

REQUIREMENTS-FIRST DEVELOPMENT – TEIL 1

Am Anfang steht die Anforderung

ChatGPT als Ersatz für Entwickler? Wohl kaum, denn es bedarf genauer Anforderungsformulierung und der Überprüfung des generierten Codes.

Muss zur Softwareentwicklung mit künstlichen Intelligenzen wie ChatGPT, GitHub Copilot oder Replit Ghostwriter noch etwas gesagt werden? Schon während ich dies im April 2023 schreibe, vibriert die Welt in einem Dreieck aus euphorischer Hektik, was den Einsatz von KI in allen beruflichen und privaten Belangen angeht, lähmender Überwältigung und Angststarre. Ein nie gesehenes Feuerwerk an guten Ratschlägen, Tools, Prognosen brennt über dem Globus ab. Auch wer bisher der Digitalisierung noch meinte ausweichen zu können, wird in den KI-Strudel gezogen. Auf die eine oder andere Weise lässt „die KI“ niemanden kalt. Was kann ich da heute sagen, das nach den Monaten, die bis zum Erscheinen dieses Artikels vergehen werden oder sogar darüber hinaus noch Relevanz haben könnte? Es fällt mir schwer, mir das vorzustellen. Stündlich lerne ich etwas dazu. Täglich kommt ein Tool heraus, das wieder einen Aspekt der Arbeitswelt verändert. Alles ist im Fluss.

Ich habe mich entschieden, nur begrenzt direkt ins Feuerwerk zu schauen. Ich probiere nicht jedes Tool aus, sondern versuche auf die Muster dahinter zu schauen. Was sind die Konstanten in all der rasenden Veränderung? Deshalb werde ich in diesem Artikel keine in die IDE integrierte KI benutzen. Stattdessen bleibe ich bei dem Tool, mit dem alles begonnen hat und dem ich die größte Stabilität zutraue: ChatGPT. Es ist für mich in seiner Einfachheit und Zugänglichkeit am vielversprechendsten. Es bietet eine riesige Bandbreite an Einsatzgebieten. Dafür nehme ich eine gewisse Ineffizienz beim Zusammenspiel mit einer IDE in Kauf. Ich wähle mit ChatGPT also eine strategische Ineffizienz.

Das will ich Ihnen nicht unbedingt empfehlen, sondern zunächst nur als Erklärung dafür geben, warum Sie in diesem Artikel keine integrierte KI sehen.

So viel zur Werkzeugkonstante. Was aber ist meine Perspektive auf die Nutzung von ChatGPT & Co. in der Softwareentwicklung? Was ergibt sich damit gar für Agilität und Clean Code Development?

KI als vertrauenswürdiger Coder?

Dass die KI passabel, gar gut codieren kann, sollte inzwischen auf der Hand liegen. Ich setze sie täglich als flexiblen Codegenerator ein. In [Bild 1](#) als triviales Beispiel die halbe Diamond-Kata. Das Ergebnis lässt sich aus dem Stand sehen. Mit



Schreibe eine Typescript-Funktion, die ein Dreieck aus ASCII-Zeichen generiert. Eingabeparameter ist die gewünschte Größe des Dreiecks; Ergebnis ist das Dreieck in Form einer mehrzeiligen Zeichenkette.

Hier sind einige Beispiele:

Dreieck mit Größe 2:

```

*
**
***

```

Dreieck mit Größe 3:

```

*
**
***
****
*****

```

Dreiecke haben eine Mindestgröße von 2. Die Punkte in den Beispielen sind Platzhalter Leerzeichen und sollen im Ergebnis nicht auftauchen.

Schreibe nun eine Typescript-Funktion, die solche Dreiecke generiert:

Klar, hier ist eine mögliche TypeScript-Funktion, die ein Dreieck aus ASCII-Zeichen generiert:

```

typescript
function generateTriangle(size: number): string {
  if (size < 2) {
    throw new Error("Triangle size must be at least 2.");
  }

  let triangle = "";
  for (let row = 0; row < size; row++) {
    const numStars = row * 2 + 1;
    const numSpaces = size - row - 1;
    triangle += " ".repeat(numSpaces);
    triangle += "**".repeat(numStars);
    triangle += " ".repeat(numSpaces);
    triangle += "\n";
  }

  return triangle;
}

```

ChatGPT generiert Code für natürlichsprachliche Anforderungen ([Bild 1](#))

```
console.log(generateTriangle(4));
```

wirft der Code das Ergebnis in [Bild 2](#) aus. Selbstverständlich funktioniert das auch in anderen Domänen; diese hier habe ich nur wegen der unmittelbaren Verständlichkeit gewählt. Schon in meinem vorherigen Artikel zum Thema finden Sie ein wachsendes, umfangreicheres Beispiel [1]. Und ChatGPT erweitert den Code auch gern für neue Anforderungen. Ich will nicht bei halben Sachen stehen bleiben: Ein gan- ▶

zer Diamant soll generiert werden (Bild 3). Leider überzeugt das Ergebnis nicht (Bild 4). ChatGPT erinnert zwar die vorher generierte Funktion, benutzt sie aber falsch. Das ist offensichtlich.

Liegt das an ChatGPT? Oder liegt das an mir? Vielleicht muss ich mir mehr Mühe beim Prompt geben? Ich probiere es noch mal mit einem Beispiel im Prompt (Bild 5).

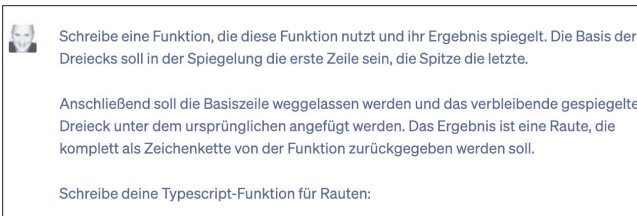
Tatsächlich verwendet ChatGPT jetzt die `generateTriangle()`-Funktion grundsätzlich



Der ChatGPT-Code läuft fehlerfrei und gibt das geforderte Dreieck aus (Bild 2)

korrekt und lässt den oberen Teil des Diamanten vollständig mit einem Aufruf generieren (Bild 6).

Aber ach, bei einem Aufruf von `generateDiamond()` zeigt sich, dass die Dinge nicht rosig sind. Es kommt zu einem Laufzeitfehler. Und bei genauerem Hinsehen zeigt sich auch, dass ChatGPT viel zu kompliziert gedacht hat: Das `lowerTriangle` wird durch Spiegelung generiert und ist damit im Grunde fertig. Warum soll es in `concat()` dann noch mal zeilenweise behandelt werden? Es geht doch viel simpler und auch fehlerfrei:



Neue Anforderungen, die den Code erweitern sollen (Bild 3)

```
const diamond = upperTriangle
    .concat(lowerTriangle).join("\n") + "\n";
```

Das Programm stürzt nicht mehr ab. Das ist ein Fortschritt. Allerdings ist das Ergebnis trotzdem nicht korrekt, wie die Beispiele in Bild 7 zeigen. Der ChatGPT-Code ist funktional knapp daneben: Eine überflüssige Leerzeile wird eingefügt

```
function generateDiamond(size: number): string {
    if (size < 2) {
        throw new Error("Diamond size must be at least 2.");
    }

    let diamond = "";

    // Generiere das obere Dreieck der Raute
    for (let i = 1; i <= size; i++) {
        diamond += generateTriangle(i);
    }

    // Generiere das untere Dreieck der Raute, gespiegelt
    for (let i = size - 1; i >= 1; i--) {
        diamond += generateTriangle(i);
    }

    return diamond;
}
```

ChatGPT hat sich Mühe gegeben – aber Mühe allein genügt nicht (Bild 4)

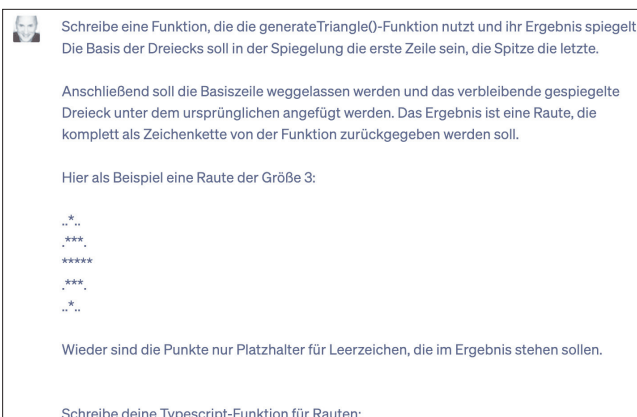
```
function generateDiamond(size: number): string {
    if (size < 2) {
        throw new Error("Diamond size must be at least 2.");
    }

    const upperTriangle = generateTriangle(size).trim().split("\n"); // Generiere das obere Dreieck
    // Spiegle das obere Dreieck
    const lowerTriangle = [...upperTriangle.slice(0, -1)].reverse(); // Entferne die Basislinie

    // Erstelle die Raute durch Kombination des oberen und unteren Dreiecks
    const diamond = upperTriangle.concat(lowerTriangle).map((line) => {
        const numSpaces = (size - line.length) / 2;
        return " ".repeat(numSpaces) + line + " ".repeat(numSpaces);
    }).join("\n") + "\n";

    return diamond;
}
```

Ein besserer Prompt führt zu einer passenden Nutzung der zuerst generierten Funktion (Bild 6)



Ein ausführlicheres Prompt führt hoffentlich zu besserem Code (Bild 5)

und die Basislinie des oberen Dreiecks ist im unteren nicht wie gefordert gelöscht.

Was nun? Was sagt uns das über die Codierungskompetenz von ChatGPT & Co.? ChatGPT & Co. können codieren und sind eine große Hilfe. Ich möchte ihre Unterstützung nicht mehr missen. Meine Reichweite hat sich deutlich vergrößert. Ich wechsele mit ihnen ohne Zögern zwischen Sprachen, zum Beispiel C#, TypeScript, Python. ChatGPT kennt die Details der Sprachen und auch Plattformen/Frameworks für die Codegenerierung (aktive Sprachkompetenz); ich kann den Code verstehen (passive Sprachkompetenz).

Die Elaborate von ChatGPT & Co. sind jedoch nicht frei von Fehlern. Anforderungen werden missverstanden, untauglicher Code wird generiert. Es ist sonnenklar: ChatGPT-Code kann nicht ungeprüft übernommen werden.

Die Konsequenz, die ich daraus ziehe, lautet: ChatGPT ist ein sehr hilfreicher und schneller Pair Programmierer für mich. Er ist der Driver, ich bin der Navigator. Bei aller Unterstützung, die ich durch ChatGPT erfahre, kann ich mich allerdings nicht blind darauf verlassen, was generiert wird. Ich muss es überprüfen.

Wer hätte das gedacht? Nein, überraschend kann das überhaupt nicht sein. Denn selbst wenn ChatGPT ein perfekter Programmierer wäre – also perfekt beim Verständnis der Anforderungen genauso wie perfekt bei ihrer Umsetzung in Code – bliebe ich, der Entwickler, immer noch imperfekt in meiner Formulierung der Anforderungen. Der Spruch *Garbage in, garbage out* gilt auch in Zeiten von KI-Co-Piloten.

Was außerdem für ChatGPT spricht: Es geht ja auch nicht immer darum, sofort Code zu generieren. Wenn ich auf *ancient code* stoße, also Code, dessen Autoren nicht mehr ansprechbar sind, kann ich ChatGPT um eine Analyse bitten. Ja, ich kann sogar darum bitten, den Code zu kommentieren (Bild 8), gar zu refaktorisieren (Bild 9).

Also warum sollte ich darauf verzichten wollen, nur weil der von ChatGPT generierte Code genauso Fehler enthalten kann wie der eines Kollegen? Allemaal, da ChatGPT schneller in der Codegenerierung ist als jeder Kollege, niemals müde wird, *egolos* ist und mich nicht in Diskussionen über Dogmen oder Stil verstrickt.

Nein, auf ChatGPT zu verzichten, wäre dumm. Allerdings dürfen wir dieses Werkzeug auch nicht naiv einsetzen. Die Fehlbarkeit von ChatGPT ist ein Aufruf an uns, darauf vorbereitet zu sein.

Wir müssen sie vorhersehen und geeignet kompensieren. Wir müssen uns ein Vorgehen zurechtlegen, in dem ChatGPT mit seinen Fähigkeiten eine hilfreiche Rolle spielt.



Auch Code, der nicht mehr abstürzt, ist nicht unbedingt fehlerfrei (Bild 7)

Die Limitationen nicht nur der KI

Wir müssen uns mit der KI arrangieren. Sie ist willig, aber hat eben ihre Grenzen. Wie gehen wir damit um? Welche Grenzen sind relevant?

Da ist zunächst die physikalische Grenze der Zahl der Tokens. Prompts und Completions können nur eine bestimmte Anzahl Tokens umfassen. Derzeit (April 2023) sind es für diejenigen, die noch keinen Zugriff im

Playground nach GPT-4 haben, um die 4000 Tokens insgesamt für Prompt und Completion. Das ist nicht so viel. Ausführliche Anforderungen lassen sich so nicht vermitteln. Und umfänglicherer Code kann so auch nicht generiert werden. Ich denke, Sie werden auch schon Fälle erlebt haben, in denen ChatGPT bei der Completion-Generierung einfach nach einer Weile angehalten hat. Das ist sehr frustrierend und unvorhersehbar. Mit GPT-4 wird sich Limitation irgendwann weiten – grenzenlos werden Input und Output deshalb jedoch nicht.

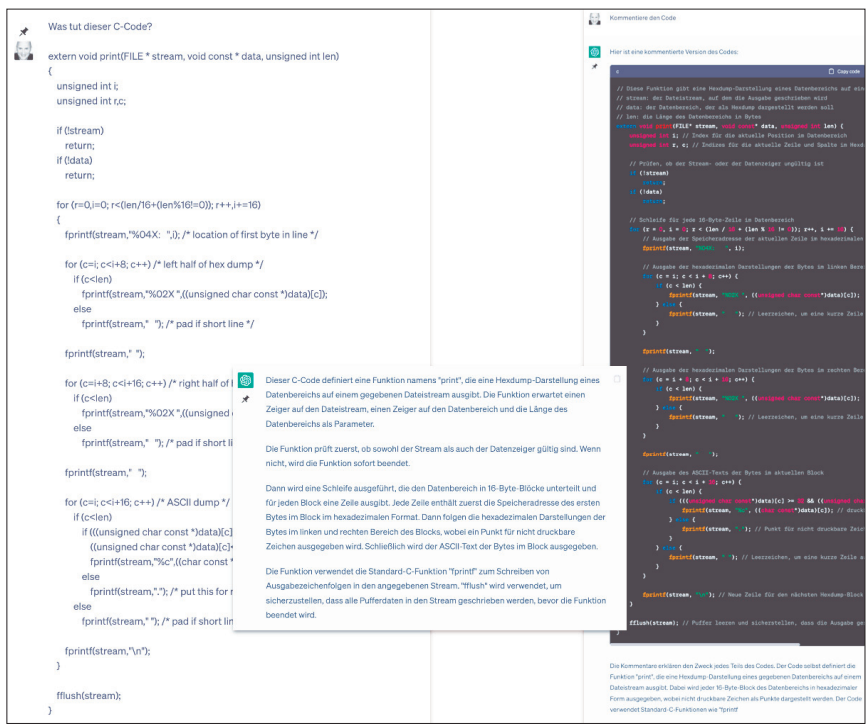
Selbst wenn die physikalische Grenze jedoch eigentlich keine mehr sein sollte, sehe ich inhaltlich immer noch eine. Codegenerierung ist etwas anderes, als einen Blogartikel zu schreiben. Bei Code braucht es Präzision und Fokus bis zum letzten Zeichen. Einem Blogartikel mit Rauschen von acht Seiten würde ich mehr vertrauen als C#-Code von gleicher Länge. Dass ChatGPT & Co. in absehbarer Zeit Tausende Zeilen hochqualitativen nicht trivialen Code in einem Rutsch generieren, glaube ich nicht. Insofern sehe ich auch inhaltlich nur eine Unterstützung bei relativ überschaubarem Code.

Und selbst wenn Tausende Zeilen in einer Completion generiert werden, bleiben Fragen:

- Ist der Code wirklich syntaktisch korrekt?
- Ist er korrekt in Nutzung von APIs?
- Ist er semantisch korrekt, das heißt, enthält er keine Laufzeitfehler und tut er wirklich, was gewünscht war?

Ganz zu schweigen davon, ob der Wunsch überhaupt korrekt, vollständig, unmissverständlich formuliert war.

Womit ich bei einer Limitation auf Entwicklerseite bin. Es ist ja nicht so, dass wir perfekt wären und nur ChatGPT fehlerbar. Meiner Erfahrung nach habe ich schlicht nur eine begrenzte Fähigkeit, Anforderungen gut zu formulieren. ►



ChatGPT analysiert, kommentiert und dokumentiert Code (Bild 8)

Manchmal gelingt mir das besser, manchmal zeigen mir gerade die Missverständnisse von ChatGPT, wie wenig klar ich mir selbst noch über Anforderungen bin.

Wir bringen all unsere menschlichen Schwächen in den Dialog mit der KI ein. Das ist nicht zu unterschätzen – auch wenn die dialogische Interaktion helfen kann, sie zu kompensieren. Wie das auch der Fall im Gespräch mit einem menschlichen Gegenüber ist.

Selbst wenn also GPT-4 oder später GPT-5 längere und längere Anforderungen von mir akzeptiert, kann ich nicht ohne Feedback wissen, welche Qualität meine Prompts wirklich haben. In der menschlichen Kommunikation, wo der Empfänger von Anforderungen zügig zu mir zurückkommen wird, wenn er ein Problem fühlt, wird die Interaktion auch vor Erzeugung von Ergebnissen in Gang gehalten. Bei ChatGPT ist das nicht der Fall. Oder zumindest nicht ohne Weiteres, denn ich kann ja auch Prompts so formulieren, dass ChatGPT mich fragt (inverted prompting), also Klarstellungen erbitten kann. Dennoch scheint mir die KI (noch) tendenziell passiv/empfangend.

Also: ChatGPT & Co. haben ihre Limitationen, wir als Entwickler, das heißt ihre Auftraggeber, haben auch unsere Limitationen. Deshalb scheint es mir ratsam, sich (zunächst) auf kleinere Brötchen zu konzentrieren. Zu denken, dass nach wenigen Zeilen Prompt eine komplette Software von Zehntausenden Zeilen herauspurzelt, inklusive Deployment-Skripten ..., das halte ich (noch) für unrealistisch.

Wir können nicht erwarten, dass Anforderungen jeder Größenordnung auf einen Prompt hin oder mit ein bisschen Dialog umgesetzt werden. Wir sind vielmehr aufgerufen, ChatGPT in begrenzten, überschaubaren Horizonten einzusetzen. Wie bisher müssen wir also große Probleme in kleinere zerlegen. So tief müssen wir zerlegen, bis zwei Bedingungen erfüllt sind:

- Die Completion, die ChatGPT generieren kann, muss das gesamte Problem lösen.
- Das Problem muss so überschaubar und klar sein, dass wir dafür einen unmissverständlichen Prompt formulieren können.

Und schließlich: In unserer Aufforderung zur Codegenerierung müssen wir die automatisierte Testbarkeit für den von ChatGPT generierten Code von Anfang an mitdenken. Denn nur über automatisierte Tests können wir Vertrauen in den Code gewinnen. ChatGPTs Code unterscheidet sich darin eigentlich nicht von dem von Kollegen oder in Open-Source-Projekten bei GitHub. Ohne Testsuite wird es sehr, sehr mühsam, seine Qualität einzuschätzen oder gar Veränderungen vorzunehmen.

Wie sicher bin ich mir?

Eine professionelle Nutzung von ChatGPT in der Programmierung fängt für mich beim Menschen an. Wir als Entwickler müssen uns fragen: Sind wir schon bereit, der KI einen Prompt vorzulegen, um Produktionscode generieren zu lassen? Oder etwas allgemeiner: Für welche Art von Codegenerierung sind wir schon bereit?

Sind wir bereit für Produktionscode? Oder sind wir nur bereit für experimentellen Code?

- Produktionscode ist der Code, der beim Kunden läuft. Der

```

c
// Gibt eine einzelne Hexdump-Spalte (8 Bytes) aus
static void print_hexdump_column(FILE* stream, void const* data, unsigned int len, unsigned int offset, unsigned int i) {
    for (i = start; i < end; i++) {
        if (i < len) {
            fprintf(stream, "%02X ", ((unsigned char const*)data)[offset + i]);
        } else {
            fprintf(stream, " "); // Leerzeichen, um eine kurze Zeile auszufüllen
        }
    }
}

// Gibt die hexadezimalen Darstellungen der Bytes im aktuellen Block aus
static void print_hexdump_hex(FILE* stream, void const* data, unsigned int len, unsigned int offset) {
    print_hexdump_column(stream, data, len, offset, 0, 8);
    fprintf(stream, " ");
    print_hexdump_column(stream, data, len, offset, 8, 16);
    fprintf(stream, " ");
}

// Gibt den ASCII-Text der Bytes im aktuellen Block aus
static void print_hexdump_ascii(FILE* stream, void const* data, unsigned int len, unsigned int offset) {
    for (i = 0; i < 16; i++) {
        if (i < len) {
            if (((unsigned char const*)data)[offset + i] >= 32 && ((unsigned char const*)data)[offset + i] <= 127) {
                fprintf(stream, "%c", ((char const*)data)[offset + i]); // ASCII-Zeichen
            } else {
                fprintf(stream, "."); // Punkt für nicht druckbare Zeichen
            }
        } else {
            fprintf(stream, " "); // Leerzeichen, um eine kurze Zeile auszufüllen
        }
    }
}

// Gibt eine Zeile des Hexdumps aus
static void print_hexdump_line(FILE* stream, void const* data, unsigned int len, unsigned int offset) {
    // Ausgabe der Speicheradresse der aktuellen Zeile im hexadezimalen Format
    fprintf(stream, "%04X: ", offset);

    // Ausgabe der hexadezimalen Darstellungen der Bytes im linken und rechten Bereich
    print_hexdump_hex(stream, data, len, offset);

    // Ausgabe des ASCII-Texts der Bytes im aktuellen Block
    print_hexdump_ascii(stream, data, len, offset);

    fprintf(stream, "\n"); // Neue Zeile für den nächsten Hexdump-Block
}

// Gibt einen Hexdump des Datenbereichs aus
extern void print(FILE* stream, void const* data, unsigned int len) {
    unsigned int offset = 0;
    unsigned int i;

    if (!stream || !data) {
        return;
    }

    // Schleife für jede 16-Byte-Zeile im Datenbereich
    for (i = 0; i < (len / 16 + (len % 16 != 0)); i++, offset += 16) {
        print_hexdump_line(stream, data, len, offset);
    }

    fflush(stream); // Puffer leeren und sicherstellen, dass die Ausgabe geschrieben wird
}

```

Refaktorisierter Code nach einigen „extractfunction“-Runden (Bild 9)

muss korrekt sein. Der muss auch sauber sein. Dafür braucht es gute Testabdeckung. Produktionscode sollte nicht leichtfertig angefasst/verändert werden. Denn jede Veränderung ist eine Belastung für seine Struktur und muss womöglich durch Refaktorisierung bereinigt werden.

- Experimenteller Code dient hingegen nur der Exploration [2]. Er dient dazu herauszufinden, ob wir Anforderungen korrekt verstanden haben und/oder wie Anforderungen

umgesetzt werden könnten. Experimenteller Code stellt Informationen für das Team her. Produktionscode stellt Nutzen für den Kunden her. Beides darf nicht verwechselt werden! Prototyp und Spike sind Begriffe der Exploration. Test ist ein Begriff aus der Produktion. Experimenteller Code muss nicht sauber und auch nicht testabgedeckt sein. Er hat keine Langlebigkeit.

Was für einen Prompt bin ich also bereit zu schreiben? Sind mir die Anforderungen schon klar genug, dass ich Produktionscode generieren lassen kann? Oder reicht es nur für ein Code-Experiment mit viel geringeren Anforderungen?

Dass Sie sich diese Frage stellen, ist für mich der erste Schritt in einem Vorgehen, das ich Requirements-First Development (RFD) nenne.

Requirements-First Development

Warum RFD und nicht TDD oder BDD? Weil für mich der Begriff Requirement (Anforderung) viel allgemeiner ist. Mit ChatGPT kann ich über Anforderungen in natürlicher Sprache sprechen und mir Code generieren lassen. Bei TDD und BDD ist die Sprache formal; ich formuliere Testcode. Und Code wird aus ihnen nicht generiert.

Durch die KI wird unser Vorgehen auf ein neues Level gehoben. Wir müssen nicht so schnell formal werden wie früher. Wir können den Regler, wie wir Anforderungen und die Überprüfung ihrer Umsetzung formulieren, über einen viel weiteren Bereich schieben.

Wir können Anforderungen in Form von Tests formulieren (Bild 10) und ChatGPT versteht tatsächlich, worum es geht (Bild 11); echte Test-first-Codierung ist also möglich.

Bei jedem Coding-Dojo wäre die Bewältigung dieser Aufgabe auf einen ganzen Abend ausgedehnt worden. Denn das zu übende classical TDD wäre nicht mit einer Liste von Akzeptanztestfällen zufrieden gewesen, sondern hätte verlangt, sich der Implementation schrittweise zu nähern.

Mit ChatGPT wird klar, dass das kein pauschales Vorgehen sein sollte. Es kostet schlicht unnötig Zeit, wenn der Schwierigkeitsgrad für ein Problem niedrig ist. In meinem Buch Test-first Codierung [3] gehe ich ausführlich darauf ein, wie das Test-first-Vorgehen den Schwierigkeitsgraden angepasst werden sollte. Nach der dortigen Kategorisierung macht ChatGPT aus vielen Problemen triviale, das heißt solche, für die lediglich Akzeptanztests formuliert werden sollten, um anschließend die Implementation einfach hinzuschreiben.

Das ist schon hilfreich – doch die Testformulierung macht etwas Mühe. Warum die Anforderungen nicht etwas informeller notieren? Wir können das Format für Testfälle wählen – solange darin eine für ChatGPT erkennbare Systematik liegt (Bild 12). Und ChatGPT übernimmt auch gern die Aufgabe, daraus lauffähigen Testcode abzuleiten (Bild 13).

Aber warum so strukturiert, wenn ChatGPT uns sogar versteht, wenn wir schreiben, wie uns der Schnabel gewachsen ist? Wir können uns Formalitäten sparen und schlicht erklären, was wir wollen (Bild 14). Testfälle für den eigenen Code liefert die KI auf Wunsch auch gleich mit (Bild 15). Über die Qualität der Testfälle lässt sich diskutieren. Als Startpunkt ►

Implementiere eine Typescript Convert() Funktion, die folgende Tests erfüllt:

```

Deno.test("Transform one number", 0 => {
  assertEquals(Convert("1"), 1);
});

Deno.test("Transform several numbers", 0 => {
  assertEquals(Convert("2 1"), 3);
  assertEquals(Convert("1 2"), 3);
});

Deno.test("Transform several numbers", 0 => {
  assertEquals(Convert("2.56 -1.5"), 1.06);
});

Deno.test("Transform several numbers", 0 => {
  assertEquals(Convert("1, 999, 42"), 1042);
});

```

Anforderungen allein spezifiziert durch Testfälle. Absichtlich geben die Testbezeichnungen und der Funktionsname keinen Hinweis auf die konkrete Aufgabe (Bild 10)

Da es sich hierbei um eine Implementierung in TypeScript handelt, gehen wir davon aus, dass die benötigten Typen (z.B. "number") bereits importiert wurden. Hier ist eine mögliche Implementierung der Convert() Funktion:

```

typescript
function Convert(input: string): number {
  // Split input string into an array of numbers
  const numbers = input.split(/[ ,]+/).map(Number);

  // Sum up the numbers
  const sum = numbers.reduce((total, num) => total + num, 0);

  return sum;
}

```

Die Funktion zerlegt den Eingabestring mit Hilfe der split() Funktion in ein Array von Zahlen. Der RegExp `/[,]+/` definiert dabei, dass die Zahlen durch mindestens ein Leerzeichen oder ein Komma getrennt sein können. Die map() Funktion wird verwendet, um jedes Element des Arrays in eine Zahl umzuwandeln.

Anschließend wird das Array mit Hilfe der reduce() Funktion aufsummiert, indem jede Zahl auf den bisherigen Gesamtsummanden addiert wird.

Die Implementierung sollte die oben beschriebenen Tests erfüllen. Beachte, dass du vor dem Ausführen der Tests die notwendigen Module importieren musst, z.B.:

ChatGPT, implementiert nach Testfällen, erklärt den Ansatz und erinnert, den Import der Testbibliothek nicht zu vergessen (Bild 11)

Schreibe eine Typscript-Funktion für Deno, die die folgenden Testfälle erfüllt:

```

Input: "1"
Output: 1

Input: "2 1"
Output: 3

Input: "1 2"
Output: 3

Input: "2.56 -1.5"
Output: 1.06

Input: "1, 999, 42"
Output: 1042

```

Generiere auch Testcode für die Testfälle.

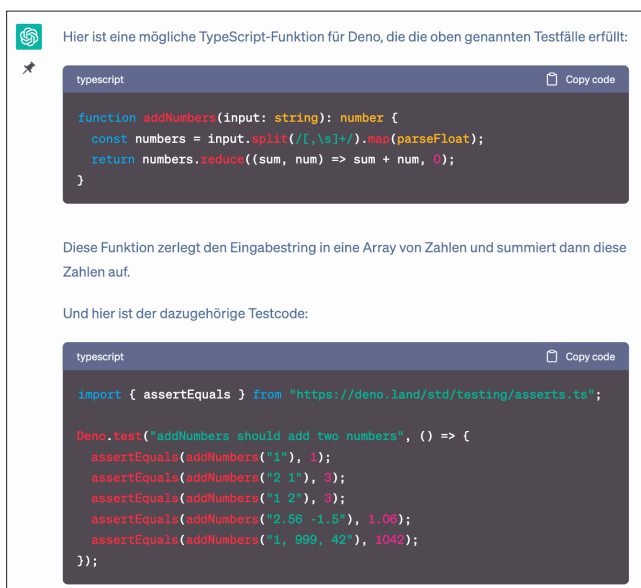
Semi-formale Testfälle versteht ChatGPT auch (Bild 12)

taugen sie auf jeden Fall. An denen kann der generierte Code schon mal gemessen werden. Wenn das Ergebnis rot ist, lohnt ein genauerer Blick – sowohl auf den funktionalen wie auf den testenden Code. In beiden können Fehler stecken.

Wer mit den Testfällen nicht zufrieden ist, kann auch um weitere bitten. ChatGPT liefert geduldig nach.

Allerdings zeigt sich hier wieder exemplarisch eine Begrenzung von ChatGPT: Einer der Testfälle schlägt fehl. Die Idee, auch Zahlen in wissenschaftlicher Notation zu prüfen, war also gut. Doch wo liegt der Fehler? In der Implementation oder im Test?

Der Test selbst ist fehlerhaft. Die Erwartung $123006,78$ ist inkorrekt. Die Summation der beiden Input-Zahlen ergibt 122999.9322 , wenn Typescript sie ausführt: `console.log(1.23e+5 + -6.78e-2)`.



Wer Zeit sparen will, lässt sich auch Testcode von ChatGPT generieren (Bild 13)

Allerdings ist der korrekte Wert nicht der, der den Testfehler auslöst. Es ist ein ganz anderer Wert: $-2,55$. Das deutet darauf hin, dass die Implementation der Lösung falsch ist. Wie sich herausstellt, zerlegt der reguläre Ausdruck (Bild 14) die Zahlen inkorrekt. Das „e“ wird als Trennzeichen interpretiert.

Das ist ein wunderbares Beispiel zur Illustration von Nutzen, aber auch Grenzen von ChatGPT:

- ChatGPT ist weit entfernt von Unfehlbarkeit bei der Generierung von Lösungen. Der Ansatz mit dem regulären Ausdruck ist grundsätzlich gut – nur wurde darin noch nicht bedacht, dass eben Zahlen in wissenschaftlicher Notation im Input vorkommen könnten. Andererseits: Ist ChatGPT dafür zu tadeln? Oder ist es eine Unschärfe in den Anforderungen gewesen, wie ich sie formuliert habe, zu der ChatGPT eine erst mal legitime Implementation generiert hat? Ich tendiere zu Letzterem.
- ChatGPT hat andererseits auch noch dokumentierte Schwäche beim Umgang mit *computation* [4]. Die KI rechnen zu

lassen, bringt nur begrenzt korrekte Antworten, wie hier zu sehen ist. Die Large Language Models (LLM) selbst sind einfach dafür (noch) nicht gemacht, Kalkulationen durchzuführen oder Algorithmen auszuführen. Obwohl ChatGPT verstanden hatte, worum es geht, und die Aufgabe (weitgehend) korrekt gelöst hatte, konnte es doch nicht die beiden Zahlen in dem Testfall selbst korrekt addieren.

Gerade diese Enttäuschung unserer Erwartung macht aber deutlich, worauf es beim Pair Programming mit der KI ankommt:

- detaillierte, vollständige, unmissverständliche Anforderungen in welcher Form auch immer
- automatisierte Tests

Wirklich gute Anforderungen stehen am Anfang der Programmierung mit ChatGPT. Nicht umsonst habe ich das Vorgehen Requirements-First Development genannt. Alles beginnt damit, dass wir uns um glasklare Anforderungen bemühen; dass wir sie selbst verstehen und auch formulieren können, ohne gleich eine Lösung zu implementieren, ist Voraussetzung.

Ob diese Anforderungen als Testcases, Pseudocode oder in natürlicher Sprache formuliert sind, ist zweitrangig. Sie können eine Form wählen, wie es am besten für die Anforderungen ist. Manche sind leichter formal zu formulieren, statt sie in Sätze zu verpacken – bei anderen ist es umgekehrt. Und manchmal sollten Sie beides tun. „Doppelt hält besser“ gilt für KI-Prompts in besonderem Maße.

Doch selbst bei aller Mühe beim Prompting dürfen Sie ohne automatisierte Tests anschließend keinen KI-Code übernehmen. Das sollte klar sein. Für den Code der Kollegen gilt natürlich dasselbe. Das bedeutet jedoch nicht nur, dass automatisierte Tests existieren, sondern vor allem, dass der KI-generierte Code überhaupt erst mal testbar ist. Es muss Ansatzpunkte für Tests geben. Ihre Anforderungen müssen also so formuliert sein, dass ChatGPT sie in einer Weise umsetzt, dass sofort Ösen vorhanden sind, in die Tests eingehängt werden können.

Ob Sie die Tests vorher formulieren oder hinterher auch von ChatGPT generieren lassen, finde ich auch nicht so wichtig. Im Pair Programming mit KI verliert Test-first-Codierung etwas an Bedeutung, weil der Produktionscode so schnell produziert wird. Viel, viel wichtiger sind die Anforderungen.

Eigentlich sollte das sogar nichts Besonderes sein. Anforderungen waren immer schon wichtig – nur sind sie bei der ganzen agilen iterativen Kommunikation ein wenig unter die Räder gekommen. Gerade bei co-located Teams schien das jederzeitige Nachfragen so billig, dass man sich keine allzu großen Gedanken über Anforderungen vor der Codierung machen musste. Wer während der Implementation eine Unklarheit bemerkt, kann ja sofort andere unterbrechen, um nachzufragen.

Bei der KI ist das nun anders. Als Entwickler wechseln Sie in die Rolle des Product Owners (PO). Jetzt sind Sie dran, Anforderungen klar zu formulieren. Ich finde, das ist eine gute, ernüchternde Entwicklung. Sie hat hoffentlich einen guten Einfluss auf Mensch-zu-Mensch Kommunikation: Entwickler dürfen lernen, von POs mehr Klarheit zu verlangen, um ihre Arbeit einfacher zu machen.

```

typescript Copy code

import { assertEquals } from "https://deno.land/std/testing/asserts.ts";

Deno.test("sumNumbers should return 0 if the input is empty", () => {
  const result = sumNumbers("");
  assertEquals(result, 0);
});

Deno.test("sumNumbers should return 0 if the input contains no numbers", () => {
  const result = sumNumbers("hello world!");
  assertEquals(result, 0);
});

Deno.test("sumNumbers should return the correct sum for a string of integers", () => {
  const result = sumNumbers("1 2 3 4 5");
  assertEquals(result, 15);
});

Deno.test("sumNumbers should return the correct sum for a string of floats", () => {
  const result = sumNumbers("1.1 2.2 3.3 4.4 5.5");
  assertEquals(result, 16.5);
});

Deno.test("sumNumbers should return the correct sum for a string of mixed numbers", () => {
  const result = sumNumbers("1 -2.5 3.75 -4 5");
  assertEquals(result, 3.25);
});

Deno.test("sumNumbers should handle semicolons and commas as delimiters", () => {
  const result = sumNumbers("1;2;3;4;5;6");
  assertEquals(result, 17.4);
});


Deno.test("sumNumbers should handle negative numbers", () => {
  const result = sumNumbers("-1 -2.2 3.3 -4.4 5");
  assertEquals(result, 0.7);
});

```


ChatGPT generiert auch ohne Vorgaben Testfälle für den eigenen Code (Bild 15)

Decomposition is king!

ChatGPT hat nur begrenzte Kapazität, ChatGPT ist bei computation schwachbrüstig, ChatGPT missversteht, ChatGPT generiert inkorrekten Code und der Mensch ist in seiner Anforderungsbeschreibung ebenfalls fehlbar und missverständlich. Kann unter diesen Umständen ChatGPT eine Hilfe sein? Absolut! Gefahr erkannt, Gefahr gebannt! Und letztlich sind

 Schreibe eine Typescript-Funktion für Deno, die ganzen Zahlen und auch Zahlen mit Nachkommastellen in einer Zeichenkette summiert. Die Zahlen können positiv oder negativ sein. Die Zahlen sind durch Leerzeichen oder Komma oder Semikolon getrennt.

Generiere auch einige Testfälle für den Deno-Testframework.

 Hier ist eine Typescript-Funktion für Deno, die ganzen Zahlen und Zahlen mit Nachkommastellen in einer Zeichenkette summiert:

```

typescript Copy code

function sumNumbers(inputString: string): number {
  const regex = /-?\d+(?:\.\d+)?/g;
  const matches = inputString.match(regex);
  if (!matches) {
    return 0;
  }
  return matches.reduce((acc, val) => acc + parseFloat(val), 0);
}

```

Diese Funktion verwendet eine reguläre Ausdrucksweise, um alle ganzen Zahlen und Zahlen mit Nachkommastellen in der Eingabezeichenkette zu finden. Dann werden alle gefundenen Zahlen zu einer Summe addiert und als Ergebnis zurückgegeben.

ChatGPT versteht auch Anforderungen in natürlicher Sprache (Bild 14)

ChatGPTs Schwächen im Grunde auch die des Menschen. Als würde der Titel Senior Software Developer davor schützen, Fehler wie ChatGPT zu begehen!

Irren ist nicht nur menschlich. Wenn wir dennoch Menschen zutrauen, mit uns und für uns zu codieren, dann können wir es ChatGPT auch zutrauen. Wir dürfen nur nicht naiv sein. Gute Vorbereitung und Erwartungsmanagement helfen, die Zusammenarbeit mit ChatGPT erquicklich zu machen.

Für mich sind weiterhin die meisten Programmierprobleme kompliziert im Sinne der Kategorisierung aus [3]. Das bedeutet, ich mache mir Gedanken über eine Problemzerlegung in komplementäre Sub-Probleme. Dadurch entsteht ein Baum von immer weniger schwierigen Problemen, an dessen Blättern die Umsetzung trivial bis einfach ist. Das heißt, es gibt eine Single-shot-Lösung oder ich muss nur ein paar Iterationsrunden drehen. Genau dabei hilft mir ChatGPT am besten.

Bei der Zerlegung überlege ich mir, wie sie funktional (Funktionen) und kategorial (Module) am besten aussieht. All meine Kompetenz als Entwickler im Bereich Softwareentwurf kommt mir hier zugute [5]. Die KI macht uns also nicht so schnell überflüssig. Sie verschiebt vielmehr unseren Fokus. Kenntnisse über Sprachfeatures, APIs, Frameworks treten zurück, Analyse und Design rücken in den Vordergrund.

Anforderungen verstehen, Lösungsansätze daraus ableiten, dann modularisieren und durch geschicktes Prompt Engineering ChatGPT den nervigen Kleinkram übernehmen lassen: So sehe ich die Softwareentwicklung in nächster Zukunft. Der Mensch fürs Grobe, für den Überblick und als Supervisor, die KI für die nitty gritty details. Am besten demonstriere ich das an einem Beispiel. Das finden Sie im nächsten Artikel. Nutzen Sie die Zeit bis dahin, um sich schon mit ChatGPT & Code vertraut(er) zu machen. Seien Sie kühn in Ihren Experimenten. Loten Sie Chancen und Grenzen aus. ■

[1] Ralf Westphal, *Pair Programming mit der KI*, dotnetpro 6/2023, Seite 82 ff., www.dotnetpro.de/A2306ChatGPT

[2] Ralf Westphal, *Codieren ohne Tests*, www.dotnetpro.de/SL2307ChatGPT1

[3] Ralf Westphal, *Test-first Codierung – Programming with Ease Teil 1*, 493 Seiten (ePub, PDF, mobi, Web), www.dotnetpro.de/SL2307ChatGPT2

[4] Stephen Wolfram, *What Is ChatGPT Doing ... and Why Does It Work?*, ISBN 978-1579550813

[5] Ralf Westphal, *Softwareentwurf mit Flow-Design – Programming with Ease Teil 2*, 584 Seiten (ePub, PDF, mobi, Web), www.dotnetpro.de/SL2307ChatGPT3



Ralf Westphal

ist Trainer, Berater und Mitgründer der Clean-Code-Developer-Initiative (clean-code-developer.de). Seine Schwerpunkte sind dauerhaft hohe Produktivität für die Softwareentwicklung und zukunftsfähige Teamorganisation.

<https://ralfw.de>

dnpCode

A2307ChatGPT