Java™ magazin
Java | Architektur | Software-Innovation

KOTLIN 2.0

Kleine Schritte, große Wirkung

entwickler.de
Ausgabe 2.2025

Wie Codeverständnis in unserem Gehirn entsteht

Code und Kognition

WTF! Wer hat diesen Code geschrieben? Git sagt: Ich. Gestern. Und warum verstehe ich den Code jetzt nicht mehr? Hoffentlich hat den Code niemand anderes gelesen … Vermutlich geht es nicht nur mir so. Aber warum? Welche kognitiven Prozesse laufen beim Verstehen ab? Und wie müssen wir Code schreiben, damit wir (und andere) ihn auch später noch verstehen?

von Stefan Mandel

Bevor wir uns praktisch mit Code auseinandersetzen, beleuchten wir in den ersten Abschnitten das Thema Verständnis aus wissenschaftlicher Sicht. Dabei lernen wir die verschiedenen Zuständigkeiten des Gedächtnisses kennen und wie die Informationen in Form von Chunks verarbeitet werden. Im darauffolgenden Abschnitt werden wir Code sehen, der unser Gedächtnis überfordert, und leiten ab, welche Vereinfachungen es weniger belasten.

Danach zeigen wir praktische Wege, wie man das in den vorigen Abschnitten erarbeitete Modell einsetzen kann. Wir untersuchen, wie man Ursachen von Unverständlichkeit identifiziert und welchen Effekt Clean Code hat. Es folgen Hinweise, wie man – individuell und im Team – die Fähigkeiten zum Schreiben verständlichen Codes kontinuierlich verbessern kann.

Verständnis und Gedächtnis

Beginnen wir mit einem Zitat aus Wikipedia: „Verstehen (auch Verständnis genannt) ist das inhaltliche Begreifen eines Sachverhalts, das nicht nur in der bloßen Kenntnisnahme besteht, sondern auch und vor allem in der intellektuellen Erfassung des Zusammenhangs, in dem der Sachverhalt steht“ [1].

Verständnis entsteht durch kognitive Prozesse in unserem Gedächtnis. Als leicht verständlich empfinden wir

Abb. 1: Merkmale für den Chunk „Igel“ (Bildquellen: [2], [3])

Aufgaben, die unser Gedächtnis gut bewältigen kann, als schwer verständlich dagegen solche, bei denen unser Gedächtnis unter Stress gesetzt wird. Wenn wir also besseres Verständnis erreichen möchten, müssen wir uns ansehen, wie das Gedächtnis optimal genutzt werden kann.

Der Begriff Gedächtnis umfasst dabei (leicht abweichend von der alltäglichen Bedeutung) alle Prozesse, Informationen aufzunehmen, umzuwandeln und anzuwenden. Das Gedächtnis hat drei Zuständigkeiten:

- Aufnehmen von Informationen (sensorisches Gedächtnis)
- Verarbeiten von Informationen (Arbeitsgedächtnis)
- Speichern von Informationen (Langzeitgedächtnis)

Langzeit- und Arbeitsgedächtnis spielen beim Verstehen von Code eine Schlüsselrolle. Wir untersuchen in den folgenden Abschnitten zunächst ihre Eigenschaften und widmen uns dann den grundlegenden Konzepten des Verständnisprozesses – sowohl allgemein als auch speziell in Bezug auf Code.

Langzeitgedächtnis – der Hauptspeicher

Das Langzeitgedächtnis speichert Informationen. Manches nur wenige Minuten, manches Stunden, Tage, Monate, Jahre oder ein Leben lang. Die Informationen werden in sogenannten Chunks gespeichert. Ein Chunk entspricht dabei einer Vorstellung (Ideen, Erfahrungen) zu einem Konzept. Während Konzepte objektiv existieren, sind Chunks (d. h. die Ideen zu diesen Konzepten) sehr individuell. Fehlt uns jeglicher Bezug zu einem Konzept, existiert dafür auch kein entsprechender Chunk in unserem Gedächtnis.

Beim Erlernen eines Chunks und beim Hinzulernen (dem Entwickeln einer Idee) wird der Chunk mit Merkmalen „verschlagwortet“, und die Suchmaschine „Langzeitgedächtnis“ kann anhand von Merkmalen effizient einen oder mehrere Chunks finden. Deren Merkmale sind dabei sehr vielfältig, z. B. handelt es sich um:

- sensorische Eigenschaften (Aussehen, Klang, Geruch …)
- Schlüsselreize, Ereignisse (z. B. ein Ball, der auf jemanden zufliegt)
- Namen

Merkmale sind selbst wieder Chunks. Oft begreifen wir den Namen eines Chunks (ein Merkmal) mit diesem selbst, so etwa wie wir Primärschlüssel mit einem Datenbankeintrag assoziieren. So kann der Eindruck entstehen, dass Chunks „einfache, griffige Informationspakete“ sind. Richtiger ist jedoch, dass sie (für diejenigen, die sie kennen) einfach und griffig (z. B. anhand ihres Namens) abgerufen werden können.

Die Merkmale in **Abbildung 1** beziehen sich auf den gleichen Chunk, nämlich unsere Vorstellung von einem Igel. Wir bemerken darin gleich mehrere Aspekte von Merkmalen und Chunks.

Beim Erfassen der Bilder/Namen haben wir sofort eine Vorstellung. Diese ist allerdings abhängig von unserer Erfahrung:

- Das erste Bild wird oft mit einem Igel assoziiert, dabei handelt es sich gar nicht um einen Igel, sondern um einen Igeltenrek (der mit dem Igel weniger verwandt ist als mit Elefant und Seekuh). Hier würden Experten einen anderen Chunk wahrnehmen.
- Das zweite Bild ist für manche nur ein „Igel“, für Experten aber ein „Europäischer Weißbrustigel“.
- „Igel“ vermittelt uns zwar eine Vorstellung, lässt aber deutlich mehr Raum für Interpretationen als die beiden Bilder. Der Chunk ist also kleiner bzw. weniger abgegrenzt als derjenige der ersten zwei Beispiele.
- „Erinaceus“ wird vermutlich nur bei Biologen und Lateinern eine Vorstellung auslösen.

Was wir ebenfalls sehen: Unser Gedächtnis interpretiert offensichtlich immer den größten verfügbaren Chunk. Wir sehen auf den Bildern zahlreiche Merkmale (Stacheln, Schnauze, Augen, Beine), und auch in den Namen sind zahlreiche Details (Buchstaben, Linien) und dennoch nehmen wie zuerst nur „Igel“ und nicht die Details wahr.

Chunks können vielfältiger Natur sein; es gibt Chunks, die

- explizites Wissen repräsentieren (Daten): Domänenwissen, Design Patterns
- Fertigkeiten beschreiben (Funktionen): Tastaturschreiben, Formulieren, Refactoring
- Reaktionen beschreiben (Event Handler): Vorstellungen, Gefühle, Tendenzen

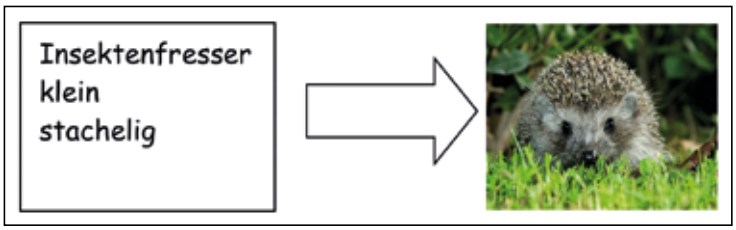

Abb. 2: Abruf von Chunks mit Merkmalen

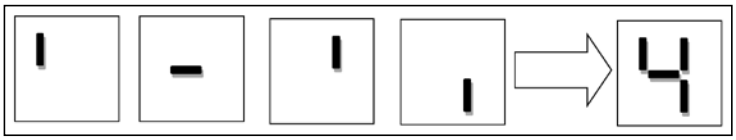

Abb. 3: Mehrere kleine Chunks werden zu einem großen

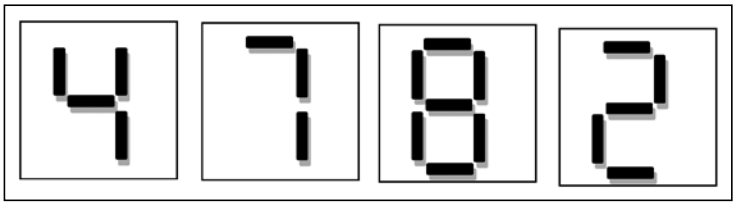

Abb. 4: Digitaldarstellung der Zahl 4782 (ein Chunk, viele Detailchunks)

Konkret würde ein fliegender Ball bei uns den Chunk „Fangreflex“ oder den „Abwehrreflex“ auslösen. Unverständlicher Code löst bei uns (je nach Natur des Lesers) einen Chunk für die Emotion „Enttäuschung“, „Frustration“ oder „Überforderung“ aus.

Oft sind die Grenzen zwischen den einzelnen Chunk-Arten aber fließend. „Addieren“ ist grundsätzlich eine Fertigkeit, aber der Mathematiker wird mit diesem Chunk noch andere explizite Details abrufen (z. B. Kommutativgesetz, Assoziativgesetz, Punkt vor Strich …).

Arbeitsgedächtnis – der Prozessor

Im Gegensatz zum Langzeitgedächtnis, das als Speicher fungiert, ist das Arbeitsgedächtnis für die Verarbeitung zuständig, z. B.

- das Ableiten von Informationen (Schließen),
- das Kombinieren von Informationen (Rechnen),
- das Reagieren auf Informationen (Ereignisse) und das Einordnen von Informationen in den besten Kontext (Lernen),
- sämtliche Anwendungen des Langzeitgedächtnisinhalts auf eine konkrete Situation.

Dabei ist das Arbeitsgedächtnis – im Gegensatz zum Langzeitgedächtnis – limitiert (Kasten: „Die Kapazität des Arbeitsgedächtnisses“). Aktuell geht die Wissenschaft davon aus, dass gleichzeitig nur etwa vier Chunks (+/- 1) aktiviert/kombiniert werden können. Da das Arbeitsgedächtnis ständig am Arbeiten ist, halten sich Chunks dort auch nur sehr kurz.

Beispielhaft würde im Arbeitsgedächtnis der Prozess aus **Abbildung 2** ablaufen. Hinter jedem der Merkmale in der Abbildung befinden sich wieder Chunks, die zusammen auf das richtige Tier hinweisen können. Und dieses Beispiel ist für uns auch dann noch nachvollziehbar, wenn unsere Chunks nicht vollständig korrekt hinterlegt sind. Das Arbeitsgedächtnis führt keine exakten Ableitungen durch und ergänzt fehlende Informationen mit dem, was es für wahrscheinlich hält.

Insbesondere können wir hier auch lernen, dass das Nachvollziehen einfacher ist, als den Zusammenhang selbst herzustellen. Beim Nachvollziehen haben wir sowohl Input-Chunks als auch Output-Chunks zur Verfügung und das Arbeitsgedächtnis kann daraus einen plausiblen Zusammenhang herstellen. Es klingt plausibel, dass ein kleiner stacheliger Insektenfresser ein Igel ist.

Beim selbstständigen Herstellen des Zusammenhangs haben wir den Output-Chunk nicht und müssen deswegen eine deutlich klarere Vorstellung von den Input-Chunks haben, um zum richtigen Ergebnis (Output-Chunk) zu kommen. Hier müssen wir aus „klein“, „stachelig“, „Insektenfresser“ auf „Igel“ kommen. Das ist insbesondere für Personen, die den biologischen Fachbegriff „Insektenfresser“ nur vage kennen, gar nicht so einfach.

Die Verarbeitung von Chunks

Auf den ersten Blick erscheint es widersprüchlich: Unser Arbeitsgedächtnis kann nur vier Chunks gleichzeitig verarbeiten – und dennoch meistern wir täglich komplexe Aufgaben, die deutlich mehr Informationen erfordern. Wie ist das möglich?

Die Antwort liegt in der bemerkenswerten Flexibilität unserer Chunks. Was zunächst wie eine einzelne Information aussieht, kann bei Bedarf neue Details zur Verfügung stellen. Das Arbeitsgedächtnis kann in seinen Verarbeitungsschritten alle verknüpften Informationen berücksichtigen, wobei genau die Details priorisiert werden, die für die Aufgabe als relevant eingestuft werden. **Abbildung 3** illustriert exemplarisch, wie sich Chunks zusammensetzen.

Alle Operanden haben ihre Entsprechung im Langzeitgedächtnis als Chunk:

- die Chunks auf der linken Seite der Abbildung würden wir vermutlich als „Balken“ abspeichern

Die Kapazität des Arbeitsgedächtnisses

Die Idee, dass das Arbeitsgedächtnis beschränkt ist, ist schon etwas älter und stammt von dem Psychologen George A. Miller, der 1956 in seinem Aufsatz „The magical number seven, plus or minus two“ die Kapazitätsgrenze bei sieben +/-2 sah [4]. Diese „magische Sieben“ galt noch bis in die 2000er Jahre als wahrscheinliche Grenze.

Die Kapazitätsgrenze von vier Chunks geht auf die Ergebnisse von Nelson Cowan („The Magical Mystery Four“, 2010) zurück, die auch unabhängig bestätigt wurden [5]. Die neueren Veröffentlichungen gehen davon aus, dass frühere Experimente oft die Bedeutung von Chunks falsch eingeschätzt haben.

- der Chunk auf der rechten Seite ist deutlich komplexer:
  - ein Muster von Balken
  - die Ziffer 4 (für alle, die arabische Ziffern und digitale Displays als Chunks verfügbar haben)
  - die Zahl 4 (für alle, die den Unterschied von Zahlen und Ziffern kennen)

Mit dem Chunk wird aus dem Langzeitgedächtnis also nicht nur die Zahl 4 geholt, sondern auch zahlreiche Merkmale, die wir direkt verarbeiten können. Wenn wir uns die Zahl 4782 auf einem Digitaldisplay vorstellen, kommen wir auf die Darstellung in **Abbildung 4**.

Das fällt uns recht leicht, weil wir arabische Zahlen kennen und wissen, wie eine Ziffer auf einem Digitaldisplay aussieht. Alle beteiligten Chunks (z. B. 19 Balken) reproduzieren wir einfach mit. Nun stellen wir uns vor, wie es Personen geht, die keine arabischen Zahlen kennen und auch keine Digitaldarstellungen von diesen Ziffern. Die Lösung erfolgt dann nicht mehr überwiegend im Langzeitgedächtnis, sondern explizit im Arbeitsgedächtnis, in etwa so:

- Zeichne Balken links oben, senkrecht
- Zeichne Balken in der Mitte, waagrecht
- Zeichne Balken rechts oben, senkrecht
- Zeichne Balken rechts unten, senkrecht usw.

Was wir an dem Beispiel ebenfalls erkennen können: Es gibt Chunks, die nur sehr generische Informationen kapseln (z. B. Bilder oder andere Sinneseindrücke), es gibt aber auch solche, die eine ganze Fülle von Details enthalten (nicht nur Formen, sondern auch Ziffern und Zahlen).

Wie viele Details wir mit einem Chunk abrufen können, ist sehr individuell damit verknüpft, wie wir die Chunks üblicherweise einsetzen:

- Für den Abiturienten ist ein Vektor ein Pfeil, eine Doktorin in linearer Algebra wird das eventuell nicht so stehen lassen wollen.
- Für eine Schulanfängerin ist ein Stück Code nur ein Haufen Buchstaben und Klammern, ein Softwareentwickler sieht darin eine Struktur- und Verhaltensbeschreibung.
- Für die Übersetzerin ist ein Chunk nur ein „Klumpen“, für jeden Leser dieses Artikels wird das am Ende womöglich nicht mehr so sein.

Chunks im Quellcode

Wo finden wir nun die Chunks im Quellcode? Im Grunde ist alles im Code, zu dem wir eine Vorstellung haben, ein Chunk, wie etwa:

- Schlüsselwörter (*class*, *struct*, *function*) vermitteln den Anfang eines Programmelements
- Trennzeichen (Semikolon, Komma) trennen Programmelemente
- Gruppierungszeichen (Klammern) gruppieren Programmelemente
- Operatoren vermitteln eine Operation auf Argumenten
- Literale werden je nach Kontext als Wert oder Variable interpretiert
- Namen vermitteln eine Semantik
- Argumentpositionen vermitteln eine Rolle

Beim Lesen eines Programms werden die einzelnen Chunks im Arbeitsgedächtnis kombiniert, es ergeben sich neue Chunks und damit eine Vorstellung, was das Programm tut. Der Verständnisprozess für das Programm in Listing 1 würde in etwa so aussehen.

1. *Struktur erkennen:* Klasse, Feld, Methode
2. *Fachliche Bedeutung erfassen:* Konto (*Account*), Guthaben (*balance*), Überweisung (*Transfer*)
3. *Ablauf verstehen:* Prüfung → Abbuchung → Gutschrift
4. *Gesamtbild:* eine Bankkontoabstraktion

Erfahrene Entwickler erfassen das in kurzer Zeit – sie profitieren von sehr detaillierten Chunks in ihrem Langzeitgedächtnis. Anfänger oder fachfremde Personen brauchen länger – in ihrem Langzeitgedächtnis sind die betreffenden Chunks eher flach, und sie müssen die Zusammenhänge überwiegend im Arbeitsgedächtnis herstellen.

Was überfordert unser Gedächtnis?

Unser Gedächtnis wird überlastet, wenn wir das Limit des Arbeitsgedächtnisses erreichen oder zu überschreiten versuchen. Das kann auf verschiedene Arten passieren:

- Eine Operation verarbeitet zu viele Input-Chunks.
- Unser Gedächtnis kann die Output-Chunks nicht mehr halten.
- Unser Gedächtnis muss Chunks von früheren Berechnungen (Zwischenergebnisse) vorhalten.

Den Hinweisen aus den vorigen Abschnitten zufolge wäre es sinnvoll, einfach die vielen Chunks zu wenigen zusammenzufassen (spontane Chunks). Das ist we-

Listing 1: Typische Klasse in Java

```
class Account {
  Money balance;

  void transferTo(Account target, Money amount) {
    if (amount > this.balance) {
      throw new IllegalTransferException();
    }
    this.balance = this.balance - amount;
    target.balance = target.balance + amount;
  }
}
```

der abwegig noch unrealistisch. Wir bilden tatsächlich spontan Chunks, in denen wir mehrere zusammenfassen. Deshalb fällt es uns zunächst leicht, den eigenen Code zu lesen und zu verstehen. Allerdings sind uns diese Chunk-Bildungen nicht bewusst, was zu folgenden Problemen führt:

- Spontane Chunks entspringen unterbewusst unserer Intuition. Wir können sie nicht kommunizieren.
- Somit haben fremde Leser nicht die Möglichkeit, diese Chunks zu übernehmen und damit zu verstehen.
- Da wir diese Chunks nicht bewusst gelernt haben, geraten sie auch einfach in Vergessenheit. Nach wenigen Wochen sind sie aus dem Langzeitgedächtnis verschwunden. Wer kennt nicht die obige Situation, dass einem der eigene Code auf einmal fremd vorkommt?

Damit ist die Variante, unterbewusst Chunks zu bilden, kein nachhaltiger Lösungsansatz. Insbesondere beim Lesen von fremdem Code geht das Gehirn aber noch zwei andere Wege.

Eine Möglichkeit, mit zu vielen Chunks umzugehen, besteht darin, sich auf die „wichtigsten" zu fokussieren. Sofern man die „Wichtigkeit" gut abschätzen kann, ist dieses Verfahren in vielen Fällen recht erfolgreich:

```
System.out.println("result=" + (2 + 3));
```

Diese Zeile Code enthält mehr als vier Chunks (Namen, Strings, Operatoren, Klammern, Semikolon). Das Fokussieren können wir uns wie folgt vorstellen:

1. Ausblenden von Klammen und Satzzeichen
2. Ignorieren von *System.out.println*
3. Konzentration auf die Kernoperation *2 + 3*

Am Ende verstehen wir, dass die Zeile "*result=5*" ausgibt. Wir brauchen für das vollständige Verständnis mehrere Schritte, aber wirklich schwierig empfinden wir es nicht, weil jeder einzelne Schritt vom Arbeitsgedächtnis gut bewältigt werden kann.

Diese naive Fokussierung funktioniert aber nicht immer so gut. In folgendem Beispiel

```
System.out.println(2 + 3 * 4 – 8 / 2);
```

bleibt nach Entfernen der unwichtigen Teile *2 + 3 * 4 – 8 / 2* übrig und die typische Fokussierung betrifft nun die am weitesten links gelegene Operation, also

```
  2 + 3 * 4 – 8 / 2
= 5 * 4 – 8 / 2
= 20 – 8 / 2
= 12 / 2
= 6
```

Das Problem ist, dass unseren internen Heuristiken nicht nach „Punkt vor Strich" priorisieren, sondern von links nach rechts. In der europäischen Gesellschaft ist diese Heuristik im Allgemeinen treffsicherer. Wir werden noch sehen, wie wir die Heuristiken auch für unsere Zwecke einsetzen können.

Eventuell gibt es Leser, die tatsächlich effizienter (richtig) fokussieren können, aber die meisten werden eher (unterbewusst) auf eine andere Variante ausweichen – die Problemzerlegung. Dabei erkennt unser Gedächtnis, dass

- es zu viele Chunks sind, um sie in einem Schritt zu kombinieren,
- die Chunks nicht spontan zu verständlichen Chunks vereinfacht werden können und
- eine naive Fokussierung nur zu falschen Ergebnissen führen würde.

Durch die Regel „Punkt vor Strich" (selbst ein Chunk) wissen wir, dass die naive Heuristik nicht zielführend ist. In diesem Fall erarbeitet das Gedächtnis mit der Problemzerlegung selbstständig eine Lösungsstrategie. Das ist jedoch mit einigen Nachteilen verbunden:

- Wir verlassen damit die Problemlösung selbst und begeben uns auf eine Metaebene (Suche nach dem Problemlöser).
- Die Suche ist anstrengend.
- Nach der Suche muss der Problemlöser auch noch schrittweise angewandt werden.
- Es ist nicht gesagt, dass wir überhaupt einen Problemlöser finden.

Für das konkrete Problem ist die Suche nach einem Problemlöser üblicherweise erfolgreich. Die Lösung des Problems besteht aus mehreren Schritten:

- Gedankliches Ersetzen von Multiplikationen und Divisionen durch geklammerte Ausdrücke
- Auswertung von Klammern von innen nach außen
- Auswertung innerhalb einer Klammer von links nach rechts

So kommen wir dann auf folgende Berechnungsschritte zu dem richtigen Ergebnis:

```
  2 + 3 * 4 – 8 / 2
= 2 + (3*4) – (8/2)
= 2 + 12 - 4
= 10
```

Bisher haben wir die Gedächtnisüberlastung hauptsächlich im Kontext des Arbeitsgedächtnisses betrachtet. Zwar zeigen sich die meisten Überlastungssituationen dort, doch die eigentliche Ursache liegt häufig im Langzeitgedächtnis begründet. Besonders kritisch wird es dann, wenn dem Leser die großen Chunks fehlen und er nur die Details wahrnimmt. Das würde z. B. passieren, wenn man **Abbildung 1** ansehen würde, ohne überhaupt

eine Vorstellung von einem Igel zu haben, oder **Abbildung 4** ohne die Kenntnis von arabischen Zahlen und Digitaldarstellungen.

Wie bereits erwähnt, ist spontanes Chunking zwar möglich, aber kurzlebig. Nachhaltiger ist es hingegen, wenn der Code Chunks adressiert, die bereits im Langzeitgedächtnis des Lesers angelegt sind. Denn dann kann das Arbeitsgedächtnis bei Bedarf sehr schnell diese – bereits verfügbaren – Chunks abrufen und kombinieren. Ideal ist es, wenn besagte Chunks dann auch noch eine Vielfalt von assoziierten Informationen aufweisen. Üblicherweise nennen wir Chunks, die Informationen zusammenfassen, in der Informatik „Abstraktionen". Diese müssen wir deshalb aktiv lernen. Nehmen wir einmal den Code in Listing 2.

Die Verständlichkeit des Codes wird erschwert, da die Namen durch Nonsens ersetzt wurden. Diese Form der Unkenntlichmachung, die Obfuskierung, basiert darauf, so viele Stellen wie möglich von den Chunks in unserem Gedächtnis zu entkoppeln. Der Code aus Listing 2 lässt sich allein durch Kenntlichmachung der Namen in den Code aus Listing 3 überführen.

Diesen Code verstehen alle Design-Patterns-Experten (Observer Pattern). Diejenigen, die das Pattern nicht kennen, sehen kaum einen Unterschied zum Beispiel in Listing 2. Doch selbst Experten erkennen das Muster in anderer Form nicht mehr, wenn man sie nicht unmissverständlich darauf stößt.

Der Code in Listing 4 (TypeScript) ist im Grunde äquivalent zu dem Java-Code in Listing 3. Dennoch hatte ich schon heiße Diskussionen darüber, ob es sich um das gleiche Muster handelt. Solche Diskussionen entstehen, wenn der eigene Chunk „Observer Pattern" im Detail anders hinterlegt ist als der des Diskussionspartners. Mit etwas Glück einigt man sich am Ende auf eine Sichtweise und gleicht damit die Chunks an. Das nennt sich dann „Lernen".

Daraus lässt sich folgern: Es erfordert viel Erfahrung, also viele im Gedächtnis griffbereit abgelegte Chunks, um Programmcode flüssig lesen und verstehen zu können. Code kann für unerfahrenere oder fremde Entwickler überfordernd wirken, während ihn Expertinnen als lesbar und verständlich einstufen.

Wie können wir einfachen Code schreiben?

Nun haben wir ein gutes kognitionspsychologisches Erklärungsmodell. Daraus auch Methoden oder Rezepte zu entwickeln, gestaltet sich leider schwierig. Die Kommunikationshürde zwischen Schreiber und Leserin lässt sich nicht einfach generalisieren. Die Vereinfachung von Code ist also immer auf die Chunks von Schreiber und Leserin limitiert – was effektiv individuelle Optimierungen für jedes Schreiber-Leserin-Paar erfordern würde.

Weiterhin können wir in Turing-mächtigen Programmiersprachen eine unglaublich große Komplexität abbilden. Die schiere Menge an möglichen Codekonstrukten wäre innerhalb eines Teams nicht über gemeinsame Chunks abbildbar. Somit ist es von vornherein sinnvoll, sich nur auf die Chunks zu beschränken, die man zum Verständnis der Domäne benötigt.

Listing 2: Obfuskiertes Design Pattern

```
class A {
  List<B> bs;

  void r(B b) {
    bs.add(b);
  }

  void t() {
    for (B b : bs) {
      b.n();
    }
  }
}
```

Listing 3: Design Pattern Observer

```
class Subject {
  List<Observer> observers;

  void register(Observer observer) {
    observers.add(observer);
  }

  void trigger() {
    for (Observer observer : observers) {
      observer.notitfy();
    }
  }
}
```

Listing 4: Design Pattern Observer in TypeScript

```
class Subject {
  List<()=>{}> callbacks;

  onEvent(callback :()=>{}) : void {
    this.callbacks.add(callback);
  }

  event() : void {
    for (var callback in this.callbacks) {
      callback();
    }
  }
}
```

Statt nach einer perfekten Formel zu suchen, konzentrieren wir uns in den folgenden Abschnitten auf drei praktische Anwendungen:

1. *Clean Code neu verstehen:* bewährte Praktiken im Licht kognitiver Prozesse analysieren
2. *Unverständlichen Code analysieren:* Vorgehen zur systematischen Analyse problematischer Codestrukturen
3. *Persönliche Entwicklung:* konkrete Strategien für besseren Code

Clean Code neu verstehen

Clean Code hat sich als Sammlung bewährter Praktiken etabliert, die zu verständlichem und wartbarem Code führen. Aber warum funktionieren diese Faustregeln? Unser kognitionspsychologisches Modell liefert die wissenschaftliche Grundlage für ihre Wirksamkeit. Betrachten wir also einige Clean-Code-Regeln und ihre kognitiven Mechanismen:

Zunächst betrachten wir das „Principle of Least Astonishment“ (Prinzip der geringsten Überraschung). Dieses Prinzip besagt, dass der Code so geschrieben sein sollte, dass der Leser/Verwender bei einer normalen Benutzung möglichst wenige Überraschungen erlebt.

Wir haben weiter oben bereits von Heuristiken gesprochen, nach denen unser Gedächtnis Informationen fokussiert. Wenn wir unseren Code nun so schreiben, dass sämtliche Heuristiken auch zum gewünschten Ergebnis führen, dann haben wir das „Principle of Least Astonishment“ erfüllt. Statt folgendem Beispiel:

```
var result = 2 + 3 * 4 - 8 / 2;
```

hätten wir z. B. folgende Darstellung wählen können:

```
var result =
   2
  + 3 * 4
  - 8 / 2;
```

In unserer Kultur verwenden wir die Heuristik: von oben nach unten, von links nach rechts. Durch die Gruppierung fokussiert das Gehirn die Zeilen separat und errechnet so unter Anwendung der Heuristik das richtige Ergebnis.

Betrachten wir nun die Regel „maximal zwei Argumente in Methoden“. Diese Heuristik lässt sich darauf zurückführen, dass man in einer typischen, objektorientierten Programmzeile, wie z. B.

```
me.give(money, you);
```

folgende Chunks aktivieren muss, um eine Anweisung zu verstehen:

- den Zustand des Subjekts (*me*)
- die Aktion des Subjekts (*give*)
- Argument 1 der Methode (*money*)
- Argument 2 der Methode (*you*)

Das sind 4 Chunks bei typischen Methodenaufrufen mit zwei Argumenten. Mit einem dritten Argument würden wir 5 Chunks kombinieren und unser Limit im Arbeitsgedächtnis (vier Chunks) sprengen. Die Regel mit den zwei Argumenten ist also keineswegs eine willkürliche Konvention, sondern begründet sich darin, dass wir mehr Informationen nicht gleichzeitig verarbeiten können.

Außerdem ist das Erklärungsmodell hier flexibler, weil es auch Erklärungen für die Verständlichkeit von Code liefert, der nicht der Faustregel entspricht:

```
me.give(20, "dollars", you);
```

aktiviert folgende Chunks:

- den Zustand des Subjekts (*me*)
- die Aktion des Subjekts (*give*)
- Argument 1 und Argument 2 der Methode (*20 dollars*)
- Argument 3 der Methode (*you*)

Es werden also mit drei Argumenten dennoch nur vier Chunks aktiviert, weil wir Zahlen, gefolgt von Einheiten automatisch zu einem neuen Chunk zusammenfassen.

Darüber hinaus ist das Erklärungsmodell auch stringenter. Mehr als vier Chunks sprengen das Limit des Arbeitsgedächtnisses und beeinflussen damit die Verständlichkeit. Das gilt dann, wenn „es einfach nicht besser geht“. Wer sich dennoch für mehr als vier Chunks entscheidet, sollte Gründe haben, die dem schlechteren Verständnis entgegenstehen.

Ein weiteres Beispiel ist die Forderung nach guten Namen. Namen sollten sprechend sein, d. h., der Begriff sollte mit einer Idee hinterlegt sein. Und zwar nicht mit irgendeiner Idee, sondern mit der richtigen. Statt:

```
var s = computeS(e);
```

lieber:

```
var salary = computeSalary(employee);
```

Und die Namen sollten kurz sein. Ein Name ist z. B. dann zu lang, wenn er Bestandteile enthält, die sich aus dem Kontext erschließen lassen. Statt:

```
var employeeSalary = computeSalary(employee);
```

lieber:

```
var salary = computeSalary(employee);
```

Natürlich ist das obere Beispiel nicht wirklich schlechter Code. Aber wenn der Kontext (Angestelltengehälter) klar

ist, dann wird hier ein Detail im Namen ausgedrückt, das die interne Heuristik ohnehin hervorgebracht hätte.

Vorsicht auch bei zusammengesetzten Wörtern. Statt:

```
class RecognizerAndSyntaxTreeBuilder {
  ..
}
```

lieber:

```
class Parser {
  ...
}
```

Formal ist das obere Beispiel präziser. Wir nehmen es aber immer als drei Chunks (*Recognizer*, *SyntaxTree*, *Builder*) wahr. Um eine Vorstellung zu entwickeln, muss stets eine Verknüpfung im Arbeitsgedächtnis erfolgen. Wer hingegen einmal gelernt hat, was ein Parser ist, speichert den Chunk ab und kann in Zukunft mit nur einem Chunk das Konzept erfassen.

Wir verwenden viele Begrifflichkeiten ganz intuitiv, obwohl sie nicht präzise sind: So nutzen wir Begriffe wie Stack und Heap intuitiv, obwohl es präzisere Bezeichner gäbe (*LocalVariableMemory*, *ExternalDataMemory*). Vielleicht erinnern sich einige Informatiker noch an die deutschen Begriffe Keller (für Stack) und Halde (für Heap), welche oft etwas Belustigung auslösten. Warum? Weil Keller und Halde bei uns bereits mit anderen Chunks hinterlegt sind und diese Begriffe im Kontext von Software sehr absurd klingen.

Eine Besonderheit sind Namen, die beim Bereinigen von dupliziertem Code entstehen. Hier wird aus:

```
var s = from(str).replace("&","&amp; ").replace("<","&lt; ");
var t = from(html).replace("&","&amp; ").replace("<","&lt; ");
```

folgender Code:

```
var s = replace(str);
var t = replace(html);
```

Zwar ist die Duplikation jetzt entfernt, aber leider auch die klare Semantik. Hier hingegen

```
var s = replaceAmpAndLt(str);
var t = replaceAmpAndLt(html);
```

ist die Semantik wieder klarer, der Name aber weder einprägsam noch verständlich. Besser wäre:

```
var encodedStr = encodeHtmlEntities(str);
var encodedHtml = encodeHtmlEntities(html);
```

und zwar dann, wenn man bereit ist, auch alle anderen Schlüsselzeichen in HTML zu kodieren. Damit haben wir nicht einfach gleiche Codestellen vereinheitlicht, sondern wir haben eine neue Operation (Abstraktion) gefunden, die mit klarer Semantik (und Namen) wiederverwendbar ist. Ich würde tatsächlich empfehlen, Duplikation nur dann zu entfernen, wenn sich eine geeignete Abstraktion findet.

Unverständlichen Code analysieren

Wenn Code schwer verständlich ist, fällt es oft nicht leicht, den genauen Grund dafür zu benennen. Das Erklärungsmodell bietet hier einen strukturierten Weg zur Analyse: Es erlaubt uns, Verständnisprobleme systematisch zu untersuchen, ihre Ursachen präzise zu identifizieren und gegebenenfalls Maßnahmen abzuleiten. Bei der Analyse können wir uns folgende Fragen stellen:

- Wie viele Chunks muss ich gleichzeitig verarbeiten?
  - Mehr als vier Chunks werden von uns als kompliziert wahrgenommen
- Kann ich mir unter den Variablen, Methoden, Klassen etwas vorstellen?
  - Gibt es Wörter zu denen mir die Vorstellung (bzw. der Chunk) fehlt?
- Habe ich echte Abstraktionen oder nur Zusammenfassungen?
  - Und sind die Abstraktionen auch in meinem Kontext verständlich?
- Gibt es Widersprüche zwischen der niedergeschriebenen Logik und den Namen?
  - Passen die Methodennamen (Methoden) nicht zu den Variablennamen (Operanden)?
- Passt der Datenfluss oder Kontrollfluss zu den Abstraktionen?

Nach der Problemidentifikation können folgende Maßnahmen ergriffen werden:

- zu viele Chunks gleichzeitig:
  - Zerlegung der Logik in kleinere Schritte
  - (Teil-)Operationen abstrahieren
  - Zwischenergebnisse möglichst früh konsumieren
  - multiple Ergebnisse abstrahieren oder aggregieren
  - falsche Namen korrigieren
  - ungenaue Namen präzisieren
  - Namenskonventionen einhalten
- falsche Abstraktionen:
  - Suche nach passenderen Abstraktionen
  - falsche Abstraktionen auflösen und neu gruppieren
- Widersprüche in Logik und/oder Namen:
  - Klärung, ob Logik oder Namen das Problem sind

Persönliche Entwicklung

Mit der Kenntnis, wie das Gedächtnis typischerweise arbeitet, lassen sich auch Maßnahmen ableiten, um von

vornherein verständlicheren Code zu erzeugen. Hier wird es ein wenig subjektiv, deswegen gebe ich Beispiele aus meiner eigenen Erfahrung.

Pair Programming ist eine in der agilen Entwicklung etablierte Methodik. Auf die Verständlichkeit von Code hat sie mehrere positive Einflüsse. Allgemein ist Pair Programming eine effiziente Möglichkeit, den Code gleichzeitig aus der Perspektive des Lesers und der Schreiberin zu erleben.

Durch den ständigen Dialog werden neue Namen stets von zwei Personen gelesen, und die Chunks dazu werden schon bei der Erstellung abgeglichen. Wenn ein Variablenname oder eine Abstraktion einmal von beiden Entwickler:innen unterschiedlich wahrgenommen wird, besteht hier noch recht früh die Möglichkeit, die Verständlichkeitsprobleme zu lösen.

Eine weitere Art, den Code bzw. das Design verständlich zu halten, ist das sprachlich einfache Design: „Designe so, dass du möglichst wenig Dokumentation schreiben müsstest, um es zu erklären". Daraus ergeben sich üblicherweise

- einfache, kurze und prägnante Namen (für Klassen und Methoden)
- einfache Zusammenhänge zwischen Objekten

Dieses Prinzip geht über „selbst dokumentierenden Code" insofern hinaus, dass wir nicht nur fordern, dass der Code die Funktion dokumentiert, sondern auch, dass die Dokumentation so kurz wie möglich ist. Klassen- und Methodennamen mit Satzlänge sind von der Lesbarkeit nicht besser als Kommentare (die man zumindest lesbarer formatieren kann).

Um das Verständnis der gemeinsamen Codebasis zu stärken, ist es zudem sehr wichtig, den Code regelmäßig im Team zu reflektieren. Dazu bieten sich verschiedene Formate an, z. B. Mob Programming oder wöchentliche Quality-Meetings (Probleme thematisieren, Maßnahmen entwickeln). Die Diskussion fördert Unterschiede in den Vorstellungen (Chunks) zutage und ermöglicht die Anpassung der eigenen Chunks und Berücksichtigung der anderen.

Damit solche Runden produktiv verlaufen, sollte eine angemessene Feedbackkultur etabliert werden. Viele Fehler beim Schreiben verständlichen Codes sind nicht vermeidbar, weil die Schreiberin andere Chunks im Gedächtnis hat als die Leser. Es ist wichtig, dass die Unterschiede klar werden, und das geht nur, wenn jeder Entwickler seine Vorstellung auch unbefangen äußern kann.

Was ist mit den Standardwerken zum Thema Code?

Die Standardwerke, wie „Refactoring" (Martin Fowler), „Clean Code" (Robert C. Martin) und „The Art of Readable Code" (Dustin Boswell) vermitteln bewährte Praktiken, um Code verständlich zu halten. Der Anspruch des Gedächtnismodells ist auch nicht, diese Praktiken zu ersetzen oder gar zu widerlegen. Es ermöglicht uns vielmehr, diese Praktiken zu reflektieren und in unbekannten Situationen neue Praktiken zu entwickeln, einfach nur durch eine Analyse der Chunks und ihrer Anzahl. Damit haben wir ein weiteres Mittel, um dem Thema „verständlicher Code" näher zu kommen.

Stefan Mandel ist Softwareentwickler und -architekt bei andrena objects ag in Karlsruhe. Seit seinem Informatikstudium interessiert er sich neben Themen aus dem agilen Software-Engineering für Compiler, Logik und Wissensmodellierung.

Links & Literatur

[1] https://de.wikipedia.org/wiki/Verstehen

[2] https://commons.wikimedia.org/wiki/File:Echinops_telfairi_Plzen_zoo_02.2011.jpg

[3] https://commons.wikimedia.org/wiki/File:Je%C5%BE_Hedgehog_1.JPG

[4] Miller, G. A.: „The magical number seven, plus or minus two: Some limits on our capacity for processing information"; in: Psychological Review 63.2 (1956)

[5] Cowan, N.: „The Magical Mystery Four: How Is Working Memory Capacity Limited, and Why?"; in: Current Directions in Psychological Science 19 (2010)