Clean Coding Cosmos:

Teil 4: Vom Urknall zum Blick in die Zukunft

Im letzten Teil unserer kosmologischen Reihe treten wir nochmals einen Schritt zurück, um das große Gesamtbild der Geschichte der Softwareentwicklung zu resümieren. Den Urknall der modernen Softwareentwicklung sehen wir in der Programmierung von architektonischen Berechnungen des Bauingenieurs Konrad Zuse auf der Z1. Von diesem Startschuss aus entwickelte sich die elektronische Datenverarbeitung zu einer unglaublich mächtigen Methode, die menschliche Arbeitseffizienz gewaltig zu steigern. Über mehr als sieben Jahrzehnte hinweg entwickelte sich die Softwareentwicklung dahin, immer größere Datenmengen in immer größerer Komplexität in immer kürzerer Zeit zu verarbeiten. Wir stellen in diesem Artikel eine grobe Skizze mit einigen Meilensteinen dieser Geschichte vor, zeigen Möglichkeiten auf, wie weitere Effizienzsteigerungen möglich werden könnten, und machen deutlich, wie der Clean Coding Cosmos dazu beitragen kann.

Seit dem Urknall der modernen Softwareentwicklung durch Konrad Zuse (**siehe Kasten 1**) sind die Möglichkeiten der Softwareentwickler und damit die Effizienz der Softwareentwicklung gewaltig expandiert. Diese Expansion betrifft alle vier Dimensionen des Entwicklerkosmos (vgl. [Obe13]). Grundlage für diese Expansion ist nicht nur die stetige Hardwareverbesserung, sondern auch die fortschreitende Evolution (**siehe Tabelle 1**) von Programmiersprachen, Entwicklungsmethoden und Entwicklungskontexten (Rahmenbedingungen wie Organisationsstrukturen).

An einem Montagmorgen im Jahr 1933 sitzt Konrad Zuse, Student des Bauingenieurswesens, an seinem Schreibtisch vor einem Stapel architektonischer Berechnungen, die er für seine Seminararbeit durchzuführen hat. Bereits aus seinem abgebrochenen Architekturstudium sind ihm die mathematischen Vorschriften für diese Aufgabe vertraut, sodass die vor ihm liegende Seminararbeit für ihn einfach nur viel lästige Fleißarbeit bedeutet. Sein Rechenschieber wartet auf den Einsatz.

Seine Gedanken beschäftigen sich aber noch mit dem Wochenende. Als ehemaliger Student des Maschinenbaus und Hobby-Tüftler arbeitet er in seiner Freizeit am Bau eines Warenautomaten, in den ein Mensch Geld einwirft, um eine Ware sowie passendes Wechselgeld zurückzubekommen. Von dieser Maschine abschweifend, taucht in seinem Kopf die Frage auf, warum es eigentlich noch keinen Rechenautomaten gibt, der ihm die immer gleichen, lästigen Berechnungen für seine Seminararbeit abnimmt.

Hauptproblem ist, dass es so viele verschiedene Rechenvorschriften im Bauingenieurswesen gibt. Diese Maschine müsste also so konzipiert sein, dass man ihr die mathematischen Vorschriften flexibel vorgeben und die nötigen Eingangsdaten bequem eingeben kann und das Ergebnis einfach auszulesen braucht. Eigentlich, denkt er, müsste es doch möglich sein, eine solche Maschine zu bauen und die ganze Fleißarbeit ihr zu überlassen. Eigentlich müsste er selbst in der Lage sein, so einen Universalrechner zu bauen und die Rechenvorschriften, die er benötigt, für diesen Rechner zu formulieren. Vor seinem inneren Auge entstehen stehen zwei Visionen: die Hardware und die Software.

1935, mittlerweile Mitarbeiter der Henschel-Flugzeugwerke, kündigt er seinen Job, um zu Hause im Wohnzimmer diese Visionen zu realisieren. 1938 ist die Z1, die erste frei programmierbare Rechenmaschine der Welt, fertig und führt (nur aufgrund unzuverlässiger Bauelemente noch unzuverlässig) ihre Rechenprogramme korrekt aus. Nach demselben Bauplan führt 1941 die Z3 (mit zuverlässigen Bauelementen) die Rechenprogramme korrekt und zuverlässig durch.

Kasten 1: Beginn der modernen Softwareentwicklung (im Sinne des ALM, vgl. [Obe13]). Der Gedankengang 1933 am Schreibtisch ist fiktiv, die genannten Tatsachen entsprechen aber der Realität.

Evolution der Programmiersprachen

Die erste Programmiersprache, die jedoch nur theoretische Bedeutung hatte, wird der Mathematikerin Ada Lovelace (1815–1852) zugeschrieben. Entwickelt wurde sie für die nie gebaute „Analytic Engine“ (vgl. [Wik-c]). Die erste praktische Programmiersprache war die Maschinensprache, mit der Konrad Zuse seine Z1 programmierte. Auf die Maschinensprachen folgten die Assembler-Sprachen und darauf die höheren Programmiersprachen (**siehe Kasten 2**). Verschiedene dieser Hochsprachen folgten sehr unterschiedlichen Paradigmen (**siehe Kasten 3**).

Die Geschichte der Programmierung zeigt, dass eine wachsende Abstraktion in den Programmierkonzepten das wesentliche Prinzip darstellt, um die Effizienz der Softwareentwicklung zu steigern. Das wird nicht nur in der Evolution von den Maschinen- zu den Hochsprachen deutlich, sondern auch in der Evolution von den nichtstrukturierten zu den multi-paradigmischen Hochsprachen (bzw. deren Konzepten):

- **50er Jahre:** Die nicht-strukturierten Programmiersprachen verwenden erstmals Sprachelemente, die aus einer natürlichen Sprache stammen (z.B. IF, WHILE).
- **60er Jahre:** Die strukturierte Programmierung zerlegt das Programm problemorientiert in eine hierarchische Struktur (schrittweise Verfeinerung, vgl. [Wir91]). Mit Hilfe der Prozedur bzw. Funktion kann ein Stück Code in einem anderen Kontext und mit verschieden Eingabewerte wiederverwendet werden.
- **70er Jahre:** Das Interface abstrahiert von der konkreten Implementierung und fördert die Idee der Kapselung und des „Information Hidings“. Des Weiteren abstrahiert die Übersetzung in Pseudocode (p-Code in Pascal) vom konkreten Betriebssystem und ist Vorbild für die „Java Virtual Machine“.
- **80er Jahre:** Die objektorientierte Programmierung schafft die Möglichkeiten, Datenmodellierung direkt mit Mitteln der Programmiersprache zu realisieren.

	Evolution der Programmiersprachen				Evolution der Entwicklungsmethoden		
Zeit	**Sprache**	**Hauptmerkmal**	**Vorteile**	**Beschränkungen**	**Methode**	**Hauptmerkmal**	**Weitere Meilensteine**
40er	Maschinen-sprachen		effektive CPU-Ausnutzung	Schwer verständlich, Prozessor-abhängig			
50er	Assembler-sprachen	Lesbarkeit durch Menschen	effektive CPU-Ausnutzung, bessere Verständlichkeit, Einführung von Variablen	benötigt Übersetzer (Assembler), nur einfache CPU-Befehle programmierbar			
50er	Fortran	Erste Hochsprache, Portabilität	bessere Verständlichkeit, Gleitkomma-Rechnungen	uneindeutige Sprache, beschränkte Ausdruckskraft, unstrukturiert	Wasserfallmodell von Benington	Einf. des Requirement Engineerings in der Softwareentwicklung	
60er	Algol	unterstützt strukturierte Programmierung	programmzeilenlos, wohldefinierte Grammatik, Prozeduren, Scoping	Einige konzeptionelle Fehler (z.B. Parameterübergabe) schwache String-Funktionen			
	Lisp	Funktionelle Programmierung	Erste funktionale Hochsprache. Metaprogrammierung möglich	Schwer erlernbar, gewöhnungsbedürftige Syntax			
	Cobol	Verarbeitung großer Datenmengen	problemorientierte Hochsprache für den kaufmännischen Bereich	Wortlastige Syntax	Dokumentation von Systemen (Architektur)	Nachvollziehbarkeit	Erste Betriebssysteme
	BASIC	Interpretiert, leicht erlernbar	auf PCs lauffähig („Volkssprache“)	unstrukturiert, nur für kleine Programme geeignet.			
	Simula	Erste objektorientierte Programmiersprache					
70er	Pascal	Große Verbreitung als Lehrsprache	Strukturierte und dynamische Datenstrukturen, Portabilität durch p-Code	Keine Module, die separat übersetzt werden	Wasserfallmodell von Royce	Feedbackschleifen zwischen den Entwicklungsphasen	
	Smalltalk	Objektorientierung nach dem Motto: Everything is an object.	Einbettung in eine integrierte Entwicklungsumgebung (IDE)	Schlechte Performance			Erste IDE
	C	Verwandtschaft zu Unix, maschinennah	Steht nahezu auf jedem Betriebssystem zur Verfügung.	schlechte Syntax, keine Range-Checks => sicherheitskritisch	V-Modell	Qualitätskontrolle durch Tests	Erste Design Patterns
80er	Modula 2	Schnittstellenkonzept, Nebenläufigkeit	Modulare Programmierung, Getrennte Übersetzung, maschinennah	fehlende Sprachmittel der Objektorientierung			
	ADA	Realtime-Fähigkeiten, Exception-Handling	Einführung von Generics	Keine Plattformunab-hängigkeit	Spiral-Modell	Iterative Entwicklung	
90er	Java	Plattform-Unabhängikeit, Memory-Modell	JVM, Standard-Libraries (JDK), Garbage Collection	kein Modulkonzept	Rational Unified Process	Vollständige Software-Entwicklungsmethode mit Toolunterstützung	Rasante Verbreitung des Internets
	JavaScript	Erste multipara-digmische Sprache	Objektorientiert (Prototyping) und funktionale Programmierung	Fehlende Sprachmittel zur Code-Strukturierung	Extreme Programming	Minimierung von Risiken	
00er	Scala	Hybrid aus objektorientierten und funktionalen Elementen	Einfachere Handhabung von Nebenläufigkeit (Aktor), JVM-Sprache, DSLs möglich	Schwer(er) erlernbar, Uneinheitliche Syntax => Dirty-Code-Gefahr	Scrum	Leichtgewichtige kundenorientierte Teamarbeit	Agiles Manifest
					Test Driven Development	Testbarkeit als Designkriterium	Clean Code Developer
10er					Team Clean Coding	Teambildungsmaßnahme	Dev-Ops

Tabelle 1: Übersicht einiger Meilensteine in Geschichte der Softwareentwicklung. Diese Liste ist unvollständig und letztlich willkürlich zusammengestellt. Allerdings sind die einzelnen Meilensteine hier unwichtig. Es geht um das Gesamtbild, um die Evolutionsgeschichte, die auch mit anderen Meilensteinen illustriert werden könnte.

Sie gestattet mittels Vererbung eine weitere Art der Wiederverwendung. Die objektorientierte Modellierung etabliert sich später als Quasi-Standard der Software-Entwicklungsmethoden.

- **90er Jahre:** Komponentenmodelle legen die Eigenschaften und die Interaktionen von Komponenten fest und ermöglichen plattformübergreifende und verteilte Anwendungen.
- **00er Jahre** Der Einsatz von *domänenspezifischen Sprachen (DSLs)*, vor allem im Bereich der *modellgetriebenen Softwareentwicklung (MDSD)*, bewirkt eine kompaktere Darstellung (Code-Einsparung) und eine Unabhängigkeit von der Zielplattform. *Service-orientierte Architekturen (SOAs)* propagieren verteilte Services auf hohem Abstraktionsniveau, die an Geschäftsprozessen ausgerichtet sind mit dem Zweck einer flexiblen Wiederverwendung.
- **10er Jahre:** Das Konzept des Cloud-Computings verfolgt das Ziel, komplett unabhängig von einer IT-Infrastruktur zu sein: Software aus der Steckdose. Die dafür notwendigen Konzepte entwickeln sich zurzeit, haben aber noch keinen Einzug in Programmiersprachen gefunden.

Außerdem kann die Höherentwicklung der Programmiersprachen gut an der Evolution der Datentypen illustriert werden:

- **50er Jahre:** Einfache statische Datentypen, wie Gleitkommazahlen oder zusammengesetzte Datentypen (Records, Arrays), abstrahieren von der Quantisierung auf Byte-Ebene.
- **60er Jahre:** Dynamische Datentypen (z.B. Strings, die Liste in LISP) überwinden virtuell die Beschränkung des endlichen Speichers.
- **70er Jahre:** Abstrakte Datentypen kapseln die Interna der Daten und spiegeln sich in den Klassen der objektorientierten Programmierung wider. Auch die funktionalen Sprachen besitzen diesbezüglich Ausdrucksmöglichkeiten auf einem hohen Abstraktionsniveau.
- **80er Jahre:** Durch generische Datentypen lassen sich (abstrakte) Datenstrukturen wie Listen, Mengen usw. typsicher wiederverwenden.
- **90er Jahre:** Standardisierte Daten-Austauschformate wie XML ermöglichen typsichere Schnittstellen zwischen Services.

Ein weiterer Aspekt der Höherentwicklung ist eine zunehmende Vereinfachung der Syntax, die zu einer größeren Kompaktheit und gleichzeitig zu einer besseren Lesbarkeit führt. Diese Entwicklung lässt sich gut beim Vergleich von **Listing 3** (COBOL),

- **Maschinensprachen (1. Generation):** Eine Folge von Nullen und Einsen, die der Prozessor entweder als eine Folge von Befehlen (so genannte OP-Codes) oder Daten versteht.
- **Assembler-Sprachen (2. Generation):** Eine Folge von lesbaren Befehlen (so genannte Mnemonics), die ein Compiler (der Assembler) eins zu eins in OP-Codes der Maschinensprache übersetzt (**siehe Listing 1**).
- **Hochsprachen:** Verwendung von Sprachmitteln, die zumindest entfernt an eine natürliche Sprache erinnern. Sie sind prozessorunabhängig und damit universell einsetzbar (portierbar).
- **Imperative Hochsprachen (3. Generation):** Verwendung von Sprachmitteln, die eine algorithmische Herangehensweise unterstützen. Das Programm besteht aus einer definierten Reihenfolge von Anweisungen, die zu einer Folge von Statusänderungen (beispielsweise der Geschäftsdaten) führen. Es beschreibt das *Wie* (den Lösungsweg).
- **Deklarative Hochsprachen (4. Generation):** Mit diesen Sprachen wird nicht das *Wie*, sondern auch das *Was* (die Lösung) beschrieben. Deklarative Sprachmittel erlauben die Definition von Regeln, welche Anforderungen von der Ausgabe erwartet werden (z. B. SQL, MathLab, UML).
- **KI-Hochsprachen (5. Generation):** Probleme werden auf einem sehr hohen abstrakten Niveau beschrieben (z. B. Prolog, **siehe Listing 2**). Weitere Sprachen befinden sich im Bereich der Künstlichen Intelligenz in der Forschung.

Kasten 2: Die Generation von Programmiersprachen (siehe auch Kasten 3).

- **Nicht-strukturierte Sprachen (imperativ):** Vorläufer der prozeduralen Sprachen (**siehe Listing 3**).
- **Prozedurale Sprachen (imperativ):** Programmiersprachen, die dem Paradigma der strukturierten Programmierung folgen (**siehe Listing 4**).
- **Objektorientierte Sprachen (imperativ):** Sie erweitern den imperativen Ansatz um die Prinzipien der Kapselung von Daten und Operationen zu Objekten sowie der Vererbung und Polymorphie. Die instanziierten Objekte (Klassen) interagieren und verändern dabei sukzessive ihre inneren Zustände, also ihre Daten (**siehe Listing 5**).
- **Funktionale Sprachen (deklarativ):** Ein Programm besteht aus verketteten, geschachtelten und rekursiven mathematischen Funktionen, die zustandslos sind. Da Datenstrukturen inhaltlich nicht geändert werden (*immutable*), können die Funktionen auch nebenläufig frei von Seiteneffekten ausgeführt werden (**siehe Listing 6**).
- **Logische Sprachen (deklarativ):** Ein Programm besteht aus einer Menge von Regeln, die mit Hilfe eines Logik-Kalküls ausgewertet werden – ähnlich einer *Rule Engine* (**siehe Listing 2**).
- **Multi-paradigmische Sprachen:** Höhere Programmiersprachen, die verschiedene Programmierparadigmen verfolgen, um die Vorteile aller zu vereinen (**siehe Listing 7**).

Kasten 3: Die Paradigmen der Hochsprachen (siehe auch Abbildung 1).

Listing 5 (C++) und **Listing 7** (Ruby) nachvollziehen.

Evolution der Entwicklungsmethoden

Die Weiterentwicklung der Hardware und die fortschreitende Evolution der Programmiersprachen erlaubten es, immer komplexere Daten zu verarbeiten. Aus Rechenmaschinen wurden so Computer. Die Komplexität der Software war schließlich so groß, dass ein „Einfach drauflos Programmieren" (Cowboy-Coding) nicht mehr effektiv war. Die Differenzierungen der Softwareentwicklung in verschiedene Phasen (Planung, Umsetzung, Qualitätssicherung, siehe **Abbildung 2** in [Obe13]) begann und schließlich wurde das Wasserfall-Modell formuliert. Von diesem Modell ausgehend, lassen sich folgende Trends feststellen:

1. Bedeutung von Tests

Anfänglich wurden Tests nur als Debugging-Methode verstanden. Ab 1956 wurden Tests zur Demonstration von Funktionalität (heute Akzeptanztests genannt) eingesetzt. Dann ab 1979 wurden sie auch gezielt zur Fehlersuche und schließlich (ab 1988) zur Prävention von Fehlern eingesetzt (vgl. [Wik-d]). Dieser Trend führte dann zum Paradigma „Test First" und zur Methode des *Test Driven Development (TDD)*.

2. Einsatz von Iterationen

Bald wurde klar, dass das generalstabsmäßige Vorgehen im Wasserfall-Modell auch mit Feedback-Schleifen zu unflexibel ist, um Software größerer Komplexität zu bewältigen. So entstand das Spiralmodell (vgl. [Wik-e]), bei dem sich die Entwicklungsphasen zyklisch wiederholen. Dieses iterative Vorgehen wurde dann von der agilen Bewegung in viel konsequenterer Form verfolgt und zum Paradigma erhoben.

3. Form der Kommunikation

Die Akteure der Softwareentwicklung lernten, immer effektiver zu kommunizieren: zum einen durch Abstraktionen und die Verdichtung von Informationen durch Modellierung und eine standardisierte Darstellung (z. B. UML) und zum anderen durch regelmäßige und intensive Kommunikation zwischen den Akteuren. Das führte zu der Entwicklung der modernen Architektenrolle mit ihren vielseitigen Kommunikationsaufgaben und auch zu den Kommunikationsformen der agilen Methoden (Standups, Retrospektive usw.).

4. Ausmaß der Vorausplanung

Zu den schwergewichtigen, planungslastigen Vorgehensweisen gab es eine Gegenbewegung, die zeigte, dass es möglich ist, komplexe Software sehr effektiv mit vergleichsweise sehr wenig Planung zu entwickeln. Das wurde erreicht, indem die Entwickler von großem Planungs-Overhead befreit wurden und ihnen großer Entscheidungsspielraum eingeräumt wurde. Das führte nicht zu Cowboy-Coding, sondern zu den leichtgewichtigen agilen Methoden, die den Lean-Prinzipien (**siehe Tabelle 2**) treu sind.

Evolution der IT-Enwicklungskontexte

Die Evolution der Entwicklungsmethoden in den letzten 20 Jahren hat deutlich gemacht, dass bestimmte Rahmenbedin-

```
.model small
.stack 256
CR equ 13d
LF equ 10d

.data
prompt1 db □Number 1: □, 0
prompt2 db CR, LF,□Number 2:□,0
result db CR, LF □Result: □, 0
num1 dw ?
num2 dw ?

.code
start:
mov ax, @data
mov ds, ax
mov ax, offset prompt1
call put_str ; display prompt1
call getn ; read first number
mov num1, ax
mov ax, offset prompt2
call put_str ; display prompt2
call getn ; read second number
mov num2, ax
mov ax, offset result
call put_str ; display result message
mov ax, num1 ; ax = num1
add ax, num2 ; ax = ax + num2
call putn ; display sum
mov ax, 4c00h
int 21h ; finished, back to dos
end start
```

Listing 1: Programm zum Einlesen und Addieren zweier Zahlen in der 8086-Assemblersprache.

```
predicates
    start.
    sum.

clauses
    start:- sum,nl.

    sum:-   write("Number 1: "),readreal(X),
            write("Number 2: "),readreal(Y),
            S = X+Y,
            write("Result: "),write(S).
goal
    start.
```

Listing 2: Programm zum Einlesen und Addieren zweier Zahlen in Prolog.

```
000100 ID DIVISION.
000200 PROGRAM-ID.  ACCEPT1.
000300*
000400 DATA DIVISION.
000500 WORKING-STORAGE  SECTION.
000600 01  WS-FIRST-NUMBER      PIC 9(3).
000700 01  WS-SECOND-NUMBER     PIC 9(3).
000800 01  WS-TOTAL             PIC ZZZ9.
000900*
001000 PROCEDURE DIVISION.
001100 0000-MAINLINE.
001200     DISPLAY 'NUMBER 1: '.
001300     ACCEPT WS-FIRST-NUMBER.
001400*
001500     DISPLAY 'NUMBER 2: '.
001600     ACCEPT WS-SECOND-NUMBER.
001700*
001800     COMPUTE WS-TOTAL = WS-FIRST-NUMBER + WS-SECOND-NUMBER.
001900     DISPLAY 'Result: ', WS-TOTAL.
002000     STOP RUN.
```

Listing 3: Programm zum Einlesen und Addieren zweier Zahlen in COBOL.

```
PROGRAM AddingTwoNumbers;

Var
   Number1, Number2, Result: Integer;

FUNCTION Add (Num1, Num2: INTEGER): INTEGER;
BEGIN
   Add := Num1 + Num2;
END;

PROCEDURE WriteResult;
BEGIN
   Write('Result: ');
   Writeln(Result);
   Readln;
END;

BEGIN
 Write('Number 1: ');
 Readln(Number1);

 Writeln('Number 2: ');
 Readln(Number2);

 Result := Add(Number1, Number2);
 WriteResult();
END.
```

Listing 4: Programm zum Einlesen und Addieren zweier Zahlen in Pascal.

```
#include <iostream>

using namespace std;

class Addition {
  int num1, num2;

public:
  void readTwoNumbers() {
    cout << "Number 1: ";
    cin >> num1;
    cout << "Number 2: ";
    cin >> num2;
  }

  void writeResult() {
    cout << "Result = " << x + y << endl;
  }
};

int main()
{
   Addition add; // Creating object of class

   add.readTwoNumbers();
   add.writeResult();

   return 0;
}
```

Listing 5: Programm zum Einlesen und Addieren zweier Zahlen in C++.

```
(defn add [a b]
  (+ a b)
)

(defn readInput [num]
   (println "Number " num ": ")
   (read-string(read-line))
)

(println "Result: "
   (add (readInput □1) (readInput □2))
)
```

Listing 6: Programm zum Einlesen und Addieren zweier Zahlen in Clojure.

```
def add(a,b)
   a + b
end

def read(num)
   puts "Number #{num}: "
   gets.to_i
end

puts "Result: #{add(read(1), read(2))}"
```

Listing 7: Programm zum Einlesen und Addieren zweier Zahlen in Ruby.

gungen bestehen müssen, damit diese modernen Entwicklungsmethoden richtig eingesetzt werden können. Das zeigte sich besonders beim Einsatz von agilen Methoden. Mit dem Paradigmenwechsel hin zu Eigenverantwortung im Team wird die Notwendigkeit der klassischen Projektleiter- und Architektenrolle immer wieder diskutiert (**siehe Kasten 6** in [Obe14-a]). Der Trend weg von diesen klassischen Rollen hin zu eigenverantwortlichen Entwicklerteams kollidiert mit dem Führungsstil klassischer Projektleiter in hierarchischen Organisationsstrukturen.

Um die agilen Methoden im großen Stile realisieren zu können, wurden Forderungen nach einem radikalen Organisationswandel laut. Allerdings zeigt die Erfahrung, dass eine schrittweise Änderung sowohl für den Einzelnen, als auch für ein Team und für die Gesamtorganisation realistischer und zielführender ist (vgl. [Roo14]).

Möglichkeiten weiterer Effizienzsteigerungen

Effizienzsteigerungen sind in allen drei genannten Bereichen möglich.

Programmiersprachen

Codegenerierung beruht auf folgenden Schritten:

1. So genannten „Glue Code“, der sich immer wieder wiederholt, in Templates auslagern.
2. Daten, die für die Implementierung spezifisch sind, extrahieren.
3. Durch Variation der Daten und Anwendung auf die Templates sehr schnell Code generieren.

Der Grund, warum die Methode nur selten eingesetzt wird, ist die Abhängigkeit von einem Generatorframework und dessen Beherrschung. Eine weitaus elegantere Möglichkeit bestünde darin, die Notwendigkeit von generierten Codeverdoppelungen durch neue Sprachmittel der Programmiersprache zu ersetzen.

Einige Sprachen – beispielsweise F#, Scala, Groovy und auch Java 8 – versuchen, die Vorteile der funktionalen Programmierung (vor allem in Bezug auf Parallelisierung) in eine an sich imperative Programmiersprache zu integrieren. Wenn es durch eine Erweiterung der sprachlichen Mittel in den etablierten Programmiersprachen möglich wird, imperativ, funktional und eventuell auch logisch zu programmieren, werden die Programmiersprachen zu flexiblen Alleskönnern. Das ermöglicht es, die Vorteile einer Programmierweise kontextspezifisch zu nutzen, ohne die Programmiersprache zu wechseln.

Für sehr spezielle Probleme sind spezielle Programmiersprachen denkbar, die Nischen besetzen, in denen die Alleskönner nicht gut genug sind. Sprachen wie Groovy oder Scala zeigen, dass zum einen das Erstellen von DSLs mühelos möglich ist, zum anderen ist auch das Erschaffen von ganz neuen Sprachen einfach(er) geworden. Der Vorteil der auf der JVM basierenden Sprachen ist, dass die komplette Infrastruktur von Java weiterverwendet werden kann – ein unschätzbarer Gewinn. Mit diesem Konzept wird die weitere Evolution von Programmiersprachen wesentlich beschleunigt.

Software, die nach den Prinzipien der Objektorientierung strukturiert ist, hat bei der Ausführung auf Multicore-Prozessoren ein Problem. Die Zustände eines Objekts können zwischen den Cores nur schwer synchronisiert werden. Das ist schlecht für die Performance. Wenn es gelingt, ein einfaches Konzept zur Parallelisierung mit der Ausdrucksstärke der objektorientierten Strukturen zu kombinieren, wird das die Effizienz der Softwareentwicklung weiter steigern.

Der Zugriff auf Datenquellen erfolgt häufig über spezielle Programmierschnittstellen, die eher unhandlich als komfortabel sind. Für jede Quelle (Datenbank, Web-Service, File-System) und jedes Format (Datenbank-Tabelle, XML, HTML, CSV, PDF) existieren zurzeit die unterschiedlichsten APIs. Auf diesem Gebiet erwarten wir in naher Zukunft erhebliche Vereinfachungen und Effizienzsteigerungen durch die Anwendung einer einheitlichen API, die auf *Method Chaining* beruht (vgl. [Wik-a]). Als wegweisendes Vorbild könnte der `Data.TypeProvider` von F# dienen.

Softwareentwicklungsmethoden

- Softskills der Akteure und die Kommunikation (vgl. „Kommunikation“ in [Obe14-a]) zwischen ihnen verbessern.
- Bildung von High-Performance-Teams (gut eingespielte Teams mit konstanter Zusammensetzung).
- Standardisierung von Teamwork, z.B. *Team Clean Coding (TCC)* (vgl. [Obe-14-b]), sodass Performance erhalten bleibt, wenn Teammitglieder wechseln.
- Einheitliches zentrales Repository (vgl. SLI in [Obe-13]) für alle Akteure im *Application Lifecycle Management (ALM)* in Kombination mit einem einheitlichen Tool, das all diesen Akteuren zur Verfügung steht.

Entwicklungskontexte

Große Organisationen, die Softwaresysteme bauen, unterliegen dem Gesetz von Conway (vgl. [Wik-f]). Die Strukturen großer Organisationen sind typischerweise sehr heterogen und damit auch ihre Systemlandschaft. Der entsprechende Mangel an Homogenität behindert die Integration der Subsysteme. Ein Mechanismus, das Gesetz von Conway unwirksam zu machen oder wenigstens abzuschwächen, würde die Effizienz der Softwareentwicklung verbessern.

Zwischen der Eigen- und der Fremdbestimmung von Entwicklerteams besteht ein gewisses Spannungsfeld (**siehe Abbildung 2**). Günstige Entwicklungskontexte geben Rahmenbedingungen vor, die für Teams einsichtig und damit leicht akzeptierbar sind, aber stark genug, um die Homogenität der Gesamtentwicklung zu erhalten. Typische Rahmenbedingungen sind z.B. Technologien und Tools sowie Entwicklungsmethoden und Qualitätssicherungsmaßnahmen, welche die Entwicklerteams einsetzen dürfen oder sollen (vgl. auch „TCC-Rahmenbedingungen“ in [Obe-14b].

Wohin mit noch größerer Effizienz?

Der Zusammenhang zwischen verschiedenen Vorgehensweisen, die daraus resultierende Effizienz und die damit verbundenen Kosten können nicht eindeutig geklärt werden. Wir stellen hier drei Ansätze vor, die diesem Zusammenhang nachgehen.

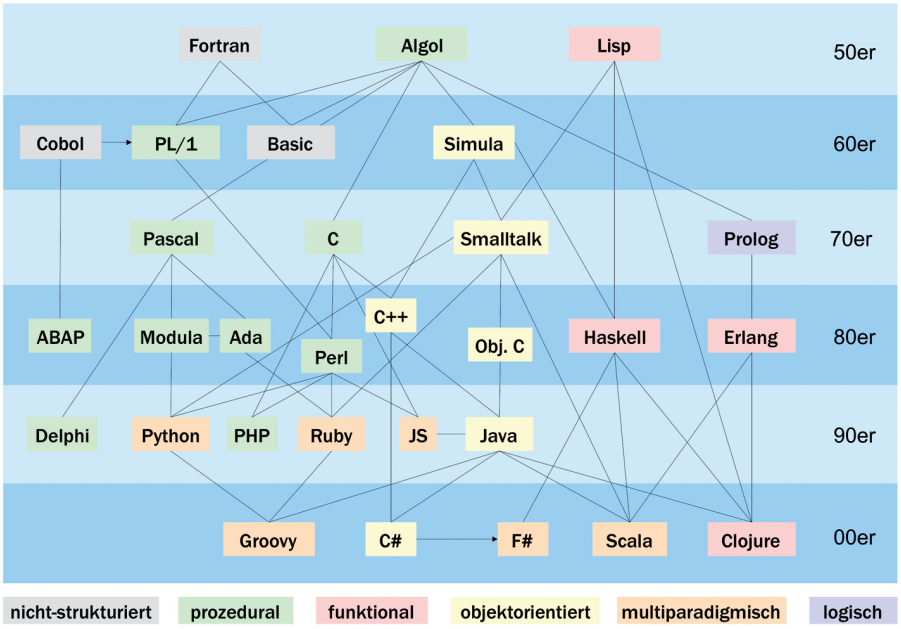

Abb. 1: Verwandtschaftsbeziehungen zwischen einigen bekannten Hochsprachen. Spätere Erweiterungen der Sprache sind hier nicht berücksichtigt.

Lean-Prinzipien	Clean Coding Cosmos Meinung
Eliminate waste	Maßnahmen gegen Softwareentwicklungsschmutz (s. Teil 2)
Amplify learning	Fortbildung der firmeneigenen Mitarbeiter fördern (s. Teil 3) und Journal Coding Club (s. Teil 3) sowie die CCD-Praktiken [Wik-c], [Wik-d] und [Wik-e]
Decide as late as possible	Die CCD-Praktiken KISS [Wik-f], „Vorsicht vor Optimierungen“ [Wik-g] und YAGNI [Wik-h]
Deliver as fast as possible	Die CCD-Praktik „Continuous Delivery“ [Wik-i]
Empower the team	Team Clean Coding (s.Teil 3)
Build integrity in	Funktionale und nicht-funktionale Merkmale der Software (s. Abb. 1 in Teil 1) sowie Softwarequalität (s. Abb. 4 und 5 in Teil 2)
See the whole	Ganzheitlicher Ansatz des Clean Coding Cosmos (s. Teil 1)

Tabelle 2: Ist lean (vgl. [Wik-b] gleich clean? Ja, aber auch die klassische Softwareentwicklung kann clean sein (vgl. auch [Obe13], [Obe14-a] und [Obe14-b] dieses Artikels.

Metrik und Zahlen

Eine Reihe von Studien misst Entwicklungsgeschwindigkeiten (z.B. *Story Points* pro Zeit), Softwarequalität (z.B. Anzahl der *Issues* pro Zeit) oder Prozessqualität (*Time-to-Market*). Das Problem mit diesen Zahlen ist, dass jedes IT-Projekt einen eigenen Stern im Entwickleruniversum und damit ein sehr individuelles Ökosystem darstellt. Vorgehensweisen oder Technologien zwischen solchen individuellen Projekten zu vergleichen, halten wir deshalb für sehr fragwürdig.

Aussage erfahrener Entwickler

Es gibt weltweit anerkannte seriöse Fachleute, die in ihrem jahrzehntelangen Entwicklerleben sehr viel vom Entwicklerkosmos erfahren haben. Wir zitieren drei der erfahrensten Entwickler:

- **Kent Beck** 2007 über XP: „[...] if you take some practices like technical collaboration [and] testing [...] there's all these nice consequences. You get very low defect rates, which means you can release software much more frequently. You get a very low cost to getting projects into an initially deployable state, low cost and short time [...] if you take this notion of incremental design, con-

Literatur & Links

[Bec07] K. Beck, XP inventor talks about agile programming, 2007, siehe: http://m.infoworld.com/d/developer-world/xp-inventor-talks-about-agile-programming-565?mm_ref=http%3A%2F%2Fwww.threeriversinstitute.org%2F

[CCC-a] Clean Code Developer, Oranger Grad, siehe: http://www.clean-code-developer.de/Oranger-Grad.ashx#Lesen_Lesen_Lesen_6

[CCC-b] Clean Code Developer, Teilnahme an Fachveranstaltungen, siehe: http://www.clean-code-developer.de/Gelber-Grad.ashx#Teilnahme_an_Fachveranstaltungen_8

[CCC-c] Clean Code Developer, Erfahrung weitergeben, siehe: http://www.clean-code-developer.de/Gr%C3%BCner-Grad.ashx#Erfahrung_weitergeben_6

[CCC-d] Clean Code Developer, Keep it simple, stupid (KISS), siehe: http://www.clean-code-developer.de/Roter-Grad.ashx#Keep_it_simple_stupid_KISS_1

[CCC-e] Clean Code Developer, Vorsicht vor Optimierungen!, siehe: http://www.clean-code-developer.de/Roter-Grad.ashx#Vorsicht_vor_Optimierungen!_2

[CCC-f] Clean Code Developer, You Ain´t Gonna Need It (YAGNI), siehe: http://www.clean-code-developer.de/Blauer-Grad.ashx#You_Ain%C2%B4t_Gonna_Need_It_YAGNI_2

[CCC-g] Clean Code Develope, Continuous Deliver, siehe: http://www.clean-code-developer.de/Blauer-Grad.ashx#Continuous_Delivery_3

[Fow14] M. Fowler, Workflows for Refactoring, Vortrag bei der OOP 2014, 2014, siehe: http://m.youtube.com/watch?v=vqEg37e4Mkw&feature=youtu.be&desktop_uri=%2Fwatch%3Fv%3DvqEg37e4Mkw%26feature%3Dyoutu.be

[Mar09] R.C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, Prentice Hall 2009

[Obe13] R. Oberrath, J. Vollmer, Clean Coding Cosmos: Teil 1: Kosmologie für Softwareentwickler, in: OBJEKTspektrum 6/2013

[Obe14-a] R. Oberrath, J. Vollmer, Clean Coding Cosmos: Teil 2: Kosmologische Suche nach Softwareentwicklungsschmutz, in: OBJEKTspektrum 1/2014

[Obe14-b] R. Oberrath, J. Vollmer, Clean Coding Cosmos: Teil 3: Kosmische Effizienz durch Team Clean Coding, in: OBJEKTspektrum 2/2014

[Roo14] S. Roock, H. Wolf, Agil skalieren – Prinzipien statt Blaupause, Vortrag bei der OOP 2014, siehe: http://de.slideshare.net/roock/agile-skalierung-prinzipien-statt-blueprint

[Wik-a] Wikipedia, Method chaining, siehe: http://en.wikipedia.org/wiki/Method_chaining

[Wik-b] Wikipedia, Lean software development, siehe: http://en.wikipedia.org/wiki/Lean_software_development

[Wik-c] Wikipedia, Analytical Engine, siehe: http://de.wikipedia.org/wiki/Analytical_Engine

[Wik-d] Wikipedia, Kurs:Software-Test/Geschichte des Testens, siehe: http://de.wikiversity.org/wiki/Kurs:Software-Test/Geschichte_des_Testens

[Wik-e] Wikipedia, Spiralmodell, siehe: http://de.wikipedia.org/wiki/Spiralmodell

[Wik-f] Wikipedia, Gesetz von Conway, siehe: http://de.wikipedia.org/wiki/Gesetz_von_Conway

[Wir91] N. Wirth, Algorithmen und Datenstrukturen, Vieweg+Teubner Verlag 1991

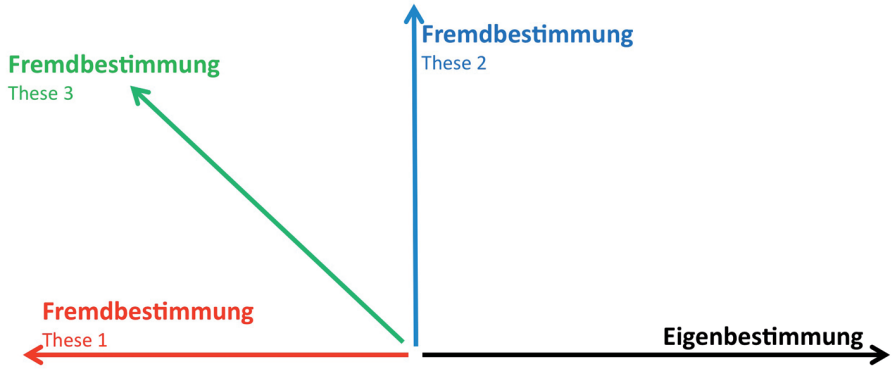

Abb. 2: Wie sind Entwicklungsteams bestimmt?
These 1: Fremdbestimmtheit widerspricht der Eigenbestimmtheit. Dieses Spannungsfeld ist häufig die gefühlte Lebenswirklichkeit von Entwicklern.
These 2 (Antithese): Fremd- und Eigenbestimmtheit sind voneinander unabhängig und können gleichzeitig verwirklicht werden, indem die Entwicklungsteams von der Sinnhaftigkeit fremdbestimmter Entscheidungen überzeugt werden.
These 3 (Synthese): Die Fremdbestimmtheit schränkt den Freiraum eines Teams ein, stellt aber nur Grenzen auf, in denen das Team Raum für Eigenbestimmtheit hat.

tinually investing in design seriously, you can continue deploying new functionality at a fairly steady rate for a very long time.“ (vgl. [Bec07]).

- **Robert C. Martin** 2009 über Clean-Code-Maßnahmen und Messy Code: „Over the span of a year or two, teams that were moving very fast at the beginning of a project can find themselves moving at a snail’s pace. Every change they make to the code breaks two or three other parts of the code. No change is trivial [...] Over time the mess becomes so big and so deep and so tall, they can not clean it up [...] As the mess builds, the productivity of the team continues to decrease, asymptotically approaching zero.“ (vgl. [Mar09]).
- **Martin Fowler** 2014 über Refactoring: „What refactoring does is, it constantly keeps your code base in a good state [...] and if you are diligent about doing this [...] you are able to deliver more functionality more quickly as somebody who has got an ill designed code base [...] If you are not doing that you are effectively stealing from your customers, you are making them spending more money, take more time for the same functionality [...] Refactoring is about an economic argument: Clean

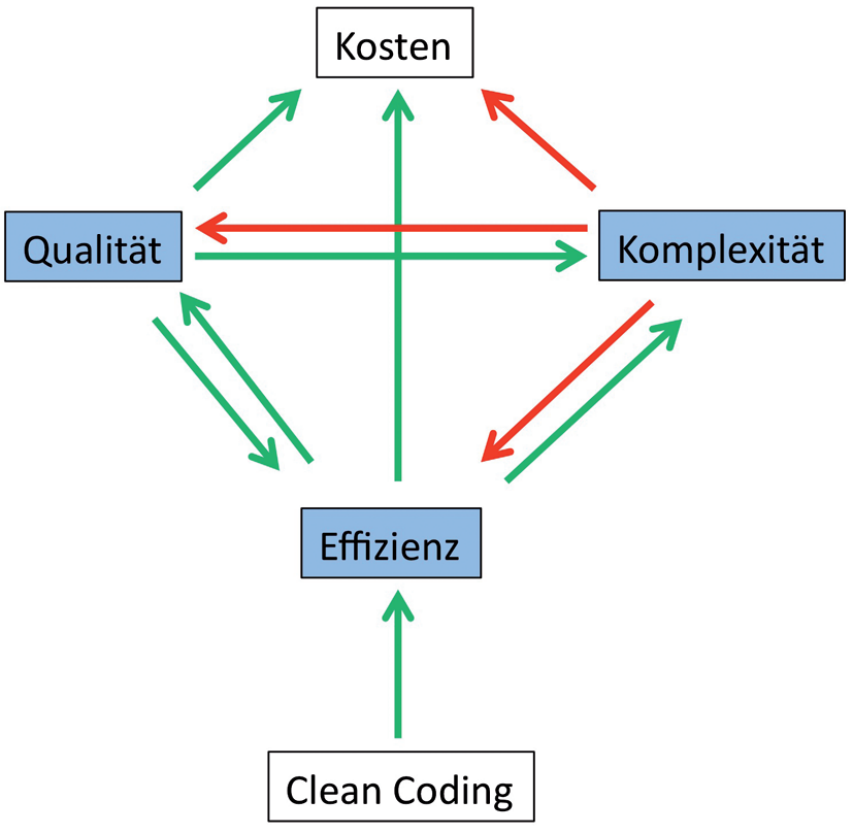

Abb. 3: Zusammenhang zwischen Qualität, Komplexität und Effizienz in der Softwareentwicklung, der sich durch Plausibilität im Gedankenexperiment ergibt (vgl. Kasten 4 und 5).

Die Qualität von Software ist hoch, wenn Sie Ihren funktionalen und nichtfunktionalen Anforderungen gerecht wird, wenn die Entwicklungskosten den gegebenen Rahmenbedingungen entsprechen und wenn die Wartbarkeit der Software sichergestellt ist. Das letzte Ziel wird typischerweise für kurzfristige Ziele als Widerspruch zu den ersten beiden Zielen betrachtet (vgl. „Termindruck“ in [Obe-14a]). Allerdings stellt das letzte Ziel langfristig sicher, die beiden ersten Ziele auch weiterhin zu erreichen. Doch leider werden in der Praxis die notwendigen Konsequenzen daraus häufig nicht gezogen.

Kasten 4: Was ist Softwarequalität und wie sehen ihre Qualitätsziele aus?

Effizienz bedeutet, unter gegebenen Rahmenbedingungen ein konkretes Ziel mit minimalem Aufwand zu erreichen. Im Gegensatz zur Effektivität spielt die Kosten-Nutzen-Betrachtung bei der Effizienz die entscheidende Rolle.

Kasten 5: Was ist Softwareentwicklungseffizienz?

> Code allows you to go faster.“ (vgl. [Fow14]).

Die gemeinsame Idee dieser drei Meinungen lautet: Mit den genannten Vorgehensweisen wird die Effizienz und damit die Softwarequalität erhöht und damit die Entwicklungskosten gesenkt.

Plausibilität im Gedankenexperiment

Abbildung 3 zeigt die Zusammenhänge zwischen Effizienz, Qualität, Komplexität und Kosten, wie sie sich aus unserer Sicht plausibel erschließen. Grundsätzlich gibt es zwei Effizienzsenken: Softwarequalität (**siehe Kasten 4**) und Komplexität. Steigende Effizienz (**siehe Kasten 5**) in der Softwareentwicklung kann eingesetzt werden, um die Qualität von Software, d.h. von Produkten und Prozessen (siehe **Abbildung 1** in [Obe13]), zu steigern. Als Resultat verringern sich Entwicklungsaufwände (Zeit und Budget) und auch das Risiko von Projektabbrüchen. Steigende Effizienz kann aber auch eingesetzt werden, um Software mit höherer Komplexität zu entwickeln. Bei gleichbleibender (guter oder schlechter) Softwarequalität ist es mit höherer Softwareentwicklungseffizienz möglich, Software zu bauen, die die vorher erreichten Datenverarbeitungsgrenzen überschreitet. Bei steigender Komplexität verringert sich aber typischerweise die Softwarequalität und damit steigt auch das Risiko von Projektabbrüchen. Ein Mittelweg, d.h. etwas mehr Qualität und etwas mehr Komplexität, ist in der Regel sinnvoll.

Fazit

Die Geschichte der IT ist die Geschichte einer immer schnelleren Verarbeitung von immer komplexeren Daten, also einer Geschichte von Effizienzsteigerungen. Beigetragen zu dieser rasanten Zunahme von Effizienz hat nicht nur die Hardwareentwicklung, sondern auch eine immer effizienter werdende Softwareentwicklung. Sowohl Programmiersprachen als auch Entwicklungsmethoden wurden (und werden immer noch) Schritt für Schritt leistungsfähiger. In den ersten Jahrzehnten dieses Wegs war die Grundlage der Effizienzsteigerung die Evolution der Programmiersprachen, in den letzten Jahren aber verstärkt die Evolution der Entwicklungsmethoden. Die ganzheitliche Weltsicht des Clean Coding Cosmos (vgl. [Obe-13]) mit seinem neuen Vorgehensmodell „Team Clean Coding“ (vgl. [Obe-14b]) verspricht, die Effizienz der Softwareentwicklung noch weiter zu steigern, eine noch bessere Softwarequalität zu erreichen und noch mehr Komplexität zu beherrschen. ||

Die Autoren

|| Dr. Reik Oberrath
(R.Oberrath@iks-gmbh.com)
ist als IT-Berater bei der iks Gesellschaft für Informations- und Kommunikationssysteme tätig. Er beschäftigt sich seit vielen Jahren mit der Entwicklung von individuellen Geschäftsanwendungen und mit der Architektur komponentenbasierter Systeme.

|| Jörg Vollmer
(info@informatikbuero.com)
ist freiberuflicher IT-Berater und verfügt über langjährige Erfahrung als Entwickler, Softwarearchitekt und Trainer im Java-Umfeld, speziell Java EE und Spring. Seine Leidenschaft gehört dem Vermitteln von Software-Qualitätsbewusstsein.