwischen Geschwindigkeit und Qua-

lität, als Eigenheit eines Systems. Sie stellen technische Schulden in Zusammenhang mit den Projektzielen und versuchen nicht, mehr abzuzahlen als nötig. Dabei behalten sie die Schulden im Auge und schreiten ein, um sie abzubauen, bevor sie zu groß werden – wenn nötig auch kreativ.

Dokumentation

Der Begriff *Dokumentation* kann sich auf mehrere Bereiche beziehen. Dabei sind nicht alle Arten für jedes Projekt von Bedeutung. Hier eine Liste der häufigsten von Engineers erstellten Dokumentationstypen und wann es sinnvoll ist, sie zu schreiben und zu pflegen.

Designdokumente/RFCs (Request for Comments)

Diese Form der Dokumentation bietet eine Art Vogelperspektive auf das System. Die Dokumente könnten Diagramme enthalten und die Überlegungen hinter technologischen Entscheidungen und Kompromissen erläutern.

Designdokumente sind am wertvollsten, wenn sie vor dem Beginn der eigentlichen Programmierung erstellt und weitergegeben werden, um Feedback zu erhalten. Schließlich sollen diese Dokumente dabei helfen, das Projekt zu erstellen und Missverständnisse möglichst früh aufzuspüren.

Testpläne, Roll-out-Pläne und Migrationspläne

Nach Abschluss der Planungsphase für ein Projekt gibt es Dokumente, die dabei helfen, eine hohe Qualität des Systems sicherzustellen, wie beispielsweise:

- **Testplan:** Wie wird das System getestet? Welche Grenzfälle müssen unbedingt berücksichtigt werden? Werden manuelle Einmaltests oder automatisierte Tests verwendet? Falls es eine Liste mit manuellen Tests, oft *Sanity Tests* (Plausibilitätsprüfungen) genannt, gibt, muss diese aktuell gehalten werden.
- **Roll-out-Plan:** Wie soll das System in Betrieb genommen werden? Welche Feature-Flags sollen dabei genutzt werden, wie wird das Experiment durchgeführt, und für welche Regionen oder Nutzergruppen wird es bereitgestellt? Für dieses Dokument ist normalerweise die Mitarbeit der Produktverantwortlichen und Data Scientists nötig.
- **Migrationsplan:** Wie soll die Migration von einem System auf ein anderes ablaufen? Wie wird überprüft, ob das neue System korrekt funktioniert, bevor Traffic dorthin geleitet wird? In welche Phasen ist die Migration unterteilt?

Das Schreiben dieser Dokumente zum Projektstart kann sehr wertvoll sein. Weniger sinnvoll ist es, sie weiterzupflegen, sobald ein Projekt bereitgestellt oder eine Migration abgeschlossen ist.

Dokumentation von Interfaces und Integrationen

API- oder Interface-Dokumentation. Wird eine API oder ein Interface für die Nutzung durch andere Engineers entwickelt, erklärt diese Dokumentation Dinge wie:

- Wie wird die API genutzt?
- Die Liste der Endpunkte und der erwarteten Ein- und Ausgaben für diese.
- Fehlercodes und zurückgegebene Meldungen.
- Codebeispiel für die Verwendung der API.

Wenn Teile des API-Codes geändert werden, sollte auch die Dokumentation entsprechend aktualisiert werden. Suchen Sie nach Möglichkeiten, dies zu automatisieren, zum Beispiel indem die Dokumentation anhand von Kommentaren erzeugt wird.

Die Dokumentation von SDKs (*Software Development Kits*), von Integrationen oder Plug-ins hat Ähnlichkeit mit der API-Dokumentation. Diese Leitfäden helfen anderen Teams, Ihr SDK, Ihre Integration oder Plug-ins besser zu nutzen.

Versionshinweise (Release Notes)

Einige Engineering-Teams stellen immer noch Versionshinweise für jedes Major Release zusammen – obwohl diese Praxis in großen Teilen der Branche nicht mehr als modern gilt. Normalerweise ist das Schreiben von Versionshinweisen recht einfach und dauert auch nicht lange. Geben Sie einfach an, auf welche Kunden sich Ihre Arbeit auswirkt, oder fassen Sie zusammen, welche Auswirkungen die ausgelieferten Features haben. Dies kann die Aktualisierung von API-, SDK- und Integrationsdokumenten deutlich erleichtern.

Versionshinweise sind eine gute Möglichkeit, über die Arbeit nachzudenken. Sie sind außerdem eine hervorragende Informationsquelle zur Weitergabe an andere Beteiligte, etwa andere Engineering-, aber auch nicht technische Teams.

Dokumentation für Neueinsteiger

Wie können neue Engineers die Funktionsweise eines Systems erlernen, für das Ihr Team zuständig ist? Die Antwort liegt in einer guten Onboarding-Dokumentation. Hierzu gehören:

- Ein grundsätzlicher Überblick darüber, wie sich das System ins »große Ganze« einfügt, seine Zuständigkeiten, mit welchen anderen Systemen es interagiert und so weiter.
- Wie das System modifiziert werden kann: Überprüfen des Codes und Vornehmen von Änderungen.
- Wie Änderungen durch die Ausführung von Tests und der Inspektion bestimmter Systembestandteile getestet und überprüft werden können.
- Wie Dinge für den produktiven Betrieb bereitgestellt werden, zum Beispiel automatisch anhand von CI/CD.

- Wie das Produktivsystem überwacht wird.
- Wie Warnmeldungen funktionieren und wie sie bei Bedarf angepasst werden können.
- Tipps zum Debuggen des Systems, falls es im Produktivbetrieb zu Problemen kommt.

Eine Onboarding- oder Einarbeitungsdokumentation ist besonders für Neueinsteiger eine wertvolle Ressource, manchmal aber auch für andere Teammitglieder! Leider gibt es nur wenig Anreize, sie zu erstellen und zu pflegen, besonders wenn kein neuer Engineer dem Team beitrifft.

Ich empfehle, in die Erstellung dieser Dokumentation zu investieren, wenn ein neuer Kollege dazukommt, und sie für jeden Neueinsteiger zu aktualisieren. Warum bitten Sie den neuen Mitarbeiter nicht gegen Ende seiner Einarbeitungsphase, die Dokumentation selbst anzupassen, falsche Details zu korrigieren und fehlende Teile zu ergänzen?

Ein »Teamhandbuch«

Wie arbeitet das Team? Wie sind die Aufgaben des Teams in die geschäftlichen Ziele integriert? Ein Teamhandbuch beantwortet diese Fragen und behandelt dabei Themen wie:

- Wie die Arbeit priorisiert wird.
- Wie die Teammitglieder Probleme ansprechen und die nächste Aufgabe in Angriff nehmen.
- Vom Team verwendete Prozesse.
- Wertvorstellungen des Teams.

Hat Ihr Team noch kein solches Handbuch, sprechen Sie mit der Teamleitung oder dem Manager und schlagen vor, eins zu erstellen. Wenn Ihr Team Ihnen genug vertraut, können Sie auch einfach mit dem Schreiben beginnen und das Team um Beiträge bitten.

Betriebshandbuch für Engineers in Rufbereitschaft

Welche Schritte müssen unternommen werden, wenn der Engineer mit Rufbereitschaft eine Warnmeldung vom System erhält? Wo befinden sich die wichtigen Dashboards und Logs? Welche Abhängigkeiten hat das System von anderen Systemen? Ein gutes Betriebshandbuch beantwortet diese Fragen.

Benutzerhandbücher und Leitfäden

Wenn die Software von Endusern verwendet wird, brauchen diese Gebrauchsanweisungen, die ihre Funktionsweise erklären. Sie werden nicht unbedingt verschiedene Programmiersprachen brauchen, um das System zu beschreiben, Screenshots und visuelle Erläuterungen können dagegen sehr hilfreich sein!

Selbst wenn diese Anleitungen schon existieren, sollten Sie bei der Anpassung von Systemteilen, die das Verhalten für Benutzer verändern, darauf hinweisen. Nach Möglichkeit sollten Sie die Benutzerhandbücher persönlich aktualisieren. Schließlich haben Sie auch die Änderungen vorgenommen und verstehen sie daher am besten!

Dokumentation als Aktivität mit großer Tragweite

Zu Beginn kann das Schreiben von Dokumentation recht zeitaufwendig sein. Dennoch hat es große Auswirkungen. Und ist ein Dokument einmal geschrieben, macht es weniger Arbeit, es aktuell zu halten. Gute Dokumentation, zum Beispiel für die Einarbeitung, kann beim Abbau technischer Schulden sehr hilfreich sein, indem sie erklärt, wie das System funktioniert.

Best Practices auf das gesamte Team ausdehnen

Als Senior-Engineer sollten Sie danach streben, hervorragende Arbeit zu leisten, und Ihr Team dabei unterstützen, dasselbe zu tun. Eine offensichtliche Methode ist die Einführung von Best Practices oder »bestmöglichen Vorgehensweisen«.

Was genau ist eine *Best Practice*? Hierbei handelt es sich um einen bewährten technischen Ansatz, der in Ihrer Arbeitsumgebung sehr gut funktioniert. Einzelpersonen und Teams, die diesen Best Practices folgen, werden schneller mit der Arbeit fertig, machen dabei weniger Fehler und erstellen leichter zu pflegenden Code.

Allerdings kann der Begriff auch missverstanden werden, denn jedes Team und jedes Unternehmen hat seine eigenen Fachkenntnisse und Dynamiken. Eine Vorgehensweise kann für ein Team genau richtig sein, während sie für ein anderes Team überhaupt nicht funktioniert.

»**Software-Engineering Practices**« (**Vorgehensweisen im Software-Engineering**) ist ein Begriff, den ich als Alternative zu »Best Practices« vorschlage. Bewährte und bewiesenermaßen praxistaugliche Vorgehensweisen finden in vielen Bereichen des Software-Engineering-Prozesses Anwendung, zum Beispiel:

- **Ein schriftlicher Planungsprozess.** Bevor Sie mit dem Coden beginnen, wird ein Plan geschrieben und herumgereicht, um Feedback zu erhalten. Dieser Plan könnte als *Request for Comments* (Aufforderung zum Kommentieren, RFC), als *Engineering Requirements Document* (technische Anforderungen, EDD) oder auch als *Architectural Decision Record* (Designdokument, ADR) umgesetzt werden.
- **Automatisierte Tests.** Das Schreiben von Unit-, Integrations-, End-to-End-, Performance-, Belastungs- oder anderen Arten von Tests. Dieses Vorgehen steigert normalerweise die Qualität und Wartbarkeit und führt oft zu einer schnelleren Auslieferung der Software, weil Regressionen schneller erkannt werden. Mehr zum Testing finden Sie im folgenden Kapitel.

- **Testbasierte Entwicklung (Test-Driven Development, TDD).** Eine Unter-
menge des automatisierten Testings, bei dem Tests vor dem eigentlichen Code
geschrieben werden.
- **Code-Reviews.** Andere Engineers überprüfen und bestätigen den Code, bevor
er freigegeben wird.
- **Post-Commit-Code-Reviews.** Hierbei werden die Code-Reviews erst nach der
Freigabe (dem *Commit*) durchgeführt. Dieser Ansatz sorgt normalerweise für
schnellere Iterationen, während Code-Reviews trotzdem durchgeführt werden.
Ein Nachteil besteht darin, dass hierdurch mehr Regressionen in den Produktiv-
betrieb gelangen können. Dieses Vorgehen funktioniert am besten in sehr klei-
nen oder besonders erfahrenen Teams.
- **Testumgebungen.** Anstatt direkt in den Produktionsbetrieb zu gehen, wird der
Code zunächst in einer zwischengeschalteten Umgebung weiter getestet. Der
Vorteil ist eine gestiegene Zuversicht, dass der Code korrekt ist. Die Nachteile
bestehen darin, dass es länger braucht, bis der Code produktiv eingesetzt wer-
den kann, und dass der Unterhalt von Testumgebungen einen höheren Arbeits-
aufwand bedeutet.
- **Stufenweise Roll-outs.** Die stufenweise Freigabe von Features und das anschlie-
ßende Einsammeln von Feedback, anstatt alle auf einmal für die Endkunden be-
reitzustellen. Feature-Flags, Experimente und A/B-Tests sind gängige Werk-
zeuge für stufenweise Roll-outs. Weitere Details hierzu finden Sie in Kapitel 17
(Teil IV), »Bereitstellung für den Produktivbetrieb«.
- **Sicheres Testen im Produktivbetrieb.** Anstatt Testumgebungen zu nutzen,
wird hierbei der Code direkt für den Produktivbetrieb bereitgestellt. Dabei
kommen sichere Testmethoden zum Einsatz, wie Mandantenfähigkeit, Feature-
Flags und mehrstufige Roll-outs.

Welche Vorgehensweisen sollte Ihr Team also nutzen? Fragen Sie besser danach,
worin die größten Herausforderungen bei der Bereitstellung von Software durch Ihr
Team bestehen.

Gibt es Probleme, weil zu viele Bugs im produktiven Betrieb landen? Dann sollten
Sie überlegen, ob Techniken wie TDD, Testumgebungen oder das Testen im Pro-
duktivbetrieb helfen können. Sind die Code-Reviews besonders langwierig? Dann
überlegen Sie, welches Vorgehen hierbei Abhilfe schaffen kann: zum Beispiel Zeit für
Reviews zu reservieren, weniger Reviews durchzuführen, kleinere Änderungen am
Code vorzunehmen oder etwas anderes.

Je besser Sie mit den Engineering Practices vertraut sind, desto besser können Sie
einschätzen, ob eine bestimmte Vorgehensweise Ihrem Team nutzt. Eine Möglich-
keit, die Praktiken kennenzulernen, besteht darin, sie zu lernen. Eine weitere Option
ist, sich Rat von Menschen zu holen, die bestimmte Vorgehensweisen schon selbst
ausprobiert haben und über entsprechende Erfahrungen verfügen!

Vorwort	23
Einleitung	25

Teil I: Grundlagen der Entwicklerkarriere

1 Karrierepfade	29
Arten von Unternehmen	29
Big Tech, technologische Großunternehmen	30
Mittlere bis große Technologieunternehmen	30
Scale-ups	30
Start-ups	30
Traditionelle, nicht technologieorientierte Unternehmen mit technischen Abteilungen	31
Traditionelle, aber technologielaastige Unternehmen	32
Kleine, nicht risikofinanzierte Unternehmen	32
Öffentlicher Sektor	33
Non-Profit-Organisationen	33
Beratungsunternehmen, Outsourcing-Firmen und Entwicklungs- agenturen	33
Akademische Einrichtungen und Forschungslabore	35
Welche Art von Unternehmen ist für Ihre Karriereziele am besten geeignet?	35
Typische berufliche Laufbahnen im Software-Engineering	35
Einspuriger Karrierepfad	35
Zweispuriger Karrierepfad	36
Alle Karrierepfade sind einmalig	37
Häufig anzutreffende Karrierepfade	38

Vergütung und Branchenebenen (Tiers) von Unternehmen	40
Ebene 1: Lokaler Markt	41
Ebene 2: Lokaler Marktführer	41
Ebene 3: Regionaler/internationaler Marktführer	41
Contractors und Freelancer	42
Vor- und Nachteile der verschiedenen Ebenen	43
Cost-Center, Profit-Center	44
Alternative Wege, über den Karriereverlauf nachzudenken	46
2 Die Karriere in die eigene Hand nehmen	49
Sie sind für Ihre Karriere verantwortlich	49
Als jemand gesehen werden, der »die Dinge anpackt«	50
Schaffen Sie etwas!	50
Erledigen Sie möglichst viele Dinge mit weitreichenden Auswirkungen	51
Lassen Sie die Leute wissen, dass Sie etwas schaffen	51
Führen Sie ein Arbeitsprotokoll	51
Ist das Führen eines Protokolls nicht etwas seltsam?	53
Bitten Sie um Feedback und geben Sie Feedback	53
Bitten Sie um Feedback	54
Geben Sie anderen Feedback	55
Schlecht formuliertes Feedback »decodieren«	56
Machen Sie sich Ihren Manager zum Verbündeten	57
Regelmäßige persönliche Treffen mit Ihrem Manager	57
Berichten Sie und gehen Sie nicht davon aus, dass Ihr Manager alles weiß	57
Verstehen Sie die Ziele Ihres Managers	57
Liefen Sie, was vereinbart war, und geben Sie rechtzeitig Bescheid, wenn Sie es nicht schaffen	58
Schaffen Sie gegenseitiges Vertrauen mit Ihrem Manager	58
Sorgen Sie dafür, dass Ihre Arbeit wahrgenommen wird	58
Teilen Sie sich Ihre Kraft gut ein	59
Dehnen	59
Ausführen	60
Leerlauf	60
3 Leistungsbeurteilungen	63
Früh beginnen: Kontext erfassen und Ziele setzen	63
Die wichtigsten Faktoren erkennen und verstehen	63
Welches System zur Leistungsbeurteilung wird verwendet?	64
Besprechen Sie Ihre Ziele mit Ihrem Manager	65
Einigen Sie sich mit Ihrem Manager auf bestimmte Ziele	65

Die Macht der Gewohnheit	66
Zeichnen Sie Ihre Erfolge auf	66
Führen Sie ein Arbeitsprotokoll	66
Lassen Sie Ihren Manager von Ihren Fortschritten wissen	67
Etwas schaffen	67
Anderen helfen	67
Notieren Sie, was Sie tun, um anderen zu helfen	68
Bitten Sie hin und wieder um konkretes Feedback	68
Vor der Leistungsbeurteilung	70
Wie sehr können Sie sich auf Ihren Manager als Ihren	
Fürsprecher verlassen?	70
Finden Sie die wichtigen Fristen heraus	70
Feedback von Kollegen bekommen	71
Nehmen Sie eine Selbstscheinschätzung vor	71
Die Leistungsbeurteilung	72
Bedenken Sie, dass Leistungsbeurteilungen nur	
Momentaufnahmen sind	72
Erkennen Sie die Dynamik der Leistungsbeurteilung	72
Machen Sie sich nicht zu abhängig vom Ergebnis	73
Voreingenommenheit ist sehr real	73
Seien Sie offen für negatives Feedback und weisen	
Sie es nicht zurück	74
Sie und Ihr Manager sitzen im selben Boot	74
Konzentrieren Sie sich auf die langfristigen Ergebnisse	74
Vergessen Sie nicht das große Ganze	75
4 Beförderungen	77
Wie Beförderungen entschieden werden	77
Arten von Beförderungsprozessen	78
Informeller Beförderungsprozess	78
Ein schlanker Beförderungsprozess	79
Ein aufwendiger Beförderungsprozess	79
Hybride Modelle	80
»Terminal Level«	80
Beförderungen in Big-Tech-Unternehmen	81
Gehalt und Beförderungen	81
Beförderung: was vom Unternehmen erwartet wird	82
Beförderungsorientierte Softwareentwicklungen	84
Ratschläge für die Beförderung	85
Die Arbeit organisieren	86
Die Bedeutung Ihres Managers	87
Bleiben Sie realistisch	88

Langfristige Karriereplanung	89
Machen Sie Ihr Selbstwertgefühl nicht von Beförderungen und Titeln abhängig	89
Versuchen Sie nicht, Menschen aus dem Weg zu drängen	89
Viele Investitionen in die Karriere zahlen sich erst später aus	89
Zufriedenheit sollte nicht durch Titel und Karrierestufen definiert werden	90
5 In verschiedenen Umgebungen erfolgreich sein	91
Produktteams und produktorientierte Engineers	91
Produktorientierte Engineers	92
Wie wird man ein produktorientierter Engineer?	93
Plattformteams	94
Vorteile der Arbeit in einem Plattformteam	95
Nachteile der Arbeit in einem Plattformteam	95
In einem Plattformteam erfolgreich sein	96
»Friedenszeiten« im Vergleich mit »Kriegszeiten«	97
In Kriegszeiten erfolgreich sein	99
In Friedenszeiten erfolgreich sein	100
Zwischen beiden Phasen wechseln	100
Arten von Unternehmen	102
Big Tech- und größere Technologieunternehmen	102
Scale-ups in einer mittleren bis späten Phase	102
Start-ups in der Frühphase	103
Ansätze für den Erfolg in jeder Art von Unternehmen	104
6 Jobwechsel	105
Neue Möglichkeiten erkunden	105
Aktive Jobsuche	105
Passiv offen für neue Möglichkeiten	106
Wenn Sie mit Ihrem Job zufrieden sind	106
Auf Beförderung warten oder den Job wechseln?	107
Beförderungen sind in der Regel rückwärtsgewandt	108
Je höher Ihr Rang, desto riskanter ist eine neue Stelle	108
Je höher die Karrierestufe, desto schwieriger die Beförderung	109
Vorbereitung auf Bewerbungsgespräche in der Technologiebranche	110
Erstes Screening	110
Technisches Screening	112
Interviews vor Ort (»On Site«)	114
Interviewverläufe auf Staff-Ebene und darüber	115
Herabstufung	116
Umgang mit Herabstufung	117
Höherstufung	118

In eine neue Stelle einarbeiten (»Onboarding«)	119
Für alle Unternehmen jeder Branchenebene	119
Einarbeiten in einer kleineren Firma	120
Einarbeiten in einem großen Unternehmen	120
Einarbeiten in eine Rolle auf Senior-Stufe oder darüber	120
Einarbeiten in eine Rolle auf Staff-Ebene oder darüber	121
Denkanstöße	122

Teil II: Der kompetente Softwareentwickler

7 Die Dinge anpacken	125
Konzentration auf die wichtigste Aufgabe	125
Machen Sie es sich zur Gewohnheit, Ihre wichtigste Aufgabe immer fertigzustellen	126
Lernen Sie, Nein zu sagen	126
Blockaden beseitigen	127
Blockaden erkennen	127
Probieren Sie verschiedene Methoden aus, um die Blockaden zu beseitigen	128
Holen Sie sich Unterstützung, um Blockaden zu beseitigen	129
Lernen Sie, zu eskalieren, ohne Federn zu lassen	129
Der »Hilfe! Ich bin blockiert!«-Schummelzettel	131
Arbeit aufteilen	134
Denken Sie in Stories, Aufgaben und Unteraufgaben	134
Priorisieren Sie die Arbeiten, die Sie der Fertigstellung näherbringen	135
Haben Sie keine Angst, Aufgaben hinzuzufügen, zu entfernen oder zu ändern	136
Zeitaufwand schätzen	136
Zeitaufwand für etwas schätzen, das Sie schon einmal getan haben	136
Zeitaufwand für etwas schätzen, das Sie noch nicht getan haben	137
Mentoren finden	139
Der »Tech Tribe of Mentors«	140
Das »Wohlwollen-Guthaben« auf einem guten Stand halten	141
Jeder hat ein Wohlwollen-Guthaben	141
Füllen Sie Ihr Wohlwollen-Guthaben regelmäßig wieder auf	141
Vermeiden Sie es nach Möglichkeit, allein zu arbeiten	142
Die Initiative ergreifen	142
Möglichkeiten, die Initiative zu ergreifen	143

8 Programmieren	145
Programmieren üben – und zwar viel!	145
Coden Sie regelmäßig	145
Bitten Sie um Code-Reviews	146
Lesen Sie so viel Code, wie Sie schreiben	147
Coden Sie etwas mehr	148
Lesbarer Code	149
Was bedeutet »lesbarer Code«?	150
Worauf Sie achten sollten	151
Hochwertigen Code schreiben	152
Den richtigen Abstraktionsgrad verwenden	152
Richtig mit Fehlern umgehen	154
Hüten Sie sich vor »unbekannten« Zuständen	155
9 Softwareentwicklung	157
Eine Sprache besonders gut beherrschen	157
Lernen Sie die Grundlagen einer Sprache	157
Eine Ebene tiefer gehen	158
Der Praktikant, der zum Go-Experten unseres Teams wurde	159
Lernen Sie das »Haupt«-Framework für eine Sprache	159
Lernen Sie eine zweite Sprache	160
In die Breite oder in die Tiefe gehen?	161
Debuggen	162
Machen Sie sich mit Ihren Debugging-Werkzeugen vertraut	162
Beobachten Sie, wie erfahrene Entwicklerinnen und	
Entwickler debuggen	163
Lernen Sie, ohne Werkzeuge zu debuggen	163
Refaktorisieren	164
Üben Sie die Refaktorisierung so oft wie möglich	164
Die Refaktorisierungsmöglichkeiten Ihrer IDE kennen und nutzen	166
Tests refaktorisieren	166
Machen Sie sich die Refaktorisierung zur täglichen Gewohnheit	167
Testen	167
10 Werkzeuge des produktiven Entwicklers	169
Ihre lokale Entwicklungsumgebung	169
Ein schneller Zyklus aus Bearbeiten → Kompilieren/Ausführen	
→ Ausgabe	170
IDE und Arbeitsablauf konfigurieren	171
Häufig verwendete Werkzeuge	172
Git	172
Kommandozeile/Terminal	172
Reguläre Ausdrücke	173
SQL	173

KI-Assistenten für die Programmierung	174
Unternehmensinterne Entwicklungswerkzeuge	174
Ihr persönlicher »Produktivitätsschummelzettel«	175
Möglichkeiten, schnell zu iterieren	175
Lesen Sie bereits vorhandenen Code und verstehen Sie, was er tut	175
Wissen, wie das Debugging der CI/CD funktioniert	176
Wissen, wie man auf Produktions-Logs und -Dashboards zugreift	176
Nehmen Sie nach Möglichkeit nur kleine Codeänderungen vor	177
Automatisierte Tests schreiben und ausführen	177
Warten Sie nicht auf Code-Reviews, sondern fordern Sie sie an	178
Holen Sie sich regelmäßiges Feedback	178
Zusammenfassung	179

Teil III: Der vielseitige Senior-Engineer

Softwareentwicklung im Vergleich mit Software-Engineering	181
Erwartungen an Senior-Engineers über Unternehmen der verschiedenen Ebenen hinweg	183
Typische Senior-Titel	183
Typische Erwartungen an Senior-Engineers	184
Senior als höchste obligatorische Karrierestufe	184
11 Aufgaben erfolgreich abschließen	185
Dinge erledigen: Wahrnehmung und Realität	185
Wahrnehmung und Realität können sich unterscheiden	186
Kommunizieren Sie, was Sie tun	186
Weniger versprechen, Erwartungen übertreffen und mehr kommunizieren	187
Kommunizieren Sie Blockaden frühzeitig und machen Sie Lösungsvorschläge	188
Ihre eigene Arbeit	188
Effizient mit eingehenden Anfragen umgehen	188
Wenn es erledigt ist, ist es auch »richtig« erledigt	192
Arbeiten Sie in kurzen Iterationen	195
Lange Arbeitsphasen	195
Ihr Team	197
Projekte gliedern und ihren Aufwand schätzen	197
Dokumentieren Sie für andere	197
Beseitigen Sie Blockaden für Ihr Team	197
Werden Sie besser im »außerhalb-der-Box-Denken«	198

Das große Ganze	199
Werden Sie produktorientiert	199
Das Geschäft verstehen	200
12 Zusammenarbeit und Teamwork	201
Code-Reviews	201
Der Ton macht die Musik	202
Vor der Freigabe Änderungen anfordern	202
Miteinander sprechen	203
Erbsenzähler-Kommentare	203
Neueinsteiger und Code-Reviews	203
Büro- und zeitzoneübergreifende Reviews	204
Arbeit im Tandem	204
Situationen, in denen Tandems sinnvoll sein können	205
Ansätze, wenn Sie der erfahrene Partner sind	206
Ansätze, wenn Sie weniger erfahren sind	207
Mentoring	208
Informelles Mentoring	208
Formelles Mentoring	208
Startpunkt für das Mentoring: das Vorstellungstreffen	209
Wenn Sie der Mentor sind	210
Wenn Sie der Mentee sind	212
Langfristige Vorteile des Mentorings	212
Feedback geben	213
Besser positives Feedback geben	213
Konstruktives oder negatives Feedback geben	213
Zusammenarbeit mit anderen Engineering-Teams	214
Zeichnen Sie eine Karte der Teams	215
Machen Sie sich mit anderen Teams bekannt	215
Einfluss auf andere	216
Liefen Sie hervorragende Arbeit ab	216
Lernen Sie andere Menschen kennen	216
Beteiligen Sie sich an funktionsübergreifenden Projekten	217
»Werben« Sie für Ihre Arbeit – aber ohne anzugeben	218
13 Software-Engineering	219
Sprachen, Plattformen und Fachgebiete	219
Lernen Sie eine imperative, eine deklarative und eine funktionale Sprache in ihrer ganzen Bandbreite	220
Machen Sie sich mit verschiedenen Plattformen vertraut	221
Steigern Sie Ihre Full-Stack-Fähigkeiten	221
KI-Assistenten können den Wechsel beschleunigen	222

Debugging	222
Wissen, welche Dashboards und Logging-Systeme man sich ansehen muss	223
Erleichtern Sie anderen das Debugging	223
Die Codebasis verstehen	224
Eignen Sie sich genug Wissen zur Infrastruktur an	225
Aus Störungen lernen	225
Technische Schulden	225
Technische Schulden abbauen	226
Die Anhäufung technischer Schulden verringern	227
»Gerade genug« technische Schulden	228
Dokumentation	229
Designdokumente/RFCs (Request for Comments)	229
Testpläne, Roll-out-Pläne und Migrationspläne	229
Dokumentation von Interfaces und Integrationen	230
Versionshinweise (Release Notes)	230
Dokumentation für Neueinsteiger	230
Ein »Teamhandbuch«	231
Betriebshandbuch für Engineers in Rufbereitschaft	231
Benutzerhandbücher und Leitfäden	231
Dokumentation als Aktivität mit großer Tragweite	232
Best Practices auf das gesamte Team ausdehnen	232
14 Testing	235
Unit-Tests	236
Der Senior-Engineer, der keinen einzigen Unit-Tests schrieb	236
Integrationstests	237
UI-Tests	238
Gedankenmodelle für automatisiertes Testen	238
Die Testpyramide	239
Der Test-Pokal	240
Spezialisierte Tests	241
Performancetests	241
Belastungstests	242
Chaostests	242
Snapshot-Tests	243
Tests auf Applikations-/Bundle-Größe	243
Smoke-Tests	243
Manuelle Tests und Plausibilitätstests	244
Andere Tests	244
Testing im Produktivbetrieb	245
Vor- und Nachteile des automatisierten Testens	246

15 Softwarearchitektur	249
Designdokumente, RFCs und Architekturdokumente	250
Das Ziel von RFCs	250
Vorteile	251
RFCs sichten	251
Architekturdokumente	252
Prototyping und Proof-of-Concept	252
Prototyping zum Erforschen	253
Erstellung mit der Absicht, das Ergebnis zu verwerfen	253
Domain-Driven Design	254
Umsetzbare Softwarearchitektur	256
Die geschäftlichen Ziele im Auge behalten	256
Einverständnis der Entscheidungsträger	256
Brechen Sie aus der Entscheidungsstarre aus	257
Änderungen korrekt einführen	258
Im Software-Engineering gibt es keine endgültigen Entscheidungen	259
Überdenken Sie Ihre architektonischen Entscheidungen	259
Zusammenfassung	260

Teil IV: Der pragmatische Tech-Lead

Typische Berufsbezeichnungen für Tech-Leads	263
Typische Erwartungen an Tech-Leads	263
Wo sich die Wege von IC und Manager trennen	264
16 Projektmanagement	267
Unternehmen, in denen Engineers Projekte leiten	267
Welchen Zweck hat das Projektmanagement?	268
Projekt-Kick-offs und Etappenziele	269
Das Projekt-Kick-off	270
Das Engineering-Kick-off	271
Etappenziele festlegen	272
Sicherstellen, dass Schätzungen nicht bindend sind	272
Die »Physik« von Softwareprojekten	273
Der Projektumfang wächst	274
Wenn sich der Zeitrahmen des Projekts ändert	275
Wenn weniger Menschen an einem Projekt arbeiten können	275
Änderungen bei den am Projekt beteiligten Personen	276
Alltag im Projektmanagement	276
Als Tech-Lead Entscheidungen treffen	278

Risiken und Abhängigkeiten	279
Technologische Risiken	279
Abhängigkeiten auf Engineering-Ebene	280
Abhängigkeiten außerhalb der Engineering-Ebene	280
Fehlende Entscheidungen oder kontextbezogene Abhängigkeiten ..	281
Unrealistischer Zeitrahmen	281
Nicht genug Leute oder Bandbreite	282
Etwas Unvorhergesehenes, das auf halber Strecke im Projekt auftritt	282
Keine Ahnung, wie lange etwas tatsächlich dauern wird	283
Projekte abschließen	283
Was ist mit Projekten, die eher gescheitert als erfolgreich sind?	285
17 Bereitstellung für den Produktivbetrieb	287
Extreme bei der Bereitstellung für den produktiven Einsatz	287
YOLO-Shipping	287
Sorgfältige mehrstufige Überprüfung	288
Typische Bereitstellungsprozesse	290
Start-ups	290
Traditionelle Unternehmen	291
Große Technologieunternehmen	291
Das Hauptprodukt von Meta	291
Prinzipien und Werkzeuge	292
Entwicklungsumgebungen	292
Testen und überprüfen	292
Überwachung, Bereitschaftsdienst und Störungsmanagement	293
Zusätzliche Kontrollebenen	294
Separate Entwicklungsumgebungen	294
Dynamisch verfügbare Testing-/Bereitstellungsumgebungen	294
Ein eigenes Team für die Qualitätssicherung (Quality Assurance, QA)	295
Exploratives Testing	295
Canary-Testing	296
Feature-Flags und Experimente	296
Etappenweise Bereitstellung	297
Mandantenfähigkeit (Multi Tenancy)	297
Automatische Rollbacks	298
Automatische Roll-outs und Rollbacks	298
Pragmatische Risiken eingehen	298
Weitere Überlegungen	300
Den richtigen Ansatz wählen	301

18 Stakeholder-Management	303
Das wahre Ziel des Stakeholder-Managements	303
Arten von Stakeholdern	304
Stakeholder nach Art der Abhängigkeit einteilen	305
Finden Sie heraus, wer die Stakeholder sind	306
Stakeholder auf dem Laufenden halten	308
Problematische Stakeholder	309
Persönliche Gespräche	310
Seien Sie transparent und klären Sie auf	310
Bitten Sie um Unterstützung	310
Von Stakeholdern lernen	310
19 Teamstruktur	313
Rollen und Titel	313
Der Unterschied zwischen Titeln und Rollen	313
»Tech-Lead« kann ein Titel oder eine Rolle sein	314
Rollen können implizit oder explizit sein	314
Entscheiden Sie, welche Rollen explizit sind	314
Teamprozesse	315
Führen Sie nicht nur Prozesse ein, sondern entfernen Sie auch welche	316
Den Fokus des Teams steigern	317
Wehren Sie sich gegen plötzliche Fokusänderungen	317
20 Teamdynamik	319
Gesunde Teams	319
Klarheit	319
Ausführung	320
Gute Moral	320
Gesunde Kommunikation	320
Ein engagiertes Team	320
Ungesunde Teams	321
Gründe für ungesunde Teams und der Umgang damit	321
Strukturelle Gründe für Schwierigkeiten des Teams	324
Teams mit wachsenden Schwierigkeiten	325
Uausgesprochene Konflikte und Reden hinter dem Rücken anderer	325
Im Verborgenen wachsende Probleme bei der Umsetzung	325
Gute Arbeit ohne Beachtung	325
Wachsendes Abnutzungsrisiko	326
Ein Team mit vielen Senior-Engineers, denen die Herausforderungen fehlen	326

Teamdynamik verbessern	326
Zuerst beobachten	327
... dann die Dynamik verbessern	327
Beziehungen zu anderen Teams	329
Zusammenfassung	330

Teil V: Vorbildliche Staff- und Principal-Engineers

Typische Erwartungen an Staff+-Engineers	334
Engineers auf Staff-Ebene und darüber sind Partner der EMs und PMs	335
21 Das Geschäft verstehen	337
North Stars, KPIs und OKRs	337
North Star	338
KPIs	338
OKRs	339
Werden die richtigen Dinge gemessen?	340
Ihr Team und Ihr Produkt	341
Den Produktmanager-Hut aufsetzen	342
Versetzen Sie sich in den Kunden hinein	342
Verstehen, warum Kunden Ihr Produkt nutzen	343
Den geschäftlichen Wert Ihres Produkts verstehen	344
Erstellen Sie eine SWOT-Analyse für Ihr Produkt	345
Ihr Unternehmen	345
Was ist das Geschäftsmodell?	345
Führen Sie persönliche Gespräche mit den Produktverantwortlichen	346
Sprechen Sie mit Leuten außerhalb der Engineering- und Produktbereiche	346
Führen Sie persönliche Gespräche mit geschäftlichen Stakeholdern	347
Beachten Sie die Kommunikation der Geschäftsleitung	348
Sprechen Sie mit Kunden und hören Sie ihnen zu	349
Beteiligen Sie sich an strategischen Diskussionen	349
Arbeiten Sie an bereichsübergreifenden Projekten	349
Führen Sie persönliche Gespräche mit Ihrem Manager und den übergeordneten Führungskräften	350
Nehmen Sie sich Zeit, um PRDs zu lesen	351
Schaffen Sie Gelegenheiten für zufällige Treffen	351
Warum treffen Engineers nur selten auf geschäftliche Entscheidungsträger?	352

Börsennotierte Unternehmen	353
Start-ups	354
Ihre Branche	354
22 Zusammenarbeit	357
Interne Politik	357
Die »falsche« Art der Politik	358
Problematische Wahrnehmungen	359
Feedback zu »schlechter« Politik geben	360
Andere beeinflussen	361
Verdienen Sie sich »Vertrauenskapital«	362
Stellen Sie Fragen und seien Sie ein aktiver Zuhörer	362
Erklären Sie Ihren Standpunkt	363
Ergreifen Sie in Designdiskussionen Partei und erklären Sie Ihre Beweggründe	363
Krempeln Sie die Ärmel hoch und machen Sie sich an die Arbeit . . .	364
Machen Sie Ihre Arbeit sichtbar	364
Übernehmen Sie die Führung und ergreifen Sie die Initiative	364
Unterstützen Sie andere uneigennützig	365
Verbessern Sie Ihre Schreibfähigkeiten	365
Zusammenarbeit mit Managern	366
Zusammenarbeit mit Kollegen auf Staff+-Ebene	367
Das Netzwerk erweitern	368
Finden Sie einen Mentor im Unternehmen	368
Arbeit an teamübergreifenden Projekten	368
An internen Schulungen teilnehmen	369
Treffen mit Menschen von außerhalb Ihres Teams	369
Netzwerk und Einfluss gehören zusammen	369
Anderen helfen	369
Stellen Sie sich als Mentor zur Verfügung	370
Förderung	370
23 Software-Engineering	373
Coding, das Sie immer noch selbst erledigen	373
Programmieren in Bursts	373
Im Tandem in Bursts programmieren	374
Problematischen Projekten helfen	374
Passen Sie Ihren Stil dem Team an	375
Gehen Sie strategisch vor	375
Hilfreiche Engineering-Prozesse	376
Definieren Sie »fertig«	376
Stilregeln für das Programmieren	377
Code-Reviews	378

Post-Commit-Reviews	378
Automatisierte Tests und Testing im Produktionsbetrieb	379
Scaffolding für neue Dienste und Komponenten nutzen	379
Roll-out-Hygiene bei Experimenten	379
Dashboards zur Kontrolle der Systemgesundheit	380
Engineering-Praktiken für schnelles Iterieren	380
Continuous Integration (CI)	380
Continuous Deployment (CD)	381
Trunk-basierte Entwicklung	382
Feature-Flags	383
Monorepos	383
Microservices oder Monolithen	384
Werkzeuge, die Engineers effizienter machen	385
Dienstkatalog	385
Codesuche	385
Entwicklerportale	386
Cloudbasierte Entwicklungsumgebungen	386
KI-Programmierungswerkzeuge	388
Kaufen, selbst erstellen oder übernehmen?	389
Compliance und Datenschutz	390
Regulierung	390
Logging	391
Auditierung	392
Sichere Entwicklung	392
24 Zuverlässige Softwaresysteme	395
Für Zuverlässigkeit zuständig sein	395
Logging	396
Verbindliche Logging-Praktiken	397
Ein Logging-Leitfaden, der sich bewährt hat	397
Nutzen Sie ein Framework, das ein »korrektes« Logging durchführt	398
Überwachung	398
50er-, 95er- und 99er-Perzentile	398
Zu überwachende Dinge	399
Geschäftliche Kennzahlen überwachen	400
Warnmeldungen	401
Dringlichkeit von Warnungen	402
Falscher Alarm	402
Statische Grenzwerte versus Anomalieerkennung	403
Bereitschaftsdienst	404
Typische Bereitschaftsdienste	405
Ein spezielles Bereitschaftsteam oder Rufbereitschaft für alle Teams?	405

Runbooks für den Bereitschaftsdienst	406
Vergütung für den Bereitschaftsdienst	407
Sollten Engineers »normale« Arbeit leisten, während sie in Bereitschaft sind?	408
Burn-out durch Bereitschaftsdienst	409
Umgang mit Störungen	409
Erkennen einer Störung	410
Beheben der Störung	410
Nachbereitung des Vorfalls	411
Resiliente Systeme schaffen	413
Planungsphase	413
Während der Programmierungsphase	414
Simulieren Sie Fehler und testen Sie die Systemantwort	414
25 Softwarearchitektur	417
Halten Sie die Sache so einfach wie möglich	418
Den Fachjargon kennen, aber nicht unnötig benutzen	419
Architektonische Schulden	420
Erstellung zusätzlicher Dienste, um schneller voranzukommen	420
Nicht aufgeteilte Monolithen	420
Nicht funktionale Probleme	421
Veraltete Sprachen oder Frameworks	421
One-Way-Door- oder Two-Way-Door-Entscheidungen	422
Two-Way-Door-Entscheidungen	422
One-Way-Door-Entscheidungen	422
Two-Way-Doors, die zu One-Way-Doors werden	424
Entscheidungen im Mittelfeld	424
Die Tragweite Ihrer Entscheidungen	425
Skalierbare Architektur	426
Skalierung, um ein Wachstum neuer geschäftlicher Anwendungsfälle zu ermöglichen	426
Skalierung, um ein Ansteigen von Datenmenge, Nutzung und Traffic zu ermöglichen	427
Architektonische Entscheidungen und geschäftliche Prioritäten	428
Architektonische Entscheidungen auf geschäftliche Ziele und Wachstum ausrichten	428
Verbinden Sie Änderungen an der Architektur mit geschäftlichen Initiativen	429
Manchmal ist gut genug besser als perfekt	429
Nahe genug an der praktischen Arbeit bleiben	429

Eigenschaften eines Softwarearchitekten	430
Eher theoretische Eigenschaften	431
Eher praktische Eigenschaften	432
Ist diese Einteilung in Archetypen nützlich?	434
Zusammenfassung	435

Teil VI: Abschluss

26 Lebenslanges Lernen	441
Bleiben Sie neugierig	442
Stellen Sie Fragen	442
Bleiben Sie neugierig und bescheiden	443
Lernen Sie weiter	444
Tandems und »Shadowing«	444
Mentoring	445
Eigenständiges Lernen	446
Teilen Sie Ihr Wissen	447
Sammeln Sie Wissen an	447
Ihre Lernweise kann sich im Laufe der Zeit ändern	448
Fordern Sie sich selbst immer wieder heraus	448
Bleiben Sie mit der Branche auf dem Laufenden	450
Machen Sie auch mal Pause	451
27 Literaturhinweise	453
Zusatzkapitel	453
Mit der Branche Schritt halten	453
Bücher	454
Errata für dieses Buch	455
Danksagungen	457
Index	459