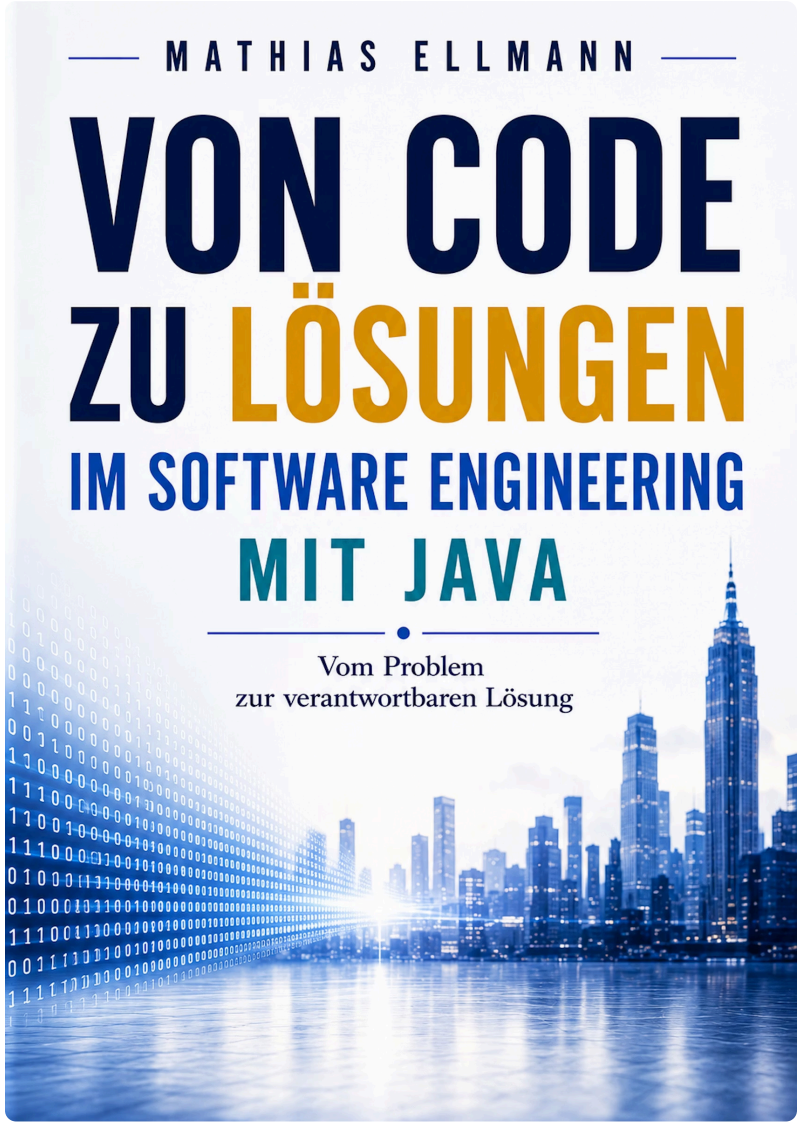


Von Code zu Lösungen im Software Engineering mit Java

Vom Problem zur verantwortbaren Lösung

Mathias Ellmann

ISBN: 978-3-6952-6205-2

Warum Software Engineering heute wichtiger ist denn je

Digitalisierung

Nahezu jede Organisation ist auf Software angewiesen – von kritischer Infrastruktur bis zu alltäglichen Geschäftsprozessen. Software ist kein Hilfsmittel mehr, sondern das Fundament moderner Organisationen.

Komplexität

Systeme wachsen schneller als das Verständnis über sie. Die Fähigkeit, Komplexität zu beherrschen und beherrschbar zu halten, ist zur Kernkompetenz geworden.

Verantwortung

Software trifft Entscheidungen mit realen Konsequenzen – für Nutzer, Organisationen und die Gesellschaft. Die ethische Dimension des Software Engineerings ist nicht optional.

Langlebigkeit

Systeme müssen über Jahre und Jahrzehnte tragfähig bleiben. Kurzfristige Lösungen erzeugen langfristige Kosten – technisch, organisatorisch und finanziell.

Das große Missverständnis

Was viele glauben

Software Engineering bedeute vor allem:

- Java · Frameworks · APIs · Syntax
- Beherrschung von Werkzeugen und Bibliotheken
- Schnelles Schreiben von funktionierendem Code

Was das Buch zeigt

Es geht um **tragfähige Lösungen** – nicht um Code. Programmieren ist das **letzte Glied** einer langen Kette von Entscheidungen.

Wer zu früh mit dem Schreiben von Code beginnt, löst möglicherweise das falsche Problem – nur sehr effizient.

Die Qualität einer Lösung wird nicht im Editor entschieden, sondern in der Analyse, im Modell und in der Architektur.

Der rote Faden des Buches

Das Buch folgt einem durchgängigen Erkenntnispfad – von der Problemwahrnehmung bis zur verantwortbaren Lösung:

1

Problem

Was ist wirklich zu lösen?

2

Anforderungen

Was wird tatsächlich gebraucht?

3

Modelle

Wie lässt sich das Problem strukturieren?

4

Entscheidungen

Welche Alternativen, welche Trade-offs?

1

Architektur

Welche Struktur trägt die Lösung?

2

Code

Wie wird die Architektur umgesetzt?

3

Qualität

Entsteht durch gute Entscheidungen, nicht durch Testen allein

4

Verantwortung

Begleitet den gesamten Prozess

Software Engineering als systemisches Problemlösen

Denk- und Entscheidungsdisziplin

Software Engineering ist keine Programmierdisziplin. Es ist die Disziplin, komplexe Probleme systematisch zu durchdringen und begründete Entscheidungen zu treffen.

Komplexität beherrschen

Komplexität beherrschbar machen, Unsicherheit reduzieren, Qualität erzeugen – das sind die zentralen Aufgaben professioneller Software Engineers.

Zusammenarbeit ermöglichen

Gute Software entsteht in Teams. Zusammenarbeit und Veränderbarkeit sichern den langfristigen Wert eines Systems.

Verantwortung übernehmen

Für Nutzer, Organisation und Gesellschaft. Verantwortung ist keine Zusatzaufgabe – sie ist integraler Bestandteil professionellen Handelns.

Probleme wirklich verstehen

Der erste und wichtigste Schritt im Software Engineering ist nicht das Schreiben von Code – es ist das gründliche Verstehen des Problems.

1 Beobachten

Symptome von Ursachen unterscheiden. Was sichtbar ist, ist selten das eigentliche Problem – es ist der Hinweis auf ein tieferliegendes Muster.

2 Analysieren

Vom Symptom zum eigentlichen Problem vordringen. Strukturierte Analyse ersetzt Vermutungen durch Erkenntnisse.

3 Zusammenhänge erkennen

Software als Teil eines größeren Systems begreifen. Kein System existiert im Vakuum – Kontext ist entscheidend.

4 Gute Fragen stellen

Gute Fragen zu stellen ist wichtiger als schnelle Antworten zu liefern. Die richtige Frage führt zur richtigen Lösung.

Anforderungen entwickeln

Anforderungen sind keine objektiven Fakten – sie sind Modelle der Wirklichkeit, geformt durch Perspektiven, Interessen und Ziele.

Stakeholder

Unterschiedliche Perspektiven, Ziele und Interessen verstehen. Wer spricht für wen? Wessen Bedürfnisse sind sichtbar, wessen verborgen?

Ziele

Von Beobachtungen zu strukturierten Anforderungen gelangen. Der Weg von der Beobachtung zur Anforderung erfordert Analyse, nicht nur Dokumentation.

Konflikte

Widersprüche und Zielkonflikte sichtbar machen. Unausgesprochene Konflikte werden zu technischen Schulden – explizite Konflikte können gelöst werden.

Prioritäten

Anforderungen sind Modelle – sie müssen priorisiert, hinterfragt und kontinuierlich verfeinert werden. Vollständigkeit ist eine Illusion.

Modelle schaffen gemeinsames Verständnis

> Modelle sind Denkwerkzeuge, keine Wahrheiten.

Domänenmodelle

Die fachliche Wirklichkeit strukturieren und kommunizieren. Ein gutes Domänenmodell schafft eine gemeinsame Sprache zwischen Fachbereich und Entwicklung.

Objektmodelle

Verantwortlichkeiten und Beziehungen klären. Wer ist für was zuständig? Welche Objekte kooperieren, welche sind voneinander unabhängig?

Architekturmodelle

Strukturentscheidungen sichtbar und diskutierbar machen. Architekturmodelle sind Kommunikationsmittel – sie ermöglichen Diskussion, bevor Entscheidungen irreversibel werden.

Entscheidungen bestimmen den Projekterfolg

Softwareprojekte scheitern selten an technischen Problemen – sie scheitern an schlechten Entscheidungen, die zu spät erkannt oder zu früh getroffen wurden.

Alternativen

Wer keine Alternativen kennt, kann nicht entscheiden – er kann nur reagieren. Professionelle Entscheidungen setzen die Kenntnis von Optionen voraus.

Trade-offs

Jede Entscheidung hat Konsequenzen – sichtbare und verborgene. Trade-offs explizit zu machen ist eine Kernkompetenz im Software Engineering.

Risiken

Unsicherheit ist kein Ausnahmefall, sondern der Normalfall. Risiken zu ignorieren bedeutet nicht, sie zu vermeiden – es bedeutet, sie unbewusst zu akzeptieren.

Verantwortung

Entscheidungen müssen nachvollziehbar und begründbar sein. Wer Entscheidungen nicht dokumentiert, verliert das institutionelle Gedächtnis des Projekts.

Architektur entsteht aus Entscheidungen

Was Architektur ist

Architektur ist kein Diagramm.

Architektur ist die **Summe bewusster Entscheidungen** über Struktur, Verantwortlichkeiten und Grenzen eines Systems.

Sie entsteht – ob bewusst gestaltet oder nicht. Die Frage ist nur, ob sie gut ist.

Was Architektur bewirkt

Gute Architektur

Ermöglicht Veränderung. Sie hält Optionen offen, trennt Verantwortlichkeiten klar und macht das System verständlich und erweiterbar.

Schlechte Architektur

Verhindert Veränderung. Sie erzeugt versteckte Abhängigkeiten, macht Änderungen riskant und teuer – und wächst mit jedem Kompromiss.

Teuerste Entscheidungen

Architekturentscheidungen sind die teuersten Entscheidungen im Projekt – weil sie am schwersten rückgängig zu machen sind.

Qualität beginnt vor dem ersten Code

> Qualität ist keine Eigenschaft des Codes – sie ist eine Eigenschaft des Prozesses.

1

In Anforderungen

Unklare Anforderungen erzeugen falsche Software – präzise und vollständig.

2

In Modellen

Schlechte Modelle führen zu schlechten Strukturen – Modellqualität ist Lösungsqualität.

3

In Architektur

Qualitätsziele müssen in der Architektur verankert sein – nicht nachträglich hinzugefügt.

4

Im Code

Code ist die letzte Ausdrucksform einer langen Kette von Qualitätsentscheidungen.

Java als Werkzeug – nicht als Ziel

Das Buch verwendet Java nicht als Lehrgegenstand, sondern als Ausdrucksmittel. Java illustriert Konzepte – es ist nicht das Konzept selbst.

Modellierung

Java unterstützt Modellierung durch Objektorientierung und klare Strukturen. Klassen, Interfaces und Pakete sind Modellierungswerkzeuge – keine bloßen Syntaxelemente.

Architektur

Java unterstützt Architektur durch Schnittstellen, Kapselung und Entwurfsmuster. Die Sprache ermöglicht es, Architekturentscheidungen im Code sichtbar zu machen.

Implementierung

Java unterstützt Implementierung durch Typsicherheit und Ausdrucksstärke. Guter Java-Code ist lesbar, verständlich und wartbar – nicht nur korrekt.

Mittel zum Zweck

Java ist Mittel zum Zweck – der Zweck ist die tragfähige Lösung. Die im Buch vermittelten Konzepte sind sprachunabhängig und auf andere Technologien übertragbar.

Objektorientierung neu denken

Objektorientierung wird häufig als technisches Konzept gelehrt. Das Buch zeigt: OO ist ein Denkmodell für Verantwortung, Zusammenarbeit und Abstraktion.

Verantwortung

Klassen übernehmen klar definierte Aufgaben. Das Single-Responsibility-Prinzip ist kein technisches Gebot – es ist ein Organisationsprinzip.

Zusammenarbeit

Objekte kommunizieren – sie kooperieren, nicht konkurrieren. Gutes OO-Design modelliert Zusammenarbeit, nicht Datenhaltung.

Abstraktion

Das Wesentliche sichtbar machen, das Unwesentliche verbergen. Abstraktion ist das mächtigste Werkzeug zur Komplexitätsbeherrschung.

Kapselung

Interna schützen, Schnittstellen stabilisieren. Kapselung schafft Vertrauen – in die Stabilität von Schnittstellen und die Unabhängigkeit von Implementierungen.

Schnittstellen verbinden Systeme

Die Rolle von Schnittstellen

Schnittstellen sind **Architekturentscheidungen** – keine technischen Details. Sie definieren, was ein System nach außen verspricht, und schützen, was es intern tut.

Entkopplung

Schnittstellen trennen Implementierung von Nutzung. Abhängigkeiten entstehen auf der Ebene der Abstraktion, nicht der Implementierung.

Zusammenarbeit

Klare Verträge ermöglichen parallele Entwicklung. Teams können unabhängig arbeiten, solange die Schnittstellen stabil sind.

Flexibilität

Implementierungen können ausgetauscht werden, ohne Abhängigkeiten zu brechen. Das ist der Kern von Erweiterbarkeit.

Kernaussage

Eine gut gestaltete Schnittstelle ist stabiler als jede Implementierung. Sie ist der Vertrag, auf dem das gesamte System aufbaut.

Schnittstellen zu gestalten bedeutet, Systemgrenzen zu definieren – und damit Architektur zu betreiben.

Persistenz und Daten

Persistenz wird häufig als technisches Randthema behandelt. Das Buch zeigt: Persistenzentscheidungen sind Architekturentscheidungen mit weitreichenden und langfristigen Konsequenzen.

Information dauerhaft nutzbar machen

Das Ziel ist nicht das Speichern von Daten – sondern das **dauerhaft nutzbar machen von Information**. Der Unterschied ist fundamental: Daten ohne Kontext sind wertlos.

Architekturentscheidung mit Langzeitwirkung

Persistenzentscheidungen sind schwer rückgängig zu machen. Die Wahl des Datenmodells, der Technologie und der Zugriffsschicht prägt das System über seinen gesamten Lebenszyklus.

Abstimmung von Domäne und Daten

Datenmodell und Domänenmodell müssen aufeinander abgestimmt sein. Divergenz zwischen beiden ist eine der häufigsten Quellen technischer Schulden.

Technische Schulden an der Grenze

Technische Schulden entstehen häufig an der Grenze zwischen Domäne und Persistenz – dort, wo Kompromisse gemacht werden, die später teuer werden.

Testen als Erkenntnisprozess

> Testen beginnt nicht nach dem Coden – es beginnt beim Verstehen des Problems.

Hypothesen prüfen

Tests formulieren Erwartungen – und decken Abweichungen auf. Ein Test ist eine formalisierte Hypothese über das Verhalten eines Systems. Wer keine Erwartungen hat, kann nicht testen.

Qualität absichern

Tests machen Qualitätsziele messbar und überprüfbar. Qualität, die nicht gemessen werden kann, kann nicht gesichert werden. Tests übersetzen abstrakte Qualitätsziele in konkrete Prüfkriterien.

Vertrauen schaffen

Gut getestete Systeme können sicher verändert werden. Tests sind kein Bürokratieaufwand – sie sind die Grundlage für nachhaltige Weiterentwicklung ohne Angst vor Regression.

Refactoring und technische Schulden

Technische Schulden sind keine Ausnahme – sie sind das unvermeidliche Ergebnis von Entscheidungen unter Unsicherheit. Die Frage ist nicht, ob sie entstehen, sondern wie bewusst man mit ihnen umgeht.

Langfristige Wartbarkeit

Code muss lesbar, verständlich und veränderbar bleiben. Lesbarkeit ist keine ästhetische Präferenz – sie ist eine wirtschaftliche Notwendigkeit. Code wird häufiger gelesen als geschrieben.

Investitionsschutz

Technische Schulden mindern den Wert jeder zukünftigen Änderung. Jede neue Funktion kostet mehr, jede Fehlerkorrektur dauert länger – bis das System nicht mehr wirtschaftlich weiterentwickelbar ist.

Nachhaltigkeit durch Refactoring

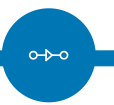
Refactoring ist keine Schwäche – es ist professionelle Disziplin. Systeme, die nicht gepflegt werden, veralten schneller als die Anforderungen. Refactoring ist Investition, nicht Aufwand.

Risiken und Unsicherheit

> Unsicherheit ist kein Planungsfehler – sie ist der Normalzustand in Softwareprojekten.

Technische Risiken

Unbekannte Technologien, Komplexität, Abhängigkeiten. Technische Risiken entstehen dort, wo Entscheidungen unter Unsicherheit getroffen werden – was in jedem Projekt der Fall ist.



Projektrisiken

Fehlende Ressourcen, unklare Ziele, Kommunikationsdefizite. Die häufigsten Projektrisiken sind keine technischen – sie sind organisatorischer und kommunikativer Natur.

Organisatorische Risiken

Wechselnde Prioritäten, fehlende Entscheidungsträger. Organisatorische Risiken sind oft die schwierigsten – weil sie außerhalb des direkten Einflussbereichs des Entwicklungsteams liegen.

Verantwortung im Software Engineering

Software Engineers gestalten Systeme, die das Leben von Menschen beeinflussen. Diese Gestaltungsmacht bringt Verantwortung mit sich – technisch, ethisch und gesellschaftlich.

Sicherheit

Software kann Menschen schützen oder gefährden. Sicherheit ist keine optionale Anforderung – sie ist eine grundlegende Pflicht gegenüber den Nutzern eines Systems.

Datenschutz

Systeme verarbeiten sensible Informationen über reale Menschen. Der Umgang mit personenbezogenen Daten ist eine ethische Frage, nicht nur eine rechtliche.

Nachhaltigkeit

Technische Entscheidungen haben ökologische und soziale Folgen. Ressourcenverbrauch, Energieeffizienz und gesellschaftliche Auswirkungen sind Teil des professionellen Kalküls.

Ethik

Software Engineers tragen Verantwortung für die Wirkung ihrer Systeme. Ethische Reflexion ist keine Zusatzaufgabe – sie ist integraler Bestandteil professionellen Software Engineerings.

Was Leser aus dem Buch mitnehmen

Das Buch vermittelt keine Rezepte – es entwickelt Denkfähigkeiten. Leser gewinnen ein Fundament für professionelles, verantwortungsvolles Software Engineering.

01

Systemisches Denken

Probleme in ihrem Kontext verstehen. Nicht isolierte Symptome behandeln, sondern Zusammenhänge erkennen und Ursachen adressieren.

02

Bessere Anforderungen

Stakeholder, Ziele und Konflikte strukturiert erfassen. Anforderungen als Modelle verstehen – nicht als unveränderliche Wahrheiten.

03

Bessere Modelle

Domäne, Objekte und Architektur klar beschreiben. Modelle als Kommunikationsmittel einsetzen – nicht als Selbstzweck.

04

Bessere Entscheidungen

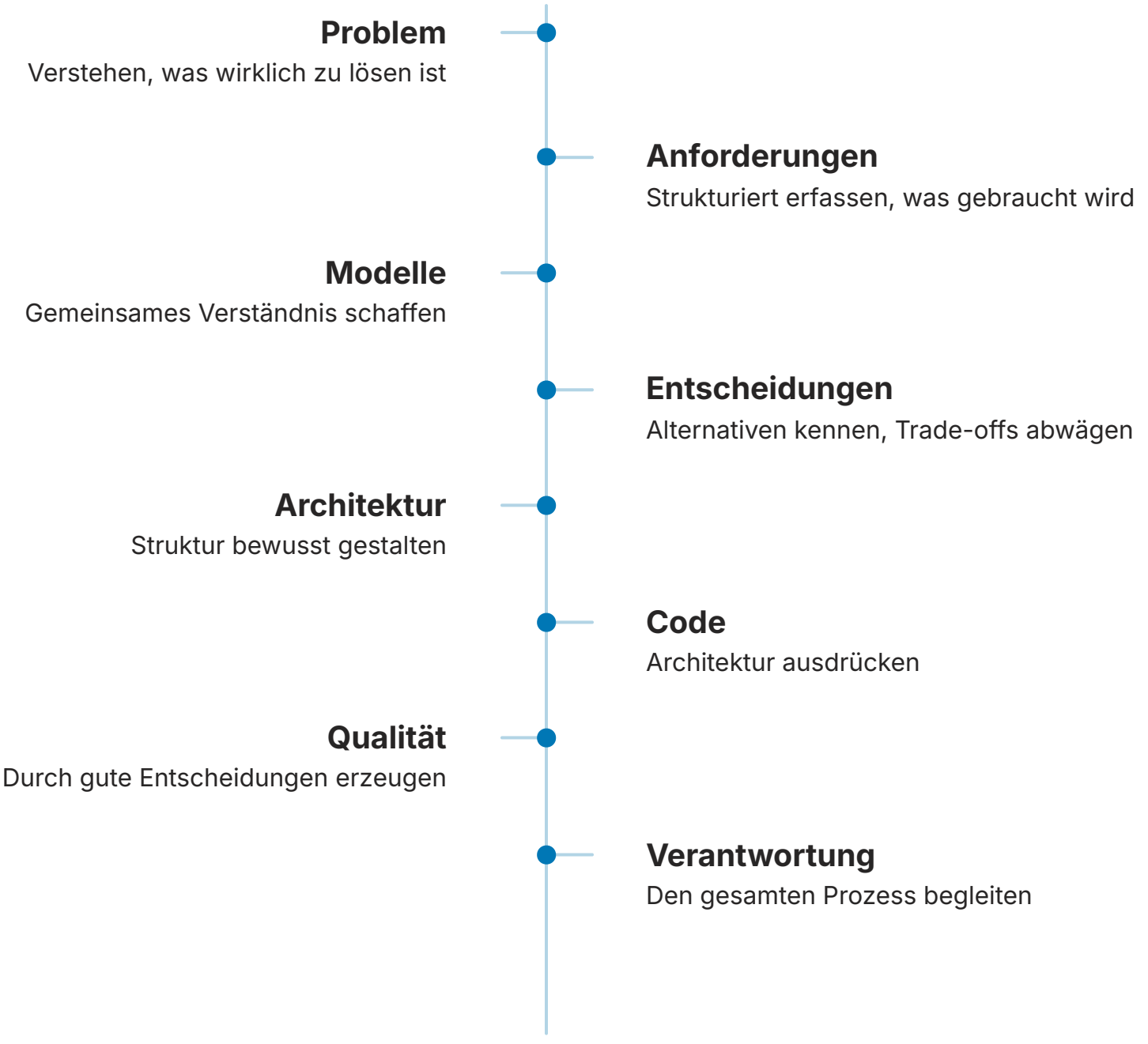
Alternativen kennen, Trade-offs bewusst abwägen. Entscheidungen begründen und dokumentieren – für das Team und die Zukunft.

05

Nachhaltige Softwareentwicklung

Qualität, Wartbarkeit und Verantwortung als Maßstab. Software entwickeln, die langfristig tragfähig, veränderbar und verantwortbar ist.

Fazit: Der eigentliche Wert entsteht vor dem ersten Code



> Wer diesen Weg bewusst geht, schreibt nicht nur besseren Code – er löst bessere Probleme. *Der eigentliche Wert entsteht lange vor dem ersten geschriebenen Code.*

Quellen und theoretische Grundlagen I

Software Engineering, Anforderungen und Architektur

Sommerville, Ian: *Software Engineering*. 10. Aufl. Pearson, 2020. Grundlage für das Verständnis von Software Engineering als systematische Disziplin zur Entwicklung langlebiger, zuverlässiger und wartbarer Softwaresysteme.

Pressman, Roger S.; Maxim, Bruce R.: *Software Engineering. A Practitioner's Approach*. 9. Aufl. McGraw-Hill, 2019. Grundlage für die praxisorientierte Sicht auf Softwareprozesse, Qualität, Projektarbeit und systematische Softwareentwicklung.

Wiegers, Karl; Beatty, Joy: *Software Requirements*. 3. Aufl. Microsoft Press, 2013. Grundlage für die Darstellung von Anforderungen, Stakeholdern, Zielen, Prioritäten und Konflikten.

Evans, Eric: *Domain-Driven Design. Tackling Complexity in the Heart of Software*. Pearson International, 2003. Grundlage für Domänenmodelle, gemeinsame Sprache und fachliche Modellierung komplexer Softwaresysteme.

Bass, Len; Clements, Paul; Kazman, Rick: *Software Architecture in Practice*. 4. Aufl. Addison-Wesley, 2021. Grundlage für die Sicht auf Architektur als Struktur, Qualitätsentscheidung und langfristig wirksame Gestaltungsdisziplin.

Fowler, Martin: *Patterns of Enterprise Application Architecture*. Pearson International, 2002. Grundlage für Architekturstrukturen, Schichten, Persistenz, Schnittstellen und Enterprise-Anwendungsdesign.

Quellen und theoretische Grundlagen II

Java, Objektorientierung, Qualität und technische Schulden

Bloch, Joshua: *Effective Java*. 3rd Edition. Addison-Wesley Professional, 2018. Grundlage für Java als Ausdrucksmittel professioneller Softwaregestaltung, insbesondere für lesbaren, robusten und wartbaren Code.

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. Grundlage für objektorientierte Entwurfsmuster, Zusammenarbeit von Objekten und wiederverwendbare Entwurfsentscheidungen.

Martin, Robert C.: *Clean Code. A Handbook of Agile Software Craftsmanship*. Pearson, 2008. Grundlage für Lesbarkeit, Wartbarkeit, Verantwortlichkeiten und professionelle Codequalität.

Martin, Robert C.: *Clean Architecture. A Craftsman's Guide to Software Structure and Design*. Pearson, 2017. Grundlage für die Trennung von Verantwortlichkeiten, Architekturgrenzen und langfristig wartbare Systemstrukturen.

Beck, Kent: *Test Driven Development. By Example*. Addison-Wesley Professional, 2002. Grundlage für Testen als Entwicklungs- und Erkenntnisprozess.

Fowler, Martin: *Refactoring. Improving the Design of Existing Code*. 2. Aufl. Pearson International, 2018. Grundlage für Refactoring als kontinuierliche Verbesserung bestehender Software.

Cunningham, Ward: „The WyCash Portfolio Management System". In: *ACM SIGPLAN OOPS Messenger* 4(2), 1992, S. 29–30. Grundlage für den Begriff und das Verständnis technischer Schulden.

Quellen und theoretische Grundlagen III

Entscheidungen, Verantwortung, Sicherheit und Betrieb

Brooks, Frederick P.: *The Mythical Man-Month. Essays on Software Engineering*. Anniversary Edition. Addison-Wesley Professional, 1995. Grundlage für die organisatorische und kommunikative Komplexität großer Softwareprojekte.

Boehm, Barry W.: *Software Engineering Economics*. Prentice Hall, 1981. Grundlage für die wirtschaftliche Sicht auf Software Engineering, Kosten, Risiken und technische Entscheidungen.

Boehm, Barry W.; Turner, Richard: *Balancing Agility and Discipline. A Guide for the Perplexed*. Addison-Wesley/Pearson Education, 2004. Grundlage für die Abwägung zwischen Flexibilität, Disziplin, Risiko und Projektkontext.

Humble, Jez; Farley, David: *Continuous Delivery. Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson International, 2010. Grundlage für Build-, Test-, Deployment- und Release-Prozesse.

Forsgren, Nicole; Humble, Jez; Kim, Gene: *Accelerate. The Science of Lean Software and DevOps*. IT Revolution Press, 2018. Grundlage für moderne DevOps-Praktiken, Softwarelieferfähigkeit und organisatorische Leistungsfähigkeit.

OWASP Foundation: *OWASP Top 10*. 2021. Grundlage für zentrale Sicherheitsrisiken moderner Webanwendungen.

Europäisches Parlament und Rat der Europäischen Union: *Verordnung (EU) 2016/679. Datenschutz-Grundverordnung*. 2016. Grundlage für Datenschutz als rechtlichen und verantwortungsethischen Rahmen.

Bundesamt für Sicherheit in der Informationstechnik: *IT-Grundschutz-Kompendium. Edition 2023*. 2023. Grundlage für systematische Informationssicherheit und organisatorische Schutzmaßnahmen.