Grundlagen Softwareentwicklung von Elmar Braun

Zusammenfassung der von Tizian Ester erlernten Notizen

Arbeiten mit KI-Assistenten in der Softwareentwicklung

Kursnotizen, geordnet und erklärt: Kontextsteuerung, Projektregeln, testgetriebene Entwicklung und die Kontrolle über den Agenten.

Dozent: Elmar Braun · Zusammenfassung: Tizian Ester

Stand: 28. August 2026

Inhalt

1 Kontextsteuerung

Das Kontextfenster ist der gesamte Arbeitsspeicher des Modells: System-Anweisungen, Projektregeln, Werkzeugbeschreibungen, gelesene Dateien und der bisherige Gesprächsverlauf. Es ist endlich. Je voller es ist, desto teurer, langsamer und unzuverlässiger arbeitet der Assistent – irrelevante Reste aus einer alten Aufgabe werden zu falschen Annahmen in der neuen. Die folgenden Befehle sind Werkzeuge der Hygiene, nicht der Bequemlichkeit.

/clear – Kontext vollständig löschen

Bedeutung: Der Verlauf wird verworfen. Das Modell startet ohne jede Erinnerung an das bisherige Gespräch; nur die dauerhaft geladenen Regeln (CLAUDE.md) und die Werkzeuge bleiben.

Warum wir es benutzt haben: Bei jedem neuen Arbeitspaket. Ein Assistent, der noch die Diskussion über das Login-Formular im Kopf hat, baut sie unaufgefordert in die Rechnungslogik ein. Sauberer Schnitt statt Themenvermischung – und der Tokenverbrauch fällt auf null zurück.

Preis: Alles Wissen ist weg. Deshalb gehört vor jedes /clear die schriftliche Übergabe aus Kapitel 3.

/compact – Verlauf verdichten

Bedeutung: Der bisherige Verlauf wird durch eine Zusammenfassung ersetzt. Anders als bei /clear bleibt das Ergebnis erhalten, die Details verschwinden.

Warum wir es benutzt haben: Nach mehreren größeren Schritten, wenn das Arbeitspaket noch läuft, der Verlauf aber aufgebläht ist. In den Notizen steht die weitergehende Idee, die KI solle nach jedem Schritt selbst verdichten – also den eigenen Kontext aktiv verwalten, statt bis zur Grenze zu laufen.

Achtung: Verdichtung ist verlustbehaftet. Was in der Zusammenfassung nicht steht, existiert danach nicht mehr. Wichtige Entscheidungen deshalb vorher in eine Datei schreiben, nicht dem Zusammenfasser überlassen.

/context – messen statt raten

Bedeutung: Zeigt, wie sich das Kontextfenster gerade zusammensetzt: System-Prompt, Regeldateien, Werkzeugbeschreibungen, gelesene Dateien, Verlauf, freier Rest.

Warum wir es benutzt haben: Zwei Fragen werden damit beantwortet, die man sonst nur vermutet: Sind meine Projektregeln überhaupt geladen? Und wer frisst den Platz? Ohne diese Messung optimiert man blind.

/effort medium – Denkaufwand dosieren

Bedeutung: Steuert, wie viel interne Überlegung das Modell pro Antwort investiert.

Warum wir es benutzt haben: *medium* war der Standard für gewöhnliche Implementierungsarbeit: gut genug für klar umrissene Aufgaben, ohne Zeit und Kosten für eine Analyse zu verbrennen, die niemand gefordert hat. Die hohe Stufe blieb echten Entwurfs- und Fehlersuchfragen vorbehalten.

Unbenutzte MCP-Server und Agenten abschalten

Bedeutung: Jedes verbundene Werkzeug und jeder registrierte Sub-Agent schreibt seine Beschreibung in den Kontext – noch bevor das erste Wort der Aufgabe gelesen wird.

Warum wir es benutzt haben: Ein Dutzend ungenutzter Server kostet leicht mehrere tausend Token pro Anfrage und vergrößert die Auswahl, aus der das Modell falsch greifen kann. Abschalten spart Geld und reduziert Fehlgriffe.

Aufgabe und Dateien exakt begrenzen

Bedeutung: Der Auftrag benennt das Ziel *und* die Dateien, die dafür angefasst werden dürfen. Eine vollständige Repository-Analyse wird nicht zugelassen.

Warum wir es benutzt haben: Ein unbegrenzter Auftrag liest hunderte Dateien, füllt den Kontext mit Belanglosem und endet in Änderungen, die niemand bestellt hat. Enge Grenzen machen das Ergebnis prüfbar: Ein Diff über drei Dateien lässt sich lesen, einer über sechzig nicht.

2 Projektregeln: CLAUDE.md

CLAUDE.md ist die Datei, die bei jedem Start automatisch in den Kontext geladen wird. Sie ist das Gedächtnis, das ein /clear überlebt. Andere Werkzeuge kennen dasselbe Prinzip unter anderem Namen (AGENTS.md, codex.md) – die Regeln sind dieselben, nur die Datei heißt anders.

Global, pro Projekt, pro Ordner

Bedeutung: Eine globale Datei im Benutzerverzeichnis enthält, wie *man* arbeitet: Sprache, Commit-Stil, Umgang mit Tests. Eine Datei je Projekt enthält, was *dieses* Projekt ausmacht: Stack, Testbefehl, Ordnerstruktur, Verbote. In größeren Repositories liegt zusätzlich in jedem relevanten Unterordner eine eigene Datei.

Warum wir es benutzt haben: Die Notiz „CLAUDE.md in JEDEN Projektordner, spezifisch zuordnen“ zielt genau darauf: Regeln gehören dorthin, wo der Code liegt, für den sie gelten. Eine einzige Regeldatei für ein Monorepo ist entweder zu allgemein, um zu helfen, oder sie trägt Frontend-Regeln in den Backend-Ordner.

Regeln vor der Arbeit festlegen

Bedeutung: Die Regeln werden geschrieben, bevor der Auftrag erteilt wird – nicht als Reaktion auf ein misslungenes Ergebnis.

Warum wir es benutzt haben: Eine Regel, die erst nach dem Schaden entsteht, hat den Schaden nicht verhindert. Und in einer neuen Sitzung ist die mündlich erteilte Ermahnung ohnehin verschwunden; nur was in der Datei steht, wirkt beim nächsten Mal noch.

Die vier Kernregeln aus dem Kurs

Regel	Was sie bedeutet	Warum sie da steht
TypeScript strict	In der tsconfig ist `strict: true` gesetzt: kein implizites `any`, strikte Null-Prüfungen, keine stillschweigenden Typaufweichungen.	Der Compiler prüft, was kein Mensch bei jeder Zeile prüfen will. Generierter Code sieht plausibel aus und ist genau dort schwach, wo Typen unklar sind – strict macht diese Stellen sichtbar, bevor der Code läuft.
Vor Implementierung Tests schreiben	Zuerst entsteht der Test, der das gewünschte Verhalten beschreibt und fehlschlägt. Erst danach der Code, der ihn erfüllt.	Der Test ist die einzige Formulierung der Anforderung, die nicht interpretierbar ist. Wer zuerst implementiert, schreibt hinterher den Test, der zufällig zum Code passt – und prüft damit nichts.
Keine API-Änderung ohne Anpassung der Tests	Ändert sich eine Signatur, ein Endpunkt oder ein Rückgabeformat, gehören die betroffenen Tests im selben Schritt angepasst.	Sonst bleibt eine grüne Testsuite zurück, die die alte Welt prüft. Die Kopplung erzwingt außerdem, dass eine Änderung an der Schnittstelle bewusst getroffen und nicht nebenbei eingeschleppt wird.
Nur angeforderte Dateien verändern	Der Assistent fasst genau die Dateien an, die im Auftrag stehen. Alles andere wird gemeldet, nicht geändert.	Verhindert stille Umformatierungen, spontane Refactorings und Änderungen an fremden Modulen. Ein enger Diff ist überprüfbar; ein breiter wird durchgewinkt.

3 Die Übergabe an einen neuen Chat

/clear und /compact vernichten Kontext – das ist ihr Zweck. Die Übergabe ist die bewusste Entscheidung darüber, was diesen Schnitt überleben soll. Sie wird am Ende eines Arbeitspakets erzeugt und zu Beginn der nächsten Sitzung eingefügt.

```
Erstelle eine Uebergabe fuer einen neuen Chat.

Enthalten:
  - aktueller Stand
  - getroffene Entscheidungen
  - geaenderte Dateien
  - offene Fehler
  - naechster konkreter Schritt

Maximal 500 Woerter. Keine Gespraechshistorie.
```

Feld	Warum genau dieses Feld
Aktueller Stand	Der Nachfolger muss wissen, was fertig ist und was nur begonnen wurde. Ohne das beginnt er entweder von vorn oder baut auf einer Hälfte auf.
Getroffene Entscheidungen	Das teuerste Wissen der Sitzung. Warum diese Bibliothek, warum dieses Datenmodell, welcher Weg wurde verworfen und weshalb. Fehlt es, wird die verworfene Variante fröhlich erneut vorgeschlagen.
Geänderte Dateien	Grenzt den Arbeitsbereich sofort ein und macht das nächste Review möglich, ohne das ganze Projekt zu lesen.
Offene Fehler	Bekannte Fehlschläge müssen explizit stehen. Sonst gilt der rote Test in der neuen Sitzung als „Altlast“ und wird stillschweigend übergangen.
Nächster konkreter Schritt	Eine ausführbare Handlung, kein Themengebiet. „Test x in y zum Laufen bringen“ statt „Fehlerbehandlung verbessern“.

Die beiden Nebenbedingungen

Maximal 500 Wörter: Eine Grenze erzwingt Priorisierung. Eine ungekürzte Zusammenfassung ist wieder ein Kontextproblem – und niemand liest sie.

Keine Gesprächshistorie: Wie man zum Ergebnis kam, ist Rauschen. Es zählt der Zustand, nicht der Weg. Historie im Übergabetext verleitet den Nachfolger dazu, alte Sackgassen erneut zu betreten.

4 Arbeitsablauf: die Acht-Punkte-Checkliste

Die Reihenfolge, in der die Werkzeuge aus Kapitel 1 im Alltag angewendet werden. Die Liste ist bewusst kurz genug, um sie vor jedem Arbeitspaket durchzugehen.

#	Schritt	Zweck
1	**/clear bei einem neuen Arbeitspaket**	Sauberer Schnitt zwischen Aufgaben. Keine Reste aus dem vorherigen Thema, keine Altlast im Tokenbudget.
2	**/effort medium**	Bewusste Dosierung des Denkaufwands. Standard für normale Umsetzungsarbeit, höhere Stufe nur bei echten Analysefragen.
3	**Aufgabe und betroffene Dateien exakt begrenzen**	Macht das Ergebnis überprüfbar und den Diff lesbar. Der Auftrag nennt Ziel und erlaubten Änderungsbereich.
4	**Keine vollständige Repository-Analyse zulassen**	Verhindert, dass der Kontext mit irrelevanten Dateien geflutet wird, bevor die eigentliche Arbeit beginnt.
5	**Nach mehreren größeren Schritten /compact**	Hält den laufenden Auftrag am Leben, ohne den gesamten Verlauf mitzuschleppen.
6	**/context prüfen**	Kontrolle: Sind die Regeln geladen? Wieviel Platz ist noch frei? Wer verbraucht ihn?
7	**Unbenutzte MCP-Server und Agenten abschalten**	Weniger Grundlast pro Anfrage, weniger falsche Werkzeugwahl.
8	**Jeder Bugfix beginnt mit einem fehlschlagenden Unit-Test**	Der rote Test beweist, dass der Fehler verstanden und reproduzierbar ist. Kein Fix ohne Regressionstest, der ihn dauerhaft absichert.

Punkt 8 verdient eine eigene Bemerkung

Die Reihenfolge **rot – grün – abgesichert** ist keine Förmlichkeit. Ein Test, der vor dem Fix fehlschlägt und danach besteht, belegt zweierlei: Der Fehler existierte wirklich, und diese Änderung hat ihn behoben. Ein Test, der erst nach dem Fix geschrieben wird, kann beides nicht zeigen – er könnte von Anfang an grün gewesen sein. Und weil der Test im Projekt bleibt, kann derselbe Fehler nicht unbemerkt zurückkehren.

5 Beweispflicht: Erklärung ist kein Beleg

> **„Eine glaubwürdige Erklärung ist kein Beweis. Ein ausgeführter, bestandener Test ist ein Beleg.“**

Der Kern des gesamten Kurses. Sprachmodelle sind darauf optimiert, überzeugende Texte zu erzeugen – nicht darauf, wahre Aussagen über den Zustand Ihres Programms zu machen. „Ich habe die Funktion angepasst, sie behandelt jetzt auch den Grenzfall“ ist eine Behauptung mit der Oberfläche eines Belegs. Erst die Ausgabe eines tatsächlich gelaufenen Testlaufs ist eine Tatsache. Zwischen beidem liegt der gesamte Unterschied zwischen Vertrauen und Kontrolle.

Der Prüfauftrag gegen die Selbstbestätigung

```
„Ich möchte, dass du unser Projekt auf Vollständigkeit prüfst, mir mindestens drei Schwachstellen nennst und nach blind spots suchst.“
```

Warum diese Formulierung: Die Frage „passt das so?“ erzeugt Zustimmung – das Modell folgt der Erwartung im Auftrag. Eine *Mindestanzahl* zu fordern dreht die Richtung um: Jetzt muss gesucht werden, Zustimmung ist keine gültige Antwort mehr. Der Zusatz „blind spots“ zielt auf die gefährlichere Kategorie – nicht auf Fehler im vorhandenen Code, sondern auf das, was gar nicht erst bedacht oder getestet wurde.

Und dann: Auch die Antwort auf diesen Auftrag ist zunächst nur eine Behauptung. Jede genannte Schwachstelle wird zu einem Testfall, oder sie zählt nicht.

Woran man eine Behauptung erkennt

Behauptung: „Der Fehler ist behoben.“ · „Die Tests laufen durch.“ · „Das ist typsicher.“ · „Ich habe alle Aufrufer angepasst.“

Beleg: Die tatsächliche Ausgabe des Testlaufs. Der Compiler ohne Fehler. Der Diff, den Sie gelesen haben. Der Testfall, der vorher rot war und jetzt grün ist.

Praxisregel: Kein Arbeitspaket gilt als abgeschlossen, solange die Belege nur aus Prosa bestehen.

6 Testgetriebene Entwicklung und die Werkzeuge

Test-Driven Development ist die praktische Umsetzung von Kapitel 5: Der Test wird zuerst geschrieben, schlägt fehl, und erst der Code macht ihn grün. Damit ist die Anforderung maschinell prüfbar, bevor irgendjemand – Mensch oder Assistent – anfängt zu implementieren.

Werkzeug	Einsatzbereich	Kurzcharakter
Cypress	Web, End-to-End und Komponententests	Läuft im Browser neben der Anwendung, sehr guter interaktiver Testlauf mit Zeitreise durch die Schritte. Stark beim Entwickeln und Debuggen einzelner Abläufe.
Playwright	Web, End-to-End über mehrere Browser	Steuert Chromium, Firefox und WebKit, wartet automatisch auf Elemente, läuft gut parallel in der CI. Die robustere Wahl für große Suiten.
Maestro	Mobile UI-Tests, iOS und Android	Testabläufe werden als kurze YAML-Dateien beschrieben statt als Programmcode. Sehr niedrige Einstiegshürde, toleriert Verzögerungen der App von Haus aus.
Detox	React Native	Grey-Box-Ansatz: kennt den inneren Zustand der App und wartet, bis diese wirklich fertig ist, statt feste Wartezeiten zu setzen. Damit deutlich weniger sprunghafte Fehlschläge.

Der Vier-Fragen-Rahmen für jeden Testfall

Aus der Kursnotiz zur Maestro-Strategie stammt ein Schema, das für jeden Testfall gilt – unabhängig vom Werkzeug. Ein Testauftrag an den Assistenten muss diese vier Punkte beantworten, sonst entsteht ein Test, der nichts entscheidet.

Frage	Bedeutung	Ohne diese Angabe passiert
1. Welches Verhalten wird erwartet?	Die Anforderung in einem Satz, formuliert als beobachtbares Verhalten der Anwendung.	Der Test prüft, was der Code tut, statt was er tun soll.
2. Bei welcher Eingabe tritt der Fehler auf?	Der konkrete, reproduzierbare Weg dorthin: Startzustand, Eingaben, Schrittfolge.	Der Fehler ist nicht reproduzierbar, der Test damit nicht schreibbar.
3. Welches Ergebnis ist falsch?	Die beobachtete Fehlwirkung, genau benannt – nicht „es geht nicht“.	Der Test wird grün, obwohl das Problem in anderer Form fortbesteht.
4. Wann gilt die Korrektur als erfolgreich?	Das Abnahmekriterium: welche Zusicherung genau bestehen muss.	Die Fertigstellung ist Verhandlungssache statt Messergebnis.

Gerüst eines Maestro-Ablaufs

```
# 01_login_leeres_passwort.yaml
# 1) Erwartung: Bei leerem Passwort erscheint eine Fehlermeldung,
#               der Nutzer bleibt auf dem Login-Bildschirm.
# 2) Eingabe:  gueltige E-Mail, Passwortfeld leer, absenden.
# 3) Falsch:   App navigiert zur Startseite (Fehlerfall durchgelassen).
# 4) Erfolg:   Dieser Ablauf laeuft durch; vor dem Fix schlaegt er fehl.

appId: com.example.app
---
- launchApp:
    clearState: true
- tapOn: { id: "email_input" }
- inputText: "test@example.com"
- tapOn: { id: "submit_button" }
- assertVisible: "Bitte Passwort eingeben"
- assertNotVisible: { id: "home_screen" }
```

Warum der Kommentarkopf zum Test gehört

Die vier Antworten stehen direkt in der Testdatei. Damit trägt der Test seine eigene Begründung: Wer ihn in einem Jahr rot vorfindet, weiß, welches Verhalten hier verteidigt wird – und kann nicht behaupten, der Test sei „veraltet“, ohne die Anforderung selbst zu widerlegen.

7 Die zehn Grundregeln

Das Grundgerüst aus dem Kurs. Die Punkte 1 und 2 legen fest, woher der Maßstab kommt; 3 bis 6 schützen ihn vor Aufweichung; 7 bis 10 regeln, wer kontrolliert und was in den Projektstand darf.

#	Regel	Bedeutung und Begründung
1	**Anforderungen werden vor der Implementierung festgelegt.**	Ohne vorher festgeschriebene Anforderung gibt es keinen Maßstab – dann wird zum Maßstab, was zufällig entstanden ist.
2	**Tests werden aus den Anforderungen abgeleitet.**	Der Test ist die maschinenlesbare Fassung der Anforderung. Wird er stattdessen aus dem fertigen Code abgeleitet, prüft er nur, dass der Code so ist, wie er ist.
3	**KI-Code wird nicht aufgrund seiner Formulierung akzeptiert.**	Flüssige Erklärungen und saubere Benennungen sind keine Korrektheitsnachweise. Bewertet wird das Verhalten, nicht der Begleittext.
4	**Tests werden tatsächlich ausgeführt.**	„Der Test würde bestehen“ ist keine Aussage über die Wirklichkeit. Nur ein realer Lauf mit sichtbarer Ausgabe zählt.
5	**Fehlgeschlagene Tests werden nicht gelöscht oder abgeschwächt.**	Ein roter Test ist Information. Wer die Zusicherung lockert oder den Test überspringt, entfernt die Information und behält den Fehler.
6	**Eine KI darf Anforderungen nicht eigenmächtig verändern.**	Kommt der Assistent nicht ans Ziel, ist das zu melden – nicht durch eine bequemere Auslegung des Ziels zu lösen. Andernfalls verschiebt sich das Projekt unbemerkt.
7	**Änderungen bleiben klein und nachvollziehbar.**	Kleine Schritte sind lesbar, testbar und einzeln zurücknehmbar. Bei einem großen Wurf ist unklar, welcher Teil das Problem verursacht hat.
8	**Ein Review-Agent prüft die Arbeit des Entwicklungs-Agenten.**	Trennung der Rollen. Wer geschrieben hat, findet die eigenen Auslassungen schlecht – ein zweiter Durchgang mit ausschließlichem Prüfauftrag findet mehr.
9	**Der Mensch kontrolliert die Agenten.**	Die Entscheidung über Anforderung, Abnahme und Übernahme bleibt beim Menschen. Agenten arbeiten zu, sie entscheiden nicht.
10	**Nur getesteter Code wird in den gültigen Projektstand übernommen.**	Die letzte Schranke. Sie hält alle vorherigen Regeln zusammen: Was sie nicht passiert hat, existiert für das Projekt nicht.

Zusammengefasst: Kontext klein halten, Regeln vor der Arbeit schreiben, den Auftrag eng fassen, jede Behauptung durch einen ausgeführten Test ersetzen – und die Entscheidung darüber, was in das Projekt übernommen wird, beim Menschen belassen.