Metaprogrammierung

Ein multiparadigmatischer Ansatz in der Softwaretechnik

Kyjiw, 2015–2025

Аннотация

Alle Programme sind Daten. Einige Daten werden als Werte interpretiert, andere als Typen dieser Werte, wieder andere als Anweisungen zur Verarbeitung der ersten beiden. Alle Programmierparadigmen und -techniken sind lediglich Methoden zur Erzeugung von Metadaten, die Regeln und den Ablauf der Datenverarbeitung festlegen. Multiparadigmatisches Programmieren vereint die Stärken aller Paradigmen und formt daraus syntaktische Konstrukte, mit denen sich Fachgebiete klar und komfortabel beschreiben lassen. Dabei spiegeln wir hochgradig spezialisierte DSLs (Domänensprachen) in niedrigstufige Maschinenanweisungen über viele Abstraktionsschichten hinweg. Entscheidend ist nicht, einer bestimmten Paradigme fanatisch zu folgen, sondern die jeweilige Aufgabe möglichst effizient für die maschinelle Ausführung darzustellen. Effizient bedeutet: mit möglichst wenigen Schichten und Abhängigkeiten, zugleich aber verständlich für den Menschen, wartbar und leicht anpassbar – unter Berücksichtigung von Zuverlässigkeit und Testbarkeit des Codes, Erweiterbarkeit, Wiederverwendbarkeit sowie Klarheit und Flexibilität der Metadatenstrukturen auf allen Ebenen.

Wir sind überzeugt, dass dieser Ansatz sowohl schnelle erste Ergebnisse bei der Entwicklung einzelner Aufgaben ermöglicht, als auch bei umfangreichen Änderungen in fortgeschrittenen und komplexen Projektphasen keine Geschwindigkeitseinbußen verursacht. Wir wollen Techniken und Prinzipien aus verschiedenen Paradigmen durch die Linse des Metaprogrammierens betrachten – nicht um das Software Engineering grundlegend zu verändern, sondern um sein Verständnis für kommende Generationen von Entwicklern zu erweitern.

Inhaltsverzeichnis

4.1. Was ist ein Technologiestack?
4.2. Entwicklungsumgebung und Debugging
4.3. Iteration: Rekursion, Iteratoren und Generatoren
4.4. Bausteine einer Anwendung: Dateien, Module, Komponenten
4.5. Objekt, Prototyp und Klasse
4.6. Partielle Anwendung und Currying, _Pipe_ und _Compose_
4.7. Chaining von Methoden und Funktionen
4.8. Mixins
4.9. Abhängigkeiten und Bibliotheken

5. **Verbreitete Programmierparadigmen**
5.1. Imperativer und deklarativer Ansatz
5.2. Strukturierte und unstrukturierte Programmierung
5.3. Prozedurale Programmierung
5.4. Funktionale Programmierung
5.5. Objektorientierte Programmierung
5.6. Prototypenbasierte Programmierung

6. **Antipatterns**
6.1. Allgemeine Antipatterns für alle Paradigmen
6.2. Prozedurale Antipatterns
6.3. Objektorientierte Antipatterns
6.4. Funktionale Antipatterns

7. **Entwicklungsprozess**
7.1. Softwarelebenszyklus, Domänenanalyse
7.2. Konventionen und Standards
7.3. Tests: Unit Tests, System- und Integrationstests
7.4. Code-Review und Refactoring
7.5. Ressourcenabschätzung, Plan und Zeitplan
7.6. Risikoanalyse, Schwachstellen, nicht-funktionale Anforderungen
7.7. Koordination und Prozessanpassung
7.8. Continuous Deployment und Continuous Delivery
7.9. Multidimensionale Optimierung

8. **Fortgeschrittene Konzepte**
   8.1. Events, Timer und _EventEmitter_
   8.2. Introspektion und Reflexion
   8.3. Serialisierung und Deserialisierung
   8.4. Reguläre Ausdrücke
   8.5. Memoisierung
   8.6. Factory und Pool
   8.7. Typisierte Arrays
   8.8. Projektionen
   8.9. I/O und Dateien

9. **Architektur**
   9.1. Dekomposition, Benennung und Verknüpfung
   9.2. Interaktion zwischen Softwarekomponenten
   9.3. Kopplung über Namensräume
   9.4. Interaktion mit Funktionsaufrufen und Callbacks
   9.5. Interaktion mit Events und Nachrichten
   9.6. Schnittstellen, Protokolle und Verträge
   9.7. Onion-Architektur bzw. Schichtenarchitektur

10. **Grundlagen paralleler Berechnungen**
   10.1. Asynchrones Programmieren
   10.2. Paralleles Programmieren, gemeinsamer Speicher (Shared Memory) und Synchronisationsprimitive
   10.3. Asynchrone Primitives: Thenable, Promise, Future, Deferred
   10.4. Coroutinen, Goroutinen, async/await
   10.5. Adapter zwischen asynchronen Verträgen
   10.6. Asynchrone und parallele Interoperabilität
   10.7. Nachrichtenbasierter Ansatz und Actor-Modell
   10.8. Asynchrone Warteschlangen und Sammlungen
   10.9. lockfreie Datenstrukturen

11. **Erweiterte Programmierparadigmen**
   11.1. Generische Programmierung

1. Einführung

Die **ständige Reflexion über die eigene Tätigkeit** – selbst über scheinbar triviale Aufgaben – sollte eine _normale Gewohnheit_ eines jeden Ingenieurs sein und ihn sein ganzes Berufsleben begleiten. Die Angewohnheit, Gedanken **schriftlich festzuhalten** und Formulierungen immer wieder zu präzisieren, hilft dabei enorm.

Dieser Text entstand aus meinen verstreuten Notizen, die ich im Laufe der Jahre sammelte und **Dutzende Male kritisch überarbeitete**. Nicht selten widersprach ich mir selbst, sobald ein Abschnitt einige Zeit „ablag“

– also schrieb ich so lange um, bis ich nach längeren Reifephasen **vollständig hinter jeder Zeile** stehen konnte. Mein erklärtes Ziel war maximale **Kürze**: Ganze Passagen wurden mehrfach gekürzt, weil ich sie knapper ausdrücken konnte.

Eine erkennbare **Struktur** – Kapitelübersicht und Inhaltsverzeichnis – formte sich erst nach dem ersten Jahr meiner Lehrtätigkeit. Im zehnten Jahr fasste ich den Entschluss, das Material nicht mehr nur als frei zugängliche Video- Vorlesungen (wie schon etwa fünf Jahre zuvor), sondern zusätzlich als **Fließtext** zu veröffentlichen. Dadurch kann jede*r aus der **Metarchia- Community** an der Entstehung des Buchs mitwirken, Tipp- und Sachfehler lassen sich dank der Leserschaft schnell finden, und vielen ist die Textform schlicht angenehmer.

Die jeweils **aktuelle Version** liegt unter <https:// github.com/ HowProgrammingWorks/ Book> und wird fortlaufend ergänzt. _Bug- Reports_ oder Korrekturvorschläge bitte als **Issue** auf Englisch einreichen: <https:// github.com/ HowProgrammingWorks/ Book/ issues> Neue Ideen gern in den **Discussions** (beliebige Sprache): <https:// github.com/ HowProgrammingWorks/ Book/ discussions> Eigene Ergänzungen und Verbesserungen bitte als **Pull Request** an das Buch- Repository schicken.

Programmieren ist die Kunst und Ingenieurwissenschaft, Probleme mithilfe von Rechentechnik zu lösen.

Es ist **Ingenieurwesen**, weil es vorhandenes Wissen nutzbar macht, und **Kunst**, weil Programmieren heute leider nicht allein auf Wissen basiert, sondern oft auf Intuition und schwer erklärbarer persönlicher Erfahrung beruht. Die Aufgabe einer*eines* Entwickler*in besteht nicht darin, ein **einzig mathematisch korrektes** Ergebnis zu finden, sondern einen **allgemeinen Lösungsmechanismus**, der in begrenzter Zeit für eine möglichst große Klasse von Problemen **akzeptable Ergebnisse** liefert – kurz: eine abstrahierte **Klasse von Lösungen**.

Zwar setzen nicht alle **Programmierparadigmen** auf strikte Schrittfolgen, doch sowohl die **physische Realisierung** von Rechenmaschinen als auch die Natur menschlichen Denkens sind grundsätzlich **schrittweise**. Die Schwierigkeit liegt darin, dass sich viele dieser Schritte nicht auf reine CPU- Operationen reduzieren lassen, sondern externe **Ein-/ Ausgabe** sowie Sensorik – und damit den äußeren Kosmos und den Menschen – einbeziehen. Diese

Unwägbarkeiten verhindern einen lückenlos **mathematischen Beweis** für die Korrektheit beliebiger Lösungswege, geschweige denn deren Ableitung aus Axiomen, wie es für die exakten Wissenschaften typisch wäre. Einzelne **Algorithmen** jedoch _können und sollen_ analytisch hergeleitet werden, wenn sie sich auf **reine Funktionen** zurückführen lassen: Funktionen, die für dieselben Eingaben **immer** exakt dasselbe Ergebnis liefern, keinerlei **Zustand** besitzen und nur andere reine Funktionen aufrufen.

Von der Mathematik hat die Informatik die Fähigkeit geerbt, **exakte Lösungen** analytisch zu finden, und der Rechner selbst arbeitet strikt innerhalb eines **formalen mathematischen Apparats**. Doch das Schreiben von Code lässt sich nicht vollständig formalisieren; wir treffen Entscheidungen unter erheblicher Unsicherheit und konstruieren Software **ingenieurmäßig**. Entwickelnde sind zudem an **Projektzeiten** gebunden. Deshalb verringern wir Ungewissheit, indem wir **konstruktive Beschränkungen** einführen, die sich nicht streng aus der Aufgabe ableiten lassen, sondern auf Intuition und Erfahrung beruhen.

Fehlt ein optimaler Algorithmus, darf die*der* Programmierer*in **jeden gangbaren Weg** wählen, der akzeptable Resultate in angemessener Zeit liefert **und** sich solange implementieren lässt, wie die Aufgabe relevant bleibt. Wir müssen daher nicht nur die Nähe zur optimalen Lösung bewerten, sondern ebenso **Kompetenz**, Werkzeugbeherrschung und sonstige Ressourcen berücksichtigen. Der Zugang zu Wissen und bestehenden Lösungen ist zudem durch **Urheberrecht, Lizenzmodelle** und proprietäre Dokumentation eingeschränkt – nicht nur bei Software, sondern auch bei Büchern, Videos und Kursmaterialien. Das alles bremst die Branche, doch der **Wissenszugang wächst unumkehrbar**: Durch Popularisierer, Enthusiasten und die Bewegung **Freier Software** sickern immer mehr Kenntnisse frei ins Netz.

1.1. Lernansätze beim Programmieren

Viele denken, dass die wichtigste Fähigkeit eines Programmierers das Schreiben von Code ist. Tatsächlich lesen und korrigieren Programmierer jedoch viel öfter Code. Die Hauptkriterien für qualitativ hochwertigen Code sind Verständlichkeit, Lesbarkeit und Einfachheit. Wie Harold Abelson sagte: „Programme sollten für Menschen geschrieben werden, die sie lesen, nicht für Maschinen, die sie ausführen."

Die wichtigsten Fähigkeiten eines Programmierers sind das Lesen und Korrigieren von Code.

Jedes Thema enthält Beispiele für guten und schlechten Code. Diese Beispiele stammen aus der Programmierpraxis und der Projektüberprüfung. Die absichtlich vorbereiteten schlechten Codebeispiele funktionieren zwar, sind aber voller Anti- Patterns und Probleme, die erkannt und behoben werden müssen. Bereits die allererste praktische Aufgabe im Kurs besteht darin, Code zu verbessern und seine Lesbarkeit zu erhöhen. Gibt man Anfängern klassische Aufgaben (z.B. eine Funktion, einen Algorithmus oder eine Klasse nach Vorgabe zu schreiben), setzen sie diese meist nicht optimal um, verteidigen ihren Code aber, weil es das erste ist, was sie geschrieben haben. Ist die Aufgabe jedoch, „fremden schlechten Code zu nehmen, Probleme zu finden und sie zu beheben“, also den Code nicht von Grund auf neu zu schreiben, sondern schrittweise zu verbessern und diese Schritte bewusst nachzuvollziehen, wird ein kritischer Denkprozess angestoßen.

Schlechten Code zu korrigieren ist eine der effektivsten Lernmethoden.

Einsteiger erhalten Beispiele für Code- Reviews und versuchen anschließend, ihre eigenen Aufgaben analog zu verbessern. Solche Iterationen wiederholen sich häufig und fördern dauerhaft eine kritische Herangehensweise. Ideal ist es, wenn ein Mentor die Verbesserungen beobachtet und gezielt korrigierend eingreifen oder Hinweise geben kann. Der Mentor sollte jedoch keinesfalls die Arbeit für den Anfänger erledigen, sondern ihn vielmehr dazu anregen, wie man über Programmierung nachdenken und nach Lösungen suchen sollte.

Ein Mentor ist ein unverzichtbarer Begleiter bei der beruflichen Weiterentwicklung.

Anschließend folgen Aufgaben, bei denen der eigene Code geschrieben wird. Wir empfehlen Anfängern ausdrücklich, ihre Lösungen untereinander auszutauschen, um Peer- Reviews durchzuführen. Zuvor sollten jedoch Linters und Formatierer eingesetzt werden, um den Code syntaktisch zu analysieren, Fehler zu finden und problematische Stellen anhand vieler Codevorlagen aufzudecken. Ziel ist es, dass der Kollege den Gedankengang versteht – und keine Zeit mit Syntax oder Formatierung

verliert.

Nutzt freundliches Code-Review, Peer-Review, Linters und Formatierer.

Im nächsten Schritt folgen Übungen zur Entkopplung verschiedener Abstraktionsebenen und anschließend von Modulen. Ziel ist es, dass möglichst wenig Wissen über Datenstrukturen eines Programmteils in einem anderen notwendig ist. Sprachfanatismus wird reduziert, indem von Anfang an mehrere Programmiersprachen parallel erlernt und gegenseitig übersetzt werden. Zwischen **JavaScript** und **Python** ist dies sehr einfach, bei **C** etwas schwieriger – aber auf diese drei Sprachen darf im Kurs keinesfalls verzichtet werden.

Vermeidet von Anfang an jeglichen Fanatismus: sprachlich, framework-basiert oder paradigmenbezogen.

Framework-Fanatismus wird reduziert, indem Anfängern die Nutzung von Bibliotheken und Frameworks untersagt und der Fokus auf möglichst nativen, abhängigkeitfreien Code gelegt wird. Paradigmenfanatismus wird reduziert, indem versucht wird, prozedurale, funktionale, objektorientierte, reaktive und zustandsbasierte Programmierung zu kombinieren. Wir werden zeigen, wie diese Kombinationen helfen, Muster und Prinzipien aus GoF und SOLID zu vereinfachen.

Ein weiterer wichtiger Bestandteil des Kurses ist das Studium von Anti-Patterns und Refactoring. Zunächst geben wir einen Überblick, anschließend wird mit echten Codebeispielen aus realen Projekten geübt.

1.2. Beispiele in JavaScript, Python und C

Wir werden Codebeispiele in verschiedenen Programmiersprachen schreiben, wobei der Fokus nicht auf den besten, schönsten oder schnellsten Sprachen liegt, sondern auf denen, ohne die man nicht auskommt. Wir nehmen **JavaScript** als die am weitesten verbreitete Sprache, **Python** , weil es Bereiche gibt, in denen man ohne sie nicht auskommt, und **C** als eine Sprache, die Assembler sehr nahe kommt, nach wie vor relevant ist und modernen Sprachen sowohl im Hinblick auf Syntax als auch auf konzeptionelle Ideen stark beeinflusst hat. Alle drei Sprachen sind weit entfernt von der Sprache meiner Träume – aber es ist

das, womit wir arbeiten.

Auf den ersten Blick wirkt **Python** sehr verschieden von **JavaScript** und anderen C-ähnlichen Sprachen. Doch dieser Eindruck täuscht: Wir werden zeigen, dass **Python** **JavaScript** sehr ähnlich ist, insbesondere aufgrund der Typensysteme, Datenstrukturen und eingebauten Collections. Die syntaktischen Unterschiede, etwa bei der Blockstruktur über Einrückungen versus geschweifte Klammern **{}**, stechen zwar ins Auge, sind in der Praxis jedoch nicht besonders entscheidend. Zwischen **JavaScript** und **Python** gibt es deutlich mehr Gemeinsamkeiten als zwischen beiden und **C**.

Wir beginnen nicht mit dem Erlernen der Syntax, sondern sofort mit dem Lesen von fehlerhaftem Code und der Fehlersuche. Sehen wir uns die folgenden Codefragmente an. Der erste ist in **JavaScript**:

```
let first_num = 2;
let secord_num = 3;
let sum = firstNum + secondNum;
console.log({ sum });
```

Versuche zu verstehen, was hier geschrieben steht, und wo mögliche Fehler liegen. Vergleiche diesen Code dann mit der entsprechenden Version in **C**:

```
#include <stdio.h>

int main() {
  int first_num = 2;
  int secord_num = 3;
  int sum = firstNum + secondNum;
  printf("%d\n", sum);
}
```

Die Fehler sind hier identisch. Sie können auch von jemandem erkannt werden, der keinerlei Programmierkenntnisse hat – sofern er sich mit dem Code beschäftigt. Der nächste Codeausschnitt ist in **Python**, erfüllt die gleiche Aufgabe und enthält die gleichen Fehler:

```
first_num = 2;
secord_num = 3;
```

```
sum = firstNum + secondNum;
print({ 'sum': sum  });
```

Im weiteren Verlauf werden wir häufig Codebeispiele aus verschiedenen Sprachen vergleichen, Fehler finden und korrigieren, den Code optimieren und dabei vor allem die Lesbarkeit und Verständlichkeit verbessern.

1.3. Modellierung: Abstraktionen und Wiederverwendung

Das Herzstück des Programmierens ist die **Modellierung** – also die Erstellung eines Domänenmodells beziehungsweise eines Modells von Objekten und Prozessen im Arbeitsspeicher des Computers. Programmiersprachen liefern dafür Syntax elemente, mit denen sich **Randbedingungen** für Modelle formulieren lassen. Jede zusätzliche Struktur, die eingeführt wird, um Funktionalität zu erweitern, bringt allerdings weitere Einschränkungen mit sich. Eine höhere **Abstraktionsebene** kann umgekehrt einige dieser Beschränkungen aufheben und die Komplexität des Modells sowie des Codes reduzieren. Wir balancieren daher ständig zwischen dem Ausweiten von Funktionen und deren Zusammenfassung zu einem allgemeineren Modell – und dieser Prozess sollte iterativ immer wieder durchlaufen werden.

Erstaunlich ist, dass Menschen mithilfe von Modellen und Abstraktionen Probleme lösen können, deren Komplexität ihre Gedächtnis- und Denk kapazität eigentlich übersteigt. Wie **präzise** ein Modell ist, bestimmt seinen Nutzen für Entscheidungen und Steuerungsmaßnahmen. Ein Modell bleibt stets unvollständig und spiegelt nur einen kleinen Ausschnitt der Realität wider – eine oder mehrere ihrer Seiten. Unter klar begrenzten Einsatzbedingungen kann ein Modell jedoch dem realen Objekt einer Domäne praktisch **gleichwertig** sein. Es gibt physikalische, mathematische, Simulations- und viele weitere Modelle; uns interessieren hier vor allem **Informations- und algorithmische Modelle**.

Abstraktion ist eine Verallgemeinerung, die viele unterschiedliche, aber ähnliche Fälle zu einem einzigen Modell zusammenführt. Uns beschäftigen Datenabstraktionen und abstrakte Algorithmen. Die einfachsten algorithmischen Abstraktionen sind **Schleifen** (iterative Verallgemeinerung) und **Funktionen** (Prozeduren, Routinen). Eine Schleife beschreibt mit einem einzigen Befehlsblock viele Iterationen,

indem sie diesen mehrfach mit unterschiedlichen Variablenwerten ausführt. Funktionen werden ebenfalls vielfach mit unterschiedlichen Argumenten aufgerufen. Typische Datenabstraktionen sind Arrays, assoziative Arrays, Listen, Mengen usw. In Anwendungen fasst man Abstraktionen zu **Schichten** (Abstraktionsebenen) zusammen:

- Niedriglevel-Abstraktionen sind direkt in der Programmiersprache eingebaut (Variablen, Funktionen, Arrays, Events).
- Höhere Abstraktionen finden sich in Plattformen, Runtimes, Standard- und externen Bibliotheken oder werden aus einfachen Abstraktionen selbst aufgebaut.
- Sie heißen „abstrakt“, weil sie **generische** Aufgaben lösen, also nicht domänenspezifisch sind.

Das **Aufbauen von Abstraktionsschichten** ist vielleicht die wichtigste Programmieraufgabe; davon hängen Flexibilität, Änderungsaufwand, Integrationsfähigkeit und Lebensdauer einer Software ab. Alle Schichten, die nicht an eine Fachdomäne gebunden sind, nennen wir **Systemschichten**. Darüber legt der Entwickler **Applikationsschichten**, deren Abstraktionsgrad und Universalität abnehmen, je konkreter sie auf Aufgaben zugeschnitten werden.

Abstraktionen verschiedener Ebenen können im selben **Adressraum** (einem Prozess bzw. einer Anwendung) oder in getrennten laufen. Ihre Trennung und Interaktion erreicht man über APIs, Modularität, Komponentenansätze – oder schlicht Disziplin, indem man direkte Aufrufe „mitten aus“ einer Komponente „mitten in“ eine andere vermeidet, sofern Sprache oder Plattform das nicht erzwingen. Das gilt selbst innerhalb eines Prozesses, wo theoretisch jede Funktion überall aufrufbar wäre. Ziel ist es, die **Kopplung** zwischen Schichten und Komponenten zu verringern, damit sie austauschbar, wiederverwendbar und separat entwickelbar bleiben. Zugleich erhöht man die **Kohäsion** innerhalb einer Schicht oder eines Moduls, was Lesbarkeit, Verständlichkeit und Änderbarkeit verbessert. Gelingt es, Abstraktionsebenen sauber zu trennen und Module so klein zu halten, dass ein Entwickler sie vollständig überblicken kann, wird der Entwicklungsprozess skalierbar, steuerbar und vorhersagbar. Dieses Prinzip liegt auch der **Microservices-Architektur** zugrunde, gilt aber genauso für Module im selben Prozess.

Generell gilt: Je besser ein System **verteilt** ist, desto besser ist es auch zentralisierbar. Aufgaben werden auf der **passenden Ebene** gelöst, wo ausreichende Informationen vorliegen; starre Verbindungen zwischen Modellen unterschiedlicher Abstraktion entfallen. Dadurch

kommt es zu keiner unnötigen Eskalation von Problemen nach oben, Entscheidungs knoten werden nicht „überhitzt“, Daten verkehr bleibt minimal und die Reaktionsgeschwindigkeit steigt.

1.4. Algorithmus, Programm, Syntax, Sprache

Es gibt viele Begriffe, die mit der Programmierung verbunden sind. Um Klarheit zu schaffen, sollten wir den Unterschied zwischen ihnen verstehen. Ein informelles Verständnis ist dabei nützlicher als eine rein formale Definition.

Algorithmus (Algorithm)

Ein Algorithmus ist noch kein Programm, sondern eine formal beschriebene Idee zur Lösung einer Aufgabe. Die Beschreibung muss so verfasst sein, dass andere sie verstehen, überprüfen und umsetzen können. Einen Algorithmus kann man nicht ausführen; er muss in Code einer Programmiersprache übertragen werden. Ein Algorithmus enthält die Beschreibung von Operationen und kann auf verschiedene Arten notiert werden: als Formel, als Blockdiagramm oder als Schrittfolge in natürlicher Sprache. Er ist immer auf eine bestimmte Klasse von Aufgaben beschränkt, die er in endlicher Zeit löst. Häufig können wir einen Algorithmus vereinfachen und optimieren, indem wir die Aufgabenklasse einschränken. Wenn wir zum Beispiel von der Addition ganzer und gebrochener Zahlen auf die Addition nur ganzer Zahlen übergehen, können wir eine effizientere Implementierung erreichen. Ebenso können wir die Aufgabenklasse erweitern, indem wir zusätzlich Zeichenketten, die Zahlen darstellen, als Eingabe zulassen; das macht den Algorithmus allgemeiner, aber weniger effizient. Wir müssen entscheiden, was genau wir optimieren. In diesem Fall ist es besser, den Algorithmus aufzuteilen: Einer wandelt alle Zahlen in einen Datentyp um, der andere führt die Addition durch.

Beispiel zur Implementierung des GGT-Algorithmus (Euclid)

In **JavaScript** kann der euklidische Algorithmus zur Bestimmung des größten gemeinsamen Teilers (GGT) so aussehen:

```
const gcd = (a, b) => {
```

```
  if (b === 0) return a;
  return gcd(b, a % b);
};
```

Oder noch kürzer:

```
const gcd = (a, b) => (b === 0 ? a : gcd(b, a % b));
```

Dieser einfache Algorithmus ist **rekursiv**: Er ruft sich selbst auf, bis **b** den Wert **0** erreicht. Für Algorithmen können wir die Rechenkomplexität bestimmen und sie nach dem benötigten Zeit- und Speicheraufwand klassifizieren.

Programm (Program)

Im vorherigen Beispiel hatten wir eine Funktion – nur einen Teil eines Programms. Damit sie wirkt, muss man sie aufrufen und Daten übergeben. Programmcode und Daten, die zu einer Einheit zusammengeführt sind, bilden ein **Programm**. Niklaus Wirth verdeutlichte das im Buchtitel: „Algorithmen + Datenstrukturen = Programme“. In den ersten Jahrzehnten der Softwareindustrie zeigte sich, dass Datenstrukturen nicht weniger wichtig sind als Algorithmen. Linus Torvalds meint: „Schlechte Programmierer kümmern sich um Code. Gute Programmierer kümmern sich um Datenstrukturen und ihre Beziehungen.“ Die Wahl der Datenstrukturen bestimmt weitgehend den Algorithmus und dessen Komplexität; über die Daten im Speicher versteht der Programmierer die Aufgabe besser als über die reine Abfolge von Operationen.

Eric Raymond formulierte: „Intelligente Datenstrukturen und dummer Code funktionieren viel besser als andersherum.“

Code als gemeinsame Sprache

Linus Torvalds sagte auch: „Reden ist billig. Zeig mir den Code.“ Programmcode ist eindeutig und ermöglicht es Entwicklern, sich zu verständigen, selbst wenn natürliche Sprachen wegen Mehrdeutigkeit versagen.

Ingenieurwesen (Engineering)

Ingenieurwesen ist die praktische Nutzung vorhandener Ressourcen durch Wissenschaft, Technik, Methoden, Organisation und Wissen. Schon früh war mir wichtig, dass Code Menschen nützt, ihr Leben verbessert und lange lebt. Wettbewerbs- oder Lehrbeispiele erschienen mir künstlich; ich wollte Software schreiben, die täglich genutzt wird: Datenbankanwendungen, Netz- und Kommunikationsprogramme, Steuerungssoftware und Entwicklerwerkzeuge.

Wie in anderen Ingenieurdisziplinen zählt der Nutzen für den Menschen mehr als die Eleganz des Konzepts. Wenn wissenschaftliche Erkenntnisse fehlen, greift man auf Intuition, Erfahrung und Versuch-und-Irrtum zurück. Das führt zu vielen unterschiedlichen und teils widersprüchlichen Lösungen. Niklaus Wirth bemerkte: „Programme werden langsamer, schneller als Hardware schneller wird“, und oft ist Neuschreiben einfacher als Reparieren.

Software-Engineering (Software engineering)

Die Anwendung ingenieurmäßiger Ansätze auf Software umfasst Architektur, Forschung, Entwicklung, Test, Bereitstellung und Wartung. Die Branche nutzt Hilfspraktiken, um Produkte verlässlich genug zu machen, um Gewinn zu bringen, aber nicht so perfekt, dass keine neuen Versionen nötig wären.

„Die meisten Programme ähneln ägyptischen Pyramiden – Millionen Steine übereinander ohne strukturelle Geschlossenheit, bloß durch Kraft und unzählige Arbeiter errichtet.“ – Alan Kay

Programmierung (Programming)

Programmierung ist die Kunst und das Ingenieurhandwerk, Aufgaben mit Rechentechnik zu lösen. Die Hardware beeinflusst stark, welche Paradigmen effizient sind – in Bezug sowohl auf Entwicklungsaufwand als auch auf Laufzeitressourcen.

Codierung (Coding)

Beschränkt man sich auf das Schreiben von Quellcode nach einer fertigen

Spezifikation, spricht man von **Codierung**. Entwicklung lässt sich in Entwurf und Codierung trennen; dennoch entsteht oft Code ohne Spezifikation – als Prototyp, MVP oder Pilot. Ihr Nutzen liegt in der Überprüfung von Hypothesen zu Anwendernutzen oder Wirtschaftlichkeit.

Programmierer verwechseln manchmal Prototyp und Produkt: Man erhält entweder einen Prototyp, der wie ein Produkt gepflegt werden muss, oder ein Produkt, das prototypisch zusammengestrickt ist. Die Branche lebt dennoch von Enthusiasten:

„Die meisten guten Programmierer programmieren nicht wegen Geld oder Anerkennung, sondern weil Programmieren Spaß macht." – Linus Torvalds

Entwickler vs. Programmierer

Es gibt Befürworter für beide Bezeichnungen. Häufig ist die Selbstbezeichnung **Entwickler** oder **Programmierer** mit einem besonderen beruflichen Stolz verbunden, nicht selten sogar mit einer gewissen Überheblichkeit gegenüber den Anhängern der anderen Bezeichnung. Es wäre sinnvoll, diese Berufe ungefähr so zu unterscheiden, wie sich die Berufe von Fahrer und Kfz- Mechaniker unterscheiden haben. Natürlich kann der Mechaniker behaupten, dass Fahrer keine Ahnung von Fahrzeugen haben, aber im Alltag werden die Menschen von Fahrern transportiert.

Ähnlich ist es in der IT: Der Programmierer sollte sich auf Abstraktionen und die Entwicklung von Softwarekomponenten konzentrieren, während sich der Entwickler auf die Anwendung fertiger Komponenten zur Lösung konkreter Aufgaben spezialisiert. Das erfordert wiederum andere Kenntnisse und Fähigkeiten außerhalb der Programmierung.

Komplexität und Einfachheit

Streben wir nach Programmen, die für Anwender und für uns als Entwickler einfach zu verstehen sind. Wenn wir unter Zeitdruck ineffizienten oder schwer verständlichen Code schreiben müssen, sollten wir eine Überarbeitung, ein Refactoring und eine Optimierung einplanen, bevor wir die Struktur vergessen oder unsere Ideen zur Verbesserung verblassen. Die Ansammlung solcher Probleme im Code nennt man **technische Schuld**. Sie macht Programme nicht nur unflexibler und schwerer verständlich, sondern sorgt auch dafür, dass neue Kollegen

beim Lesen des Codes schlechte Praktiken übernehmen. Die Einfachheit in der Lösung komplexer Aufgaben ist das Ziel eines guten Programmierers, das Verbergen dieser Komplexität hinter Abstraktionen ist die Methode eines erfahrenen Ingenieurs.

„Ich habe mir immer gewünscht, mein Computer wäre so einfach zu bedienen wie mein Telefon. Mein Wunsch ist in Erfüllung gegangen: Jetzt weiß ich nicht mehr, wie ich mein Telefon bedienen soll.“ – Bjarne Stroustrup

1.5. Декомпозиция и разделение ответственности

No translation

1.6 Überblick über den Beruf Softwareingenieur

Rund um das Programmieren hat der Mensch – wie um jede andere Tätigkeit – im Laufe der Zeit eine Fülle von Vorurteilen und Irrtümern aufgebaut. Die wichtigste Quelle solcher Probleme ist die **Terminologie**: Unterschiedliche **Paradigmen**, Programmiersprachen und **Ökosysteme** bringen ihren eigenen Wortschatz mit, der nicht nur gegenseitig widersprüchlich ist, sondern oft schon innerhalb einer Community unlogisch wirkt. Hinzu kommt, dass viele autodidaktische Entwickler ihre ganz eigene, unverwechselbare Terminologie erfinden – häufig doppeln sich dadurch Konzepte nur unter neuem Namen.

Auch übereifrige **Marketingabteilungen** wirken destruktiv auf die **IT-Branche**, indem sie offensichtliche Sachverhalte verdrehen, Konzepte unnötig verkomplizieren und so einen endlosen Strom an Problemen erzeugen, auf dem große Teile des Softwaregeschäfts überhaupt erst beruhen. Um Nutzer- und Entwickler-gemeinschaften an sich zu binden, entwerfen Branchengiganten verlockende, scheinbar stimmige Konzepte, die letztlich zu **Inkompatibilitäten**, einem „Krieg der Standards“ und zu deutlicher **Vendor-Lock-in** führen. Solche Gruppierungen besetzen über Jahre – manchmal Jahrzehnte – die Aufmerksamkeit und Budgets der Menschen. Getrieben von Stolz und Eitelkeit verbreiten einige Entwickler selbst fragwürdige oder von vornherein sackgassenartige Ideen; wirklich gutes Software-Engineering lohnt sich für den Hersteller schließlich selten.

Zwar sieht die Lage im Bereich **Freie Software** und **Open-Source-Code** erheblich besser aus, doch sind dezentrale Enthusiasten zu zersplittert, um der massiven Propaganda der Großunternehmen wirksam entgegenzutreten.

Sobald sich eine Technologie oder ein Ökosystem genügend weit entwickelt hat, um darauf hochwertige Lösungen aufzubauen, veraltet es unweigerlich, wird vom Hersteller nicht mehr gepflegt – oder es wächst sich zu einer unbeherrschbaren Komplexität aus. In meiner Erinnerung haben sich bereits mehr als fünf solcher Technologie-Ökosysteme vollständig abgelöst.

1.7 Überblick über Programmierparadigmen

Ein **Mathematiker** betrachtet ein Programm als **Funktion**, die sich in einfachere Funktionen **dekomponieren** lässt, sodass das Gesamtprogramm eine **Komposition** dieser Teilfunktionen bildet. Vereinfacht gesagt ist ein Programm dann eine komplexe Formel – ein **Daten-Transformator**, der am Eingang die Aufgabenparameter erhält und am Ausgang die Lösung liefert. Nicht jeder Entwickler kennt diese Sichtweise; sie ist nicht perfekt, aber nützlich, um das eigene Tun neu zu reflektieren.

Verbreiteter ist die gegenteilige Perspektive, die sich direkt aus der **Praxis des Programmierens** ergibt. Hier entwickelt man Anwendungen ausgehend von den Bedürfnissen der **Benutzer**, von **UI-Mock-ups** sowie von den verfügbaren Werkzeugen, Sprachen, Plattformen und Bibliotheken. Das Ergebnis ist weniger eine Programm-Funktion als vielmehr ein großes **Zustandssystem**, in dem eine **kombinatorische Explosion** von Übergängen entsteht und dessen Verhalten selbst für die Autorin oder den Autor schwer vorherzusagen ist – geschweige denn für den Nutzer.

Diesen scheinbar problematischen Ansatz darf man jedoch nicht vorschnell verwerfen. Sein konstruktiver Kern liegt darin, dass sich nicht jedes Programm in absehbarer Zeit als reine **funk tionale Daten transformation** realisieren lässt. Darüber hinaus besteht menschliches Handeln aus **Schritten** und der Veränderung des Zustands unserer Umgebung durch **schrittweise Manipulation** – eine Darstellung als reine Funktionen wäre für unser Denken unnatürlich.

Ein **Paradigma** definiert einen Satz von Ideen und Begriffen, Annahmen und Einschränkungen, Konzepten, Prinzipien, Postulaten,

Mustern und Techniken, die wir zum Lösen von Aufgaben auf einem Rechner heranziehen.

In diesem Abschnitt skizzieren wir einige Paradigmen; weiter hinten im Buch folgt für jedes eine eigene, ausführlichere Kapitel. Manche **Programmiersprachen** unterstützen nur ein Paradigma, andere sind **multiparadigmatisch**. Wir betrachten verschiedene Sprachen und wie sie die Paradigmen konkret umsetzen.

Für Menschen ist es intuitiv, jede Tätigkeit als **Abfolge von Schritten** – also als **Algorithmus** – zu sehen: das ist der **imperative Ansatz**. Diese Schritte können linear sein oder per **Verzweigung** zu einem anderen Teilplan führen. Der Entscheidungs vorgang besteht für die Maschine in einem **Vergleichs operator**, der meist zwei Alternativen eröffnet.

Aktionen lassen sich grob in **interne** und **externe** unterteilen.

- **Interne Operationen** betreffen ausschließlich Prozessor und Speicher; sie sind sofort abgeschlossen und ihr Ergebnis steht im nächsten Algorithmusschritt bereit.
- **Externe Operationen** richten sich an **Ein-/Ausgabe-Geräte** (Netzwerk, Festplatten, Sensoren usw.) und erfordern das Warten auf eine Reaktion – deren Dauer ist im Voraus meist unbekannt.

Wir senden also ein Steuersignal an ein Peripheriegerät und übermitteln die nötigen Daten. Anschließend haben wir zwei Möglichkeiten:

1. Wir **warten** auf das Ergebnis (**blockierender I/O**), oder
2. wir fahren mit dem nächsten Algorithmusschritt fort, ohne das Ergebnis abzuwarten (**nichtblockierender I/O**).

Diese Unterscheidung ist den großen Laufzeitunterschieden zwischen internen und externen Aktionen geschuldet. Externe Vorgänge gehen oft mit **physischen** oder sogar **analogen** Prozessen einher (etwa Funksignal, Plattenzugriff, Sensorablesung), die zusätzliche Daten umwandlung, **Übergangsvorgänge** oder ein bestimmtes Gerät signal abwarten. Viele Geräte besitzen eigene **Controller**, auf denen ein separater Befehlsstrom läuft; die Abstimmung zwischen **CPU** und Peripherie kostet Zeit und kann scheitern.

Ein Paradigma liefert somit ein **abstraktes Modell** zur Problemlösung, einen charakteristischen **Stil**, Beispiele guter und schlechter Lösungen sowie erprobte **Pattern**, die beim Schreiben von Code Anwendung

finden.