

REQUIREMENTS-FIRST DEVELOPMENT - TEIL 2

Das Zugpferd steuern

Anforderungen in die KI kippen und das fertige Programm herausholen – nein, so einfach geht es nicht. Aber helfen kann ChatGPT an vielen Stellen.

Es gibt einen Rahmen, in dem ich das Pair Programming mit KIs wie ChatGPT sehe. Im ersten Teil dieser Artikelserie [1] habe ich ihn so beschrieben: „Anforderungen verstehen, Lösungsansätze daraus ableiten, daraufhin modularisieren und dann 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.“

Dass Sie einer KI Hunderte Seiten Anforderungen vorlegen und die daraufhin fehlerfreien Code produziert, ist in nächster Zeit wohl nicht zu erwarten. Das hat weniger mit der Leistungsfähigkeit von ChatGPT zu tun – begrenztes Kontextfenster, schwachbrüstig bei Computation –, sondern mit der Impräzision von Anforderungen und Grenzen der Testabdeckung.

- Anforderungen schon für kleine Probleme so zu formulieren, dass sie unmissverständlich sind, ist sehr schwierig. Solange ein lebendiger Dialog zwischen Anforderer und Entwickler besteht, kann das jedoch ausgeglichen werden. Selbst ChatGPT kann das [2] (Bild 1). Aber wie ein Mensch ist ChatGPT dabei stimmungsbeeinflusst: Heute findet es dies unklar, morgen jenes. Bild 2 [3] zeigt, dass nicht immer dieselben Fragen gestellt werden. Je umfanglicher Anforderungen sind, desto dichter der Nebel, in dem sich eine Lösung verbirgt. Auch mit ChatGPT wird daher ein inkrementelles Vorgehen nötig sein. Dabei kann ChatGPT ebenfalls Kreativität entwickeln, wie der Chat in [4] beweist, doch insgesamt ist das Vorgehen (noch) wackelig und nicht alle vorgeschlagenen Inkremente sind Inkremente. Deshalb: Auf der Anforderungsseite ist eine menschliche Planung mit Augenmaß nötig.
- Die Aussagekraft von Tests ist abhängig von der Menge des Codes, den sie abdecken. Je tiefer der Code geschachtelt ist, desto schwieriger ist es, ihn aussagekräftig abzudecken. Es reichen also nicht allein Akzeptanztests an den Entry Points einer Software; es lassen sich dort nicht genügend vielfältige Tests ansetzen. Und auch der Definitions-/Überprüfungsaufwand für nichttriviale Datenmengen/-strukturen ist dort zu hoch, um facettenreich zu testen. Deshalb ist Modularisierung unerlässlich: Software muss aus Bausteinen bestehen, die für sich getestet werden können und erst in mehreren Integrationsstrata [5] übereinander die gesamte Software konstituieren. Hochqualitative Bausteine in leicht überprüfbarer Weise korrekt zusammengesteckt ergeben dann ein hochqualitatives Gesamtprodukt. Sicher kann dabei ChatGPT mit Modulvorschlägen helfen – doch

bei der Formulierung von Schnittstellen und Tests würde ich die Codierung (noch) nicht ChatGPT überlassen. Deshalb: Softwareentwurf bleibt nötig und ist eine menschliche Aktivität, die Erfahrung braucht.

Daraus folgt für mich: ChatGPT ist ein Programming Buddy, den ich nicht missen möchte. Doch es braucht ein Vorgehen, das seinen Fähigkeiten und Grenzen gerecht wird. Das nenne ich Requirements-First Development (RFD) und habe es im vorhergehenden Artikel skizziert. Nachfolgend möchte ich es an einem nichttrivialen Beispiel demonstrieren.

Das Beispielszenario

Da meine Frau und ich gerade dabei sind, Spanisch zu lernen, wähle ich als Beispielszenario ein Programm, das helfen soll, Vokabeln zu pauken: eine sogenannte Flashcard-App. Vokabeln mit Karteikarten zu lernen ist effektiver, als sie in einer Liste immer wieder durchzugehen. Den Ansatz des Spaced Repetition Learning [6] habe ich bereits als Schüler im Buch „So lernt man lernen“ von Sebastian Leitner kennengelernt. Natürlich gibt es schon viele Flashcard-Apps online oder fürs

✦ Sei ein erfahrener Softwareentwickler mit einem Gespür für unvollständige Anforderungen. Schreibe mir ein Programm in Typescript für die Deno Runtime, das die folgenden Anforderungen erfüllt:

###

Das Programm begrüßt den Anwender mit seinem Namen.

###

Wenn du diese Anforderungen unvollständig oder schwammig findest, stelle mir gern Fragen, bevor du den Code generierst.

🌀 Basierend auf Ihrer Anforderung, fehlen hier ein paar Details. Lassen Sie uns diese klären, bevor ich fortfahre:

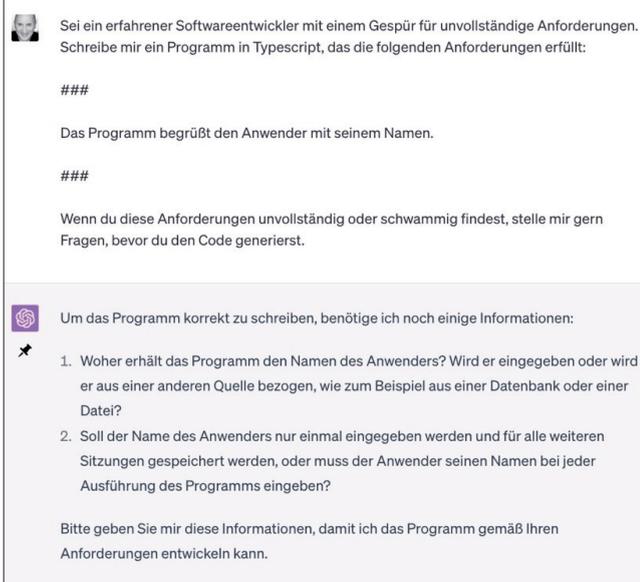
✦

1. Wie wird der Name des Benutzers erhalten? Wird der Name von der Kommandozeile, aus einer Datei, oder durch eine andere Methode eingegeben?
2. Soll das Programm eine grafische Benutzeroberfläche haben oder wird es ausschließlich auf der Kommandozeile ausgeführt?
3. Soll das Programm irgendwelche Fehlerbehandlungen oder spezielle Bedingungen berücksichtigen, wenn der Name nicht gegeben ist? Zum Beispiel, was passiert, wenn der Benutzer keinen Namen eingibt oder einen ungültigen Namen eingibt?

Es wäre hilfreich, diese Punkte zu klären, um Ihnen einen genauen und funktionsfähigen Code zu liefern.

👤 2. Das Programm braucht keine grafische Benutzeroberfläche

ChatGPT erkennt Unklarheiten in den Anforderungen und fragt nach (Bild 1)



Je nach Stimmung fragt ChatGPT anders nach (Bild 2)

```
ralfw@Hereafter req-first 2 % deno run --allow-all fc.ts
Front: Hund
Do you know the back of the card? (yes/no/maybe) yes
Front: Katze
Do you know the back of the card? (yes/no/maybe) no
Front: Haus
Do you know the back of the card? (yes/no/maybe) maybe
Review session complete
ralfw@Hereafter req-first 2 %
```

Die von ChatGPT erfundene Benutzerschnittstelle (Bild 3)

Smartphone. Mir gefällt Repetico [7] besonders gut. Doch eine eigene App ist nochmal etwas anderes, oder?

Ich stelle ich mir die App wie im Kasten **Anforderungsskizze für eine Flashcard-App** skizziert vor. In dem Szenario ist ein bisschen Benutzerschnittstelle dabei, sie kann in einem Ter-

minal-Fenster laufen, es gibt ein bisschen Persistenz und etwas Domänenlogik.

Wie würden Sie jetzt vorgehen? Wie lange würden Sie allein für die Implementation brauchen? Ich meine eine saubere Implementation, keinen Prototyp, also eine, in der es auch Tests gibt und Struktur erkennbar ist. Denn wer weiß ... vielleicht soll der Code ja weiterentwickelt werden. Mir schwebt da zum Beispiel für die Zukunft eine Unterstützung beim Lernen durch KI vor, wie ich sie schon in meinem Online-Spanischkurs Spanish Day by Day [8] einsetze.

Wenn ich just for fun diese Anforderungen eins zu eins an ChatGPT gebe, dann ist das Ergebnis schon beeindruckend, finde ich [9]. Das Programm ist aus dem Stand lauffähig und erfüllt die Anforderungen an Spaced Repetition Learning grundsätzlich.

Allerdings hat sich ChatGPT einige Freiheiten genommen beziehungsweise nicht genau hingeschaut:

- Die Benutzerschnittstelle passt (Bild 3). Ich hatte sie nicht näher beschrieben und ChatGPT hat das Beste daraus gemacht. Ich möchte zwar einen Kartensatz auf der Kommandozeile angeben können, doch so genau hatte ich das nicht gesagt. Also hat ChatGPT einen Dateinamen fixiert. Auch möchte ich meine Antworten kürzer geben können; doch für den Start ist auch ein komplettes Wort wie *maybe* okay.
- Die Benutzerschnittstelle hat allerdings eine Lücke: Ich kann die Karten nicht umdrehen. Diesen Wunsch hatte ich ausdrücklich in den Anforderungen formuliert.
- Die Datei des Kartensatzes wird aktualisiert (Bild 4). Allerdings folgen die Wiedervorlageintervalle nicht meiner Beschreibung, die Konfigurierbarkeit wünscht. Stattdessen ►

● Anforderungsskizze für eine Flashcard-App

Auf der Vorderseite jeder Flashcard steht eine Vokabel auf Deutsch, auf der Rückseite auf Spanisch. Natürlich kann man die App auch nutzen, um eine andere Sprache oder etwas ganz anderes zu lernen, zum Beispiel Clean-Code-Developer-Prinzipien und -Praktiken. Mit der Flashcard-App starte ich eine Lernsitzung für einen Satz von Flashcards. Dann zeigt mir die App die Vorderseite einer Karte – also die deutsche Vokabel – und ich überlege, ob ich weiß, was auf der Rückseite steht, also wie die spanische Übersetzung lautet. Ich kann die Karte umdrehen, um meine Antwort zu überprüfen. Schließlich bewerte ich meine Antwort: Habe ich die spanische Übersetzung gewusst, habe ich sie nicht gewusst oder habe ich sie ungefähr gewusst? Die Flashcard wird dann im Kartensatz auf Wiedervorlage gelegt, je nachdem, wie meine Antwort ausgefallen ist.

Zu diesem Zweck gehört jede Flashcard einem Level an:

- Level 0: Ich habe die Karte noch nicht gesehen.
- Level 1–n: Ich habe die Antwort auf dem vorherigen Level gewusst.

Wenn mir eine Karte auf Level k vorgelegt wird und ich weiß die Antwort, dann steigt sie auf Level $k+1$ auf. Wenn ich die Antwort

nicht weiß, fällt sie zurück auf Level 1. Wenn ich die Antwort ungefähr wusste, fällt sie auf Level $k-1$ (≥ 1) zurück.

Eine Karte wird mir dann wieder in einer Lernsitzung vorgelegt, wenn ihr Wiedervorlagedatum erreicht oder überschritten ist.

Das Wiedervorlagedatum wird aufgrund des Levels berechnet, auf dem eine Karte ist, und hängt von meiner Antwort ab:

- Wenn ich die Antwort nicht weiß, ist die Wiedervorlage morgen.
- Wenn ich die Antwort nur „ungefähr weiß“, ist die Wiedervorlage auch morgen.
- Wenn ich die Antwort weiß, ist die Wiedervorlage je nach aktuellem Level in einigen Tagen in der Zukunft. Beispiel: Level 0: ein Tag, Level 1: zwei Tage, 2: vier, 3: acht, 4: 16, 5: 32, 6: 64, 7: 9999. Eine Wiedervorlage in 9999 Tagen sorgt effektiv dafür, dass die Karte nicht wieder vorgelegt wird.

Wie viele Level es gibt und welche Wiedervorlage-Intervalle, ist eine Sache der Konfiguration. Die Zahlen sind nur Beispiele.

Ein Kartensatz kann in einer simplen Textdatei gespeichert sein (CSV oder JSON). Es gibt keine speziellen Skalierungsanforderungen und auch keinen Mehrbenutzerbetrieb. Wenn die Anwendung auf meinem Desktop läuft, bin ich (erst mal) zufrieden.

hat ChatGPT meine Beispiele für Wiedervorlageintervalle verallgemeinert (Bild 5), indem die auf das Quadrat des Levels gesetzt werden. Das finde ich clever aus meinem Beispiel abgeleitet — auch wenn ich es so nicht gemeint hatte.

- ChatGPTs Code ist nicht modularisiert, aber durchaus testbar. Es gibt Funktionen für einzelne Aspekte der Anwendung. Andererseits sind Verantwortlichkeiten auch vermischt (Bild 6): Benutzerinteraktion und Spaced-Repetition-Algorithmus, also Domänenlogik, stehen in derselben Funktion.

Mit diesem Code könnte ich jetzt weiterarbeiten. ChatGPT würde ihn für mich refaktorisieren und auch Tests schreiben. Doch mir scheint das erstens umständlicher, als wenn ich gleich strukturierter vorgegangen wäre. Zweitens ist das Ergebnis auch nur so positiv, weil die Anforderungen immer noch überschaubar sind. Für einen Prototyp ist das Ergebnis völlig okay. Insofern bin ich auch froh, es mit den kompletten Anforderungen einmal probiert zu haben. ChatGPT hat mir Informationen beschafft:

- Ich habe eine Idee für Datenstrukturen bekommen.
- Ich habe ein Gefühl für die Bedienung bekommen.
- Ich habe einen alternativen Vorschlag für die Wiedervorlage bekommen.
- Ich habe Hinweise darauf bekommen, was eben nicht vermischt werden sollte, um es testbar zu machen.

Anforderungsanalyse

Um ChatGPT auf testbare Module ansetzen zu können, muss ich diese identifizieren. Ich muss selbst die Anforderungen analysieren und daraus Modulideen ableiten. Damit folge ich dem Prinzip Separation of Concerns. Das Single Responsibility Principle [10] hilft mir, Module zu erkennen: Sie stehen für die großen Entscheidungen des Kunden, die in den Anforderungen stecken.

Ich bevorzuge beim Herangehen an Anforderungen ein Softwarezellendiagramm wie in Bild 7. Es stellt das Softwaresystem in einen Kontext: Anwender benutzen das Softwaresystem, das Softwaresystem benutzt Ressourcen. Indem ich die Kontaktpunkte des Softwaresystems mit der Umwelt aus den Anforderungen herausarbeite, kann ich tiefgehende Fragen dazu stellen, wie die Kommunikation dort genau aussehen soll.

Benutzer in ihren Rollen und alle Ressourcen werden später im Code durch Module repräsentiert. Dazu

```

1  [
2  {
3      "front": "Hund",
4      "back": "Perro",
5      "level": 1,
6      "reviewDate": "2023-06-01"
7  },
8  {
9      "front": "Katze",
10     "back": "Gato",
11     "level": 0,
12     "reviewDate": "2023-05-31"
13  },
14  {
15     "front": "Haus",
16     "back": "Casa",
17     "level": 1,
18     "reviewDate": "2023-06-01"
19  }
20 ]
21
    
```

Der Kartensatz nach der Lernsitzung (Bild 4)

kommt mindestens ein Modul für die Domäne, den Kern des Softwaresystems. Wenn ich mit dieser Brille an Anforderungen herantrete, wird bereits ein Teil des Entwurfs sichtbar. Das finde ich sehr bequem und geradlinig.

Dass es ein Modul geben sollte, das ein Kartenset lädt/speichert, ist nicht überraschend. Aber in der Umwelt liegen weitere Ressourcen, die ich daran erkenne, dass APIs nötig sind, um Daten zu beschaffen:

- Die Zeit ist eine Ressource, auf deren Inhalt ich mit der API-Klasse *Date* zugreife. Das will gekapselt sein – weil ich nur so die Logik, die die Zeit braucht, einfach testen kann.
- Auch die Wiedervorlageintervalle kommen aus einer Datei. Die Anforderungen erwähnen die Konfigurierbarkeit, das heißt, sie sollen veränderbar sein, ohne den Code anfassen zu müssen. Das API für den Zugriff darauf ist ebenfalls zu kapseln. Aus Prinzip, denn erstens ist das eine ganz eigene Entscheidung, Wiedervorlageintervalle konfigurierbar zu machen, und zweitens soll Logik, die sich darauf bezieht, ohne Abhängigkeit von einer Datei testbar sein.

Nicht überraschend ist ebenfalls, dass die Benutzerschnittstelle mindestens ein eigenes Modul braucht. Darin stecken so viele Entscheidungen, dafür sind wieder andere APIs nötig, diese Logik muss für Tests anderer ausgeblendet werden können. Aufgefallen ist mir bei der Strukturierung der Softwarezelle jedoch, dass die Anforderungen gar keine konkrete Aussage zur Interaktion mit dem Programm machen. Eine Desktop-Anwendung reicht aus. Aber was für eine? Soll es ein GUI geben, oder reicht eine Ausführung in einem Terminal-Fenster? Ich entscheide mich für die einfachere Variante: das Terminal-Fenster, also eine Konsolenanwendung. Und

```

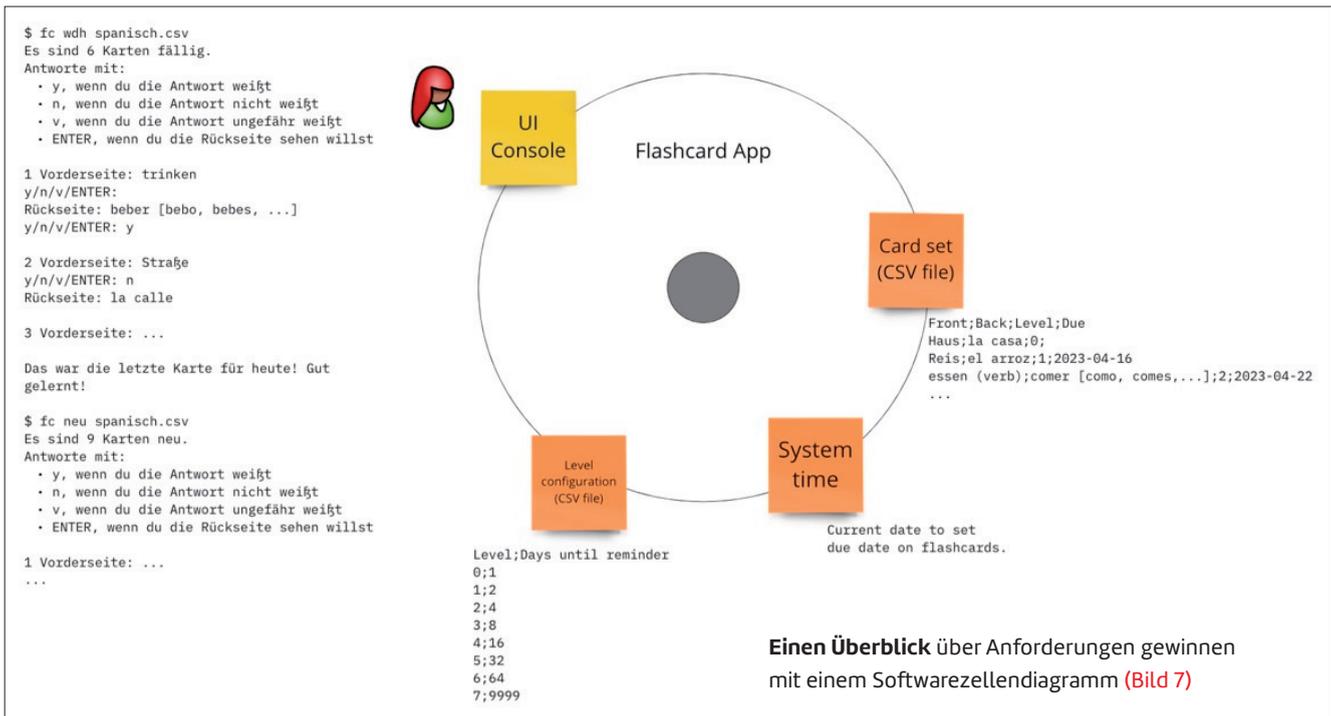
16  function getNextReviewDate(level: number): string {
17      const daysToNextReview = Math.pow(2, level); // Change this to customize review intervals
18      const nextReviewDate = new Date();
19      nextReviewDate.setDate(nextReviewDate.getDate() + daysToNextReview);
20      return nextReviewDate.toISOString().slice(0,10);
21  }
22
23  async function reviewCard(card: Flashcard, promiseId: Promise<void> {
    
```

ChatGPTs Algorithmus für die Wiedervorlage (Bild 5)

```

23  async function reviewCard(card: Flashcard): Promise<void> {
24      console.log(`Front: ${card.front}`);
25      const input = prompt('Do you know the back of the card? (yes/no/maybe)');
26      switch (input) {
27          case 'yes':
28              card.level = card.level + 1;
29              break;
30          case 'no':
31              card.level = 0;
32              break;
33          case 'maybe':
34              card.level = Math.max(1, card.level - 1);
35              break;
36          default:
37              console.log('Invalid input. Please answer with "yes", "no", or "maybe."');
38              return reviewCard(card);
39      }
40      card.reviewDate = getNextReviewDate(card.level);
41  }
    
```

Verantwortlichkeiten sind stellenweise vermischt (Bild 6)



wie sehen dort die Dialoge aus? Denn Dialoge gibt es nicht nur bei GUI-Anwendungen. Mir schweben zwei Dialoge vor (Bild 8): einer, um eine Lernsitzung zu starten, der andere, um die Lernsitzung durchzuführen. In der Lernsitzung werden die Karten eine nach der anderen vorgelegt und beurteilt.

Im nächsten Schritt muss ich die Dialoge verfeinern: Wie sehen die Interaktionen genau aus? Was zeigt das Programm, was gibt der Benutzer ein, wie und wann triggert er Logik im Programm, damit etwas mit seinen Eingaben passiert? Mich interessieren die Entry Points der App. Bild 9 zeigt, wie ich mir die Konkretisierung der Dialoge vorstelle:

- Der Start der Sitzung findet beim Programmstart statt und wird mit Kommandozeilenparametern eingestellt.
- Durchgeführt wird die Sitzung mit Konsolenein-/ausgaben.

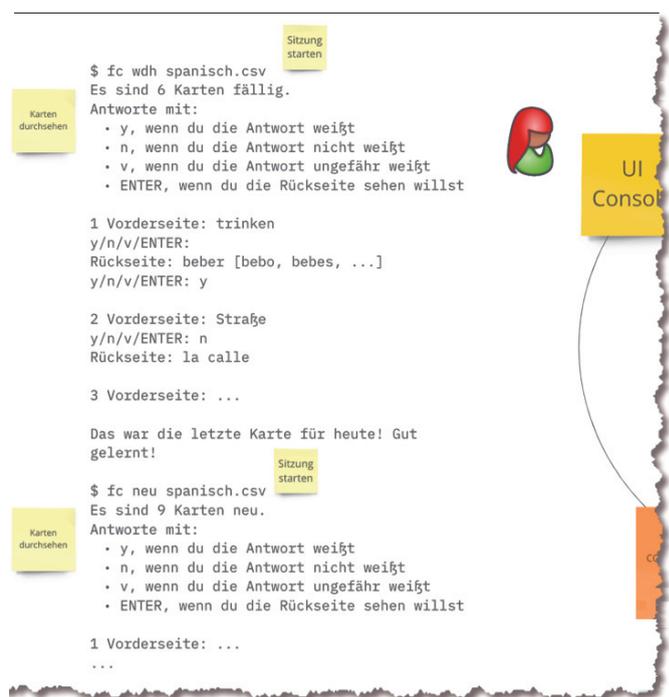
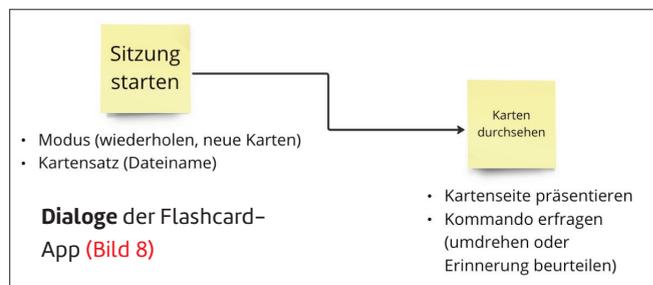
Technisch wird die Umsetzung der Dialoge einfach sein. Dennoch brauche ich Klarheit darüber, wann welche Entry Points getriggert werden. Ich erkenne zwei (Bild 10):

- Start der Sitzung
- Auswertung eines Kommandos

Das ist ein Ergebnis der Analyse – aber sie sagt etwas über eine Modulschnittstelle aus, wie später zu sehen sein wird.

Nicht nur die Benutzerschnittstelle hat Struktur. Die Ressourcen haben auch eine, die ich in der Analyse erkunden kann. Wie sehen zum Beispiel Lernkarteikarten aus, um sie mit einem Spaced-Repetition-Learning-Algorithmus wieder vorzulegen? Die Softwarezelle mit den auf ihrer Grenze und darin platzierten Modulen ist für mich wie eine Schnittstelle. Ich muss alle Berührungspunkte mit der Umwelt und den Kern nacheinander abarbeiten und mir dazu passende Fragen stellen.

Die erste Frage ist die nach dem Zweck: Was wird in dem ►



Konkretisierung der Dialoge für das Konsolenprogramm (Bild 9)

Modul gekapselt? Die zweite ist die nach den dafür nötigen Datenstrukturen. Kann ich die schon in der Analyse erkennen, weil sie irgendwie in den Anforderungen stecken? Oder ist das eine Sache des Entwurfs?

Für die Kartensätze liegt die Struktur nahe, denke ich. Alle Angaben, die für eine Sitzung und den Wiedervorlagealgorithmus wichtig sind, stehen in den Anforderungen. Bild 11 zeigt die Struktur mit ein paar Beispielen.

Der Einfachheit halber habe ich die Struktur als CSV-Text notiert. Eine JSON-Codierung ist aber genauso möglich. Der Vorteil von CSV: Es lässt sich sehr leicht mit Excel herstellen. Der Vorteil von JSON: Es lässt sich leichter als CSV einlesen und speichern.

Die Datenstrukturen für die Zeit und die Wiedervorlagekonfiguration sind noch einfacher.

Wenn die Entry Points das API der Benutzerschnittstelle sind, wie sehen dann die APIs der Ressourcenadapter aus? Das ist keine Sache der Anforderungsanalyse. Sie richten sich nach dem Bedarf, den das Innere der App hat. Dass die Ressourcen gebraucht werden, ist klar. Wie genau sie gebraucht werden, ist nicht klar. Das stellt sich erst im Entwurf heraus, der das Innenleben modelliert.

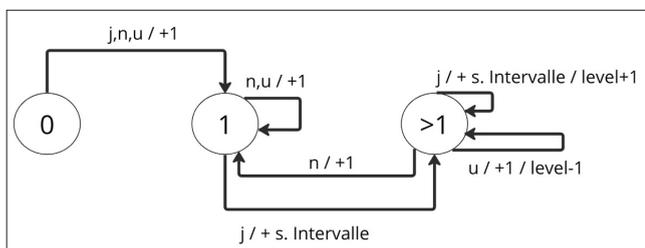
Das API der Benutzerschnittstelle hingegen befriedigt den Bedarf der Benutzer. Was diese wollen, beschreiben die Anforderungen. Deshalb sind die dortigen Entry Points ein Ergebnis der Anforderungsanalyse.

Selbstverständlich gehört zur Anforderungsanalyse auch eine Betrachtung dessen, was wo passieren soll. Wenn schon Module identifiziert werden können, dann kann ich mir auch eine Idee davon machen, was sie tun sollen. Das ist für die Adapter auf der Peripherie der Softwarezelle trivial:

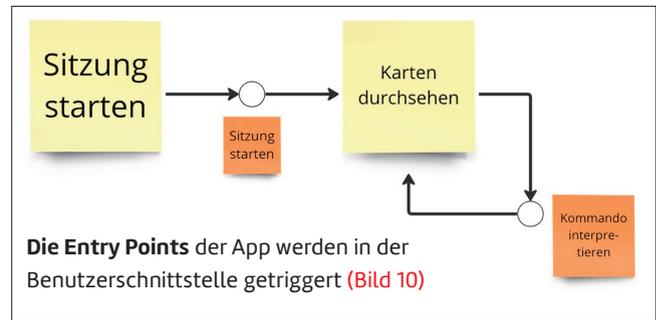
- Die Benutzerschnittstelle sammelt Eingaben vom Benutzer und projiziert Ergebnisse von deren Verarbeitung für ihn.
- Der Adapter für die Kartensätze lädt und speichert sie. Er

Aktuelles Level	Antwort			Wiedervorlageintervalle	
	ja	nein	ungefähr	Level	Tage in der Zukunft
>0	(+1 / s. Intervalle)	(1 / +1)	(-1, >0 / +1)	0	1
0	(neues Level / Wiedervorlageintervall)	(1 / +1)	(1 / +1)	1	2
				2	3
				3	5
				4	8
				5	13
				6	21
				7	34
				8	9999

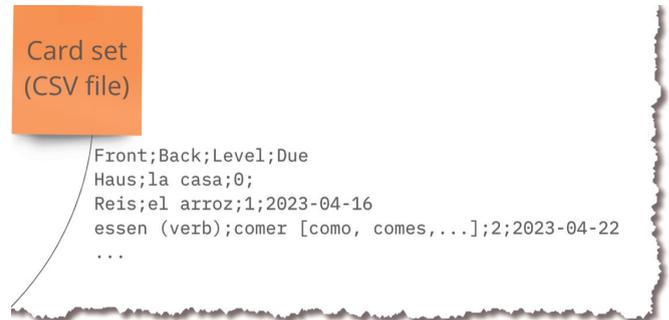
Der Wiedervorlagealgorithmus übersetzt in Tabellenform für mehr Übersichtlichkeit (Bild 12)



Die Wiedervorlage übersetzt in einen Automaten, dessen Zustände die Levels sind (Bild 13)



Die Entry Points der App werden in der Benutzerschnittstelle getriggert (Bild 10)



Struktur eines Kartensatzes (Bild 11)

macht aus toten Daten auf der Platte lebendige im Speicher.

- Der Adapter für die Zeit liefert die Zeit.
- Der Adapter für die Wiedervorlagekonfiguration liefert die Intervalle von der Festplatte.

Für diese Anwendung ist das alles einfach nachvollziehbar, denke ich. Interessanter ist die Domäne, das heißt der Lernalgorithmus. Der steckt im Kern der Softwarezelle. Worum geht es dabei?

- Es werden Karten ausgewählt für die Vorlage in einer Lernsitzung. Entweder sind das die fälligen oder die neuen Karten. Und ich füge kühn hinzu: Die ausgewählten Karten könnten auch noch in zufälliger Reihenfolge vorgelegt werden, um das Lernen zu intensivieren.
- Karten werden nach Level und Erinnerungserfolg für die Wiedervorlage eingeplant. Das geschieht vor allem gemäß der Konfiguration. Die Anforderungen beschreiben das schon sehr detailliert. Um mir die Anforderungen aber wirklich anzueignen, mache ich eine Übersetzung in eine andere Darstellung, zum Beispiel Tabellen (Bild 12) oder Zustandsautomat (Bild 13).

Nach diesem genaueren Blick auf die Anforderungen fühle ich mich bereit für die nächste Phase, den Entwurf.

Entwurf

Im Entwurf geht es darum, die Lösung zu modellieren. Sie wird nicht codiert, sondern nur durch ihre Bausteine deklarativ beschrieben. Der Entwurf beantwortet die Fragen:

- Welche Bausteine hat die Lösung?
- Wie stehen diese Bausteine in Beziehung zueinander?

Ausführlich habe ich den Entwurf in meinem Buch „Softwareentwurf mit Flow-Design“ [11] beschrieben. Hier kann ich darauf nur mit Skizzen und Beispielen eingehen. Da die



Um die Anforderungen der Flashcard-App zu erfüllen und eine modulare und wartbare Code-Struktur zu gewährleisten, könnte man den folgenden Ansatz verfolgen:

1. **User Interface (UI)-Modul:** Dieses Modul ist für die Darstellung der Benutzeroberfläche und die Interaktion mit dem Benutzer verantwortlich. Es sollte die Funktionalität bereitstellen, um Karten anzuzeigen, Benutzerantworten zu erfassen und den Lernfortschritt anzuzeigen. Das UI-Modul sollte eng mit den anderen Modulen kommunizieren, um die benötigten Daten abzurufen und Aktionen auszuführen.
2. **Karten-Modul:** Dieses Modul kümmert sich um die Verwaltung der Flashcards, ihrer Eigenschaften und ihres Status (Level und Wiedervorlagdatum). Es sollte Funktionen bereitstellen, um neue Karten hinzuzufügen, vorhandene Karten abzurufen und den Kartensatz zu aktualisieren. Das Karten-Modul sollte auch die Berechnung der Wiedervorlage-Daten basierend auf den Antworten des Benutzers implementieren.
3. **Dateiverwaltungs-Modul:** Dieses Modul ist für das Lesen und Schreiben von Kartensets in einer Textdatei (CSV oder JSON) zuständig. Es sollte Funktionen bereitstellen, um Kartensets zu importieren und zu exportieren. Das Dateiverwaltungs-Modul kann eng mit dem Karten-Modul zusammenarbeiten, um die Daten zwischen der Textdatei und dem Karten-Modul zu synchronisieren.
4. **Config-Modul:** Dieses Modul enthält die Konfigurationsparameter für die Flashcard-App, z.B. die Anzahl der Level, die Wiedervorlage-Intervalle und andere Einstellungen. Es sollte Funktionen bereitstellen, um die Konfigurationsdaten abzurufen und zu aktualisieren. Das Config-Modul kann von anderen Modulen verwendet werden, um auf die App-Einstellungen zuzugreifen.
5. **Main-Modul:** Dieses Modul dient als Einstiegspunkt der Anwendung. Es initialisiert die anderen Module, stellt die Anfangskonfiguration bereit und startet den Lernprozess. Das Main-Modul kann die Verbindung zwischen den Modulen herstellen und die Abläufe koordinieren.

Mit dieser Modularisierung haben wir die Verantwortlichkeiten klar aufgeteilt und das Single Responsibility Principle befolgt:

ChatGPT kann auch bei der Modularisierung helfen (Bild 14)

Zielsprache TypeScript ist, können Module nicht nur Klassen sein, sondern auch `.ts`-Dateien. Für beide lassen sich Interfaces definieren, das heißt Unterschiede machen zwischen Öffentlichem und Privatem (Gekapseltem).

Beziehungen sind vor allem Nutzungsbeziehungen: Welcher Baustein benutzt welchen anderen als Dienstleister? Anders ausgedrückt: Welche Abhängigkeitsverhältnisse existieren zwischen den Bausteinen? Hier muss ich aufpassen, damit der Code sauber bleibt. ChatGPT traue ich nur begrenzt ein Bewusstsein in dieser Hinsicht zu. Das Integration Operation Segregation Principle (IOSP) [12] und die IODA-Architektur [13] beziehungsweise die darauf basierende Sleepy-Hollow-Architektur wird es nicht ohne weitere Anleitung berücksichtigen.

Aber was leistet ChatGPT eigentlich in Bezug auf den Entwurf? Kann ich die KI fragen, welche Module die Anwendung haben sollte? Ich habe das mal versucht [14] und finde das Ergebnis nicht schlecht (Bild 14). Hier hat ChatGPT sogar erkannt, dass es eine Konfiguration gibt. Was allerdings fehlt: ein Modul für die Systemzeit. Unbedarft sollte ich also eine solche Frage nicht stellen. Mindestens sollte ich den Prompt mit einigen allgemeinen Hinweisen zur Modularisierung spicken. Allerdings finde ich die Modularisierung jedoch nicht so aufwendig. Ich habe nicht den Eindruck, Zeit verschwendet zu haben. Letztlich bin ich als Entwickler für die Software verantwortlich; da steht es mir gut zu Gesicht, dass ich mich mit den Anforderungen solide auseinandersetze.

Für diese Anforderungen sind die Module also im Wesentlichen bekannt. Die Arbeitsperde sind identifiziert.

Worauf es im Entwurf ankommt, ist die Definition ihrer Schnittstellen. Was muss ich tun, um sie zu bestimmen?

Im Flow-Design [11] schaue ich mir eine Funktion an und überlege, wie sich ihre Arbeitsweise als Datenfluss realisieren lassen könnte. Nach der Anforderungsanalyse sind mir nur die Entry Points als Funktionen bekannt; sie zu finden ist das Ziel der entwicklerorientierten Anforderungsanalyse, die ich Slicing nenne. Formalisiert sehen die Entry Points zum Beispiel in TypeScript so aus:

- `startSession(cardsetFilename:string, useDueCards:boolean):Card`
- `scheduleCard(knowledgeAssessments:Assessments):Card`

Sie gehören nach der IODA-Architektur [13] zum Prozessor-Modul, das die Leistungen des Body der Sleepy Hollow Architecture [13] definiert. Das Prozessor-Modul integriert sowohl die Module der Domäne wie auch den Ressourcenadapter (Bild 15).

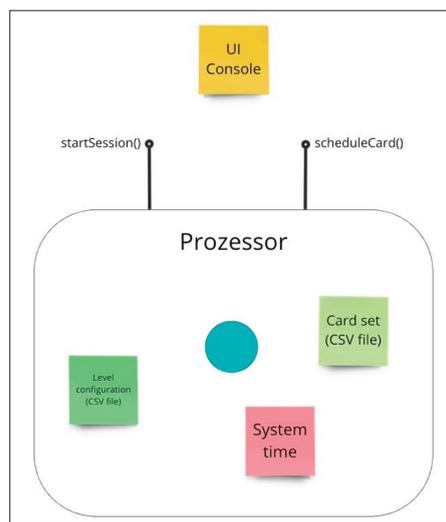
Wie komme ich nun zu einem Datenfluss-Transformationsprozess für die Entry Points, der den Input von der Benutzerschnittstelle umwandelt in einen Output für sie – plus Änderungen des Anwendungszustands in der Datenbank, dem gewählten Kartensatz?

Das ist kreative Entwurfsarbeit. Das ist meine Domäne als Mensch. Oder? Vielleicht kann ChatGPT aber auch dabei unterstützen? Auf Nachfrage liefert ChatGPT tatsächlich ein sehr brauchbares Ergebnis (Bild 16). Dafür war allerdings ein bisschen mehr Aufwand nötig; zero-shot prompting war mir zu riskant [15]. Stattdessen habe ich ChatGPT über mehrere Prompts hinweg zuerst die Anforderungen gegeben und die Module genannt, die ich gefunden habe, dann habe ich die Entry Points beschrieben und schließlich habe ich mir einen Datenfluss für die Start-Funktion gewünscht [16]:

„Die Schnittstelle des Prozessors ist definiert. Aber die Schnittstellen der anderen Module sind noch nicht definiert. Dabei brauche ich deine Hilfe. Ich möchte jetzt diese Schnittstellen nach dem Bedarf bestimmen, den die beiden Prozessor-Funktionen haben.“

„Lass uns mit der ersten Funktion beginnen, dem Start einer Lernsitzung. Finde die Schritte, die durchlaufen werden sollten, um eine Lernsitzung zu starten. Für jeden Schritt sollte eines der anderen Module zuständig sein. Nenne jeden Schritt und das dazugehörige Modul.“

Um ChatGPT nicht auf falsche Gedanken zu bringen, habe ich den Begriff Datenfluss jedoch vermieden und einfach nur von Schritten gesprochen. Ich habe mich des Prompt Stacking [17] bedient: Das war für mich einfacher, weil ich nicht alles in einen Prompt stecken musste. Das war für ChatGPT einfacher, weil dadurch sein Kontext langsam wachsen konnte. Wie beim ▶



Der Prozessor implementiert die Entry Points und integriert die Module von Domäne und Ressourcenzugriff (Bild 15)

Chain-of-Thought Prompting [18, 19] hilft eine solche Aufteilung nach meiner Erfahrung, größere Zusammenhänge für ChatGPT verdaubar zu machen, auch wenn keine Zwischenergebnisse produziert werden (das wäre Prompt Chaining gewesen [20]).

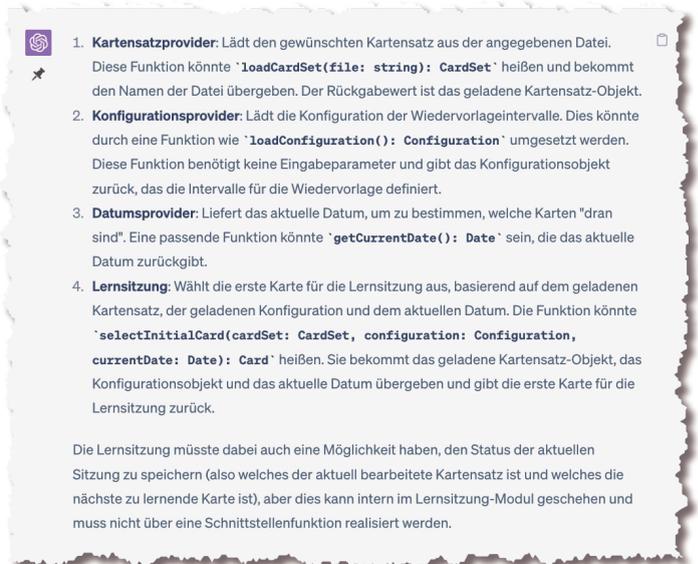
Integriert bilden die von ChatGPT vorgeschlagenen Schritte den Datenfluss in Bild 17. Das hätte ich kaum besser entworfen. Alle Funktionseinheiten passen zusammen.

Nur eine kleine Ungereimtheit fällt auf: Meine Definition des Start-Entry-Point enthält ein Flag (*useDueCards*) für die Auswahl der Karten für die Sitzung: Sollen es die fälligen (*true*) oder die neuen (*false*) sein? Wer hat hier nicht aufgepasst? Das war ich. Dass man sich bei Sitzungsstart aus einem Kartensatz die eine oder andere Untermenge wünscht, taucht zwar in meiner Anforderungsanalyse auf (Bild 9), doch davon steht nichts im Anforderungstext. Also kann ChatGPT es nicht wissen und nicht bei der Signaturdefinition der Schritte berücksichtigen. Ein klassischer Fall von fehlender Information. Für ChatGPT war das ein unknown unknown.

Doch das Problem ist ja klein. Ich muss mich nur entscheiden, ob ich diese Information dem Provider für Kartensätze mitgebe (*loadCardSet*), damit der gleich die gewünschte Untermenge liefert. Oder ob die Information besser an die Domäne übergeben werden sollte (*selectInitialCard*).

Eine Übergabe an den Provider scheint mir nur sinnvoll, wenn die Menge der Karten in einem Satz sehr groß ist, so dass eine Filterung möglichst nah am Persistenzmedium stattfinden sollte (zum Beispiel mit einer SQL-Abfrage). Das ist jedoch nicht der Fall, selbst dann nicht, wenn es sich um 10000 oder mehr Karten handeln sollte. Für mehr Flexibilität geht deshalb diese Information an die Domäne.

Die Funktionseinheiten passen also zusammen. Sie stellen komplementäre Bausteine dar, die in einem Fluss verdrahtet zusammen die Leistung von *startSession* erbringen. ChatGPT kennt schon die Modulzugehörigkeit. Damit sind deren Inter-



ChatGPT entwirft einen Datenfluss für den Entry Point des Sitzungsstarts (Bild 16)

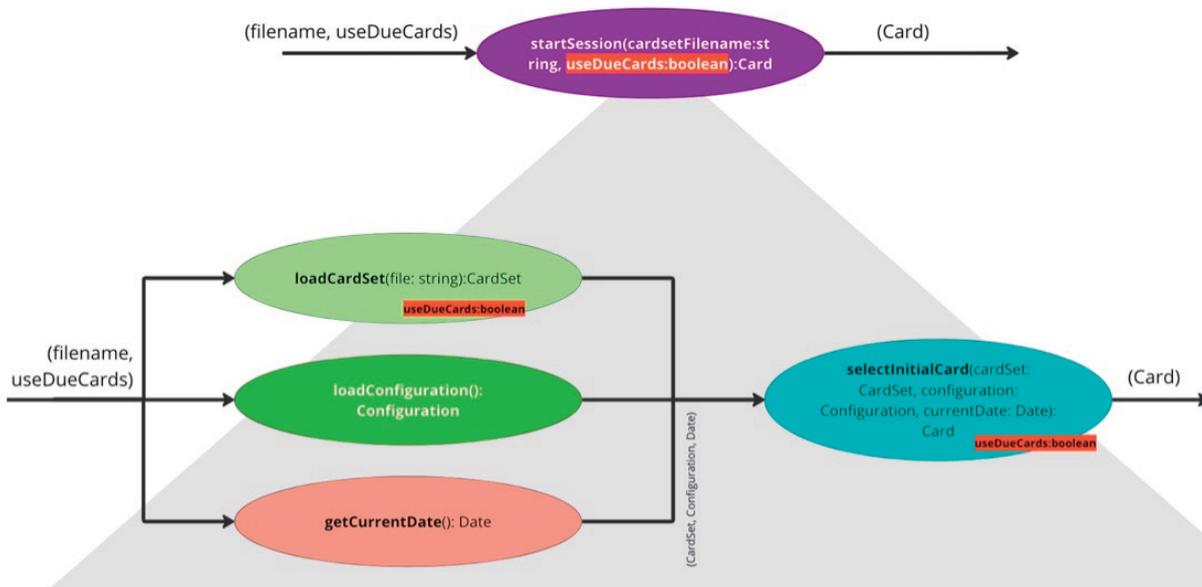
faces für dieses erste Inkrement fast definiert. Es fehlen nur noch die Details für die In-Memory Datenstrukturen.

Danach gefragt [21] liefert ChatGPT die Antwort in Bild 18. Auch das passt für mich weitgehend:

- Die *id* für die Karten finde ich allerdings überflüssig. Sie taucht in den persistenten Daten nicht auf und wird in-memory nicht gebraucht.
- Bei der *Configuration* sind die Level implizit die Position der Intervalle im Array. Solange die Level zusammenhängend und ≥ 0 sind, passt das.

Erst abschließen, dann implementieren

Für einen ersten Durchstich ohne Benutzerschnittstelle liegen alle Funktionen und Module vor. Soll ich zur Implementation voranschreiten? Im Sinne inkrementellen Vorgehens wäre das normal. An dieser Stelle entscheide ich mich jedoch dagegen, weil ich für das Medium Artikelserie lieber die Phasen abschließen möchte.



ChatGPTs Schrittfolge als Datenfluss. Die Funktionseinheiten sind farbcodiert nach Modulzugehörigkeit (vergleiche Bild 15) (Bild 17)

```

1  type Card = {
2      id: string; // Eindeutige ID der Karte
3      front: string; // Vorderseite der Karte (Deutsche Vokabel)
4      back: string; // Rückseite der Karte (Spanische Übersetzung)
5      level: number; // Aktuelles Level der Karte
6      dueDate: Date; // Wiedervorlagedatum
7  };
8
9  type CardSet = {
10     name: string; // Name des Kartensatzes
11     cards: Card[]; // Liste aller Karten in diesem Satz
12 };
13
14 type Configuration = {
15     intervals: number[]; // Wiedervorlageintervalle je Level
16 };

```

Von ChatGPT entworfene Datenstrukturen (Bild 18)

Aus Gründen der Darstellung wähle ich also den Wasserfall. Deshalb weiter mit dem Datenfluss für den zweiten Entry Point. Dazu führe ich am besten den Chat fort, der mir schon die ersten Funktionen geliefert hat [21]. Ich frage nun nach den Schritten für den zweiten Entry Point (Bild 19). Dieses Mal bin ich etwas spezifischer, indem ich die Signatur des Entry Points angebe und nochmal seine Funktion skizziere.

Das Ergebnis kann sich wieder sehen lassen, finde ich. Die Schritte hören sich sinnig an. Bild 20 zeigt den daraus abgeleiteten Datenfluss. Es passt alles zusammen.

Allerdings – und jetzt bin ich froh, dass ich nicht schon zur Implementation fortgeschritten bin – irgendetwas ist doch nicht ganz so, wie es sein sollte. Ein genauerer Blick ist nötig.

- Mir fällt jetzt auf, dass `loadConfiguration` schon im ersten Datenfluss vorkommt. Warum eigentlich? Die Wiedervorlageintervalle sind erst beim zweiten Entry Point relevant. `selectInitialCard` braucht sie nicht wirklich. Was hatte sich ChatGPT dabei gedacht?
- Die Konfiguration wird bei jedem Aufruf des zweiten Entry Points geladen. Ist das gut, ist das nötig? Auf diese Weise wird der Zustand verringert, der im Prozessor gehalten wird. Letztlich ist das für die Performance angesichts so überschaubarer Daten jedoch unerheblich. Die wiederholten Aufrufe können drinbleiben. Oder: Das Modul *Lernsitzung* kann mit der Konfiguration initialisiert werden. Dann würde sie zu Recht beim ersten Entry Point geladen und an `selectInitialCard` übergeben; hier beim zweiten Entry Point wäre das allerdings nicht mehr nötig. Ich denke, ich muss mich entscheiden, ob die Konfiguration als Zustand über Entry-Point-Aufrufe gehalten werden soll. Und wenn ja, von wem.
- Die *Lernsitzung* ist zustandsbehaftet. ChatGPT hat das korrekt erkannt, indem es `getCurrentCard` im zweiten Entry Point aufruft. Die aktuelle Karte, für die eine Bewertung abgegeben wurde, fließt ja nicht hinein.
- Andererseits gehört `updateCard` auch zu *Lernsitzung* und hätte Zugriff auf den Zustand. Warum muss die aktuelle Karte zuerst aus dem Modul herausgeholt und dann wieder hineingesteckt werden?
- Nach jeder Bewertung wird der Kartensatz gespeichert. Ist das eine gute Sache? Vorteil: Auf der Platte ist stets der aktuelle Stand, auch wenn das Programm abstürzen sollte oder der Benutzer es mit [Ctrl]+[C] abbricht. Nachteil: Das Speichern kostet Zeit. Doch ist das relevant, bemerkt das

der Benutzer? Ich denke, nein. Selbst wenn in einem Kartensatz 2500 Karten sein sollten, ist das Speichern nicht spürbar.

- Warum liefert `updateCard` eine Karte zurück, wenn die in keinem Downstream-Schritt genutzt wird?

Was bisher auch fehlt, ist Funktionalität auf Datenstrukturen. Die *Configuration* und das *CardSet* sind für mich Kandidaten, die Logik enthalten dürfen. Sie als Datenstrukturen dumm zu halten würde einer Primitive Obsession [22] Vorschub leisten.

Zum Entwurf gehört auch, die Logik klug auf Verhaltensbeziehungsweise Datenmodule zu verteilen. Datenmodule (Datenstrukturen) sind Daten, können als abstrakte Datentypen aber auch etwas Logik [23] enthalten. Verhaltensmodule haben gegebenenfalls Daten, sind jedoch vor allem Container für Logik. Ich denke, damit bin ich an einem Punkt im Entwurf, wo ich meine Erkenntnisse aus der Analyse und ChatGPTs Vorschläge konsolidieren sollte. Ich will es selbst unternehmen, die finalen Modulschnittstellen zu definieren.

Schnittstellendefinition

Welche Module gibt es, was sollen sie leisten?

- *UserInterface*: Sammelt Benutzereingaben von der Kommandozeile und präsentiert Karten für die Beurteilung. ▶

 Lass uns nun die zweite Funktion des Prozessor-Moduls näher anschauen. Mit ihr wird die Bewertung des Benutzers für eine Karte übermittelt. Er kann sie als gewusst, nicht gewusst oder ungefähr gewusst klassifizieren. Das Ergebnis ist die nächste zu lernende Karte.

Die Funktion soll diese Signatur haben:
`scheduleCard(knowledgeAssessments: Assessments): Card`

In der Funktion wird einerseits die aktuelle Karte nach der Bewertung durch den Benutzer mit einem neuen Level und einem Wiedervorlagedatum versehen. Zum anderen wird die nächste Karte bestimmt, die dem Benutzer vorgelegt werden soll zur Bewertung.

Bitte finde jetzt die Schritte, die in der Funktion durchlaufen müssen. Für jeden Schritt sollte eines der anderen Module zuständig sein. Nenne jeden Schritt und das dazugehörige Modul.

 1. **Lernsitzung**: Erhält die Bewertung des Benutzers für die aktuelle Karte. Eine passende Methode könnte `getCurrentCard(): Card` sein, die die aktuelle Karte zurückgibt, damit sie bewertet werden kann.

2. **Konfigurationsprovider**: Lädt die Wiedervorlageintervalle, um das neue Level und das Wiedervorlagedatum der aktuellen Karte zu bestimmen. Dies könnte durch eine bereits vorhandene Funktion wie `loadConfiguration(): Configuration` erfolgen, die das Konfigurationsobjekt zurückgibt.

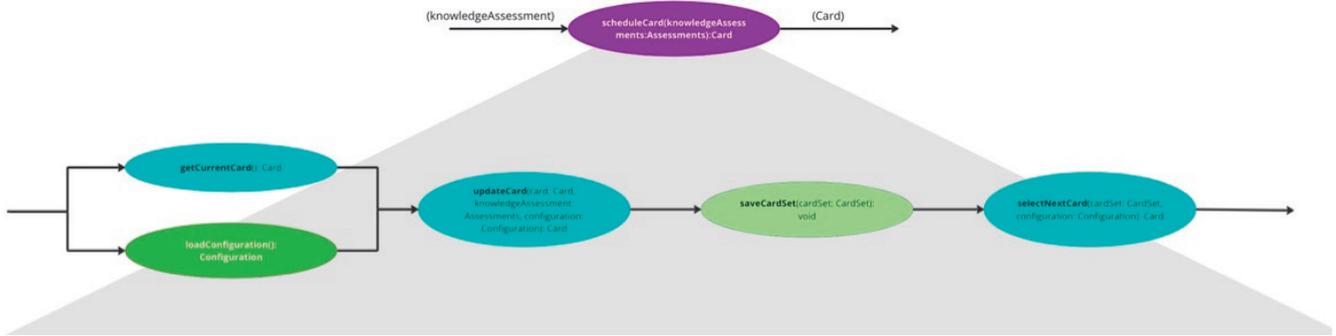
3. **Lernsitzung**: Aktualisiert das Level und das Wiedervorlagedatum der aktuellen Karte basierend auf der Bewertung des Benutzers und den Wiedervorlageintervallen aus der Konfiguration. Eine mögliche Funktion könnte `updateCard(card: Card, knowledgeAssessment: Assessments, configuration: Configuration): Card` sein. Sie erhält die aktuelle Karte, die Bewertung des Benutzers und die Konfiguration und gibt die aktualisierte Karte zurück.

4. **Kartensatzprovider**: Speichert den aktualisierten Kartensatz, nachdem die aktuelle Karte aktualisiert wurde. Dies könnte durch eine Funktion wie `saveCardSet(cardSet: CardSet): void` realisiert werden, die das aktualisierte Kartensatz-Objekt erhält.

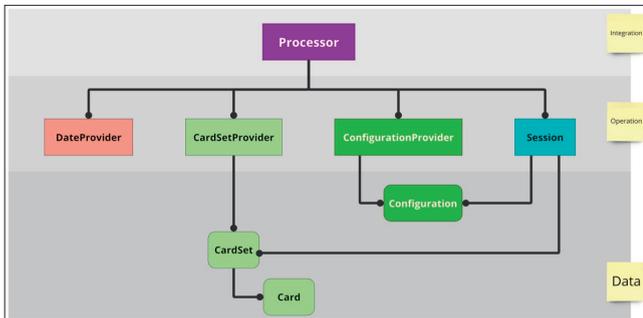
5. **Lernsitzung**: Wählt die nächste Karte für die Lernsitzung aus, basierend auf dem aktualisierten Kartensatz und der geladenen Konfiguration. Eine passende Funktion könnte `selectNextCard(cardSet: CardSet, configuration: Configuration): Card` sein, die das aktualisierte Kartensatz-Objekt und das Konfigurationsobjekt erhält und die nächste Karte zurückgibt.

Beachten Sie, dass die genauen Funktionen und deren Signatur von den genauen Anforderungen und der gewählten Architektur Ihrer Anwendung abhängen können. Diese Liste stellt nur einen groben Überblick dar, wie die einzelnen Schritte durch die verschiedenen Module umgesetzt werden könnten.

ChatGPTs Vorschlag für die Schritte innerhalb des zweiten Entry Points (Bild 19)



ChatGPTs Idee von einem Datenfluss für den zweiten Entry Point (Bild 20)



Abhängigkeiten zwischen den Modulen (Bild 21)

Mit dem Modul werde ich mich erst ganz zum Schluss näher beschäftigen, denke ich. Vorher will ich den *body* funktionstüchtig haben.

- *CardSetProvider*: Lädt und speichert ganze Kartensätze (*CardSet*).
- *DateProvider*: Liefert nur das aktuelle Datum. Das gehört zum Sitzungszustand, würde ich sagen.
- *ConfigurationProvider*: Lädt die Wiedervorlageintervalle (*Configuration*). Ich denke, auch die gehören zum Sitzungszustand.
- *Configuration*: Die Konfiguration ist für mich nur eine Datenstruktur, aus der ich über den Level als Index das zugehörige Wiedervorlageintervall heraushole. Wie das Intervall dann genutzt wird, ist Sache der Domänenlogik.
- *Session*: Die Session repräsentiert eine Lernsession für den gewählten Kartensatz. Hier werden die relevanten Karten selektiert und Wiedervorlagen nach den Benutzerbewertungen bestimmt. Der Kartensatz (beziehungsweise eine Untermenge) gehören auch zum Sitzungszustand. Die Domäne ist also zustandsbehaftet.
- *CardSet*: Ein Satz von Karten (*Card*) wie geladen von der Festplatte. Als Logik sehe ich hier die Selektion von neuen Karten (Level 0) beziehungsweise Karten, die eine Wiedervorlage zu einem Datum haben.
- *Card*: Eine Karte mit einem zu lernenden Wort. Sollte sie Logik enthalten? Nein, ich denke, die Domänenlogik sollte auf die *Session* beschränkt sein.
- *Processor*: Dieses Modul bietet die Entry Points an, die die anderen Module (außer Benutzerschnittstelle) zu Datenflüssen integrieren. Der *Processor* kann allerdings auch Zustand halten. Ich bin noch etwas unentschieden, ob ich den Sitzungszustand hier verorte oder wirklich in die Sitzung (*Session*) stecke. Ein Vorteil von Zustand im *Processor* ist

die einfachere Testbarkeit der Domänenlogik; sie könnte ich dann als Pure Functions implementieren (vergleiche Functional Core, Imperative Shell [24]).

Die Zusammenarbeit dieser Module sieht für mich nach der IODA-Architektur wie in Bild 21 aus. Entscheidend für die Testbarkeit ist, dass keine Abhängigkeiten zwischen den Arbeitsperden der Ebene *Operation* existieren [12]. Der Prozessor integriert diese nur noch zu übersichtlichen Prozessen in Form von Datenflüssen.

Mit dieser Übersicht fühle ich mich gut aufgestellt, um die Schnittstellen selbst in Code zu gießen. Sie sollen meine späteren Ausgangspunkte für ChatGPTs Codierungsdienste sein. Dass ich ChatGPT ein TypeScript-Projekt mit den Dateien für die Module und den Schnittstellendefinitionen generieren lasse, sehe ich nicht. Dafür gibt es doch noch hier und da Dinge, die ich geradeziehen muss. ChatGPT die zu erklären ist umständlicher, als es selbst zu machen. Und dieser Teil der Codierung ist auch nicht aufwendig. Vielmehr sehe ich ihn als Gelegenheit zur Reflexion; damit bekomme ich einen soliden Stand für die nächste Phase.

Bild 22 wirft ein paar Schlaglichter auf die Implementation der Schnittstellen. Ich habe dafür in WebStorm ein TypeScript-Projekt für die Deno Runtime aufgesetzt. Die meisten Verhaltensmodule haben ein Interface als Schnittstellendefinition. Nur *Processor* und *Session* weichen davon ab. Beide müssen in Tests nicht durch Surrogate ersetzt werden.

Nach dem Bisherigen war es geradlinig, die Schnittstellen zu codieren. Nur bei einem Modul bin ich vom Entwurf ab-

```

import {CardSet, Card} from "./Data.ts"

no usages new*
interface ICardSetProvider {
  loadCardSet(filename:string):CardSet;
  saveCardSet(filename:string, cardSet:CardSet): void;
}

5x usages new*
export class CardSet {
  private cards: Card[];

  no usages new*
  constructor(cards: Card[]) { this.cards = cards; }

  no usages new*
  get dueCards(dueDate:Date): Card[] {}

  no usages new*
  get newCards(): Card[] {}
}

export type Card = {
  front: string;
  back: string;
  level: number;
  dueDate: Date;
};
    
```

Die Schnittstellendefinition mit Daten und Interfaces in Projektdateien (Bild 22)

```

Session.ts x
import {CardSet, Card, Configuration, Assessments} from "./Data.ts"

no usages new *
class Session {
  private cards: Card[];
  private config: Configuration;
  private dueDate: Date;

  no usages new *
  constructor(config: Configuration, dueDate: Date) {
    this.config = config;
    this.dueDate = dueDate;
  }

  no usages new *
  initialize(cardSet: CardSet, useDueDate: boolean): void {
  }

  no usages new *
  getNextCard(): Card {
  }

  no usages new *
  updateCard(knowledgeAssessment: Assessments): void {
  }
}

```

Die Lernsitzung nach der Überarbeitung (Bild 23)

gewichen. Ich denke, hier hat sich ausgezahlt, dass ich nochmal darüber nachdenken konnte. Die *Session* sieht nun durchaus anders als die erste Idee aus (Bild 23).

- Ich habe mich dafür entschieden, die Lernsitzung zustandsbehaftet zu machen. Das macht sie etwas schwerer zu testen, doch ein Zustand passt zu einer Sitzung. Ich finde den Zustand dort natürlicher aufgehoben als im *Processor*.
- Die Methoden sehen anders aus als von ChatGPT vorgeschlagen, weil ich mich für den Zustand entschieden habe.
- Einen Moment habe ich mit mir gerungen, wo die Auswahl der zu lernenden Karten stattfinden soll. Der Konstruktor und eine eigene Methode standen zur Verfügung. Ich habe mich dann für die eigene Methode (*initialize*) entschieden, um der Empfehlung zu folgen, dass ein Konstruktor nie fehlschlagen und deshalb frei von Logik sein sollte. Damit muss der *Processor* ein Protokoll befolgen: Nach dem Konstruktor ist zuerst die Initialisierung aufzurufen, bevor die anderen Methoden genutzt werden. Für mich erscheint das jedoch überschaubar und durch die Namensgebung offensichtlich.

Bereit zur Implementierung

Analyse und Entwurf sind damit abgeschlossen. Die Implementierung kann beginnen. Für ChatGPT werden die Interfaces und Klassenrumpfe eine solide Grundlage sein, um die Methoden eine nach der anderen zu codieren und mit Tests zu versehen. Für jede einzelne werde ich gezielt Anforderungen formulieren.

Ich denke, der Aufwand bisher war sehr überschaubar. ChatGPT hat im Konzeptionellen sogar mehr unterstützen können, als ich zunächst gedacht hatte. Doch es ist mir auch noch mal klar geworden, dass mein Beitrag als Mensch nicht überflüssig wird. ChatGPT will an die Hand genommen werden. Es braucht Augenmaß. Und es braucht auch ein Gespür für den Trade-off zwischen manueller und beauftragter Arbeit: ChatGPT könnte vielleicht sogar mehr – doch dafür müsste ich die Prompts sehr ausführlich formulieren und in einem Dialog noch nachlegen. Da ist es in vielen Fällen schneller, das Ergebnis selbst zu finden. Im nächsten Teil der

Artikelserie kann ChatGPT aber endlich zeigen, aus welchem Holz es als Codierer geschnitzt ist. Ich bin gespannt, wie es mit meiner Vorarbeit zurechtkommt. ■

- [1] Ralf Westphal, *Am Anfang steht die Anforderung*, dotnetpro 7/2023, Seite 39 ff., www.dotnetpro.de/A2307ChatGPT
- [2] ChatGPT Chat 1, www.dotnetpro.de/SL2308ChatGPT1
- [3] ChatGPT Chat 2, www.dotnetpro.de/SL2308ChatGPT2
- [4] ChatGPT Chat 3, www.dotnetpro.de/SL2308ChatGPT3
- [5] Ralf Westphal, *Stratified Design over Layered Design*, www.dotnetpro.de/SL2308ChatGPT4
- [6] *Spaced Repetition bei Wikipedia*, www.dotnetpro.de/SL2308ChatGPT5
- [7] *Repetico*, www.repetico.de
- [8] *Spanish Day by Day – Jeden Tag kostenlose Spanischlektionen mit der Unterstützung von ChatGPT*, <https://spanishdaybyday.substack.com/>
- [9] ChatGPT Chat 4, www.dotnetpro.de/SL2308ChatGPT6
- [10] Ralf Westphal, *Entmystifiziert*, dotnetpro 11/2020, Seite 26 ff., www.dotnetpro.de/A2011SRP
- [11] Ralf Westphal, *Softwareentwurf mit Flow-Design*, www.dotnetpro.de/SL2308ChatGPT7
- [12] Ralf Westphal, *IOSP 2.0*, www.dotnetpro.de/SL2308ChatGPT8
- [13] Ralf Westphal, *IODA Architecture*, www.dotnetpro.de/SL2308ChatGPT9
- [14] ChatGPT Chat 5, www.dotnetpro.de/SL2308ChatGPT10
- [15] *Zero-Shot Prompting*, www.dotnetpro.de/SL2308ChatGPT11
- [16] ChatGPT Chat 6, www.dotnetpro.de/SL2308ChatGPT12
- [17] *Stacked Prompts*, www.dotnetpro.de/SL2308ChatGPT13
- [18] *Zero Shot Chain of Thought*, www.dotnetpro.de/SL2308ChatGPT14
- [19] *Teach LLMs How To Reason With Chain-Of-Thought Prompting*, www.dotnetpro.de/SL2308ChatGPT15
- [20] *A Guide to Smarter Prompts for Unparalleled AI-Generated Content*, www.dotnetpro.de/SL2308ChatGPT16
- [21] ChatGPT Chat 6v2, www.dotnetpro.de/SL2308ChatGPT17
- [22] *Primitive Obsession*, www.dotnetpro.de/SL2308ChatGPT18
- [23] Ralf Westphal, *Logic Makes the Software Turn Around*, www.dotnetpro.de/SL2308ChatGPT19
- [24] *Kenneth Lange – The Functional Core, Imperative Shell Pattern*, www.dotnetpro.de/SL2308ChatGPT20



Ralf Westphal

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

<https://ralfw.de>

dnpCode A2308ChatGPT