Tom Hombergs

Clean Architecture
Praxisbuch

für saubere Software-Architektur
und wartbaren Code

Übersetzung
der 2. Auflage

Mit Codebeispielen in Java

mitp

Inhaltsverzeichnis

Vorwort

Das Paradies fürs Entwicklungsteam: Domänenlogik ist einfach zu testen, Infrastruktur und Technologie für Tests einfach zu simulieren (zu »mocken«) und es gibt eine ganz saubere Trennung zwischen Domänencode und technischem Code. Selbst die Migration von einer Technologie auf eine andere geht leicht von der Hand. Keine endlosen Diskussionen mehr, welcher Teil des Codes dieses verzwickte kleine Feature implementieren soll, das die Verkaufsleute schon der Fachabteilung versprochen haben. Dieses Paradies heißt »Clean Architecture«, und Tom wird Sie auf Ihrer Reise dorthin begleiten.

Seit einigen Jahren gibt es diese Clean Architecture unter verschiedenen Namen (etwa: Hexagonale Architektur, Ports-und-Adapter-Architektur, Zwiebelarchitektur). Dem Ganzen liegt eine ziemlich einfache Idee zugrunde: zwei konzentrische Kreise, die Domäne und Technik innerhalb der Software trennen. Abhängigkeiten fließen grundsätzlich nach innen, von der Technologie zur Domäne. Domänenklassen dürfen niemals Abhängigkeiten zu technischen Klassen haben.

Zu schade nur, dass die meisten der Originalquellen es versäumt haben, die Details der Umsetzung zu erklären, beispielsweise wie Packages und Code organisiert sein sollten. Toms Buch füllt diese Lücke perfekt aus. Anhand eines anschaulichen Beispiels führt Tom Sie (und Ihr Entwicklungsteam) zu einer ausgesprochen gut wartbaren und sauberen architektonischen Struktur.

Tun Sie sich selbst und Ihrem Entwicklungsteam einen Gefallen, geben Sie der Clean Architecture eine Chance. Ich verspreche Ihnen, dass Sie es nicht bereuen werden!

Gernot Starke
Köln, Juni 2023
Pragmatischer Softwarearchitekt seit den 1990er-Jahren, Gründer von arc42, Mitgründer von iSAQB und Nerd

Einleitung

Wenn Sie dieses Buch in die Hand genommen haben, dann machen Sie sich vermutlich Gedanken um die Software, die Sie entwickeln. Sie möchten nicht nur, dass Ihre Software die expliziten Anforderungen Ihrer Kunden erfüllt, sondern auch die implizite Anforderung der Wartbarkeit sowie Ihre eigenen Ansprüche hinsichtlich Struktur und Ästhetik.

Es ist schwer, diesen Anforderungen gerecht zu werden, da Softwareprojekte (oder Projekte ganz allgemein, wenn wir ehrlich sind) üblicherweise nicht so verlaufen wie geplant. Manager ziehen um das ganze Projektteam eine Deadline[1], externe Partner bauen ihre APIs anders als versprochen und die Softwareprodukte, von denen wir abhängig sind, funktionieren nicht so, wie wir es erwarten.

Und dann ist da noch Ihre eigene Softwarearchitektur. Zu Anfang war sie so einfach. Alles war klar und schön. Dann zwang die Deadline Sie dazu, Abkürzungen zu nehmen. Die Abkürzungen sind nun alles, was von der Architektur geblieben ist, und es dauert länger und länger, neue Features abzuliefern.

Ihre von Abkürzungen belastete Architektur macht es schwer, auf eine API zu reagieren, die geändert werden musste, weil ein externer Partner Mist gebaut hat. Es scheint leichter, einfach Ihren Projektmanager vorzuschicken, um diesem Partner mitzuteilen, dass er die API abliefern soll, auf die Sie sich geeinigt hatten.

Inzwischen haben Sie vollständig die Kontrolle über die Situation verloren. Mit hoher Wahrscheinlichkeit wird eines der folgenden Dinge passieren:

- Der Projektmanager ist nicht stark genug, um den Kampf gegen den externen Partner zu gewinnen.
- Der externe Partner findet ein Schlupfloch in der Spezifikation der API, das ihm recht gibt.
- Der externe Partner braucht weitere <hier Zahl einsetzen> Monate, um die API anzupassen.

1 Das Wort »Deadline« stammt angeblich aus dem 19. Jahrhundert und beschrieb eine Linie, die um ein Gefängnis oder Gefangenenlager gezogen wurde. Ein Gefangener, der diese Linie überquerte, wurde erschossen. Behalten Sie das im Hinterkopf, wenn wieder einmal jemand eine »Deadline« zieht! Im Deutschen gibt es den viel harmloseren – wenn auch bürokratischer klingenden – Begriff des »Fälligkeitstermins«. Falls Sie gern gefährlicher leben, können Sie auch von einer »Galgenfrist« sprechen – das ist die kurze Zeit, die einem noch bis zu einem unangenehmen Ereignis (dem Gang zum Galgen) bleibt.

All das führt zum selben Ergebnis ? Sie müssen schnell Ihren Code ändern, weil die Deadline droht.

Sie fügen eine weitere Abkürzung ein.

Anstatt den Zustand Ihrer Softwarearchitektur durch äußere Umstände diktieren zu lassen, setzt dieses Buch darauf, dass Sie selbst die Kontrolle übernehmen. Sie erreichen dies, indem Sie eine Architektur erschaffen, die Ihre Software »soft«, also »weich« macht, im Sinne von »flexibel«, »erweiterbar« und »anpassbar«. Eine solche Architektur erlaubt es Ihnen problemlos, auf externe Faktoren zu reagieren und nimmt eine große Last von Ihren Schultern.

Das Ziel dieses Buches

Ich habe dieses Buch geschrieben, weil ich von der Praktikabilität der verfügbaren Ressourcen über domänenzentrierte Architekturstile enttäuscht war. Zu diesen Architekturstilen gehören *Clean Architecture* von Robert C. Martin und *Hexagonal Architecture* von Alistair Cockburn.

Viele Bücher und Online-Ressourcen beschreiben wertvolle Konzepte, aber nicht, wie man diese tatsächlich implementieren kann. Das liegt vermutlich daran, dass es mehr als eine Möglichkeit gibt, einen Architekturstil zu implementieren.

Mit einer praktischen Diskussion über das Herstellen einer Webanwendung im Stil einer Hexagonalen Architektur bzw. im »Ports-und-Adapter«-Stil versuche ich, diese Lücke zu füllen. Um dieses Ziel zu erreichen, demonstrieren die Codebeispiele und Konzepte in diesem Buch meine Interpretation der Implementierung einer Hexagonalen Architektur. Es gibt ganz sicher andere Interpretationen und ich erhebe auch nicht den Anspruch, dass meine *die* allgemeingültige Lösung ist.

Ich hoffe jedoch, dass die Konzepte in diesem Buch Ihnen eine gewisse Inspiration bieten, sodass Sie Ihre eigene Interpretation der Hexagonalen/Clean Architecture finden.

An wen sich dieses Buch richtet

Dieses Buch ist für Softwareentwickler aller Erfahrungsstufen gedacht, die sich mit dem Erstellen von Webanwendungen befassen.

Als Einsteiger oder Einsteigerin lernen Sie, wie Sie Softwarekomponenten und ganze Anwendungen auf saubere und wartbare Art und Weise entwerfen. Sie erfahren auch etwas darüber, wann Sie eine bestimmte Technik einsetzen können und sollten. Um wirklich den größten Nutzen aus diesem Buch zu ziehen, sollten Sie allerdings bereits einmal an der Erstellung einer Webanwendung mitgewirkt haben.

Als Entwicklerin und Entwickler mit größerer Erfahrung haben Sie Gelegenheit, die Konzepte aus dem Buch mit Ihrer eigenen Vorgehensweise zu vergleichen und hoffentlich einige davon in Ihren eigenen Entwicklungsstil zu integrieren.

Die Codebeispiele sind in Java und Kotlin geschrieben, aber alle Erörterungen lassen sich gleichermaßen auf andere objektorientierte Programmiersprachen anwenden. Falls Sie kein Java-Programmierer sind, aber objektorientierten Code in anderen Sprachen lesen können, ist alles gut. An den wenigen Stellen, an denen wir Java- oder Framework-Spezifika benötigen, werde ich diese erklären.

Die Beispielanwendung

Um einen thematischen roten Faden in diesem Buch zu haben, zeigen die meisten Codebeispiele Ausschnitte einer beispielhaften Webanwendung für Online-Überweisungen. Sie trägt den Namen »BuckPal«[2].

Die BuckPal-Anwendung erlaubt es einem Benutzer, ein Konto zu registrieren, Geld zwischen Konten zu transferieren und die Aktivitäten auf dem Konto (Ein- und Auszahlungen) einzusehen.

Ich bin kein Finanzspezialist, also versuchen Sie nicht, den Beispielcode anhand seiner rechtlichen oder funktionalen Korrektheit zu bewerten. Bewerten Sie ihn lieber anhand seiner Struktur und seiner Wartbarkeit.

Der Fluch von Beispielanwendungen für Softwaretechnikbücher und Online-Ressourcen liegt darin, dass sie zu einfach sind, um tatsächlich die realen Probleme anzusprechen, mit denen Sie sich jeden Tag auseinandersetzen müssen. Andererseits muss eine Beispielanwendung einfach genug bleiben, um die diskutierten Konzepte effektiv zu vermitteln.

Ich hoffe, das richtige Gleichgewicht zwischen »zu einfach« und »zu komplex« gefunden zu haben, wenn wir die Anwendungsfälle der BuckPal-Anwendung in diesem Buch diskutieren.

Der Code der Beispielanwendung ist auf GitHub zu finden.[3]

2 Eine schnelle Online-Suche hat ergeben, dass ein Unternehmen namens PayPal meine Idee gestohlen und sogar einen Teil des Namens kopiert hat. Scherz beiseite: Versuchen Sie einmal, einen Namen zu finden, der so ähnlich ist wie »PayPal«, aber nicht der Name eines existierenden Unternehmens ist. Zum Schreien komisch!

3 Das BuckPal-GitHub-Repository: `https://github.com/thombergs/buckpal`.

Treten Sie in Kontakt

Falls Sie etwas zu diesem Buch sagen möchten, würde ich mich freuen, von Ihnen zu hören. Schreiben Sie mir direkt eine E-Mail an `tom@reflectoring.io` oder auf Twitter über `@TomHombergs`.

Sollte es zu diesem Buch bereits eine Errata-Liste geben, ist sie unter `www.mitp.de/0814` zu finden.

Kapitel 1

Wartbarkeit

Dieses Buch dreht sich um Softwarearchitektur. Eine der Definitionen von *Architektur* ist *die Struktur eines Systems oder Prozesses*. In unserem Fall ist es die Struktur eines Softwaresystems.

Architektur ist das Entwerfen dieser Struktur mit einem Zweck. Sie gestalten Ihr Softwaresystem bewusst so, dass es bestimmte Anforderungen erfüllt. Es gibt funktionale Anforderungen, die eine Software erfüllen muss, um einen Nutzen für ihre Benutzer zu haben. Ohne Funktionalität ist Software wertlos, da sie keinen Nutzen besitzt.

Außerdem gibt es **Qualitätsanforderungen** (auch **nichtfunktionale Anforderungen** genannt), die eine Software erfüllen sollte, um von ihren Benutzern, Entwicklern und anderen Stakeholdern als qualitativ hochwertig betrachtet zu werden. Eine solche Qualitätsanforderung ist die **Wartbarkeit**.

Was würden Sie sagen, wenn ich behauptete, dass Wartbarkeit als Qualitätsmerkmal in gewisser Weise viel wichtiger ist als die Funktionalität und Sie die Wartbarkeit beim Entwurf von Software daher über alles andere stellen sollten? Nachdem wir die Wartbarkeit als wichtige Qualität etabliert haben, werden wir im Rest des Buches untersuchen, wie wir diese Wartbarkeit verbessern können, indem wir die Konzepte von Clean Architecture und Hexagonaler Architektur anwenden.

1.1 Was bedeutet Wartbarkeit überhaupt?

Bevor Sie mich für verrückt erklären und versuchen, das Buch zurückzugeben, lassen Sie mich Ihnen erklären, was ich mit Wartbarkeit meine.

Wartbarkeit ist nur eine der vielen Qualitätsanforderungen, die potenziell eine Softwarearchitektur ausmachen. Ich habe ChatGPT nach einer Liste von Qualitätsanforderungen gefragt und folgendes Ergebnis erhalten:

- Skalierbarkeit
- Flexibilität
- Wartbarkeit
- Sicherheit
- Zuverlässigkeit

- Modularität
- Performance
- Interoperabilität
- Testbarkeit
- Kosteneffizienz

Und das ist nur eine kleine Auswahl.[1]

Als Softwarearchitekten entwerfen Sie Ihre Software so, dass sie die Qualitätsanforderungen erfüllt, die für die Software am wichtigsten sind. Für eine Trading-Anwendung mit hohem Durchsatz würden Sie sich zum Beispiel auf Skalierbarkeit und Zuverlässigkeit konzentrieren. Befasst sich die Anwendung mit persönlichen Informationen in Deutschland, müsste die Sicherheit an erster Stelle stehen.

Ich glaube, dass es falsch ist, die Wartbarkeit mit den restlichen Qualitätsanforderungen in einen Topf zu werfen, da Wartbarkeit etwas Besonderes ist. Ist eine Software wartbar, dann bedeutet dies, dass sie leicht zu ändern ist. Ist sie leicht zu ändern, dann ist sie flexibel und vermutlich auch modular. Wahrscheinlich ist sie auch kosteneffizient, da leichte Änderungen gleichbedeutend sind mit günstigen Änderungen. Ist sie wartbar, kann man sie vermutlich weiterentwickeln, sodass sie skalierbar, sicher, zuverlässig und performant ist, wenn sich das alles als notwendig erweisen sollte. Man kann die Software so verändern, dass sie mit anderen Systemen interoperabel wird, weil dies leicht zu ändern ist. Und schließlich impliziert die Wartbarkeit auch eine Testbarkeit, da wartbare Software mit hoher Wahrscheinlichkeit aus kleineren und einfacheren Komponenten besteht, die das Testen erleichtern.

Sie erkennen sicher, was ich hier gemacht habe. Ich habe die KI nach einer Liste von Qualitätsanforderungen gefragt und diese dann alle wieder auf Wartbarkeit zurückgeführt. Mit ähnlich plausiblen Argumenten könnte ich vermutlich noch viel mehr Qualitätsanforderungen an Wartbarkeit koppeln. Das ist natürlich ein bisschen vereinfacht, aber im Kern stimmt es: Wenn die Software wartbar ist, dann ist es einfacher, sie in alle Richtungen weiterzuentwickeln, funktional und nichtfunktional. Und wir alle wissen, dass Änderungen im Leben eines Softwaresystems unausweichlich sind.

1.2 Wartbarkeit führt zu Funktionalität

Kommen wir zurück zu meiner Behauptung vom Anfang dieses Kapitels, dass Wartbarkeit wichtiger ist als Funktionalität.

1 Wenn Sie ein paar Ideen in Bezug auf Softwarequalität haben möchten (die von Menschen und nicht von einem Sprachmodell stammen), schauen Sie unter `https://quality.arc42.org` nach.

Fragen Sie einen Produktmenschen, was an einem Softwareprojekt am wichtigsten ist, dann wird er oder sie Ihnen sagen, dass der Wert, den die Software den Benutzern bietet, die wichtigste Sache ist. Mit Software, die ihren Benutzern keinen Wert bietet, ist kein Geld zu verdienen. Und ohne zahlende Benutzer gibt es kein funktionierendes Geschäftsmodell, das das wesentliche Maß für den Erfolg im Geschäftsleben ist.

Die Software muss also einen Wert bieten. Dies darf sie aber nicht auf Kosten der Wartbarkeit tun.[2] Denken Sie einmal darüber nach, wie viel effizienter und befriedigender es ist, Funktionalität zu einem Softwaresystem hinzuzufügen, das leicht zu ändern ist, als zu einem, bei dem Sie sich zeilenweise durch den Code kämpfen müssen! Ich bin mir ziemlich sicher, dass auch Sie schon einmal an einem dieser Softwareprojekte mitgearbeitet haben, bei denen es so viel überflüssigen Kram im Code gibt, dass es Tage oder Wochen dauert, um ein Feature zu entwickeln, das eigentlich nur ein paar Stunden zur Fertigstellung brauchen sollte.

Entsprechend ist Wartbarkeit also eine entscheidende Stütze für die Funktionalität. Schlechte Wartbarkeit bedeutet, dass Änderungen in der Funktionalität mit der Zeit immer teurer werden, wie Abbildung 1.1 zeigt:

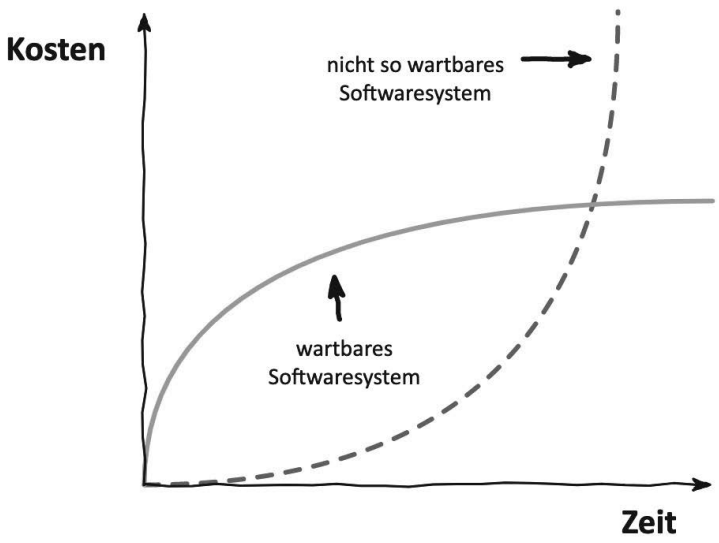

Abb. 1.1: Ein wartbares Softwaresystem verursacht über seine Lebenszeit geringere Kosten als ein nicht so gut wartbares Softwaresystem.

2 Im Kontext dieses Buches benutze ich den Begriff »Wartbarkeit« synonym zu »Veränderlichkeit einer Codebasis«. Siehe `https://quality.arc42.org/qualities/maintainability` für einige Definitionen von Wartbarkeit (die alle mit dem Ändern der Software zu tun haben).

In einem weniger gut wartbaren Softwaresystem werden Änderungen an der Funktionalität schon bald so teuer, dass jede Änderung ein Ärgernis wird. Die Produktleute beschweren sich bei den Entwicklern über die Kosten der Änderungen. Die Entwickler verteidigen sich wiederum damit, dass die Auslieferung neuer Features immer eine höhere Priorität hatte als die Verbesserung der Wartbarkeit. Mit zunehmenden Änderungskosten steigt auch die Wahrscheinlichkeit von Konflikten.

Wartbarkeit ist ein Friedensstifter. Sie ist umgekehrt proportional zu den Kosten von Änderungen und daher auch zur Wahrscheinlichkeit eines Konflikts. Haben Sie schon einmal darüber nachgedacht, die Wartbarkeit eines Softwaresystems zu verbessern, nur um einen Konflikt zu vermeiden? Ich denke, das allein ist schon eine gute Investition.

Was ist jedoch mit den großen Softwaresystemen, die trotz ihrer schlechten Wartbarkeit erfolgreich sind? Es stimmt, dass es kommerziell erfolgreiche Softwaresysteme auf dem Markt gibt, die kaum gewartet werden können. Ich habe an Systemen gearbeitet, bei denen das Hinzufügen eines einzigen Feldes zu einem Formular ein Projekt war, das Wochen an Entwicklerzeit kostete – und der Kunde hat meinen geforderten Stundensatz bezahlt, ohne mit der Wimper zu zucken.

Diese Systeme fallen üblicherweise in eine von zwei Kategorien (manchmal auch in beide):

- Sie sind am Ende ihres Lebens, sodass nur noch ausgesprochen selten Änderungen an ihnen vorgenommen werden.
- Sie werden von einem finanziell gut aufgestellten Unternehmen gestützt, das bereit ist, Geld in das Problem zu stecken.

Selbst wenn ein Unternehmen viel Geld aufwenden kann, erkennt es normalerweise, dass es die Wartungskosten reduzieren könnte, indem es in die Wartbarkeit investiert. Üblicherweise sind in solchen Unternehmen daher bereits Initiativen auf dem Weg, um die Software besser wartbar zu machen.

Sie sollten sich immer um die Wartbarkeit der Software sorgen, die Sie entwickeln, damit diese nicht zu dem gefürchteten **Big Ball of Mud** (einer großen Matschkugel) degeneriert. Falls Ihre Software jedoch nicht in eine der zwei genannten Kategorien fällt, müssen Sie sich sogar noch mehr Sorgen um sie machen.

Heißt das, Sie brauchen viel Zeit, um eine wartbare Architektur zu planen, bevor Sie überhaupt mit dem Programmieren beginnen? Müssen Sie ein **Big Design Up Front** (**BDUF**) – eine vorher festgelegte Architektur – schaffen, was oft mit der Wasserfall-Methode gleichgesetzt wird?! Nein, das müssen Sie nicht. Aber Sie müssen zumindest ein bisschen was an Architektur planen (vielleicht sollten wir dies dann **Some Design Up Front** bzw. **SDUF** nennen?), um den Samen der Wartbarkeit in die Software einzupflanzen, der es Ihnen erleichtern dürfte, die Architektur mit der Zeit in die gewünschte Richtung zu entwickeln.

Teil dieses Vorab-Designs ist die Wahl eines Architekturstils, der den Rahmen für die Software definiert, die Sie entwickeln. Dieses Buch wird Ihnen helfen zu entscheiden, ob eine *Clean Architecture* – oder *Ports-und-Adapter-/Hexagonale*-Architektur – für Ihren Kontext eine gute Entscheidung ist.

1.3 Wartbarkeit erzeugt Entwicklerfreude

Würden Sie als Entwicklerin oder Entwickler lieber an Software arbeiten, bei der Änderungen einfach sind, oder an Software, bei der Änderungen schwer sind? Nicht antworten, es ist eine rhetorische Frage!

Abgesehen vom direkten Einfluss auf die Änderungskosten hat Wartbarkeit noch einen weiteren Vorteil: Sie macht Entwickler glücklich (oder zumindest macht sie sie weniger traurig – je nachdem, wie das aktuelle Projekt gerade läuft).

Der Begriff, mit dem ich dieses Glück beschreiben möchte, lautet **Developer Joy**, also quasi Entwicklerfreude. Man könnte auch **Developer Experience** (das Erlebnis des Entwicklers) oder **Developer Enablement** (Ermächtigung oder Förderung des Entwicklers) dazu sagen. Doch ganz egal, wie Sie es nennen, es bedeutet, dass Sie den Kontext zur Verfügung stellen, den Entwickler brauchen, um ihre Arbeit gut zu erledigen.

Freude und Spaß der Entwickler stehen in einer direkten Beziehung zur Produktivität. Wenn die Entwickler glücklich sind, dann arbeiten sie im Allgemeinen besser. Und wenn sie gute Arbeit verrichten, dann sind sie glücklicher. Es gibt eine wechselseitige Korrelation zwischen Entwicklerfreude und Entwicklerproduktivität:

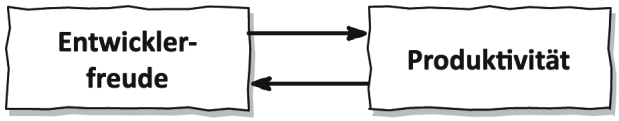

Abb. 1.2: Entwicklerfreude beeinflusst die Entwicklerproduktivität und umgekehrt.

Diese Korrelation wird im *SPACE-Framework* für Entwicklerproduktivität anerkannt.[3] SPACE liefert zwar keine einfache Antwort darauf, wie man die Entwicklerproduktivität messen könnte, stellt aber fünf Kategorien für bestimmte Metriken zur Verfügung, sodass wir bewusst einen Satz an Metriken wählen können, die all diese Kategorien abdecken, um die Entwicklerproduktivität im Kontext

3 *The SPACE of Developer Productivity* von Nicole Forsgren et al., 6. März 2021. »SPACE« ist eine Abkürzung für Satisfaction und Well-Being, Performance, Activity, Communication und Collaboration sowie Efficiency und Flow (Zufriedenheit und Wohlbefinden, Performance, Aktivität, Kommunikation und Zusammenarbeit sowie Effizienz und Fluss).
Siehe `https://queue.acm.org/detail.cfm?id=3454124`.

unseres Unternehmens und unserer Projekte am besten zu messen. Eine dieser Kategorien (das **S** in **SPACE**) ist **Satisfaction und Well-Being** (Zufriedenheit und Wohlbefinden), die ich für dieses Kapitel in Entwicklerfreude und Developer Joy übersetze.

Entwicklerfreude führt nicht nur zu besserer Produktivität, sondern natürlich auch zu einer größeren Treue gegenüber dem Arbeitgeber (zu Englisch »developer retention«, also die Verweildauer beim gleichen Arbeitgeber). Ein Entwickler, der Freude an seiner Arbeit hat, bleibt beim Unternehmen. Oder anders ausgedrückt, ein Entwickler, dem seine Arbeit keinen Spaß macht, sieht sich wahrscheinlich schnell nach etwas Besserem um.

An welcher Stelle kommt hier nun die Wartbarkeit ins Spiel? Nun, wenn Ihr Softwaresystem wartbar ist, dann brauchen Sie weniger Zeit, um eine Änderung zu implementieren, sind also produktiver. Ist Ihr Softwaresystem wartbar, dann macht es Ihnen mehr Freude, Änderungen vorzunehmen, weil es effizienter ist und es Ihnen mehr Genugtuung bringt. Selbst wenn Ihr Softwaresystem nicht so gut wartbar ist, wie Sie es gern hätten (ehrlich gestanden, ist das eine Tautologie), aber Sie die Gelegenheit haben, im Laufe der Zeit die Wartbarkeit zu verbessern, sind Sie glücklicher und produktiver. Und wenn Sie glücklich sind, dann bleiben Sie wahrscheinlich auch länger im Unternehmen.

In Diagrammform ausgedrückt, sieht das so aus wie in Abbildung 1.3 gezeigt.

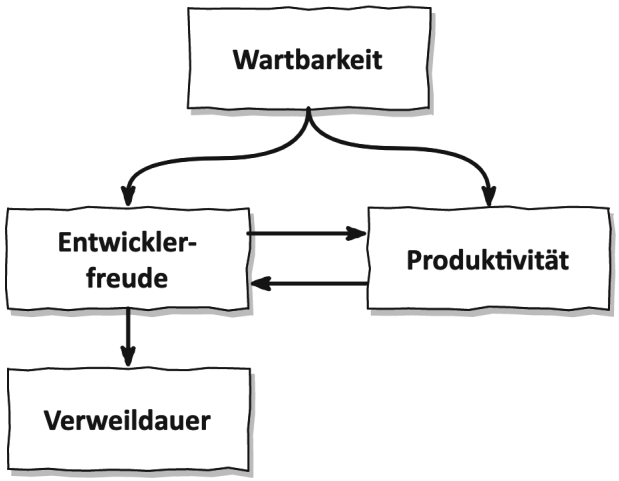

Abb. 1.3: Wartbarkeit hat einen direkten Einfluss auf Entwicklerfreude und -produktivität, während die Entwicklerfreude die Verweildauer der Entwickler beim selben Unternehmen beeinflusst.

Stichwortverzeichnis