Table of Contents

Claude Code – Engineering Guide: Neue Atomium Krankenkasse (NAK) 4
Zweck dieses Dokuments 4
1. Projektüberblick 4
Hauptbereiche 4
Navigation 5
2. Technischer Stack 5
Lokale Frameworks 5
3. Architektur 5
3.1 Composer-Pattern 6
3.2 DI-Flow 6
3.3 Modulstruktur 6
3.4 Live/Mock-Architektur 7
3.5 Navigation 8
4. Claude Code Setup 9
4.0 Wie Claude Code funktioniert – das Wichtigste zuerst 9
4.1 Installation 12
4.2 Projektstruktur für Claude Code 12
4.3 settings.json 14
4.4 Eingebaute Slash-Commands 15
4.5 CLI-Flags beim Start 17
4.6 Keyboard Shortcuts 17
4.7 Fortgeschrittenes 18
4.8 Best Practices – effizient und kostengünstig arbeiten 19
4.9 Memory – was Claude Code zwischen Sessions merkt 20
4.10 Konfigurationsdateien im Projektalltag pflegen 22
4.11 Claude Code fortlaufend verbessern – bessere Ergebnisse mit der Zeit 22
5. CLAUDE.md – Projektgedächtnis 24
5.1 Was CLAUDE.md ist und warum sie die wichtigste Datei ist 24
5.2 CLAUDE.md ist nicht einmalig – sie existiert auf mehreren Ebenen 24
5.3 Was auf welcher Ebene steht 25
5.4 Wie man CLAUDE.md erstellt und bearbeitet 25
5.5 Beispiel: Root CLAUDE.md für Atomium 26
Beispiel: Feature-spezifische CLAUDE.md 27
6. Skills – Wiederverwendbare Anweisungen 27
6.1 Was ein Skill ist – und was nicht 27

6.2 Wie ein Skill erstellt wird – Schritt für Schritt 28
6.3 Wie ein Skill benannt wird 28
6.4 Wie Claude Code einen Skill verwendet 28
6.5 Format eines Skills 29
6.6 Die Skills für Atomium 29
6.7 Kurzübersicht: Skills anlegen 32
7. Rules – Harte Projektregeln 33
7.1 Was Rules sind und warum sie existieren 33
7.2 Wie Rules erstellt werden 33
7.3 Unterschied Rules vs. Skills: wo was hingehört 33
7.4 Wie Claude Code die Rules kennt 34
7.5 Die Rules für Atomium 34
7.6 Kurzübersicht: Rules anlegen 36
8. Commands – Anlegen, Verstehen und für die Qualitätssicherung einsetzen 37
11.0 Was sind Commands – und warum brauchen wir sie? 37
11.1 Wie Commands technisch funktionieren 37
11.2 Command anlegen – so geht es 37
11.3 Alle Atomium-Commands – Dateiinhalt und Verwendung 38
11.4 Kurzreferenz – alle Commands 46
9. Hooks – Automatische Aktionen nach Claude-Code-Schritten 47
18.1 Was sind Hooks? 47
18.2 Wann werden Hooks ausgelöst? 47
18.3 Hooks konfigurieren 47
18.4 Nützliche Hooks für Atomium 48
18.5 Vollständiges settings.json mit allen Atomium-Hooks 49
18.6 Was Hooks nicht können 50
10. Auto-Memory – Claude Code merkt sich dein Projekt 50
19.1 Was ist Auto-Memory? 50
19.2 Welche Informationen speichert Auto-Memory? 50
19.3 Wie funktioniert das – Schritt für Schritt 51
19.4 Wo liegen die Dateien? 52
19.5 Was Auto-Memory für Atomium konkret bringt 52
19.6 Was Claude Code sich nicht merkt 52
11. Agents – Spezialisierte Sub-Instanzen 52
20.1 Was sind Agents? 53
20.2 Wann lohnen sich Agents im Atomium-Projekt? 53
20.3 Wie du Agents einsetzt 53

20.4 Was Agents nicht können 54
20.5 Agents vs. Commands vs. Skills – der Unterschied 54
12. Neues Feature entwickeln 55
8.1 Wie der Workflow funktioniert 55
8.2 Beispiel: Feature „Bescheinigungsdownload" 55
8.3 Zusammenfassung: Was du tust, was Claude Code tut 60
13. Bestehendes Feature erweitern 60
9.1 Erweiterung: Neue Seite in bestehendem Feature 61
9.2 Legacy-Code: Was tun wenn du auf ATLegacy stößt? 62
9.3 Refactoring: Legacy-Klasse migrieren 62
9.4 Zusammenfassung: Was du tust, was Claude Code tut 63
14. UnitTest-Strategie 63
10.1 Grundlagen – Was getestet wird, was nicht, und warum 63
10.2 Das Grundgerüst – jeder Test sieht so aus 64
10.3 Repository-Tests – was sie prüfen und warum 66
10.4 ViewModel-Tests – was sie prüfen und warum 67
10.5 Mapper-Tests – die einfachsten Tests 69
10.6 Fixtures – warum und wie 70
10.7 Tests mit Claude Code erstellen 71
10.8 Zusammenfassung: Was du tust, was Claude Code tut 73
15. Qualitätssicherung im Alltag 73
12.1 Vor einem Merge-Request 73
12.2 Während der Feature-Entwicklung 73
12.3 Pre-Merge-Workflow – Schritt für Schritt 74
12.4 Commands anlegen – einmalige Einrichtung 74
16. Dokumentationsstandard 74
13.1 Was wird dokumentiert – und was nicht? 75
13.2 Wie sieht korrekter Apple-Stil aus? 75
13.3 Wie Claude Code Dokumentation erzeugt 76
17. Code-Stil und Formatierung 76
14.1 Werkzeuge und wer sie ausführt 77
14.2 Pflicht-Struktur jeder Swift-Datei 77
14.3 Code-Stil mit Claude Code prüfen 78
18. Automatisierung 78
15.1 CI-Pipeline – automatische Prüfungen bei jedem Pull Request 79
15.2 Wiederkehrende Claude-Code-Workflows 79
19. Debug-Architektur 80

16.1 Wo das Debug-Menü liegt und was es enthält ... 80
16.2 Neues Debug-Feature hinzufügen – mit Claude Code ... 81
16.3 Welche Debug-Komponenten existieren ... 81
16.4 FakeLogin – was es ist und wie man es nutzt ... 82
20. Onboarding neuer Entwickler ... 82
Tag 1 – Setup und erstes Gespräch mit Claude Code ... 82
Tag 2 – Ein bestehendes Feature vollständig verstehen ... 83
Tag 3 – Erste eigene kleine Änderung ... 83
Was Claude Code im Alltag ersetzt – und was nicht ... 84
Fazit ... 84

Claude Code – Engineering Guide: Neue Atomium Krankenkasse (NAK)

Zweck dieses Dokuments

Claude Code am praktischen Beispiel – das ist der Ansatz dieser Anleitung. Sie zeigt anhand von Atomium, einer fiktiven iOS-App der NAK, wie sich ein LLM sinnvoll in einen iOS-Workflow einbinden lässt. Kein Konzept ohne konkreten Code, kein Pattern ohne Anwendungsfall.

---

1. Projektüberblick

Atomium ist eine fiktive iOS-App, gebaut wie eine echte Krankenkassen-App – mit SwiftUI, modularer Architektur und Composer-Pattern. Sie dient als durchgängiges Praxisbeispiel für diese Anleitung. Alle Funktionen, Namen und Abläufe sind frei erfunden.

Hauptbereiche

Jeder Bereich ist ein eigenständiges Feature-Modul in Atom/Features/:

Bereich	Beschreibung
IAM-Login	Authentifizierung, Session-Management, Biometrie
Dashboard	Kachel-basiertes Home-Screen-Layout
Bonusprogramm	Punkte, Aktionen, Einlösungen
Postfach	Nachrichten, Dokumente, Lesebestätigung
Serviceleistungen	Erstattungen, Anträge, Formulare
Antragserstellung	Multi-Step-Flows, Uploads
Meine Daten	Persönliche Daten, Datenschutzeinstellungen
Einstellungen	Benachrichtigungen, Sprache, Sicherheit
Profilwechsel	Familienmitglieder wechseln, Kinderprofile

Navigation

TabBar-Hauptnavigation mit fünf Tabs, Profil rechts oben:

```
TabBar:
├── Home          (Dashboard + Kacheln)
├── Service       (Leistungen, Anträge, Kontakt)
├── Gesundheit    (Themen, Bescheinigungen)
├── Bonusprogramm (Punkte, Aktionen)
└── Postfach      (Nachrichten, Dokumente)

Navigation Top-Right:
└── Profil
    ├── Einstellungen
    ├── Meine Daten
    └── Profilwechsel
```

---

2. Technischer Stack

Damit Claude Code versionsspezifisch und korrekt generiert, muss es den Stack kennen. Die relevanten Eckdaten:

```
Sprache:      Swift 5.9+
UI-Framework: SwiftUI
Reaktiv:      Combine + Async/Await (gemischt, Übergangsphase)
Tests:        XCTest + Custom Test Utilities
Min. iOS:     16.0
```

Lokale Frameworks

Vier lokale Swift-Frameworks. Wichtigste Regel: neuer Code nur in Atom.

Framework	Rolle
ATCore	Übergreifende Infrastruktur: Netzwerk, Logging, Security, Persistence, Extensions
ATUI	Design-System: Farben, Typografie, Komponenten, Icons
ATLegacy	Legacy-Architektur (MVC/MVVM-Mischung). Kein neuer Code hier.
Atom	Neue Architektur, alle neuen Features, Composer-Pattern

Regel: Neuer Code entsteht ausschließlich in Atom. Wer in ATLegacy schreibt, braucht eine explizite Begründung im PR.

---

3. Architektur

Bevor Claude Code für neue Features eingesetzt wird, muss klar sein wie das Projekt aufgebaut ist. Composer-Pattern, DI-Flow, Feature-Modulstruktur, Live/Mock-Trennung – Claude Code kennt diese Strukturen nur, wenn sie hier stehen. Das ist kein optionales Hintergrundwissen.

3.1 Composer-Pattern

Das Composer-Pattern ist die zentrale Architekturentscheidung in Atom. Es ersetzt klassische Coordinator- und Factory-Ansätze.

Ein Composer hat genau eine Aufgabe: Er verbindet alle Abhängigkeiten eines Features und gibt eine View zurück. Er kennt den Kontext, die Navigation und die Umgebung. Die View weiß nichts davon.

```swift
// MyFeature/MyFeatureComposer.swift
enum MyFeatureComposer {
  static func compose() -> some View {
    WithContext {
      try MyFeatureEnvironment()
    } content: { environment in
      MyFeatureView(
        loadData: environment.loadData,
        submitAction: environment.submit
      )
    }
  }
}
```

Warum enum? Kein Instanzzustand, kein versehentliches Retain, klare statische Factory-Semantik.

Was WithContext macht: Löst die Environment auf, fängt Initialisierungsfehler ab und stellt den Scope sicher.

3.2 DI-Flow

Der DI-Flow zeigt wie Abhängigkeiten von der Root bis in ein einzelnes Feature fließen. Claude Code muss diese Hierarchie kennen um einen neuen Composer korrekt anzubinden – in die richtige Ebene, ohne Seiteneffekte auf andere Features.

```
RootFlowComposer
└── AtomAppFeatureComposer
  └── FeatureComposer (z.B. BonusProgramComposer)
    ├── Environment      (Abhängigkeiten live/mock)
    ├── ViewModel       (@MainActor, @Observable)
    ├── Repository      (async throws Protokoll)
    ├── View          (dumm, actions as closures)
    └── Navigator       (sheet/push/fullScreenCover)
```

RootFlowComposer kennt den App-State, Session, Auth-Status.

AtomAppFeatureComposer koordiniert die TabBar und den globalen Navigationsbaum.

FeatureComposer ist in sich geschlossen. Er testet sich selbst.

3.3 Modulstruktur

Jedes Feature folgt dieser Ordnerstruktur:

```
MyFeature/
├── Domain/
│  ├── MyFeatureModel.swift      // Fachliches Modell, kein API-Mapping
│  └── MyFeatureError.swift
```

```
├── Repository/
│   ├── MyFeatureRepository.swift    // Protokoll
│   ├── MyFeatureRepositoryLive.swift // Echte Implementierung
│   └── MyFeatureRepositoryMock.swift // Mock für Tests und Previews
├── API/
│   ├── MyFeatureAPIClient.swift     // URLSession, Codable
│   └── MyFeatureAPIModel.swift      // API-Modelle (Response/Request)
├── Mapper/
│   └── MyFeatureMapper.swift        // API → Domain
├── Interactor/
│   └── MyFeatureInteractor.swift    // Business-Logik, kombiniert Repos
├── UI/
│   ├── MyFeatureComposer.swift
│   ├── MyFeatureEnvironment.swift
│   ├── MyFeatureView.swift
│   ├── MyFeatureViewModel.swift
│   └── MyFeatureNavigator.swift
└── Tests/
    ├── MyFeatureRepositoryTests.swift
    ├── MyFeatureViewModelTests.swift
    └── MyFeatureMapperTests.swift
```

MyFeature vs. MyFeatureUI: Wenn ein Feature in mehrere Targets eingebunden wird, trennt man Domain-Logik (MyFeature) von der View-Schicht (MyFeatureUI). Damit bleibt das Domain-Framework frei von SwiftUI-Imports und testbar ohne UI-Host.

3.4 Live/Mock-Architektur

Jedes Repository hat genau drei Implementierungen: ein Protokoll, eine Live-Klasse (echte Netzwerkanbindung) und einen Mock (für Tests und SwiftUI-Previews). Claude Code generiert immer alle drei – nie nur das Protokoll oder nur die Live-Implementierung.

```swift
// Repository-Protokoll
protocol BonusProgramRepository {
  func fetchPoints() async throws -> BonusPoints
  func redeemReward(_ reward: Reward) async throws
}

// Live-Implementierung
struct BonusProgramRepositoryLive: BonusProgramRepository {
  private let apiClient: BonusProgramAPIClient

  func fetchPoints() async throws -> BonusPoints {
    let response = try await apiClient.getPoints()
    return BonusProgramMapper.map(response)
  }
}

// Mock-Implementierung
struct BonusProgramRepositoryMock: BonusProgramRepository {
```

```swift
  var fetchPointsResult: Result<BonusPoints, Error> = .success(.preview)

  func fetchPoints() async throws -> BonusPoints {
    try fetchPointsResult.get()
  }
}
```

FakeLogin: Für Development Builds existiert ein FakeLoginComposer, der den IAM-Flow überspringt und direkt in ein Testprofil einloggt. Er ist mit #if DEBUG abgesichert und darf nie in Release-Builds landen.

```swift
#if DEBUG
enum FakeLoginComposer {
  static func compose(profile: FakeProfile = .standard) -> some View {
    FakeLoginView(profile: profile, onLogin: { /* inject session */ })
  }
}
#endif
```

3.5 Navigation

Navigation läuft über ATUIFlow – ein eigener Wrapper um SwiftUI's NavigationStack und Modals.

```swift
// Navigator definiert die Destinations
enum MyFeatureDestination: Hashable {
  case detail(id: String)
  case edit(model: MyModel)
}

// Navigator-Protokoll
protocol MyFeatureNavigator {
  func navigateToDetail(id: String)
  func presentEdit(model: MyModel)
  func dismiss()
}

// In der View
struct MyFeatureView: View {
  let onDetailTap: (String) -> Void
  // ...
}
```

sheet vs. fullScreenCover: Modals, die Kontext behalten (z.B. Filterdialoge) → sheet. Eigenständige Flows (z.B. Antragserstellung) → fullScreenCover.

Kein NavigationLink direkt in Views. Navigation-Entscheidungen gehören in den Composer oder Navigator, nie in die View selbst.

---

4. Claude Code Setup

Alles was du brauchst, bevor du das erste Mal claude eintippst: wie Claude Code funktioniert, wie die Installation läuft, welche Projektstruktur angelegt wird, welche Befehle es gibt. Reihenfolge ist wichtig – von Grundlagen zu Details.

4.0 Wie Claude Code funktioniert – das Wichtigste zuerst

Drei Punkte vorab – wer sie kennt, hat mit dem Rest kein Problem.

---

1. Es gibt zwei Orte wo du etwas eintippst – Terminal und Chat. Sie sind verschieden.

```
Terminal (macOS)       → hier startest du Claude Code, erstellst Dateien, führst Builds aus
Claude Code Chat       → hier stellst du Fragen, gibst Prompts ein, rufst Commands auf
```

Wo hier Öffne ein Terminal steht: macOS-Terminal (Terminal.app oder iTerm). Wo Im Claude Code Chat eingeben steht: das Textfeld innerhalb der laufenden Claude-Code-Session.

---

2. Was sofort verfügbar ist – und wann ein Neustart nötig ist.

Claude Code liest Konfigurationsdateien **beim Start einer Session**. Das bedeutet:

Aktion	Wann wirksam
CLAUDE.md erstellt oder geändert	Beim nächsten Start von Claude Code
.claude/commands/ Datei erstellt	Beim nächsten Start – dann per Tab-Completion verfügbar
.claude/skills/ Datei erstellt	Sofort – diese Dateien werden auf Anfrage gelesen, kein Neustart nötig
.claude/rules/ Datei erstellt	Beim nächsten Start (wenn in CLAUDE.md referenziert)
settings.json geändert	Beim nächsten Start

Neustart: Im Terminal Ctrl+C drücken, dann claude erneut eingeben.

Schneller Weg in einer laufenden Session: Im Claude Code Chat /clear tippen. Löscht den Konversationsverlauf und lädt CLAUDE.md neu – ohne das Terminal zu schließen.

---

3. Commands mit / werden im Chat eingegeben – nicht im Terminal.

```
Im Terminal:    claude              ← startet Claude Code
Im Terminal:    touch .claude/commands/check-async.md    ← erstellt Datei

Im Chat:      /check-async Atom/Features/BonusProgram  ← ruft Command auf
Im Chat:      /clear                         ← setzt Session zurück
Im Chat:      Erstelle eine Datei unter .claude/...   ← Claude Code schreibt die Datei
```

Alles was mit / beginnt und als Command aufgeführt ist, wird **im Chat-Fenster** eingegeben.

---

4. Desktop-App vs. Terminal – was ist der Unterschied?

Claude Code gibt es in zwei Varianten: als **Terminal-CLI** (diese Anleitung) und als **Desktop-App** (separates Programm für macOS und Windows). Beide sprechen dasselbe Modell an – aber sie unterscheiden sich in Bedienung und Funktionsumfang.

	Terminal-CLI	Desktop-App
Start	claude im Terminal eintippen	App öffnen, Projekt wählen
Dateizugriff	Arbeitet im aktuellen Verzeichnis	Projektordner per GUI wählen
Commands	/command tippen, Tab-Completion	Seitenleiste mit Command-Liste
CLAUDE.md, Skills, Rules	Werden automatisch aus .claude/ geladen	Werden genauso geladen – kein Unterschied
Hooks	Laufen im Terminal-Kontext	Laufen im App-Kontext
Syntax-Highlighting	Nein (reiner Text)	Ja – Code wird farbig angezeigt
Mehrere Projekte	Ein Terminal-Fenster = ein Projekt	Tabs für mehrere Projekte
Diff-Ansicht	Kein eingebauter Diff	Änderungen werden visuell markiert
Für Einsteiger	Weniger intuitiv	Einfacher einzusteigen
Für Profis	Schneller, scriptbar, in CI nutzbar	Komfortabler im Alltag

Einsteiger: Desktop-App. Sie zeigt was Claude Code macht, markiert Änderungen visuell, kein Terminal-Wissen nötig.

Projektalltag: Terminal-CLI. Mächtiger, in Shell-Scripts einbindbar, läuft in CI-Pipelines. Diese Anleitung nutzt die CLI.

Skills, Rules, Commands in der Desktop-App: Dieselben Dateien unter .claude/. Die App liest CLAUDE.md und .claude/commands/ automatisch – /check-async funktioniert im Chat der Desktop-App genauso wie im Terminal.

Download: claude.ai/download → „Claude Code Desktop" wählen.

---

5. Abonnement & Kosten – was kostet Claude Code?

Claude Code hat zwei Zugangswege – und der Unterschied kann teuer werden wenn man ihn nicht kennt.

Option A: Abonnement (empfohlen für Entwickler)

Plan	Preis	Was du bekommst
Pro	~20 USD/Monat	Claude Code mit Nutzungslimit – gut für gelegentliche Nutzung
Max	~100 USD/Monat	Höheres Limit, beste Modelle, für intensive tägliche Nutzung

Mit Abonnement zahlst du einen festen Monatsbetrag. Nutzungslimit erreicht? Warten bis zum nächsten Tag – keine Überraschungsrechnung.

Option B: API-Zugang (pay-per-use)

Du zahlst pro Token, kein festes Limit, keine Wartezeit – aber die Kosten steigen mit der Nutzung.

Aktuelle API-Preise (ungefähr, Stand 2026 – unter anthropic.com/pricing nachschlagen):

Modell	Eingabe	Ausgabe	Wann einsetzen
Claude Haiku	günstig	günstig	Schnelle Checks, einfache Aufgaben

Modell	Eingabe	Ausgabe	Wann einsetzen
Claude Sonnet	mittel	mittel	Tägliche Arbeit, Code-Generierung
Claude Opus	teuer	teuer	Komplexe Architektur, tiefes Reasoning

Was ist besser für Atomium?

Für ein Team das täglich mit Claude Code arbeitet: **Max-Abonnement**. Planbare Kosten, kein Token-Zählen, beste Modelle verfügbar.

Für Einzelentwickler die gelegentlich testen: **Pro-Abonnement** als Einstieg.

API direkt lohnt sich für automatisierte Workflows oder CI – dort ist pay-per-use oft günstiger als ein Pauschalabo.

---

6. Welches Modell wann – Haiku, Sonnet oder Opus?

Claude Code wählt automatisch ein Modell – du kannst gezielt wechseln. Je komplexer die Aufgabe, desto besser das Modell, aber auch teurer.

Modell	Stärken	Typische Aufgaben im Atomium-Projekt
Haiku	Schnell, günstig	Kurze Erklärungen, einfache Umbenennungen, schnelle Suche
Sonnet	Ausgewogen (Standard)	Feature-Entwicklung, Tests schreiben, Code-Reviews
Opus	Tiefes Reasoning, beste Qualität	Architektur-Entscheidungen, DSGVO-Analysen, Refactoring-Pläne

```
Im Chat: /model haiku     ← für schnelle einfache Aufgaben
Im Chat: /model sonnet    ← Standard (empfohlen für den Alltag)
Im Chat: /model opus      ← für komplexe Analysen
```

Praktischer Tipp: Starte mit Sonnet. Wenn die Antwort nicht gut genug ist, wechsle auf Opus. Für reine Suchaufgaben oder schnelle Erklärungen reicht Haiku.

---

7. Plan-Modus – erst planen, dann handeln

Bei großen Aufgaben – neues Feature, Refactoring, Migration – Claude Code **nicht sofort loslegen lassen**. Plan-Modus: Claude Code analysiert, erstellt einen Plan, wartet auf Freigabe bevor es Code schreibt.

So aktivierst du den Plan-Modus:

```
Im Chat: /plan Implementiere das Certificate-Feature komplett mit Composer, Repository, Tests
```

Oder mit dem Keyword in deinem Prompt:

```
Erstelle einen Plan für das neue Bonusprogramm-Feature.
Noch kein Code – erst den Plan zeigen, ich prüfe ihn und gebe dann frei.
```

Der Ablauf:

1. Du gibst die Aufgabe mit Planungsanweisung ein
2. Claude Code analysiert das Projekt und erstellt einen Schritt-für-Schritt-Plan
3. Du liest den Plan, korrigierst ihn bei Bedarf
4. Du gibst frei: „Plan sieht gut aus, jetzt umsetzen"

5. Claude Code schreibt den Code

Warum das wichtig ist:

Ohne Plan-Modus fängt Claude Code direkt an Dateien zu schreiben. Wenn die Richtung falsch ist, sind es 20 geänderte Dateien. Mit Plan-Modus erkennst du Missverständnisse bevor Code existiert.

Hotkey: Es gibt keinen festen Plan-Modus-Hotkey. Schreibe einfach „erst planen" oder /plan vor deine Aufgabe.

4.1 Installation

Claude Code ist ein CLI-Tool, installierbar über npm:

```
# Im Terminal: Claude Code global installieren (einmalig)
npm install -g @anthropic-ai/claude-code

# Im Terminal: Ins Projektverzeichnis wechseln und Claude Code starten
cd /path/to/Atomium
claude
```

Nach claude öffnet sich das Chat-Fenster im Terminal. Ab hier tippst du Fragen und Prompts – keine Terminal-Befehle, außer Ctrl+C um Claude Code zu beenden.

Claude Code liest beim Start automatisch CLAUDE.md und alle Dateien in .claude/. Das passiert im Hintergrund – danach kennt Claude Code den Projektkontext.

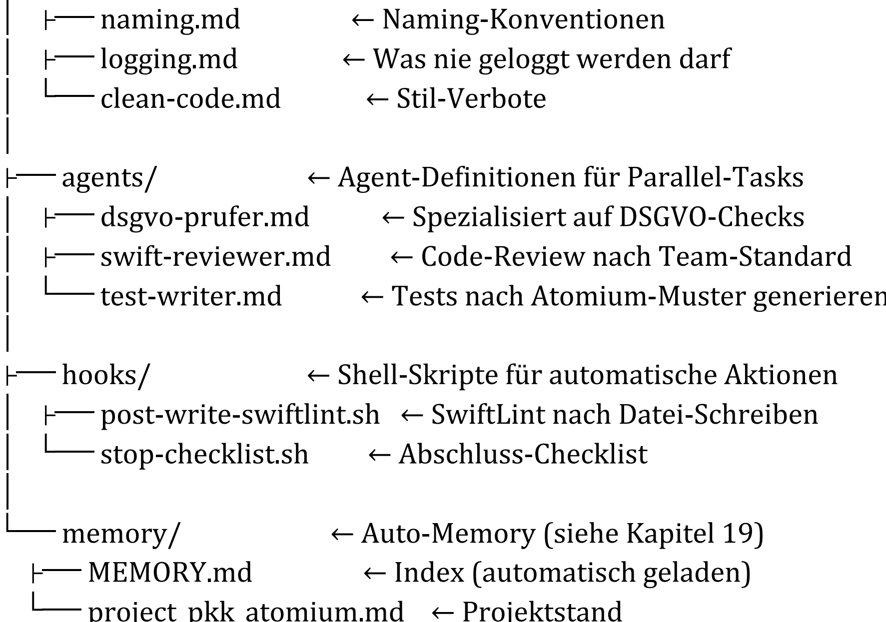
4.2 Projektstruktur für Claude Code

Claude Code braucht zwei Dinge im Projektverzeichnis: CLAUDE.md im Root und den .claude/-Ordner. Den Rest – Swift-Code, Xcode-Projektdateien, Tests – lässt es unangetastet.

Vollständige Struktur für Atomium:

```
Atomium/
├── CLAUDE.md                  ← Projektanweisungen (in Git, für alle)
├── CLAUDE.local.md            ← Persönliche Anweisungen (NICHT in Git)
│                          Beispiel: eigene Debugging-Präferenzen,
│                          lokale Pfade, persönliche Stilvorlieben
│
├── Atomium.xcodeproj
│
├── Atom/                      ← Aktives Framework (neuer Code kommt hierher)
│   ├── Features/
│   │   ├── IAMLogin/          ← Auth, Session, Biometrie
│   │   ├── Dashboard/         ← Kachel-Layout, Home Screen
│   │   ├── Postfach/          ← Nachrichten, Dokumente
│   │   ├── Antragserstellung/ ← Multi-Step-Flows
│   │   ├── Bescheinigungen/   ← Zertifikate, Downloads
│   │   ├── BonusProgram/      ← Punkte, Aktionen
│   │   ├── Gesundheit/        ← Gesundheitsthemen
│   │   ├── Serviceleistungen/ ← Erstattungen, Formulare
│   │   ├── MeineDaten/        ← Persönliche Daten, Datenschutz
```

```
│   │   ├── Einstellungen/        ← Benachrichtigungen, Sprache
│   │   └── Profilwechsel/        ← Familienmitglieder
│   │
│   └── Debug/                   ← Nur #if DEBUG, nie im Release
│       ├── DebugMenuComposer.swift
│       └── Features/
│           ├── LoginDebug/
│           ├── NetworkDebug/
│           └── FeatureFlags/
│
├── ATCore/                      ← Infrastruktur (Auth, Netzwerk, Persistence)
│                                  Kein neuer Code hier – nur lesen und nutzen
├── ATUI/                        ← Design-System, UIComponents
│                                  Kein neuer Code hier – nur lesen und nutzen
├── ATLegacy/                    ← Legacy (NICHT mehr verwenden)
│
└── .claude/
    ├── settings.json            ← Berechtigungen + Hooks (in Git, für alle)
    ├── settings.local.json      ← Persönliche Overrides (NICHT in Git)
    │
    ├── commands/                ← Custom Slash-Commands
    │   ├── check-async.md           → /check-async
    │   ├── check-dsgvo.md           → /check-dsgvo
    │   ├── check-security.md        → /check-security
    │   ├── check-memory.md          → /check-memory
    │   ├── check-threading.md       → /check-threading
    │   ├── check-deadcode.md        → /check-deadcode
    │   ├── check-tests.md           → /check-tests
    │   ├── check-architecture.md    → /check-architecture
    │   ├── check-all.md             → /check-all
    │   ├── review-feature.md        → /review-feature
    │   ├── make-mock.md             → /make-mock
    │   ├── make-tests.md            → /make-tests
    │   ├── add-docs.md              → /add-docs
    │   └── refactor.md              → /refactor
    │
    ├── skills/                  ← Prozesswissen (Team-Konvention, in CLAUDE.md referenziert)
    │   ├── composer.md              ← Wie Composer-Pattern funktioniert
    │   ├── unittest.md              ← UnitTest-Standard
    │   ├── repository.md            ← Repository-Pattern + Mock
    │   ├── async-await.md           ← Async/Await-Regeln
    │   ├── security.md              ← Sicherheits-Checks
    │   ├── dsgvo.md                 ← DSGVO-Checkliste
    │   └── debug-feature.md         ← Debug-Feature anlegen
    │
    ├── rules/                   ← Harte Verbote (in CLAUDE.md eingebunden)
    │   ├── architecture.md          ← Kein Code in ATLegacy
```



```
│   ├── naming.md          ← Naming-Konventionen
│   ├── logging.md         ← Was nie geloggt werden darf
│   └── clean-code.md      ← Stil-Verbote
│
├── agents/                ← Agent-Definitionen für Parallel-Tasks
│   ├── dsgvo-prufer.md    ← Spezialisiert auf DSGVO-Checks
│   ├── swift-reviewer.md  ← Code-Review nach Team-Standard
│   └── test-writer.md     ← Tests nach Atomium-Muster generieren
│
├── hooks/                 ← Shell-Skripte für automatische Aktionen
│   ├── post-write-swiftlint.sh  ← SwiftLint nach Datei-Schreiben
│   └── stop-checklist.sh  ← Abschluss-Checklist
│
└── memory/                ← Auto-Memory (siehe Kapitel 19)
    ├── MEMORY.md          ← Index (automatisch geladen)
    └── project_pkk_atomium.md  ← Projektstand
```

Automatisch geladen: CLAUDE.md, CLAUDE.local.md und .claude/commands/. Alles andere – skills/, rules/, agents/, memory/ – wird in CLAUDE.md referenziert oder explizit im Chat aufgerufen.

Was CLAUDE.local.md ist: Eine persönliche CLAUDE.md die nur auf deinem Rechner gilt und **nicht** ins Git-Repo committed wird. Trage sie in .gitignore ein. Ideal für persönliche Präferenzen: „Ich bevorzuge kürzere Antworten", „Erkläre mir Architekturentscheidungen immer mit einem Beispiel".

Was settings.local.json ist: Persönliche Overrides für Berechtigungen. Beispiel: du willst lokal git push erlauben, aber im Team-settings.json bleibt es gesperrt.

Was du selbst anlegst: Drei Wege:

- **Terminal:** touch .claude/commands/check-async.md → Datei erstellen, dann in Xcode befüllen
- **Xcode:** Rechtsklick auf .claude/commands/ → New File → leere Textdatei
- **Claude Code Chat:** „Erstelle die Datei .claude/commands/check-async.md mit folgendem Inhalt: ..." → Claude Code schreibt die Datei direkt ins Projekt

```
# Im Terminal: Ordnerstruktur einmalig anlegen (dann Claude Code neu starten)
mkdir -p .claude/commands .claude/skills .claude/rules .claude/agents .claude/hooks .claude/memory
```

Nach dem Anlegen: Claude Code neu starten (Ctrl+C, dann claude), damit die neue Struktur erkannt wird.

.gitignore-Einträge für persönliche Dateien:

```
# .gitignore
CLAUDE.local.md
.claude/settings.local.json
.claude/memory/user_*.md
.claude/memory/feedback_*.md
```

4.3 settings.json

settings.json legt fest welche Shell-Befehle Claude Code ohne Rückfrage ausführen darf – und welche grundsätzlich gesperrt sind. Ohne diese Datei fragt Claude Code bei jedem find, grep oder xcodebuild nach Bestätigung. Das wird schnell nervig.

Mit der Allowlist läuft das automatisch. Destruktive Befehle wie rm -rf bleiben dauerhaft gesperrt – egal was im Prompt steht.

```json
{
 "permissions": {
  "allow": [
   "Bash(swiftlint:*)",
   "Bash(xcodebuild:*)",
   "Bash(swift test:*)",
   "Bash(find:*)",
   "Bash(grep:*)",
   "Bash(cat:*)",
   "Bash(ls:*)"
  ],
  "deny": [
   "Bash(rm -rf:*)",
   "Bash(git push:*)",
   "Bash(git reset --hard:*)"
  ]
 }
}
```

allow: Claude Code darf diese Befehle ohne Bestätigung ausführen. Der *-Platzhalter erlaubt alle Argumente zu diesem Befehl.

deny: Diese Befehle sind immer gesperrt, egal was der Entwickler tippt. Schützt vor versehentlichem Datenverlust.

Wo die Datei liegt: .claude/settings.json gilt für alle (liegt im Repo). Persönliche Einstellungen: ~/.claude/settings.json im Home-Verzeichnis – gilt nur für den eigenen Rechner.

Wann wirksam: Beim nächsten Start. Wenn Claude Code gerade läuft: Ctrl+C, dann claude.

Erste Einrichtung Schritt für Schritt:

1. Im Terminal: touch .claude/settings.json
2. Datei in Xcode oder einem Editor öffnen
3. Den JSON-Inhalt (siehe oben) einfügen und speichern
4. Claude Code neu starten: Ctrl+C, dann claude
5. Claude Code fragt beim nächsten Bash-Befehl nicht mehr nach – die Allowlist greift

---

4.4 Eingebaute Slash-Commands

Diese Commands kommen mit Claude Code – nichts anlegen, direkt nach dem Start verfügbar.

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

Session & Kontext verwalten

Command	Was es tut
/init	Analysiert das aktuelle Projektverzeichnis und generiert eine erste CLAUDE.md
/context	Zeigt an wie viel Kontext-Prozent belegt sind und was geladen ist
/compact [Anweisung]	Fasst den bisherigen Kontext zusammen und verkleinert ihn – ohne Session zu verlieren
/clear	Löscht den gesamten Kontext. CLAUDE.md wird neu geladen. Günstigster Neustart.
/memory	Öffnet die Memory-Verwaltung: was Claude Code sich gemerkt hat, was gelöscht werden kann
/resume	Listet vorherige Sessions auf – du kannst eine wieder aufnehmen
/cost	Zeigt die Token-Kosten der aktuellen Session an

Arbeit planen und steuern

Command	Was es tut
/plan [Aufgabe]	Startet den Plan-Modus: Claude Code plant zuerst, handelt erst nach deiner Freigabe
/ultraplan	Maximale Planungstiefe: Claude Code durchdenkt mehrere Lösungswege bevor es beginnt
/focus	Blendet Zwischenschritte aus – du siehst nur das Endergebnis
/batch	Mehrere unabhängige Aufgaben gebündelt übergeben, Claude Code arbeitet sie durch
/copy	Kopiert den letzten Output in die macOS-Zwischenablage

Modell & Verhalten anpassen

Command	Was es tut
/model sonnet	Wechselt das Modell (z.B. sonnet, opus, haiku)
/effort low|medium|high|max	Stellt die Reasoning-Tiefe ein – max für komplexe Architekturanalysen, low für schnelle Antworten
/simplify	Weist Claude Code an, eine Antwort oder Code einfacher zu formulieren

---

Qualität & Sicherheit

Command	Was es tut
/security-review	Eingebautes Security-Review des aktuell geöffneten Kontexts
/agents	Sub-Agent-Manager: laufende Agents anzeigen, stoppen, überwachen
/hooks	Hook-Verwaltung: aktive Hooks anzeigen und testen
/permissions	Zeigt aktuelle Berechtigungen aus settings.json an
/fewer-permission-prompts	Häufig genehmigte Befehle werden zur Allowlist hinzugefügt – reduziert Rückfragen

---

System & Setup

Command	Was es tut
/mcp	MCP-Server (Model Context Protocol) verwalten – externe Tools anbinden
/doctor	System-Diagnose: überprüft ob Claude Code korrekt konfiguriert ist
/statusline	Status-Zeile im Terminal anpassen (Modell, Kosten, Kontext)
/insights	Analysiert Muster aus vergangenen Sessions: welche Aufgaben häufig vorkommen
/voice	Sprachsteuerung aktivieren (experimentell)
/install-github-app	GitHub-App für Claude Code installieren (Zugriff auf Issues, PRs)

▶ **Du gibst ein:**

```
/init
```

◀ **Claude Code liest** alle Swift-Dateien, die Projektstruktur und die Composer-Muster und generiert eine erste CLAUDE.md für Atomium. Das ist der schnellste Einstieg wenn noch keine CLAUDE.md existiert.

✎ **Du prüfst** den generierten Inhalt und ergänzt projektspezifische Regeln (DSGVO, ATLegacy-Verbote, etc.).

4.5 CLI-Flags beim Start

Diese Flags tippst du im **Terminal** – nicht im Chat.

```
claude                    # Normal starten
claude --resume           # Session auswählen (Liste der letzten Sessions)
claude --continue         # Letzte Session direkt weiterführen
claude --enable-auto-mode # Auto-Modus: Claude Code handelt selbstständig
claude --model sonnet     # Modell direkt beim Start setzen
claude --print "Frage"    # Einmalige Frage, kein interaktiver Modus
claude --no-auto-updates  # Automatische Updates deaktivieren
```

Empfehlung für den Alltag im Atomium-Projekt:

```
# Letzten Stand direkt weiterführen (kein Session-Auswahl-Dialog):
claude --continue

# Gezieltes Modell für schnelle Code-Reviews:
claude --model sonnet
```

4.6 Keyboard Shortcuts

Diese Shortcuts funktionieren im Chat-Eingabefeld, solange Claude Code im Terminal läuft.

Shortcut	Was es tut
Tab	Command-Autocomplete: nach / zeigt Tab alle verfügbaren Commands

Shortcut	Was es tut
↑ / ↓	Navigiert durch frühere Eingaben in der Session
Ctrl+C	Bricht die aktuelle Antwort ab (Claude Code hört sofort auf)
Ctrl+C (zweimal)	Beendet Claude Code vollständig
Ctrl+A	Cursor zum Anfang der Zeile
Ctrl+E	Cursor zum Ende der Zeile
Ctrl+K	Löscht alles ab Cursor-Position bis zum Zeilenende
Ctrl+L	Scrollt das Terminal-Fenster zurück nach oben
Shift+Enter	Neue Zeile in der Eingabe (für mehrzeilige Prompts)
Esc (zweimal)	Unterbricht – ähnlich wie Ctrl+C, ohne Claude Code zu beenden

Wichtigste Shortcuts im Atomium-Alltag:

- Tab nach / → zeigt alle Custom-Commands (/check-async, /review-feature etc.)
- Ctrl+C → wenn Claude Code in die falsche Richtung läuft, sofort stoppen
- Shift+Enter → mehrzeilige Prompts für komplexe Feature-Anfragen

---

4.7 Fortgeschrittenes

Nicht für den ersten Tag – aber sie steigern die Effizienz erheblich sobald die Grundlagen sitzen.

Auto-Modus

Im normalen Modus fragt Claude Code bei jedem Schritt nach: „Darf ich diese Datei schreiben?" Das ist sicher, aber langsam bei großen Aufgaben.

Auto-Modus: Claude Code arbeitet die Aufgabe durch, ohne zu unterbrechen. Sinnvoll für klar abgegrenzte, vollständig definierte Aufgaben.

```
# Terminal: Auto-Modus beim Start
claude --enable-auto-mode
```

Oder im Chat:

```
/effort max
Implementiere jetzt alle 5 Schritte aus dem Plan ohne weitere Rückfragen.
```

Wann Auto-Modus sinnvoll ist:

- Vollständig klare Aufgabe: „Erstelle für alle 6 bestehenden ViewModels in Atom/Features/BonusProgram/ je eine Test-Datei nach unserem Standard"
- Du weißt genau was du willst und hast einen Commit-Stand als Fallback

Wann Auto-Modus riskant ist:

- Vage Aufgaben ohne klare Grenzen
- Wenn du den Projektkontext noch nicht vollständig in CLAUDE.md hinterlegt hast
- Bei Aufgaben die bestehenden Code verändern (lieber mit Bestätigung)

Think-Keywords – Reasoning-Tiefe steuern

Claude Code kann unterschiedlich tief nachdenken – das beeinflusst Qualität und Kosten.

Keyword im Prompt	Was passiert	Wann einsetzen
(kein Keyword)	Direkte Antwort, kein tiefes Nachdenken	Einfache Fragen, Erklärungen
think	Claude Code denkt kurz nach	Code-Fragen, einfache Implementierungen
think hard	Tieferes Nachdenken, mehrere Ansätze abwägen	Architektur-Entscheidungen
think harder	Intensive Analyse, Edge-Cases bedenken	Komplexe Refactorings, Sicherheits-Checks
ultrathink	Maximale Tiefe – analysiert alle Konsequenzen	Kritische DSGVO-Prüfungen, komplexe Composer-Umbauten

Beispiel für Atomium:

```
think hard: Welche Auswirkungen hätte es, CertificateRepository von Combine auf async/await
umzustellen?
Welche anderen Teile des Systems sind betroffen?

ultrathink: Prüfe ob unsere aktuelle Token-Speicherung im Keychain DSGVO-konform ist.
Bedenke auch den Fall eines Profilwechsels und was mit dem alten Token passiert.
```

4.8 Best Practices – effizient und kostengünstig arbeiten

Vier Regeln, die Claude Code im Projektalltag spürbar besser machen. Kommen aus der Praxis.

Die 4 Karpathy-Regeln

Andrej Karpathy (KI-Forscher, ehemals OpenAI/Tesla) hat vier Grundregeln für produktives Arbeiten mit AI-Coding-Assistenten formuliert:

Regel 1 – Vor jeder Aufgabe committen Bevor du Claude Code eine Aufgabe gibst: git commit. Dann hast du immer einen Stand, auf den du zurücksetzen kannst. Claude Code kann in die falsche Richtung laufen – dann machst du einfach einen Reset.

```
# Im Terminal vor jeder Claude-Code-Session:
git add -A && git commit -m "WIP: vor Claude-Code-Session"
```

Regel 2 – Jeden Diff lesen Lies jede Änderung, bevor du sie übernimmst. Claude Code kennt den Kontext nicht so gut wie du. Zwei Minuten Diff-Lesen sparen Stunden Debugging.

Regel 3 – Aufgaben klein halten Lieber fünf kleine, klare Aufgaben nacheinander als eine vage große. „Schreibe einen ViewModel für Certificates" ist besser als „Implementiere das ganze Certificate-Feature".

Regel 4 – Neu starten wenn es hakelt Wenn Claude Code im dritten Anlauf dasselbe falsche Ergebnis produziert: Session beenden, /clear, von vorne mit einem präziseren Prompt. Ein frischer Kontext ist fast immer effizienter als Korrekturrunden.

Kontext & Kosten im Griff behalten

Claude Code rechnet pro Token ab. Ein voller Kontext kostet mehr als ein leerer – und lange Sessions akkumulieren schnell.

Wie Kosten entstehen:

- Jede Nachricht liest den bisherigen Kontext neu
- Je mehr Dateien geladen sind, desto teurer wird jede Anfrage
- Lange Sessions akkumulieren schnell

Wann /compact aufrufen:

```
/compact Behalte nur: aktueller Feature-Stand, letzte Architektur-Entscheidung, offene TODOs.
```

Rufe /compact auf wenn:

- /context zeigt über 60–70% Kontext-Auslastung
- Du eine lange Debugging-Session hattest und jetzt eine neue Aufgabe startest
- Die Session mehr als 2–3 Stunden läuft

Wann /clear besser ist als /compact: /compact fasst zusammen und behält den Gesprächsverlauf komprimiert. /clear löscht alles und startet frisch – CLAUDE.md wird neu geladen. /clear ist günstiger, aber du verlierst den Session-Kontext.

Situation	Empfehlung
Neue Aufgabe, kein Zusammenhang zur letzten	/clear
Selbe Aufgabe, Kontext wird zu groß	/compact
Anderer Entwickler übernimmt die Session	claude --resume → andere Session auswählen
Tages-Ende, morgen weiterarbeiten	/clear oder neue Session mit claude
Kostenkontrolle zwischendurch	/cost aufrufen

Faustregel: Eine Claude-Code-Session = eine Aufgabe. Nach Abschluss /clear. Das hält Kosten niedrig und Kontext sauber.

4.9 Memory – was Claude Code zwischen Sessions merkt

Claude Code hat kein Langzeitgedächtnis. Nach einem Neustart ist alles weg – Architektur, Entscheidungen, aktueller Stand. Du erklärst dasselbe immer wieder von vorne.

Memory löst das. Du bittest Claude Code einmal, wichtige Informationen unter .claude/memory/ zu speichern – beim nächsten Start werden diese Dateien automatisch geladen. Claude Code schreibt das Notizbuch selbst und schlägt es beim nächsten Gespräch wieder auf.

Was du damit gewinnst:

Ohne Memory	Mit Memory
Jede Session: Projekt von vorne erklären	Claude Code kennt Projekt, Architektur, letzten Stand
Präferenzen immer wiederholen	Einmal gesagt, dauerhaft gemerkt
Sprint-Übergang: Kontext verloren	Offene Aufgaben bleiben bekannt

So aktivierst du Memory – einmalig im Chat:

► **Du gibst ein:**

```
Merke dir: Wir arbeiten an Atomium, einer SwiftUI-App für eine private
Krankenkasse (NAK). Architektur: Composer-Pattern in Atom/.
ATLegacy ist Legacy – kein neuer Code.
Aktuell: Certificate-Feature in Entwicklung.
```

◄ **Claude Code legt** .claude/memory/project_pkk_atomium.md und MEMORY.md an. Beim nächsten Start mit claude werden diese Dateien automatisch geladen.

Laufend nutzen:

```
Merke dir: CertificateViewModel ist fertig, Tests stehen noch aus.
```

```
Wo haben wir aufgehört?
```

Wie Memory persistiert wird – technisch:

Memory sind gewöhnliche Markdown-Dateien auf deiner Festplatte unter .claude/memory/. Es gibt keine Datenbank, keine Cloud, keine Verschlüsselung. Was Claude Code sich „merkt", liegt als lesbarer Text in einer .md-Datei. Du kannst diese Dateien jederzeit in Xcode oder einem Texteditor öffnen, lesen und manuell bearbeiten.

```
.claude/memory/
├── MEMORY.md              ← Index: wird automatisch beim Start geladen
├── project_pkk_atomium.md    ← Projektkontext
├── feedback_code_style.md    ← Deine Code-Präferenzen
└── user_role.md             ← Dein Erfahrungsstand
```

Wann Memory löschen:

Situation	Was löschen
Information ist veraltet (Feature abgeschlossen, Entscheidung revidiert)	Die entsprechende Memory-Datei öffnen und den veralteten Abschnitt entfernen
Falsches gespeichert (Claude Code hat etwas missverstanden)	Datei öffnen, korrigieren oder löschen
Neuer Entwickler übernimmt (persönliche Präferenzen nicht relevant)	user_*.md und feedback_*.md löschen
Kompletter Neustart (Projekt-Umbau, neue Architektur)	Ganzen .claude/memory/-Ordner löschen

So löschst du eine einzelne Memory-Eintragung:

```
Im Chat: Vergiss bitte was du dir über das Certificate-Feature gemerkt hast.
Es ist jetzt abgeschlossen und der Eintrag ist veraltet.
```

Oder direkt in der Datei: .claude/memory/project_pkk_atomium.md öffnen, den betreffenden Abschnitt manuell löschen, speichern.

So löschst du alles auf einmal:

```
# Im Terminal – löscht den gesamten Memory-Ordner:
rm -rf .claude/memory/
```

Beim nächsten Start beginnt Claude Code mit einem leeren Gedächtnis.

Die vollständige Beschreibung – was genau gespeichert wird, welche Dateitypen es gibt und was in .gitignore muss – steht in **Kapitel 19**.

4.10 Konfigurationsdateien im Projektalltag pflegen

CLAUDE.md, Skills, Rules, Commands und Hooks entwickeln sich mit dem Projekt. Wer sie nicht pflegt, bekommt veralteten Code – Claude Code kennt dann schlicht nicht mehr den aktuellen Stand.

Datei	Wann aktualisieren
CLAUDE.md	Neues Feature fertig → Eintrag ergänzen · Architekturregel geändert → anpassen · Neues Framework eingebunden → dokumentieren
.claude/skills/	Neues Muster etabliert (z.B. neuer Test-Stil) → Skill anlegen oder anpassen · Alter Skill veraltet → löschen oder ersetzen
.claude/rules/	Neue harte Regel vom Team beschlossen → Rule-Datei ergänzen · Altes Verbot nicht mehr relevant → entfernen
.claude/commands/	Neuer wiederkehrender Check wird gebraucht → Command anlegen · Bestehender Command liefert falsches Ergebnis → Prompt verbessern
.claude/settings.json	Neues Tool im Projekt (z.B. neuer Build-Befehl) → allow-Liste ergänzen · Neuer Hook → hooks-Block erweitern
.claude/memory/	Sprint-Ende → laufende Aufgaben aktualisieren · Teamentscheidung gefallen → in Memory festhalten

Typische Auslöser im Projektalltag:

- **Nach einem Sprint:** CLAUDE.md um abgeschlossene Features ergänzen, offene Punkte in Memory aktualisieren
- **Wenn Claude Code Fehler wiederholt:** Prüfen ob eine Rule oder ein Skill fehlt – dann anlegen, nicht jedes Mal im Chat korrigieren
- **Wenn ein neuer Entwickler anfängt:** CLAUDE.md und Skills müssen vollständig und aktuell sein
- **Wenn ein Befehl in CI neu dazukommt:** settings.json um den allow-Eintrag ergänzen und Claude Code neu starten
- **Wenn das Team einen neuen Standard beschließt:** Skill oder Rule anlegen – nicht nur im Chat erwähnen

Faustregel: Wenn du dasselbe Claude Code zweimal erklärst, gehört es in eine Datei.

4.11 Claude Code fortlaufend verbessern – bessere Ergebnisse mit der Zeit

Claude Code wird besser, je mehr Projektwissen du ihm gibst. Das passiert nicht von selbst – du musst aktiv daran arbeiten. Drei Hebel helfen dabei.

Die drei Hebel für bessere Ergebnisse:

Hebel 1 – CLAUDE.md präziser machen

Wenn Claude Code etwas falsch macht (falsches Muster, falsche Struktur, altes Framework benutzt), schreib die Korrektur in CLAUDE.md. Damit passiert derselbe Fehler nie wieder.

Schlechtes Ergebnis → analysieren warum → fehlende Information in CLAUDE.md ergänzen.

Im Chat: Was fehlt dir in der CLAUDE.md, damit du diesen Fehler nicht wiederholt?

Hebel 2 – Schlechte Antworten als Feedback nutzen

Wenn Claude Code falsch liegt, sag es direkt:

```
Das war falsch weil du ATLegacy verwendet hast.
Neue Features kommen immer nach Atom/.
Merke dir diese Regel für künftige Anfragen.
```

Claude Code korrigiert sich und speichert die Regel in Memory – sofern du es so formulierst.

Hebel 3 – Skills und Rules iterativ verfeinern

Nach jedem Sprint: Prüfe ob die bestehenden Skills noch aktuell sind.

- Neues Muster eingeführt? → Skill anlegen
- Altes Verbot nicht mehr relevant? → Rule löschen
- Naming-Konvention geändert? → CLAUDE.md und Rule anpassen

Praktisches Verbesserungs-Ritual (alle zwei Wochen):

► **Im Chat eingeben:**

```
Analysiere unsere .claude/-Konfiguration:
1. Welche CLAUDE.md-Einträge sind möglicherweise veraltet?
2. Welche Fehler habe ich dir in den letzten Sessions korrigiert,
   die noch nicht in einer Rule oder Skill festgehalten sind?
3. Was sollte ich ergänzen damit du bessere Ergebnisse lieferst?
```

◄ **Claude Code analysiert** die Konfigurationsdateien und gibt konkrete Verbesserungsvorschläge. Es erkennt Lücken zwischen dem was du täglich machst und dem was in den Konfigurationsdateien steht.

✎ **Du entscheidest** welche Vorschläge sinnvoll sind und setzt sie um.

Was Teams nach 4 Wochen typischerweise berichten:

- Claude Code kennt das Composer-Pattern so gut dass es neue Features ohne Korrektur korrekt erstellt
- DSGVO-Checks werden automatisch angewendet ohne explizit erwähnt zu werden
- Tests folgen immer dem richtigen Given/When/Then-Muster
- Kein Legacy-Import in ATLegacy mehr

Was ohne aktives Zutun nie besser wird:

- Informationen die nur in Gesprächen vorkamen aber nie in Dateien landeten
- Teamentscheidungen die mündlich beschlossen aber nicht dokumentiert wurden
- Veraltete Einträge in CLAUDE.md die niemand löscht

---

5. CLAUDE.md – Projektgedächtnis

CLAUDE.md ist die wichtigste Datei im Projekt. Alle anderen Konfigurationsdateien – Skills, Rules, Commands – bauen darauf auf. Was reingehört, auf welchen Ebenen sie existiert und wie man sie für Atomium befüllt: das kommt jetzt.

5.1 Was CLAUDE.md ist und warum sie die wichtigste Datei ist

CLAUDE.md ist die einzige Datei, die Claude Code bei jedem Start automatisch und vollständig liest – ohne dass du etwas tippen musst. Alles, was darin steht, kennt Claude Code von der ersten Sekunde an.

„Beim Start" bedeutet: Jedes Mal wenn du im Terminal claude eintippst. Die laufende Session liest CLAUDE.md nicht erneut – dafür /clear im Chat eingeben.

Ohne CLAUDE.md erklärt man Claude Code bei jeder Session von vorne: Architektur, Regeln, Naming. Mit einer gepflegten CLAUDE.md entfällt das komplett.

Faustregel: Was ein neuer Entwickler in Woche 1 lernen muss, gehört in CLAUDE.md.

5.2 CLAUDE.md ist nicht einmalig – sie existiert auf mehreren Ebenen

CLAUDE.md kann in jedem Unterverzeichnis des Projekts existieren. Claude Code lädt automatisch alle CLAUDE.md-Dateien entlang des Pfads, in dem du arbeitest.

Wie das Laden funktioniert:

Beim Start liest Claude Code alle CLAUDE.md-Dateien vom Projektroot bis zum aktuellen Arbeitsverzeichnis. Die Dateien werden **additiv** kombiniert – keine überschreibt die andere.

```
Atomium/
├── CLAUDE.md                ← Immer geladen (globale Regeln)
│
├── Atom/
│   ├── CLAUDE.md             ← Geladen wenn in Atom/ gearbeitet wird
│   │                    (ergänzt die Root-CLAUDE.md)
│   └── Features/
│     ├── IAMLogin/
│     │   └── CLAUDE.md        ← Geladen wenn in IAMLogin/ gearbeitet wird
│     │                  (ergänzt Root + Atom-CLAUDE.md)
│     └── BonusProgram/
│       └── CLAUDE.md        ← Nur für BonusProgram spezifisches Wissen
│
└── ATCore/
    └── CLAUDE.md            ← Nur für ATCore-spezifisches Wissen
```

Was "additiv" bedeutet:

Wenn Claude Code in Atom/Features/IAMLogin/ arbeitet, lädt es:

1. Atomium/CLAUDE.md → globale Projektregeln
2. Atomium/Atom/CLAUDE.md → Atom-spezifische Regeln
3. Atomium/Atom/Features/IAMLogin/CLAUDE.md → IAMLogin-spezifisches Wissen

Alle drei Dateien sind gleichzeitig aktiv. Widersprüche vermeiden – im Zweifel hat die spezifischere Datei mehr Gewicht.

5.3 Was auf welcher Ebene steht

Root CLAUDE.md (immer geladen):

- Projektüberblick und Ziel der App
- Framework-Rollen (ATCore, ATUI, ATLegacy, Atom)
- Globale Architekturregeln (Composer-Pattern, kein neuer Code in ATLegacy)
- Naming-Konventionen
- Sicherheits- und DSGVO-Grundregeln
- Verweis auf Skills und Rules (damit Claude Code weiß, dass sie existieren)

Framework-Level CLAUDE.md (z.B. Atom/CLAUDE.md):

- Spezifische Regeln für dieses Framework
- Welche Patterns hier gelten
- Welche Klassen/Protokolle zentral sind

Feature-Level CLAUDE.md (z.B. Atom/Features/IAMLogin/CLAUDE.md):

- Spezifisches Wissen über dieses Feature
- Besonderheiten die kein anderes Feature hat
- API-Endpoints, Sonderfälle, bekannte Bugs und ihre Workarounds
- Welche Teile noch Legacy sind und warum

5.4 Wie man CLAUDE.md erstellt und bearbeitet

CLAUDE.md ist eine gewöhnliche Markdown-Datei. Es gibt zwei Wege sie zu erstellen:

Weg 1 – Manuell (du schreibst den Inhalt selbst):

```
# Im Terminal: Datei erstellen
touch CLAUDE.md

# Dann: Datei in Xcode oder einem Texteditor öffnen und befüllen
# Danach: Claude Code neu starten damit sie geladen wird
```

Weg 2 – Claude Code schreibt sie für dich (empfohlen für Feature-spezifische CLAUDE.md):

Tippe im Claude Code Chat:

```
Analysiere das Feature in Atom/Features/IAMLogin/ und erstelle eine
CLAUDE.md speziell für dieses Feature. Sie soll enthalten:
- Übersicht was das Feature macht
- Besonderheiten der Implementierung
- Bekannte Einschränkungen
- Wichtige Klassen und ihre Rollen
```

Claude Code liest den Code, analysiert das Feature und schreibt die Datei direkt ins Projekt. Nach dem nächsten Start (oder /clear) ist sie aktiv.

CLAUDE.md nach jedem großen Architekturentscheid aktualisieren. Eine veraltete CLAUDE.md ist schlimmer als keine – sie liefert falsches Wissen. Claude Code im Chat: „Aktualisiere CLAUDE.md, wir haben folgendes geändert: ..."

5.5 Beispiel: Root CLAUDE.md für Atomium

```markdown
# Atomium – iOS App

## Projekt
iOS-App der Neuen Atomium Krankenkasse (NAK). SwiftUI, Combine, Async/Await.
Frameworks: ATCore, ATUI, ATLegacy (Legacy, kein neuer Code!), Atom (aktiv).

## Architektur
Composer-Pattern in Atom. Kein neuer Code in ATLegacy.
Jeder Composer gibt eine View zurück. Environments lösen Dependencies auf.
Repositories haben immer Live + Mock-Implementierungen.

## Ordnerstruktur
Atom/Features/[FeatureName]/
  Domain/, Repository/, API/, Mapper/, Interactor/, UI/, Tests/

## Kernregeln
- Views sind dumm. Keine Business-Logik in Views.
- Kein direkter URLSession-Aufruf außerhalb von APIClient-Klassen.
- Kein UserDefaults für sensible Daten. Keychain via ATCore.KeychainService.
- Kein print() in Produktionscode. Nur ATCore.Logger verwenden.
- @MainActor konsequent auf ViewModels.
- Kein force unwrap (!) in neuem Code.

## Naming
- Composer: [Feature]Composer
- Environment: [Feature]Environment
- Repository-Protokoll: [Feature]Repository
- Live: [Feature]RepositoryLive
- Mock: [Feature]RepositoryMock
- ViewModel: [Feature]ViewModel
- View: [Feature]View
- Navigator: [Feature]Navigator

## Sensible Daten
Nie loggen: Tokens, Passwörter, Krankenversicherungsnummern, Diagnosen, Medikamente.
Keychain-Keys sind in ATCore.KeychainKey definiert.
Alle API-Calls laufen über HTTPS. Kein HTTP erlaubt.

## Tests
```

```
XCTest. Async/Await-Tests mit async throws. Kein DispatchQueue.main.async in Tests.
Mocks kommen aus dem jeweiligen Feature-Mock-Repository.

## Weitere Regeln (in separaten Dateien)
Die folgenden Dateien enthalten detailliertere Regeln. Lese sie wenn du in
dem jeweiligen Bereich arbeitest:
- .claude/rules/architecture.md  → Architekturverbote
- .claude/rules/logging.md       → Was nie geloggt werden darf
- .claude/rules/naming.md        → Naming-Konventionen im Detail
- .claude/skills/composer.md     → Wie ein Composer aufgebaut wird
- .claude/skills/dsgvo.md        → DSGVO-Checkliste
- .claude/skills/security.md     → Security-Checkliste
```

Beispiel: Feature-spezifische CLAUDE.md

```
# IAMLogin Feature

## Überblick
Authentifizierung via externen IAM-Provider (OAuth 2.0 + PKCE).
Biometrischer Login nach erstem Passwort-Login möglich.

## Besonderheiten
- Token-Refresh läuft automatisch via ATCore.TokenRefreshInterceptor
- Session-Ablauf wird zentral in IAMSessionManager behandelt
- FakeLogin für Debug-Builds: IAMLoginComposer prüft #if DEBUG

## Bekannte Einschränkungen
- Der IAM-Provider liefert bei Netzwerkfehler manchmal HTTP 200 mit Fehler im Body.
  Parsing-Sonderfall in IAMLoginAPIClient.parseAuthResponse() – nicht anfassen.

## Wichtige Klassen
- IAMLoginComposer   → Einstiegspunkt
- IAMSessionManager  → Session-Verwaltung (Singleton, bewusst so)
- IAMTokenStore      → Keychain-Wrapper für Tokens
```

---

6. Skills – Wiederverwendbare Anweisungen

Skills sind Markdown-Dateien mit fertigen Prompt-Vorlagen – kein Plugin, keine Magie. Was einmal in .claude/skills/ liegt, muss nicht mehr im Chat neu getippt werden.

6.1 Was ein Skill ist – und was nicht

Ein „Skill" im Kontext dieses Projekts ist **eine einfache Markdown-Datei**, die eine wiederverwendbare Anweisung oder ein Pattern enthält. Es gibt keine Magie dahinter. Kein Plugin. Kein spezielles Format. Nur Text in einer .md-Datei.

Claude Code hat kein eingebautes „Skill"-System. .claude/skills/ ist **unsere eigene Team-Konvention** – Prompt-Vorlagen in einem Ordner, damit sie nicht bei jeder Session neu getippt werden.

6.2 Wie ein Skill erstellt wird – Schritt für Schritt

Option A: Manuell erstellen (du schreibst die Datei selbst)

```
# Im Terminal: Datei erstellen
touch .claude/skills/dsgvo.md
```

Dann: Datei in Xcode oder einem Texteditor öffnen, Inhalt hineinschreiben, speichern.

Neustart nötig? Nein. Skills werden nicht beim Start geladen, sondern wenn du sie explizit erwähnst. Die Datei ist sofort nach dem Speichern verfügbar.

Option B: Claude Code erstellt die Datei für dich (empfohlen)

Öffne Claude Code und tippe **im Chat**:

```
Erstelle mir eine Skill-Datei unter .claude/skills/dsgvo.md.
Sie soll eine DSGVO-Prüf-Checkliste enthalten speziell für Atomium:
- Was nie geloggt werden darf (KV-Nummer, Diagnosen, Tokens)
- Wann Keychain statt UserDefaults verwendet werden muss
- Welche API-Calls Certificate Pinning benötigen
Schreib die Datei direkt ins Projekt.
```

Claude Code schreibt die Datei direkt ins Projekt. Du siehst im Chat welche Datei erstellt wurde und kannst den Inhalt sofort lesen. **Kein Neustart nötig.** Die Datei ist ab sofort bei jeder Referenz im Chat verfügbar.

6.3 Wie ein Skill benannt wird

Der **Dateiname** ist der Name des Skills. Das #-Heading in der Datei ist nur für Menschen lesbar – Claude Code interessiert nur der Dateiinhalt.

```
.claude/skills/dsgvo.md      → du referenzierst ihn als "dsgvo-Skill" oder
                               "die Datei .claude/skills/dsgvo.md"
.claude/skills/composer.md   → du referenzierst ihn als "composer-Skill" oder
                               "Datei .claude/skills/composer.md"
```

Namensregeln:

- Kleinbuchstaben, Bindestriche statt Leerzeichen
- Sprechend: async-await.md ist besser als rules1.md
- Kein Präfix nötig – der Ordner skills/ gibt den Kontext

6.4 Wie Claude Code einen Skill verwendet

Skills werden **nicht automatisch geladen**. Claude Code liest sie nur, wenn du sie explizit erwähnst:

Variante 1 – Direkt referenzieren im Chat:

```
Erstelle einen Composer nach dem Muster in .claude/skills/composer.md
```

Variante 2 – In CLAUDE.md einbinden (empfohlen):

Füge in die Root-CLAUDE.md einen Abschnitt ein, der Claude Code mitteilt, welche Skills es gibt und wann es sie lesen soll:

```
## Skills (Prompt-Vorlagen)

Wenn du einen Composer erstellst, lese zuerst .claude/skills/composer.md.
Wenn du Tests schreibst, lese zuerst .claude/skills/unittest.md.
Wenn du DSGVO-relevanten Code schreibst oder prüfst, lese .claude/skills/dsgvo.md.
Wenn du Security-relevanten Code schreibst oder prüfst, lese .claude/skills/security.md.
```

Damit kennt Claude Code die Skills und wendet sie automatisch an – kein manuelles Erinnern mehr.

Variante 3 – Als Basis für Commands:

Commands in .claude/commands/ können Skills referenzieren. Beispiel in check-dsgvo.md:

```
Führe die DSGVO-Prüfung aus .claude/skills/dsgvo.md durch für $ARGUMENTS.
```

6.5 Format eines Skills

Ein Skill ist einfacher Markdown-Text. Kein spezielles Format, keine Pflichtfelder. Der Inhalt ist der Prompt, den Claude Code ausführen soll.

Was in einen Skill gehört:

- Konkrete Anweisungen was Claude Code tun soll
- Regeln die immer gelten (z.B. Namenskonventionen)
- Code-Beispiele als Vorlage
- Checklisten was geprüft werden soll

Was nicht in einen Skill gehört:

- Allgemeines Hintergrundwissen (gehört in CLAUDE.md)
- Absolute Verbote (gehören in rules/)
- Projektstruktur-Beschreibungen (gehören in CLAUDE.md)

6.6 Die Skills für Atomium

Die vollständigen Inhalte aller sieben Skills für Atomium. Manuell anlegen oder Claude Code beauftragen (Kapitel 6.2). Danach in CLAUDE.md referenzieren (Kapitel 6.4), damit Claude Code sie automatisch anwendet.

.claude/skills/composer.md

```
# Composer-Pattern Atomium

Wenn du einen neuen Composer für ein Feature erstellst:

1. Datei: Atom/Features/[Feature]/UI/[Feature]Composer.swift
2. Immer enum (kein struct, kein class) – kein versehentlicher Zustand
```

3. Nutze WithContext { try Environment() } content: { env in ... }
4. Übergib der View nur Closures, keine konkreten Service-Typen
5. Keine Navigation-Logik im Composer – gehört in den Navigator

Beispiel:

````
\`\`\`swift
enum BonusProgramComposer {
    static func compose() -> some View {
        WithContext {
            try BonusProgramEnvironment()
        } content: { environment in
            BonusProgramView(
                onLoadPoints: environment.loadPoints,
                onRedeem: environment.redeem
            )
        }
    }
}
\`\`\`
````

.claude/skills/unittest.md

UnitTest-Pattern Atomium

Teststruktur für alle Tests:

````
\`\`\`swift
final class [Feature][Component]Tests: XCTestCase {
    private var sut: [Typ]!
    private var [mock]: [MockTyp]!

    override func setUp() async throws {
        try await super.setUp()
        [mock] = [MockTyp]()
        sut = [Typ](repository: [mock])
    }

    override func tearDown() async throws {
        sut = nil
        [mock] = nil
        try await super.tearDown()
    }

    func test_[methode]_[bedingung]_[ergebnis]() async throws {
        // Given
        // When
        // Then
    }
}
````

\`\`\`

Regeln:
- Naming: test_[methode]_[bedingung]_[ergebnis]
- Immer Given / When / Then Kommentare
- Kein sleep(), kein DispatchQueue.main.async in Tests
- @MainActor auf Testklasse wenn ViewModel @MainActor ist

.claude/skills/async-await.md

Async/Await-Regeln Atomium

1. ViewModels sind immer @MainActor
2. Repository-Methoden sind async throws
3. Task { } in Views nur für einmalige Ladevorgänge (.task Modifier bevorzugen)
4. Nie Task.detached – nur mit expliziter schriftlicher Begründung im PR
5. Nie await in init() – Laden startet immer in .task { } oder einer Methode

Retain-Cycle verhindern:
\`\`\`swift
// Falsch – strong capture:
Task { await self.loadData() }

// Richtig:
Task { [weak self] in await self?.loadData() }
\`\`\`

MainActor-Muster:
\`\`\`swift
@MainActor
final class BonusProgramViewModel: ObservableObject {
  @Published var points: Int = 0

  func loadPoints() async {
    isLoading = true
    defer { isLoading = false }
    do {
      points = try await repository.fetchPoints().points
    } catch {
      self.error = error
    }
  }
}
\`\`\`

.claude/skills/dsgvo.md

DSGVO-Prüfung Atomium

Prüfe den Code auf:

```
1. LOGGING – Was nie geloggt werden darf:
  KV-Nummer, Name, Geburtsdatum, Adresse, Diagnosen (ICD/OPS),
  Medikamente, Auth-Tokens, Passwörter, Kontodaten
  Erlaubt: technische IDs (UUID), anonyme Fehlercodes

2. PERSISTENZ – Sensible Daten:
  Verboten: UserDefaults oder unverschlüsseltes FileSystem
  Pflicht: ATCore.KeychainService für Tokens und Credentials

3. ANALYTICS – Nie an Analytics weitergeben:
  Diagnosen, Medikamente, Behandlungsarten

4. NETZWERK – Kein HTTP:
  Alle Requests via HTTPS. Certificate Pinning aktiv.

5. DATENMINIMIERUNG:
  API-Request-Modelle nur mit notwendigen Feldern.

Ausgabe: Datei, Zeile, Kategorie, Schwere (Kritisch/Hoch/Mittel), Fix.
```

.claude/skills/security.md

```
# Security-Checkliste Atomium

1. Keychain – Tokens nie in UserDefaults
2. Biometrie – nur via ATCore.BiometricService
3. Certificate Pinning – ATCore.NetworkClient hat Pinning aktiv
4. Screenshot Protection – .screenshotProtection() auf sensiblen Views
5. Background Snapshot – Blur-Screen im App Switcher
6. Deeplinks – Parameter validieren, nie direkt in Navigation nutzen
7. Clipboard – Passwort-/KV-Felder sperren Clipboard-Verlauf
8. Token Lifetime – abgelaufene Tokens erneuern oder invalidieren
9. Jailbreak Detection – aktiv und nicht umgehbar
10. Kein force unwrap in Auth-/Token-Pfaden

Ausgabe: Severity High/Medium/Low, Datei, Zeile, Beschreibung, Fix.
```

6.7 Kurzübersicht: Skills anlegen

Kurzform für wer direkt loslegen will: Ordner anlegen, Skill-Datei erstellen, in CLAUDE.md registrieren. Vollständige Erklärung: 6.1–6.5.

```
# Ordner erstellen (einmalig)
mkdir -p .claude/skills

# Skill-Datei anlegen – entweder:
touch .claude/skills/dsgvo.md        # dann manuell befüllen
```

```
# oder Claude Code direkt beauftragen:
# "Erstelle .claude/skills/dsgvo.md mit folgendem Inhalt: ..."

# In CLAUDE.md eintragen damit Claude Code die Skills kennt:
## Skills
Wenn du einen Composer baust → .claude/skills/composer.md lesen
Wenn du Tests schreibst → .claude/skills/unittest.md lesen
Wenn du DSGVO-Code prüfst → .claude/skills/dsgvo.md lesen
```

---

7. Rules – Harte Projektregeln

Rules sind absolute Verbote – keine Empfehlungen, keine Präferenzen. Hier steht wie sie angelegt werden, wie sie sich von Skills unterscheiden und welche Rules für Atomium gelten.

7.1 Was Rules sind und warum sie existieren

Rules definieren harte Grenzen. Kein „bevorzuge X" – sondern „X ist verboten, immer, ohne Ausnahme". Die rote Linie, die auch unter Zeitdruck nicht überschritten werden darf.

Der Unterschied in der Praxis:

- **Skill** → „Wenn du einen Composer baust, folge diesem Muster" (Anleitung, Vorlage)
- **Rule** → „Kein neuer Code in ATLegacy. Nie." (absolutes Verbot)

7.2 Wie Rules erstellt werden

Rules sind gewöhnliche Markdown-Dateien – genau wie Skills. Manuell erstellen oder Claude Code beauftragen.

Manuell:

```
mkdir -p .claude/rules
touch .claude/rules/architecture.md
# Dann in Xcode oder einem Editor befüllen
```

Über Claude Code:

```
Erstelle .claude/rules/logging.md mit den Logging-Verboten für Atomium:
- Nie KV-Nummer, Diagnosen, Tokens loggen
- Kein print() in Produktionscode
- Nur ATCore.Logger verwenden
Schreib die Datei direkt.
```

Dateiname: Beliebig. Sprechend und kleingeschrieben. Kein Präfix nötig – der rules/-Ordner gibt den Kontext.

7.3 Unterschied Rules vs. Skills: wo was hingehört

Inhalt	Gehört in

Inhalt	Gehört in
„Folge diesem Muster wenn du X machst"	skills/
„X ist immer verboten"	rules/
„Projektüberblick, Architektur, Naming"	CLAUDE.md
„Führe diese Prüfung durch für $ARGUMENTS"	commands/

7.4 Wie Claude Code die Rules kennt

Genau wie Skills werden Rules **nicht automatisch geladen**. Sie müssen in CLAUDE.md referenziert werden:

```
## Regeln (immer einhalten)

Lese die folgenden Dateien und halte ihre Regeln immer ein:
- .claude/rules/architecture.md  → Architekturverbote
- .claude/rules/logging.md       → Was nie geloggt werden darf
- .claude/rules/naming.md        → Naming-Pflichten
- .claude/rules/clean-code.md    → Code-Struktur-Regeln
```

Mit diesem Abschnitt in CLAUDE.md kennt Claude Code die Regeln von Anfang an – und du musst sie nicht bei jedem Chat neu erwähnen.

Alternative: Du kannst die Rules auch direkt in CLAUDE.md einbetten statt in separate Dateien. Für kleine Projekte ist das einfacher. Für ein großes Projekt wie Atomium lohnen sich separate Dateien, weil sie gezielt aktualisiert werden können ohne die gesamte CLAUDE.md anzufassen.

7.5 Die Rules für Atomium

Hier sind die vollständigen Inhalte der vier Rules-Dateien. Leg sie unter .claude/rules/ an – entweder manuell oder über Claude Code (Kapitel 7.2). Danach müssen sie in CLAUDE.md eingebunden werden (Kapitel 7.4), damit Claude Code sie bei jeder Session kennt.

.claude/rules/architecture.md

```
# Architekturverbote Atomium

Diese Regeln gelten ohne Ausnahme für neuen Code in Atom/:

VERBOTEN – Architektur:
- Kein neuer Code in ATLegacy. Migrations-PRs brauchen explizite Lead-Freigabe.
- Keine Business-Logik in SwiftUI-Views (body, private computed vars mit Logik).
- Kein direkter URLSession-Aufruf außerhalb von *APIClient.swift Dateien.
- Kein Singleton-Pattern für neue Features. DI läuft über Composer und Environment.
- Kein NotificationCenter für Feature-zu-Feature-Kommunikation. Closures oder Combine.
- Kein @EnvironmentObject in neuen Views. Nur explizite Parameter und Closures.

VERBOTEN – Swift:
- Kein force unwrap (!) in neuem Code außerhalb von Tests.
```

- Kein implicitly unwrapped optional (Type!) als Property-Typ in neuen Klassen.
- Kein try! außer in Unit-Test-Fixtures.

.claude/rules/logging.md

Logging-Regeln Atomium

Erlaubt:
- ATCore.Logger.debug(), .info(), .warning(), .error()
- Technische Fehlermeldungen ohne personenbezogene Daten
- Request-URLs (ohne Query-Parameter die Nutzerdaten enthalten)
- Anonyme Fehlercodes und HTTP-Statuscodes

VERBOTEN – darf nie geloggt werden:
- Krankenversicherungsnummer (KVK, KV-Nummer, Mitgliedsnummer)
- Name, Geburtsdatum, Adresse, E-Mail
- Diagnosen (ICD-Codes), Medikamente, Behandlungscodes (OPS)
- Auth-Tokens, Refresh-Tokens, Session-IDs
- Passwörter, PINs, biometrische Daten
- Kontodaten, IBAN

Produktionscode:
- Kein print(). Kein NSLog(). Kein debugPrint().
- Nur ATCore.Logger.

.claude/rules/naming.md

Naming-Pflichten Atomium

Typen (struct, class, enum, protocol):
- UpperCamelCase
- Keine Abkürzungen außer: URL, HTTP, HTTPS, API, ID, UI, IAM, KV
- Kein Präfix (kein AtomiumView, kein ATBonusViewModel außer in ATCore/ATUI)
- Kein ungarische Notation: nicht mValue, strName, bIsLoading

Funktionen und Methoden:
- lowerCamelCase
- Immer Verben: loadCertificates(), submitForm(), navigateToDetail()
- Kein generisches get/set wenn spezifischer möglich: fetchBonusPoints() statt getData()

Properties:
- lowerCamelCase
- Booleans: Präfix is/has/can/should → isLoading, hasError, canSubmit, shouldRefresh
- Kein negierter Boolean-Name: isLoggedOut statt isNotLoggedIn

Feature-Struktur (Pflicht-Namen):
- [Feature]Composer, [Feature]Environment, [Feature]ViewModel, [Feature]View
- [Feature]Repository (Protokoll), [Feature]RepositoryLive, [Feature]RepositoryMock
- [Feature]Navigator, [Feature]Destination

```
Tests:
- test_[methode]_[bedingung]_[ergebnis]
- Beispiel: test_fetchPoints_whenUnauthorized_throwsAuthError
- Kein test_1, test_erfolg, testFetchData
```

.claude/rules/clean-code.md

```
# Code-Struktur-Regeln Atomium

Größenbeschränkungen (Warning bei Überschreitung, Error bei deutlicher Überschreitung):
- Funktion: max. 30 Zeilen
- Datei: max. 300 Zeilen
- Parameter pro Funktion: max. 4 → dann struct verwenden

Kommentare:
- Kein Kommentar für "Was" – nur für "Warum" (nicht-offensichtliche Gründe)
- Kein Kommentar der den Funktionsnamen wiederholt
- Kein auskommentierter Code im Commit

Struktur:
- Extensions in eigene Dateien: [Typ]+[Kontext].swift
- MARK-Kommentare für Sektionen: // MARK: - Properties
- Keine verschachtelten Typen außer wenn der innere Typ nur im äußeren Sinn macht

Verboten:
- Kein dead code – löschen oder als TODO markieren
- Keine Magic Numbers ohne benannte Konstante
- Keine doppelte Logik – nicht kopieren, extrahieren
```

7.6 Kurzübersicht: Rules anlegen

Komprimierte Schritte-Referenz: Ordner anlegen, Rules-Datei erstellen, in CLAUDE.md einbinden. Vollständige Erklärung zu jedem Schritt steht in 7.1–7.4.

```
# Ordner erstellen (einmalig)
mkdir -p .claude/rules

# Datei anlegen – entweder manuell:
touch .claude/rules/architecture.md     # dann in Xcode befüllen

# oder Claude Code beauftragen:
# "Erstelle .claude/rules/architecture.md mit diesen Architekturverboten: ..."

# In CLAUDE.md eintragen damit Claude Code die Rules kennt:
## Regeln (immer einhalten)
- .claude/rules/architecture.md lesen und einhalten
- .claude/rules/logging.md lesen und einhalten
- .claude/rules/naming.md lesen und einhalten
- .claude/rules/clean-code.md lesen und einhalten
```

---

8. Commands – Anlegen, Verstehen und für die Qualitätssicherung einsetzen

Commands sind gespeicherte Prompts, per Slash aufrufbar. Einmal angelegt, nie wieder neu getippt. Hier steht wie sie funktionieren, wie man sie anlegt – und alle 14 Atomium-Commands mit vollständigem Dateiinhalt.

11.0 Was sind Commands – und warum brauchen wir sie?

Das Problem: Jedes Mal wenn du eine DSGVO-Prüfung machen willst, tippst du denselben langen Prompt neu. Du erinnerst dich vielleicht nicht an alle Punkte. Das Ergebnis ist jedes Mal leicht anders.

Die Lösung: Ein Command ist eine Markdown-Datei mit einem vollständigen Prompt. Einmal angelegt, ab dann per Slash aufgerufen. Immer derselbe Prompt, immer dasselbe strukturierte Ergebnis.

```
Ohne Command:    Du tippst 20 Zeilen Prompt neu ein
Mit Command:     Du tippst:  /check-dsgvo Atom/Features/IAMLogin
```

11.1 Wie Commands technisch funktionieren

Wo Command-Dateien liegen:

```
.claude/commands/check-dsgvo.md     ← Dateiname wird der Command-Name
.claude/commands/check-async.md     ← wird zu /check-async im Chat
.claude/commands/review-feature.md ← wird zu /review-feature im Chat
```

Der einzige Parameter: $ARGUMENTS

Wenn du in der Command-Datei $ARGUMENTS schreibst, ersetzt Claude Code diesen Platzhalter durch alles was du nach dem Command-Namen eingibst:

```
Command-Datei enthält:   Analysiere $ARGUMENTS auf DSGVO-Probleme...
Du tippst im Chat:       /check-dsgvo Atom/Features/IAMLogin
Claude Code führt aus:   Analysiere Atom/Features/IAMLogin auf DSGVO-Probleme...
```

Ein Command kann genau einen $ARGUMENTS-Platzhalter haben. Mehr gibt es nicht – keine benannten Parameter, keine Flags.

Commands ohne Parameter haben kein $ARGUMENTS und laufen immer mit demselben fixen Prompt – nützlich für Checks die das gesamte Projekt betreffen.

11.2 Command anlegen – so geht es

Weg A: Claude Code erstellt die Datei für dich (empfohlen)

Tippe **im Claude Code Chat**:

```
Erstelle die Command-Datei .claude/commands/check-dsgvo.md.
Der Command soll prüfen ob in $ARGUMENTS personenbezogene Daten geloggt werden,
ob sensible Daten sicher gespeichert sind, und ob alle Requests über HTTPS laufen.
Ausgabe: Datei, Zeile, Problem, Schwere, Fix.
```

Claude Code schreibt die Datei direkt ins Projekt. Danach: **Claude Code neu starten** (Ctrl+C im Terminal, dann claude). Erst nach dem Neustart erscheint /check-dsgvo als Command.

Weg B: Manuell im Terminal

```
# Im Terminal:
mkdir -p .claude/commands
touch .claude/commands/check-dsgvo.md
```

Datei in Xcode öffnen, Prompt-Inhalt einfügen, speichern. Dann Claude Code neu starten.

Alle Atomium-Commands auf einmal anlegen:

Tippe im Chat:

```
Erstelle alle Command-Dateien aus Kapitel 12 dieser Anleitung unter .claude/commands/.
```

Danach einmalig neu starten – alle Commands sind verfügbar.

Wann ist ein Command verfügbar?

- Command-Datei erstellt → Claude Code neu starten → Command verfügbar
- Im Chat: / eintippen und Tab drücken → alle verfügbaren Commands werden angezeigt
- Command-Dateien liegen unter .claude/commands/ – jede Datei ist ein Command

---

11.3 Alle Atomium-Commands – Dateiinhalt und Verwendung

Jeder Command: Dateiinhalt, Aufruf im Chat, was Claude Code zurückgibt.

Alle Commands im Chat verwenden: relativ zum Projektroot, z.B. Atom/Features/BonusProgram – nicht der absolute Mac-Pfad.

/check-async – Async/Await-Analyse

► **Anlegen:** Datei .claude/commands/check-async.md erstellen mit diesem Inhalt:

```
Analysiere alle Swift-Dateien in $ARGUMENTS auf Async/Await-Probleme.

Prüfe auf:
1. Task { } ohne [weak self] Capture – Retain-Cycle-Risiko
2. @MainActor-Verletzungen: async Methoden, die UI-State ändern ohne @MainActor-Kontext
3. DispatchQueue.main.async statt await MainActor.run {}
4. Blocking Calls im async Kontext: Thread.sleep() statt Task.sleep()
5. Fehlende Cancellation-Behandlung bei long-running Tasks

Ausgabe pro Fund: Datei · Zeile · Problem · Fix als Swift-Snippet
Abschluss: Anzahl Funde pro Kategorie.
```

► **Im Chat aufrufen:**

```
/check-async Atom/Features/BonusProgram
/check-async Atom/Features/Certificate/UI/CertificateViewModel.swift
/check-async Atom/Features
```

◄ **Claude Code prüft und gibt aus:** Datei, Zeile, Problemtyp, konkretes Code-Beispiel, Fix-Snippet. Am Ende eine Zusammenfassung.

/check-memory – Retain-Cycles und Speicherlecks

► **Anlegen:** Datei .claude/commands/check-memory.md erstellen:

```
Analysiere alle Swift-Dateien in $ARGUMENTS auf Speicherprobleme und Retain-Cycles.

Prüfe auf:
1. Starke self-Captures in Closures, die vom selben Objekt gehalten werden
2. delegate-Properties ohne weak var
3. Combine sink { } ohne [weak self]
4. Timer oder NotificationCenter.addObserver ohne späteres Aufräumen
5. AnyCancellable-Instanzen außerhalb von Set<AnyCancellable>

Für jeden Fund: Datei · Zeile · Warum es ein Problem ist · Fix
Abschluss: Gesamtanzahl, Severity (Critical / Warning).
```

► **Im Chat aufrufen:**

```
/check-memory Atom/Features/Mailbox
/check-memory Atom/Features/IAMLogin/UI/IAMLoginViewModel.swift
/check-memory Atom/Features
```

◄ **Claude Code gibt aus:** Pro Fund: welches Objekt hält welches, warum das ein Leak ist, korrekter Fix-Code.

/check-threading – MainActor und Threading

► **Anlegen:** Datei .claude/commands/check-threading.md erstellen:

```
Analysiere $ARGUMENTS auf Threading-Probleme und @MainActor-Verletzungen.

Wenn $ARGUMENTS ein ViewModel-Name ohne Pfad ist, suche die Datei in Atom/Features/.
Wenn $ARGUMENTS ein Pfad ist, analysiere direkt diese Datei.

Prüfe auf:
1. ViewModel-Klassen ohne @MainActor-Annotation
2. @Published Properties, die von background async Methoden gesetzt werden
3. UI-relevante State-Änderungen außerhalb des MainActor-Kontexts
4. Race-Conditions: mehrere parallele Tasks die denselben State schreiben

Für jeden Fund: Datei · Zeile · Code-Ausschnitt · Erklärung · Fix
Bewertung: Ist dieses ViewModel thread-safe? Ja / Nein / Bedingt
```

► **Im Chat aufrufen:**

```
/check-threading BonusProgramViewModel
/check-threading Atom/Features/BonusProgram/UI/BonusProgramViewModel.swift
```

◄ **Claude Code gibt aus:** Konkrete Threading-Probleme mit Erklärung, warum sie gefährlich sind, und den korrekten Fix.

/check-dsgvo – DSGVO-Prüfung

► **Anlegen:** Datei .claude/commands/check-dsgvo.md erstellen:

Führe eine DSGVO-Prüfung für $ARGUMENTS durch.

Prüfe auf:
1. LOGGING – Personenbezogene Daten geloggt?
  Verboten: KV-Nummer, Name, Geburtsdatum, Diagnosen (ICD/OPS), Medikamente, Tokens, Passwörter
  Erlaubt: technische IDs (UUID), anonyme Fehlertypen

2. PERSISTENZ – Sensible Daten sicher gespeichert?
  Verboten: UserDefaults oder unverschlüsseltes FileSystem
  Pflicht: Keychain via ATCore.KeychainService

3. ANALYTICS – Gesundheitsdaten an Analytics?
  Verboten: Diagnosen, Medikamente, Behandlungsarten

4. NETZWERK – Alle API-Calls über HTTPS?
  Verboten: http:// URLs. Certificate Pinning muss aktiv sein.

5. DATENMINIMIERUNG – Nur notwendige Daten gesendet?

Ausgabe pro Fund: Datei · Zeile · Kategorie · Bewertung (Kritisch/Hoch/Mittel) · Fix
Abschluss: Bestanden / Nicht bestanden.

► **Im Chat aufrufen:**

/check-dsgvo Atom/Features/IAMLogin
/check-dsgvo Atom/Features/MyData/UI/MyDataViewModel.swift
/check-dsgvo Atom/Features

◄ **Claude Code gibt aus:** Pro Fund: Kategorie, Schwere, konkreter Code, Fix. Am Ende Gesamturteil.

/check-security – Security-Review

► **Anlegen:** Datei .claude/commands/check-security.md erstellen:

Führe ein Security-Review für $ARGUMENTS durch.

Prüfe auf:
1. KEYCHAIN – Tokens/Credentials in Keychain, nicht UserDefaults
2. BIOMETRIE – ATCore.BiometricService verwendet, nicht direkt LocalAuthentication
3. CERTIFICATE PINNING – ATCore.NetworkClient mit aktivem Pinning
4. SCREENSHOT PROTECTION – .screenshotProtection() auf sensiblen Views
5. BACKGROUND SNAPSHOT – Blur-Screen im App Switcher

6. DEEPLINK VALIDATION – Parameter validiert, nicht direkt verwendet
7. CLIPBOARD – Passwort-/KV-Felder sperren Clipboard-Verlauf
8. TOKEN LIFETIME – Abgelaufene Tokens erneuert oder invalidiert
9. JAILBREAK DETECTION – Aktiv und nicht umgehbar
10. Kein force unwrap in Auth-/Token-Pfaden

Ausgabe pro Fund: Severity (High/Medium/Low) · Datei · Zeile · Beschreibung · Fix
Abschluss: Security-Score, Liste offener High-Severity-Punkte.

► **Im Chat aufrufen:**

```
/check-security Atom/Features/IAMLogin
/check-security Atom/Features/IAMLogin/Repository/IAMLoginRepositoryLive.swift
/check-security Atom/Features
```

◄ **Claude Code gibt aus:** Severity-bewertete Liste aller Sicherheitsprobleme mit Fix-Vorschlägen.

/check-deadcode – Dead-Code-Analyse

► **Anlegen:** Datei .claude/commands/check-deadcode.md erstellen:

Analysiere das Feature $ARGUMENTS in Atom/Features/$ARGUMENTS/ auf toten Code.

Prüfe auf:
1. Funktionen die nirgendwo in Atom/ aufgerufen werden
2. Properties die nirgendwo gelesen oder geschrieben werden
3. Typen (struct, class, enum) die nicht referenziert werden
4. Enum-Cases die in keinem switch/if-case verwendet werden
5. Auskommentierter Code

Unterscheide:
- "Definitiv unused" – kann sicher gelöscht werden
- "Möglicherweise extern genutzt" – aus ATCore/ATUI möglicherweise verwendet, manuell prüf
en

Ausgabe: Datei · Zeile · Symbol-Name · Kategorie (Definitiv / Möglicherweise)

► **Im Chat aufrufen** (Feature-Name ohne Pfad – Claude Code sucht in Atom/Features/[Name]/):

```
/check-deadcode Certificate
/check-deadcode BonusProgram
```

◄ **Claude Code gibt aus:** Liste aller ungenutzten Symbole mit Bewertung ob sicher löschbar.

/check-architecture – Architekturprüfung

► **Anlegen:** Datei .claude/commands/check-architecture.md erstellen:

Prüfe das Feature in $ARGUMENTS auf Einhaltung der Atom-Architektur.

Erwartete Struktur (Atom/Features/[Feature]/):

- Domain/   → reine Modelle, kein SwiftUI, kein Netzwerk
- Repository/ → Protokoll + *Live.swift + *Mock.swift
- API/   → *APIClient.swift + *APIModel.swift
- Mapper/   → *Mapper.swift, mappt API → Domain
- UI/   → *Composer.swift, *Environment.swift, *View.swift, *ViewModel.swift
- Tests/   → *Tests.swift für Repository, ViewModel, Mapper

Prüfe auf Verstöße:
1. Fehlende Dateien laut Struktur oben
2. Repository ohne Live- oder Mock-Implementierung
3. Business-Logik direkt in einer View-Datei
4. URLSession-Aufruf außerhalb einer APIClient-Datei
5. Import ATLegacy in Atom/Features/
6. Composer ohne WithContext
7. ViewModel ohne @MainActor

Ausgabe: Checkliste ✅/❌ pro Punkt + Fundstellen bei ❌.

► **Im Chat aufrufen:**

/check-architecture Atom/Features/BonusProgram
/check-architecture Atom/Features/Certificate

◄ **Claude Code gibt aus:** Checkliste mit ✅/❌ für jeden Prüfpunkt, bei ❌ konkrete Datei und Zeile.

/check-tests – Test-Coverage

► **Anlegen:** Datei .claude/commands/check-tests.md erstellen:

Prüfe die Testabdeckung für das Feature $ARGUMENTS.

Schritt 1: Lese alle public und internal Methoden aus:
- Atom/Features/$ARGUMENTS/UI/*ViewModel.swift
- Atom/Features/$ARGUMENTS/Repository/*RepositoryLive.swift
- Atom/Features/$ARGUMENTS/Mapper/*Mapper.swift

Schritt 2: Lese alle Testdateien in Atom/Features/$ARGUMENTS/Tests/

Schritt 3: Prüfe für jede Methode:
- Gibt es einen Success-Testfall?
- Gibt es einen Error/Failure-Testfall?
- Wird isLoading getestet?
- Werden Edge-Cases getestet?

Ausgabe:
- ✅ Methoden mit ausreichender Abdeckung
- ❌ Methoden mit fehlenden Tests + welche Testfälle fehlen
- ⚠️ Fehlende Testdateien

Erstelle anschließend die fehlenden Tests nach dem Projekt-Standard:
- Naming: test_[methode]_[bedingung]_[ergebnis]
- Struktur: Given / When / Then
- Kein DispatchQueue, kein sleep()

► **Im Chat aufrufen** (Feature-Name ohne Pfad):

/check-tests Certificate
/check-tests BonusProgram

◄ **Claude Code gibt aus:** Coverage-Bericht + direkt die fehlenden Tests als fertiger Swift-Code.

/review-feature – Vollständiges Pre-Merge-Review

► **Anlegen:** Datei .claude/commands/review-feature.md erstellen:

Führe ein vollständiges Pre-Merge-Review für das Feature $ARGUMENTS durch.
Pfad: Atom/Features/$ARGUMENTS/

Führe diese Checks der Reihe nach aus:
1. Architektur – Composer-Aufbau, Live/Mock vorhanden, keine ATLegacy-Imports
2. Async/Await – weak-self-Fehler, MainActor-Verletzungen, DispatchQueue-Missbrauch
3. Speicher – Retain-Cycles, Combine-Leaks, Timer-Deregistrierung
4. DSGVO – Logging sensibler Daten, unsichere Persistenz, HTTP-Calls
5. Security – Keychain, Pinning, Screenshot-Protection
6. Tests – ViewModel, Repository und Mapper ausreichend getestet
7. Naming und Clean Code – Konventionen, Funktionslänge, Dateilänge

Ausgabe als Report:

Pre-Merge Review: $ARGUMENTS
Architektur ✅/❌
Async/Await ✅/❌
Speicher ✅/❌
DSGVO ✅/⚠️/❌
Security ✅/⚠️/❌
Tests ✅/❌
Code-Qualität ✅/❌

Offene Punkte (müssen vor Merge behoben werden):
[Liste]

Empfehlung: Merge freigeben / Merge blockiert

► **Im Chat aufrufen** (vor jedem Merge):

/review-feature Certificate
/review-feature BonusProgram
/review-feature IAMLogin

◀ **Claude Code führt alle 7 Checks aus** und gibt einen strukturierten Report zurück, der direkt als PR-Kommentar verwendet werden kann.

/make-mock – Mock für Repository erstellen

► **Anlegen:** Datei .claude/commands/make-mock.md erstellen:

Erstelle einen vollständigen Mock für das Repository-Protokoll $ARGUMENTS.

Suche das Protokoll in Atom/Features/ – es heißt $ARGUMENTS.swift.

Generiere eine Datei [Name]Mock.swift im Repository/-Ordner mit:
1. Result<T, Error> Property für jede async throws Methode
   Standardwert: .success([sinnvoller Default])
2. Call-Counter (Int) für jede Methode: [methodenName]CallCount
3. Parameter-Capture für letzten Aufruf: last[MethodenName][Parameter]
4. Implementierung: callCount erhöhen, Parameter speichern, Result zurückgeben
5. final class für Referenzsemantik in Tests

Gib die fertige Datei direkt aus.

► **Im Chat aufrufen:**

/make-mock BonusProgramRepository
/make-mock CertificateRepository

◀ **Claude Code erstellt** die Mock-Datei direkt in Atom/Features/[Feature]/Repository/.

/make-tests – UnitTests generieren

► **Anlegen:** Datei .claude/commands/make-tests.md erstellen:

Generiere vollständige UnitTests für $ARGUMENTS.

Suche die Datei in Atom/Features/ – sie heißt $ARGUMENTS.swift.
Suche den passenden Mock im Repository/-Ordner des gleichen Features.

Erstelle Tests für jede public und internal Methode:
- Success-Pfad
- Error-Pfad (jeden throw-Pfad separat)
- Loading-State (isLoading true während Laden, false danach)

Test-Standard:
- Klasse: [Name]Tests: XCTestCase
- setUp/tearDown async throws
- Naming: test_[methode]_[bedingung]_[ergebnis]
- Given / When / Then Kommentare
- @MainActor auf Testklasse wenn ViewModel @MainActor ist
- Kein DispatchQueue, kein sleep()

Gib die fertige Testdatei direkt aus.

► **Im Chat aufrufen:**

/make-tests CertificateViewModel
/make-tests BonusProgramViewModel
/make-tests CertificateRepositoryLive
/make-tests CertificateMapper

◄ **Claude Code erstellt** die vollständige Testdatei direkt in Atom/Features/[Feature]/Tests/.

/add-docs – Dokumentation generieren

► **Anlegen:** Datei .claude/commands/add-docs.md erstellen:

Füge Apple-konforme Dokumentationskommentare zu $ARGUMENTS hinzu.

Dokumentiere alle public und internal Deklarationen.
Private Deklarationen bleiben undokumentiert.

Stil:
- Englisch
- Erster Satz: kurze Zusammenfassung was es macht (nicht wie)
- Kein Boilerplate: nicht "This class is responsible for..."
- Parameter: nur wenn Name nicht selbsterklärend
- Returns: nur wenn Rückgabewert nicht offensichtlich
- Throws: nur wenn Fehlertypen für den Aufrufer wichtig

Gib die Datei mit eingefügten Kommentaren vollständig zurück.

► **Im Chat aufrufen:**

/add-docs Atom/Features/Certificate/Repository/CertificateRepository.swift
/add-docs Atom/Features/BonusProgram/UI/BonusProgramViewModel.swift

◄ **Claude Code gibt** die Datei mit eingefügten Kommentaren zurück – du prüfst und speicherst sie.

/refactor – Refactoring-Plan erstellen

► **Anlegen:** Datei .claude/commands/refactor.md erstellen:

Analysiere $ARGUMENTS für ein sicheres Refactoring.

Schritt 1 – Analyse:
- Was macht die Klasse/Datei aktuell?
- Welche Verantwortlichkeiten hat sie?
- Welche Abhängigkeiten hat sie?
- Wer ruft diese Datei auf (suche in Atom/ und ATCore/)?

Schritt 2 – Bewertung:

- Entspricht sie dem Composer/ViewModel/Repository-Pattern?
- Welche Teile können extrahiert werden?
- Liegt sie in ATLegacy? → Migrationsstrategie nötig

Schritt 3 – Plan:
Erstelle einen schrittweisen Refactoring-Plan.
Noch keinen Code schreiben. Nur Plan.
Warte auf Bestätigung bevor du mit der Umsetzung beginnst.

▶ **Im Chat aufrufen:**

/refactor ATLegacy/Sources/LoginManager.swift
/refactor Atom/Features/BonusProgram/UI/BonusProgramView.swift

◀ **Claude Code liest** die Datei, analysiert Abhängigkeiten und erstellt einen strukturierten Migrationsplan als Text – noch keine Codeänderungen.

✎ **Du liest den Plan** und gibst ihn frei (oder verlangst Anpassungen).

▶ **Du gibst ein:** Plan sieht gut aus. Fange mit Schritt 1 an.

◀ **Claude Code beginnt** erst jetzt mit der Implementierung.

/check-all – Alle Features prüfen (kein Parameter)

▶ **Anlegen:** Datei .claude/commands/check-all.md erstellen:

Prüfe alle Features in Atom/Features/ auf Architektur-Konformität.

Für jeden Feature-Ordner:
1. Gibt es Domain/, Repository/, UI/, Tests/?
2. Hat das Repository Live + Mock?
3. Gibt es einen Composer?
4. Liegt kein Import ATLegacy in der UI-Schicht?

Ausgabe als Tabelle:
| Feature | Domain | Repo+Mock | Composer | Kein Legacy | Tests |
⚠️ = vorhanden aber unvollständig, ❌ = fehlt komplett

▶ **Im Chat aufrufen** (kein Parameter nötig):

/check-all

◀ **Claude Code scannt** alle Feature-Ordner in Atom/Features/ und gibt eine Tabelle zurück.

11.4 Kurzreferenz – alle Commands

Alle 14 Commands auf einen Blick. Im Chat / gefolgt vom Command-Namen tippen. Die meisten erwarten Pfad oder Feature-Name als Parameter – durch Leerzeichen getrennt. Einzige Ausnahme: /check-all hat keinen Parameter.

Im Chat eingeben (nicht im Terminal):

```
/check-async      [pfad]          Async/Await-Probleme
/check-memory     [pfad]          Retain-Cycles, Leaks
/check-threading  [viewmodel|pfad]  MainActor, Race-Conditions
/check-dsgvo      [pfad]          Datenschutz
/check-security   [pfad]          Security
/check-deadcode   [feature-name]    Toter Code
/check-tests      [feature-name]    Test-Coverage
/check-architecture [pfad]        Composer-Struktur
/review-feature   [feature-name]    Vollständiges Pre-Merge-Review
/make-mock        [protokoll-name]  Mock generieren
/make-tests       [klassen-name]    Tests generieren
/add-docs         [pfad]          Dokumentation einfügen
/refactor         [pfad]          Refactoring-Plan erstellen
/check-all                        Alle Features prüfen
```

Tab-Completion: Im Chat / eintippen und Tab drücken – alle verfügbaren Commands werden angezeigt.

9. Hooks – Automatische Aktionen nach Claude-Code-Schritten

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

18.1 Was sind Hooks?

Hooks sind Shell-Befehle, die Claude Code automatisch ausführt – ohne dass du danach fragen musst. Du konfigurierst sie einmalig in .claude/settings.json. Danach passiert der Hook bei jedem passenden Claude-Code-Schritt selbsttätig im Hintergrund.

Ohne Hook: Claude Code erstellt eine neue Swift-Datei → du merkst erst beim nächsten CI-Lauf, dass SwiftLint Fehler wirft.

Mit Hook: Claude Code erstellt eine neue Swift-Datei → SwiftLint läuft sofort automatisch auf genau dieser Datei → du siehst den Fehler noch bevor du die Datei öffnest.

18.2 Wann werden Hooks ausgelöst?

Event	Wann es auslöst
PostToolUse	Nachdem Claude Code ein Tool verwendet hat (z.B. Datei geschrieben, Datei bearbeitet)
PreToolUse	Unmittelbar bevor Claude Code ein Tool einsetzt
Stop	Wenn Claude Code die Aufgabe abgeschlossen hat und wartet

Für Atomium sind PostToolUse-Hooks am nützlichsten: SwiftLint soll laufen nachdem eine Datei erstellt oder geändert wurde.

18.3 Hooks konfigurieren

Hooks werden in .claude/settings.json eingetragen. Die Datei existiert bereits (siehe Kapitel 4.3). Du ergänzt einen hooks-Block:

```
{
 "permissions": {
  "allow": ["Bash(swiftlint:*)", "Bash(xcodebuild:*)", "Bash(find:*)"],
  "deny": ["Bash(rm -rf:*)", "Bash(git push:*)"]
 },
 "hooks": {
  "PostToolUse": [
   {
    "matcher": "Write|Edit",
    "command": "if [[ \"$CLAUDE_FILE_PATH\" == *.swift ]]; then swiftlint lint --path \"$CLAUDE_FILE_PATH\"; fi"
   }
  ]
 }
}
```

Was hier passiert:

- matcher: "Write|Edit" → der Hook greift bei Datei-Schreiben und Datei-Bearbeiten
- $CLAUDE_FILE_PATH → die Umgebungsvariable enthält den Pfad der gerade bearbeiteten Datei
- if [[... == *.swift]] → Hook läuft nur bei Swift-Dateien, nicht bei Markdown oder JSON

Nach der Änderung: Claude Code muss neu gestartet werden (Kapitel 4.3 – Neustart-Tabelle).

18.4 Nützliche Hooks für Atomium

Hook 1 – SwiftLint nach jeder Dateiänderung:

```
{
 "matcher": "Write|Edit",
 "command": "if [[ \"$CLAUDE_FILE_PATH\" == *.swift ]]; then swiftlint lint --path \"$CLAUDE_FILE_PATH\"; fi"
}
```

◀ **Claude Code schreibt** eine neue ViewModel-Datei → SwiftLint prüft sofort die Datei und gibt Meldungen direkt im Chat aus: warning: Line Length Violation (140/120).

✎ **Du siehst** die SwiftLint-Meldungen ohne die CI-Pipeline abzuwarten.

Hook 2 – Test-Erinnerung wenn ein ViewModel geändert wird:

```
{
 "matcher": "Write|Edit",
 "command": "if [[ \"$CLAUDE_FILE_PATH\" == *ViewModel.swift ]]; then echo \"Hinweis: ViewModel wurde geändert – Tests prüfen: Atom/Features/${CLAUDE_FILE_PATH##*/ViewModel.swift}Tests/\"; fi"
}
```

◀ **Claude Code ändert** CertificateViewModel.swift → Im Chat erscheint automatisch: Hinweis: ViewModel wurde geändert – Tests prüfen: ...

✎ **Du entscheidest** ob du die Tests sofort laufen lässt oder erst nach weiteren Änderungen.

Hook 3 – Abschluss-Zusammenfassung nach jeder Aufgabe:

```
{
 "PostToolUse": [],
 "Stop": [
  {
   "command": "echo \"--- Claude Code fertig. Checklist: 1) Tests grün? 2) SwiftLint sauber? 3) D
SGVO-Daten geprüft? ---\""
  }
 ]
}
```

◀ **Claude Code meldet** Aufgabe erledigt → Im Chat erscheint automatisch die Checklist.

✎ **Du arbeitest** die drei Punkte durch bevor du den PR öffnest.

18.5 Vollständiges settings.json mit allen Atomium-Hooks

```
{
 "permissions": {
  "allow": [
   "Bash(swiftlint:*)",
   "Bash(xcodebuild:*)",
   "Bash(find:*)"
  ],
  "deny": [
   "Bash(rm -rf:*)",
   "Bash(git push:*)"
  ]
 },
 "hooks": {
  "PostToolUse": [
   {
    "matcher": "Write|Edit",
    "command": "if [[ \"$CLAUDE_FILE_PATH\" == *.swift ]]; then swiftlint lint --path \"$CLAUD
E_FILE_PATH\"; fi"
   },
   {
    "matcher": "Write|Edit",
    "command": "if [[ \"$CLAUDE_FILE_PATH\" == *ViewModel.swift ]]; then echo \"ViewModel
 geändert – denk an Tests!\"; fi"
   }
  ],
  "Stop": [
   {
    "command": "echo \"Aufgabe abgeschlossen. Checklist: Tests grün? SwiftLint sauber? DSGV
```

```
0 geprüft?\""
    }
   ]
  }
}
```

Wichtig: Nach jeder Änderung an settings.json Claude Code neu starten – erst dann sind die Hooks aktiv.

18.6 Was Hooks nicht können

Einschränkung	Erklärung
Kein Zugriff auf Claude-Code-Kontext	Der Hook-Befehl ist ein reiner Shell-Befehl – er weiß nicht, was Claude Code gerade macht
Kein automatischer Fix	Hooks können melden, aber nicht selbst Code korrigieren
Kein Xcode-Start	Hooks laufen im Terminal-Kontext, nicht in Xcode
Nur bei Tool-Einsatz	Hooks laufen nicht wenn du selbst eine Datei änderst – nur wenn Claude Code es tut

---

10. Auto-Memory – Claude Code merkt sich dein Projekt

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

19.1 Was ist Auto-Memory?

Claude Code hat kein dauerhaftes Gedächtnis zwischen Sessions. Nach einem Neustart kennt es weder Projekt noch aktuellen Stand – du erklärst alles von vorne.

Auto-Memory löst das. Claude Code legt unter .claude/memory/ Markdown-Dateien an, die es beim nächsten Start automatisch liest. Die Index-Datei MEMORY.md wird als erstes geladen – Claude Code weiß damit sofort, was zuletzt passiert ist.

19.2 Welche Informationen speichert Auto-Memory?

Typ	Inhalt	Beispiel für Atomium
Projekt	Laufende Aufgaben, Deadlines, Entscheidungen	„Sprint 12: Zertifikate-Feature in Arbeit, PR offen seit 2026-05-18"
Feedback	Deine Präferenzen, was Claude Code anders machen soll	„Keine auskommentierten Code-Blöcke – wir löschen stattdessen"
Nutzer	Deine Rolle, dein Kenntnisstand	„iOS-Entwickler mit 5 Jahren Swift-Erfahrung, neu bei Compose-Patterns"
Referenzen	Wo Informationen zu finden sind	„Bugs werden in Jira-Projekt Atomium verfolgt"

19.3 Wie funktioniert das – Schritt für Schritt

Einmalig aktivieren:

► Du gibst ein:

Merke dir bitte folgendes für unsere nächsten Sessions:
Wir arbeiten an Atomium, der iOS-App der Neuen Atomium Krankenkasse (NAK).
Architektur: Composer-Pattern mit enum-basierten Composern.
Aktiver Framework-Ordner: Atom/
Legacy (kein neuer Code): ATCore, ATUI, ATLegacy
Wir verwenden das Maker-Mock-Pattern für alle Repositories.

◄ Claude Code legt an:

- .claude/memory/project_pkk_atomium.md mit Projektkontext
- .claude/memory/MEMORY.md als Index-Datei

Diese Dateien werden beim nächsten Start automatisch geladen.

✎ **Du prüfst:** War die gespeicherte Information korrekt? Kannst du die Dateien unter .claude/memory/ im Finder oder Xcode einsehen.

Laufende Arbeit merken lassen:

► Du gibst ein:

Merke dir: Wir haben heute das Zertifikate-Feature angefangen.
CertificateComposer ist fertig, CertificateViewModel fehlt noch.
Nächster Schritt: ViewModel schreiben und testen.

◄ Claude Code aktualisiert .claude/memory/project_pkk_atomium.md mit dem aktuellen Stand.

Beim **nächsten Start** sagst du:

► Du gibst ein:

Wo haben wir aufgehört?

◄ Claude Code liest MEMORY.md, öffnet die verlinkten Dateien und gibt eine Zusammenfassung: „Zertifikate-Feature, CertificateViewModel noch ausstehend."

Präferenzen dauerhaft speichern:

► Du gibst ein:

Merke dir: Ich möchte keine Kommentare im Code außer wenn der Grund
wirklich nicht offensichtlich ist. Kein //MARK: bei Dateien unter 50 Zeilen.

◄ Claude Code schreibt eine Feedback-Datei unter .claude/memory/feedback_code_style.md und verlinkt sie in MEMORY.md.

Ab sofort gilt diese Präferenz in jeder neuen Session – ohne erneut erklären zu müssen.

19.4 Wo liegen die Dateien?

```
Atomium/
└── .claude/
    ├── settings.json        ← Berechtigungen, Hooks
    ├── commands/            ← Deine /commands
    └── memory/
        ├── MEMORY.md        ← Index (wird automatisch geladen)
        ├── project_pkk_atomium.md
        ├── feedback_code_style.md
        └── user_role.md
```

Alle Dateien sind plain Markdown – du kannst sie jederzeit öffnen, lesen und manuell anpassen. Claude Code überschreibt sie bei der nächsten Session nicht, sondern ergänzt sie.

19.5 Was Auto-Memory für Atomium konkret bringt

Situation	Ohne Memory	Mit Memory
Neue Session starten	Claude Code kennt nichts, du erklärst alles von vorne	Claude Code kennt Projekt, Architektur und letzten Stand
Präferenz mitteilen	Musst du jede Session wiederholen	Einmal gesagt, dauerhaft gemerkt
Sprint-Übergang	Context geht verloren	Offene Tasks sind weiterhin bekannt
Neuer Entwickler	Muss selbst Projekt erklären	Claude Code kennt das Projekt bereits

---

19.6 Was Claude Code sich nicht merkt

Auto-Memory ersetzt weder git noch ein Ticketsystem. Es speichert Kontext – keine Code-Änderungen. Für Code bleibt git die einzige Quelle der Wahrheit.

Die .claude/memory/-Dateien mit persönlichen Präferenzen gehören in .gitignore. Projektweite Erinnerungen (Architekturentscheidungen, Teamstandards) können committet werden – dann haben alle dieselbe Basis.

```
# In .gitignore ergänzen:
.claude/memory/user_*.md       ← persönliche Präferenzen nicht committen
.claude/memory/feedback_*.md  ← persönliches Feedback nicht committen

# Committen:
.claude/memory/project_*.md   ← Projektkontext für alle
.claude/memory/MEMORY.md      ← Index für alle
```

---

11. Agents – Spezialisierte Sub-Instanzen

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

20.1 Was sind Agents?

Wenn du Claude Code bittest, alle 12 Features auf DSGVO zu prüfen, arbeitet normalerweise eine Instanz sie nacheinander durch. Das dauert lang und überfüllt den Kontext.

Agents lösen das. Claude Code kann für Teilaufgaben eigenständige Sub-Instanzen starten. Jeder Agent bekommt genau eine Aufgabe, seinen eigenen Kontext, und läuft unabhängig. Der Haupt-Claude koordiniert und sammelt die Ergebnisse – wie ein Tech Lead, der Alice auf Authentication, Bob auf Postfach und Carlos auf Bonusprogramm ansetzt.

20.2 Wann lohnen sich Agents im Atomium-Projekt?

Aufgabe	Ohne Agent	Mit Agent
DSGVO-Check aller 12 Features	Nacheinander, lange Wartezeit, voller Kontext	12 Agents parallel, jeder prüft ein Feature
Refactoring-Analyse vor einem großen Umbau	Eine lange Antwort mit allem vermischt	Jeder Agent analysiert ein Subsystem getrennt
Test-Coverage-Bericht für alle Features	Kontextverschmutzung durch 30+ Dateien	Jeder Agent liest nur seinen Feature-Ordner
Migrations-Check (z.B. ATLegacy → Atom)	Unübersichtlich bei vielen Dateien	Separater Agent pro altem Modul

---

20.3 Wie du Agents einsetzt

Keine Konfiguration nötig – du rufst Agents im Chat auf. Du beschreibst die Aufgabe so, dass parallele Sub-Aufgaben sinnvoll sind.

Beispiel: DSGVO-Prüfung aller Features

► **Du gibst ein:**

```
Starte für jedes Feature unter Atom/Features/ einen separaten Agent.
Jeder Agent soll:
1. Den Feature-Ordner auf DSGVO-Verstöße prüfen (KV-Nummer, Diagnose, Tokens in Logs)
2. Ein kurzes Ergebnis zurückgeben: Feature-Name + Befunde (oder "keine Befunde")

Features: Certificate, BonusProgram, Mailbox, Dashboard, UserData, Settings,
          FamilySwitch, IAMLogin, ServiceLeistungen, Antragserstellung,
          Gesundheit, Profilwechsel

Sammle alle Ergebnisse am Ende in einer Tabelle.
```

◄ **Claude Code startet** für jedes der 12 Features einen Agent. Die Agents arbeiten unabhängig, lesen jeweils nur ihren Feature-Ordner, und geben ein Ergebnis zurück. Am Ende fasst Claude Code die Berichte zusammen:

```
| Feature         | DSGVO-Befunde                                |
|-----------------|----------------------------------------------|
```

Certificate	Keine Befunde
BonusProgram	⚠️ Logger.debug("KV: \(user.kvNummer)") – Zeile 47
Mailbox	Keine Befunde
...	...

✎ **Du gehst** die ⚠️-Befunde durch und behebst sie.

Beispiel: Refactoring-Analyse vor einem großen Umbau

▶ **Du gibst ein:**

```
Analysiere folgende drei Subsysteme jeweils getrennt:
- Atom/Features/IAMLogin/ → welche Teile davon könnten in ATCore verschoben werden?
- Atom/Features/UserData/ → welche Teile sind noch von ATLegacy abhängig?
- Atom/Features/Settings/ → wo wird direkt auf UserDefaults zugegriffen statt über ein Repository?

Starte für jeden Bereich einen eigenen Analyse-Durchlauf und gib drei separate Berichte zurück.
```

◀ **Claude Code analysiert** die drei Bereiche unabhängig voneinander und gibt drei saubere Berichte zurück – ohne dass sie sich gegenseitig im Kontext stören.

✎ **Du priorisierst** welchen Umbau du zuerst angehst.

20.4 Was Agents nicht können

Einschränkung	Erklärung
Kein gemeinsamer Schreib-Zugriff	Zwei Agents sollten nicht gleichzeitig dieselbe Datei schreiben
Kein persistenter State zwischen Agents	Agents sehen sich gegenseitig nicht – jeder startet mit eigenem Context
Keine UI-Interaktion	Agents können keine Xcode-Aktionen starten oder Simulatoren bedienen
Abhängige Aufgaben nicht parallelisierbar	Wenn Agent B das Ergebnis von Agent A braucht, müssen sie sequenziell laufen

Faustregel: Agents lohnen sich wenn die Teilaufgaben vollständig unabhängig voneinander sind. Wenn das Ergebnis von Teil A die Eingabe für Teil B ist, arbeite lieber sequenziell.

20.5 Agents vs. Commands vs. Skills – der Unterschied

Konzept	Was es ist	Wann einsetzen
Command	Gespeicherter Prompt-Template mit /name	Wiederkehrende Einzelaufgabe (1 Feature, 1 Datei)
Skill	Gespeichertes Prozesswissen in CLAUDE.md	Architekturwissen, das immer verfügbar sein soll
Agent	Eigenständige Sub-Instanz für eine Teilaufgabe	Viele unabhängige Aufgaben gleichzeitig

Ein typischer Workflow kombiniert alle drei: /check-dsgvo als Command → Claude Code startet für jeden Feature-Ordner einen Agent → jeder Agent nutzt das DSGVO-Skill-Wissen aus CLAUDE.md.

---

12. Neues Feature entwickeln

Du beschreibst, Claude Code schreibt. Ein einziger Prompt erzeugt ein vollständiges Feature – Composer, Repository, Mock, ViewModel, View und Tests inklusive. So sieht das in der Praxis aus.

8.1 Wie der Workflow funktioniert

Es gibt genau **einen Moment**, in dem du etwas eintippst. Alles andere – Code, Dateien, Analysen – kommt von Claude Code. Du liest, prüfst und korrigierst bei Bedarf.

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

8.2 Beispiel: Feature „Bescheinigungsdownload"

Versicherte sollen eine Liste ihrer Bescheinigungen sehen und einzelne als PDF herunterladen können.

Schritt 1 – Du startest das Feature mit einem einzigen Prompt

▶ Du gibst ein:

```
Erstelle ein neues Feature "Bescheinigungsdownload" nach unserem Atom-Standard.

Feature-Beschreibung:
- Versicherte sehen eine Liste ihrer verfügbaren Bescheinigungen
- Jede Bescheinigung kann als PDF heruntergeladen werden
- Typen: Mitgliedsbescheinigung, Steuerbescheinigung, Behandlungsnachweis

API (bereits vorhanden):
- GET /api/v1/certificates        → Liste aller Bescheinigungen
- GET /api/v1/certificates/{id}/pdf → PDF als Data

Erstelle alle Dateien nach unserem Standard:
- Domain-Modelle (Certificate, CertificateType)
- Repository-Protokoll + RepositoryLive + RepositoryMock
- APIClient + APIModelle + Mapper
- ViewModel mit @MainActor
- Environment
- Composer
- View (einfache Liste mit Download-Button)
- UnitTests für ViewModel und Repository

Orientiere dich an .claude/skills/composer.md und .claude/skills/unittest.md.
```

Ab hier arbeitet Claude Code. Du wartest.

Schritt 2 – Claude Code analysiert zuerst

◀ Claude Code liest zuerst das Projekt:

Claude Code liest zuerst bestehende Features (z.B. Postfach, BonusProgram), die CLAUDE.md und die Skills – bevor es eine Datei schreibt. Dann zeigt es dir seinen Plan: welche Dateien es anlegen wird, in welcher Reihenfolge.

Schritt 3 – Claude Code erstellt alle Dateien

◀ Claude Code erstellt: Domain/CertificateModel.swift

```
struct Certificate: Identifiable, Equatable {
  let id: String
  let title: String
  let issuedAt: Date
  let type: CertificateType
  let downloadURL: URL
}

enum CertificateType: String, Codable {
  case membershipProof = "MEMBERSHIP_PROOF"
  case taxStatement    = "TAX_STATEMENT"
  case treatmentRecord = "TREATMENT_RECORD"
}
```

◀ Claude Code erstellt: Repository/CertificateRepository.swift

```
protocol CertificateRepository {
  func fetchCertificates() async throws -> [Certificate]
  func downloadCertificate(id: String) async throws -> Data
}
```

◀ Claude Code erstellt: Repository/CertificateRepositoryLive.swift

```
struct CertificateRepositoryLive: CertificateRepository {
  private let apiClient: CertificateAPIClient

  func fetchCertificates() async throws -> [Certificate] {
    let response = try await apiClient.getCertificates()
    return response.map(CertificateMapper.map)
  }

  func downloadCertificate(id: String) async throws -> Data {
    try await apiClient.downloadCertificate(id: id)
  }
}
```

◀ Claude Code erstellt: Repository/CertificateRepositoryMock.swift

```swift
final class CertificateRepositoryMock: CertificateRepository {
    var fetchResult: Result<[Certificate], Error> = .success(Certificate.previews)
    var downloadResult: Result<Data, Error> = .success(Data())
    var fetchCallCount = 0
    var lastDownloadedID: String?

    func fetchCertificates() async throws -> [Certificate] {
        fetchCallCount += 1
        return try fetchResult.get()
    }

    func downloadCertificate(id: String) async throws -> Data {
        lastDownloadedID = id
        return try downloadResult.get()
    }
}
```

◀ Claude Code erstellt: UI/CertificateViewModel.swift

```swift
@MainActor
final class CertificateViewModel: ObservableObject {
    @Published var certificates: [Certificate] = []
    @Published var isLoading = false
    @Published var error: Error?
    @Published var downloadingID: String?

    private let repository: CertificateRepository

    init(repository: CertificateRepository) {
        self.repository = repository
    }

    func loadCertificates() async {
        isLoading = true
        defer { isLoading = false }
        do {
            certificates = try await repository.fetchCertificates()
        } catch {
            self.error = error
        }
    }

    func download(certificate: Certificate) async {
        downloadingID = certificate.id
        defer { downloadingID = nil }
        do {
            let _ = try await repository.downloadCertificate(id: certificate.id)
        } catch {
```

```
            self.error = error
        }
    }
}
```

◀ **Claude Code erstellt: UI/CertificateEnvironment.swift**

```
struct CertificateEnvironment {
    let viewModel: CertificateViewModel

    init() throws {
        let repository = CertificateRepositoryLive(
            apiClient: CertificateAPIClient(
                networkClient: try ATCore.NetworkClient.shared()
            )
        )
        viewModel = CertificateViewModel(repository: repository)
    }
}
```

◀ **Claude Code erstellt: UI/CertificateComposer.swift**

```
enum CertificateComposer {
    static func compose() -> some View {
        WithContext {
            try CertificateEnvironment()
        } content: { environment in
            CertificateView(viewModel: environment.viewModel)
        }
    }
}
```

◀ **Claude Code erstellt: UI/CertificateView.swift**

```
struct CertificateView: View {
    @ObservedObject var viewModel: CertificateViewModel

    var body: some View {
        Group {
            if viewModel.isLoading {
                ATUILoadingView()
            } else {
                certificateList
            }
        }
        .task { await viewModel.loadCertificates() }
        .alert(error: $viewModel.error)
    }

    private var certificateList: some View {
```

```
        List(viewModel.certificates) { certificate in
            CertificateRow(
                certificate: certificate,
                isDownloading: viewModel.downloadingID == certificate.id,
                onDownload: {
                    Task { [weak viewModel] in
                        await viewModel?.download(certificate: certificate)
                    }
                }
            )
        }
    }
}
```

◀ **Claude Code erstellt: Tests/CertificateViewModelTests.swift**

Claude Code schreibt die Tests nach dem Pattern aus .claude/skills/unittest.md. Alle öffentlichen Methoden werden getestet, Success- und Error-Pfad.

Schritt 4 – Du prüfst den generierten Code

✎ **Du überprüfst:**

Der einzige manuelle Schritt. Du öffnest die erzeugten Dateien in Xcode und prüfst:

- Compiliert der Code?
- Stimmen die API-Endpunkte mit dem Backend überein?
- Sind die Domain-Modelle vollständig?
- Entspricht die View dem gewünschten Design?

Wenn etwas nicht stimmt, sagst du es Claude Code direkt:

▶ **Du korrigierst im selben Chat:**

```
Der API-Endpunkt für den Download ist /api/v1/certificates/{id}/download, nicht /pdf.
Bitte korrigiere CertificateAPIClient entsprechend.
```

◀ **Claude Code ändert nur die betroffene Datei** und zeigt dir den Diff.

Schritt 5 – Navigation anbinden

✎ **Du entscheidest:**

Wo soll das Feature eingehängt werden? Im Service-Tab, im Gesundheits-Tab? Das ist eine Produktentscheidung, die du triffst.

▶ **Du gibst ein:**

```
Binde CertificateComposer in den ServiceComposer ein.
Die Destination heißt .certificates.
```

◀ **Claude Code ergänzt:** ServiceComposer.swift

```
case .certificates:
  CertificateComposer.compose()
```

Und erweitert ServiceDestination um den neuen Case – ohne dass du das manuell machst.

Schritt 6 – CLAUDE.md aktualisieren

► Du gibst ein:

Aktualisiere CLAUDE.md und trage das neue Certificate-Feature ein.

◄ Claude Code ergänzt CLAUDE.md automatisch:

```
## Certificate (Bescheinigungsdownload)
Composer: CertificateComposer
Pfad: Atom/Features/Certificate/
Einstieg: Service-Tab → Bescheinigungen
API: GET /api/v1/certificates, GET /api/v1/certificates/{id}/download
```

Du musst CLAUDE.md nicht manuell bearbeiten. Claude Code macht das – aber nur wenn du es bittest. Es ändert nichts ohne deinen Auftrag.

8.3 Zusammenfassung: Was du tust, was Claude Code tut

Zur Orientierung: wer welchen Schritt bei der Feature-Entwicklung übernimmt.

Schritt	Wer	Was
Feature beschreiben	**Du**	Einen Prompt schreiben mit Feature-Anforderungen und API
Code analysieren	**Claude Code**	Bestehende Features lesen, Pattern erkennen
Alle Dateien erstellen	**Claude Code**	Domain, Repository, Mock, ViewModel, Environment, Composer, View, Tests
Code prüfen	**Du**	Xcode öffnen, compilieren, visuell prüfen
Korrekturen	**Du**	Fehler im Chat beschreiben → Claude Code ändert
Navigation anbinden	**Du sagst wo**	Claude Code ändert die betroffenen Dateien
CLAUDE.md aktualisieren	**Du beauftragst**	Claude Code schreibt den Eintrag

Du schreibst keinen Swift-Code von Hand. Du beschreibst, entscheidest, prüfst. Claude Code schreibt.

---

13. Bestehendes Feature erweitern

Drei typische Alltagssituationen: neue Seite in einem bestehenden Feature ergänzen, Legacy-Code in ATLegacy verstehen ohne ihn anzufassen, Legacy-Klasse in die neue Atom-Architektur migrieren.

Legende: ► Du gibst ein = Eingabe im Claude Code Chat · ◄ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

9.1 Erweiterung: Neue Seite in bestehendem Feature

Beispiel: Das Bonusprogramm soll eine „Aktionsverlauf"-Seite bekommen.

Schritt 1 – Du lässt Claude Code zuerst analysieren

► **Du gibst ein:**

```
Analysiere das Bonusprogramm-Feature in Atom/Features/BonusProgram/.
Zeige mir:
1. Welche Composer, ViewModels, Repositories bereits existieren
2. Welche Patterns verwendet werden
3. Wo Abweichungen vom Standard-Pattern vorliegen
4. Was ich bei einer Erweiterung beachten muss
```

◄ **Claude Code liest alle Dateien in Atom/Features/BonusProgram/** und gibt dir eine strukturierte Übersicht zurück – Klassennamen, Verantwortlichkeiten, Auffälligkeiten. Du musst keine Datei selbst öffnen.

Schritt 2 – Du entscheidest ob die Analyse passt

✎ **Du liest den Überblick** und stellst sicher, dass Claude Code die Struktur korrekt verstanden hat. Wenn etwas fehlt oder falsch ist, korrigierst du es im Chat.

Schritt 3 – Du beauftragst die Erweiterung

► **Du gibst ein:**

```
Ich möchte dem Bonusprogramm eine "Aktionsverlauf"-Seite hinzufügen.
Der Nutzer soll sehen, welche Bonus-Aktionen er wann abgeschlossen hat.

API: GET /api/v1/bonus/history → Liste von BonusAction (id, title, points, completedAt)

Analysiere zuerst wie das Postfach (Atom/Features/Mailbox/) aufgebaut ist,
da es eine ähnliche Listenstruktur hat.
Dann erstelle:
- BonusHistoryAction (Domain-Modell)
- Erweiterung des BonusProgramRepository-Protokolls um fetchHistory()
- Erweiterung der RepositoryLive-Implementierung
- Erweiterung des Mocks
- BonusHistoryViewModel
- BonusHistoryView
- Anbindung im bestehenden BonusProgramComposer als neuer Navigationspfad
```

◄ **Claude Code:**

- Liest Atom/Features/Mailbox/ als Referenz
- Erweitert das bestehende BonusProgramRepository-Protokoll (fügt fetchHistory() hinzu)
- Erweitert BonusProgramRepositoryLive und BonusProgramRepositoryMock

- Erstellt BonusHistoryViewModel.swift und BonusHistoryView.swift als neue Dateien
- Bindet die neue View in den bestehenden Composer ein

Du siehst in Claude Code welche Dateien geändert und welche neu erstellt wurden.

Schritt 4 – Du prüfst die Änderungen

✎ **Du prüfst in Xcode:**

- Compiliert alles ohne Fehler?
- Sind die bestehenden Tests noch grün?
- Entspricht die neue View dem Design-System (ATUI-Komponenten verwendet)?

9.2 Legacy-Code: Was tun wenn du auf ATLegacy stößt?

Wenn eine Aufgabe dich in ATLegacy führt: kein neuer Code dort. Aber du musst oft verstehen was dort passiert, bevor du weißt wie du es umgehst.

► **Du gibst ein:**

```
Ich muss die Funktionalität von ATLegacy/Sources/PolicyManager.swift
verwenden, darf aber keinen Code in ATLegacy schreiben.

Analysiere PolicyManager:
1. Was macht diese Klasse?
2. Welche Methoden brauche ich konkret?
3. Gibt es bereits eine Atom-Entsprechung?
4. Wie kann ich die benötigte Funktionalität nutzen ohne neuen Code in ATLegacy zu schreiben?
```

◄ **Claude Code liest PolicyManager.swift** und gibt dir eine Einschätzung:

- Beschreibung der Klasse und ihrer Methoden
- Ob es eine Atom-Entsprechung gibt
- Einen konkreten Vorschlag: z.B. „Rufe die Methode aus dem bestehenden Atom-Kontext auf" oder „Erstelle einen dünnen Wrapper in Atom der ATLegacy intern nutzt"

✎ **Du entscheidest** welchen Weg du gehst, basierend auf der Analyse.

9.3 Refactoring: Legacy-Klasse migrieren

Beispiel: DashboardViewController aus ATLegacy soll zu einem Atom-Composer werden.

► **Du gibst ein:**

```
Ich möchte DashboardViewController aus ATLegacy zu einem
DashboardComposer in Atom migrieren.

Erstelle mir zunächst nur den Plan – noch keinen Code:
1. Einen schrittweisen Migrationsplan ohne Funktionsverlust
2. Welche Tests ich vor der Migration schreiben muss (Safety Net)
```

```
3. Eine Strategie beide Implementierungen parallel laufen zu lassen
   bis die Migration vollständig ist
4. Risiken und wie wir sie minimieren

Erst nach meiner Bestätigung des Plans fangen wir mit dem Code an.
```

◀ **Claude Code liest DashboardViewController**, analysiert Abhängigkeiten und erstellt einen strukturierten Migrationsplan als Text – noch keine Codeänderungen.

✎ **Du liest den Plan** und gibst ihn frei (oder verlangst Anpassungen).

▶ **Du gibst ein:**

```
Plan sieht gut aus. Fange mit Schritt 1 an: Schreibe zuerst die Tests
für die bestehende Funktionalität.
```

◀ **Claude Code schreibt zuerst Tests** für den bestehenden Code, damit Regressions auffallen – erst dann kommt die Migration.

9.4 Zusammenfassung: Was du tust, was Claude Code tut

Überblick: wer was übernimmt.

Schritt	Wer	Was
Analyse beauftragen	**Du**	Prompt mit Feature-Name und was du wissen willst
Analyse durchführen	**Claude Code**	Dateien lesen, Patterns erkennen, Bericht schreiben
Analyse bewerten	**Du**	Lesen und korrigieren wenn nötig
Erweiterung beauftragen	**Du**	Prompt mit konkreter Anforderung und API
Code schreiben	**Claude Code**	Bestehende Dateien erweitern, neue anlegen
Ergebnis prüfen	**Du**	Xcode öffnen, compilieren, Tests laufen lassen
Legacy-Strategie	**Du entscheidest**	Claude Code gibt Optionen, du wählst
Refactoring-Plan	**Claude Code**	Du bestätigst bevor Code geändert wird

---

14. UnitTest-Strategie

Das Testkonzept für Atomium von Grund auf: was getestet wird, was nicht, wie Tests aufgebaut sind, was sut und Mocks sind – und wie Claude Code Tests generiert. Auch wer noch keine XCTest-Tests geschrieben hat, findet hier einen direkten Einstieg.

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

10.1 Grundlagen – Was getestet wird, was nicht, und warum

Bevor Claude Code auch nur eine Testzeile schreibt, muss klar sein, was überhaupt getestet wird. In Atomium gilt folgende Aufteilung:

Was wird getestet	Warum
RepositoryLive	Prüft ob der Netzwerkaufruf korrekt läuft und ob API-Antworten richtig ins Domain-Modell übersetzt werden
ViewModel	Prüft ob sich der UI-State korrekt ändert (isLoading, Daten, Fehler) wenn Methoden aufgerufen werden
Mapper	Prüft ob API-Modelle korrekt in Domain-Modelle umgewandelt werden – reine Ein/Ausgabe, keine Netzwerke
View	Wird **nicht** automatisiert getestet – wird manuell in Xcode Preview und auf dem Gerät geprüft
Composer / Environment	Wird **nicht** unit-getestet – wird durch Integration getestet

Was sind Mocks?

Ein Mock ist eine Ersatz-Implementierung eines Protokolls, die im Test kontrolliertes Verhalten liefert. Statt eines echten Netzwerkaufrufs gibt der Mock sofort ein vorbereitetes Ergebnis zurück.

```
Echter Ablauf:   ViewModel → RepositoryLive → APIClient → Netzwerk → Antwort
Test-Ablauf:     ViewModel → RepositoryMock (gibt sofort .success([...]) zurück)
```

Damit testest du das ViewModel isoliert – ohne Netzwerk, ohne Server, ohne Wartezeit.

Was ist sut?

sut steht für „System Under Test" – die Klasse, die du gerade testest. Der Name ist Konvention im iOS-Umfeld. In einem ViewModel-Test ist das ViewModel der sut. In einem Repository-Test ist es das Repository.

Was ist Given / When / Then?

Jeder Test ist in drei Teile gegliedert:

- **Given** – Ausgangszustand vorbereiten (Mock konfigurieren, Testdaten setzen)
- **When** – Die Methode aufrufen, die getestet werden soll
- **Then** – Das Ergebnis prüfen (XCTAssertEqual, XCTAssertNil, etc.)

Diese drei Kommentare stehen in jedem Test. Kein Test ohne sie.

10.2 Das Grundgerüst – jeder Test sieht so aus

Alle Tests im Projekt folgen exakt dieser Struktur. Claude Code generiert sie automatisch in diesem Format.

```swift
final class CertificateViewModelTests: XCTestCase {

    // MARK: - Properties
    // sut = die Klasse die wir testen
    // repository = der Mock, der sut's Abhängigkeit ersetzt
    private var sut: CertificateViewModel!
    private var repository: CertificateRepositoryMock!

    // MARK: - Lifecycle
```

```swift
    // setUp läuft vor JEDEM einzelnen Test – frischer Zustand, kein State-Überlauf
    override func setUp() async throws {
        try await super.setUp()
        repository = CertificateRepositoryMock()
        sut = CertificateViewModel(repository: repository)
    }

    // tearDown läuft nach JEDEM Test – Speicher freigeben, Retain-Cycles vermeiden
    override func tearDown() async throws {
        sut = nil
        repository = nil
        try await super.tearDown()
    }

    // MARK: - loadCertificates
    func test_loadCertificates_success_populatesCertificates() async throws {
        // Given – Mock soll Erfolg zurückgeben
        repository.fetchResult = .success([.fixture(id: "1"), .fixture(id: "2")])

        // When – die zu testende Methode aufrufen
        await sut.loadCertificates()

        // Then – erwartete Auswirkung prüfen
        XCTAssertEqual(sut.certificates.count, 2)
        XCTAssertFalse(sut.isLoading)
        XCTAssertNil(sut.error)
    }
}
```

Warum async throws in setUp/tearDown? Weil das ViewModel @MainActor ist und der Mock asynchron aufgerufen werden kann. async throws in setUp/tearDown ist Pflicht sobald der Code async ist – sonst gibt es Threading-Probleme im Test.

Warum final class, nicht struct? XCTestCase erfordert eine Klasse. final verhindert unbeabsichtigte Vererbung.

Naming-Pflicht:

```
test_[methode]_[bedingung]_[ergebnis]

test_loadCertificates_success_populatesCertificates  ✅
test_loadCertificates_whenNetworkFails_setsError     ✅
test_1                                               ❌
testLaden                                            ❌
testFetchData                                        ❌
```

10.3 Repository-Tests – was sie prüfen und warum

Das Repository ist der Übergang zwischen Netzwerk und Domain-Modell. Es gibt zwei Dinge zu testen:

1. **Happy Path:** Kommt eine gültige API-Antwort, wird sie korrekt ins Domain-Modell gemappt
2. **Error Path:** Kommt ein Fehler vom Netzwerk, wird er korrekt weitergegeben

Der APIClient wird hier mit einem eigenen Mock ersetzt – nicht der Repository-Mock (das wäre eine Ebene zu hoch).

```
final class CertificateRepositoryLiveTests: XCTestCase {
  private var sut: CertificateRepositoryLive!
  private var apiClient: CertificateAPIClientMock!   // ← APIClient wird gemockt, nicht Repositor
y

  override func setUp() async throws {
    try await super.setUp()
    apiClient = CertificateAPIClientMock()
    sut = CertificateRepositoryLive(apiClient: apiClient)
  }

  override func tearDown() async throws {
    sut = nil
    apiClient = nil
    try await super.tearDown()
  }

  // MARK: - fetchCertificates

  // Test 1: Erfolgsfall – API antwortet, Mapper läuft, Domain-Modelle kommen raus
  func test_fetchCertificates_success_returnsMappedCertificates() async throws {
    // Given – API-Mock liefert Fixture-Daten
    apiClient.getCertificatesResult = .success(CertificateAPIResponse.fixtures)

    // When
    let result = try await sut.fetchCertificates()

    // Then – Anzahl stimmt, Typen wurden korrekt gemappt
    XCTAssertEqual(result.count, 3)
    XCTAssertEqual(result.first?.type, .membershipProof)
  }

  // Test 2: Fehlerfall – Netzwerkfehler wird durchgereicht
  func test_fetchCertificates_networkError_throwsError() async throws {
    // Given – API-Mock simuliert Netzwerkausfall
    apiClient.getCertificatesResult = .failure(URLError(.notConnectedToInternet))

    // When / Then – Repository muss den Fehler weiterwerfen
    await XCTAssertThrowsErrorAsync(
      try await sut.fetchCertificates()
```

```
        }
    }
}
```

Was hier NICHT getestet wird:

- Der Inhalt des Mappings – das ist Aufgabe der Mapper-Tests (10.5)
- Die View-Reaktion auf Fehler – das ist Aufgabe der ViewModel-Tests (10.4)

10.4 ViewModel-Tests – was sie prüfen und warum

Das ViewModel ist für den UI-State zuständig. Tests prüfen ausschließlich: Verändert sich der State korrekt wenn eine Methode aufgerufen wird?

Das Repository wird hier mit dem RepositoryMock ersetzt – kein Netzwerk, kein API-Client.

Was in jedem ViewModel getestet werden muss:

- isLoading ist true während einer laufenden Anfrage und false danach
- error ist nil bei Erfolg und gesetzt bei Fehler
- Die Daten-Property (certificates, points etc.) enthält nach Erfolg die richtigen Daten
- Wenn eine Aktion einen ID-Zustand hat (z.B. downloadingID), wird er korrekt gesetzt und zurückgesetzt

```
@MainActor                    // ← weil CertificateViewModel @MainActor ist
final class CertificateViewModelTests: XCTestCase {
    private var sut: CertificateViewModel!
    private var repository: CertificateRepositoryMock!

    override func setUp() async throws {
        try await super.setUp()
        repository = CertificateRepositoryMock()
        sut = CertificateViewModel(repository: repository)
    }

    override func tearDown() async throws {
        sut = nil
        repository = nil
        try await super.tearDown()
    }

    // MARK: - loadCertificates

    // Test: Erfolgsfall setzt Daten und beendet Loading
    func test_loadCertificates_success_populatesCertificates() async {
        // Given
        repository.fetchResult = .success([.fixture(id: "1"), .fixture(id: "2")])

        // When
```

```swift
    await sut.loadCertificates()

    // Then
    XCTAssertEqual(sut.certificates.count, 2)
    XCTAssertFalse(sut.isLoading)
    XCTAssertNil(sut.error)
}

// Test: Fehlerfall setzt error und beendet Loading
func test_loadCertificates_networkError_setsError() async {
    // Given
    repository.fetchResult = .failure(URLError(.notConnectedToInternet))

    // When
    await sut.loadCertificates()

    // Then
    XCTAssertNotNil(sut.error)
    XCTAssertTrue(sut.certificates.isEmpty)
    XCTAssertFalse(sut.isLoading)
}

// MARK: - download

// Test: Nach erfolgreichem Download wird downloadingID zurückgesetzt
func test_download_success_clearsDownloadingID() async {
    // Given
    let certificate = Certificate.fixture(id: "cert-123")
    repository.downloadResult = .success(Data())

    // When
    await sut.download(certificate: certificate)

    // Then – ID wird nach Abschluss wieder nil
    XCTAssertNil(sut.downloadingID)
    // Repository wurde mit der richtigen ID aufgerufen
    XCTAssertEqual(repository.lastDownloadedID, "cert-123")
}

// Test: Fehler beim Download setzt error
func test_download_failure_setsError() async {
    // Given
    let certificate = Certificate.fixture(id: "cert-456")
    repository.downloadResult = .failure(URLError(.cannotConnectToHost))

    // When
    await sut.download(certificate: certificate)
```

```swift
        // Then
        XCTAssertNotNil(sut.error)
        XCTAssertNil(sut.downloadingID)
    }
}
```

Warum @MainActor auf der Testklasse? Das ViewModel ist @MainActor-isoliert. Wenn du im Test Properties wie sut.certificates liest, musst du das ebenfalls auf dem MainActor tun – sonst gibt es Compiler-Warnings oder Abstürze. @MainActor auf der Testklasse löst das automatisch.

10.5 Mapper-Tests – die einfachsten Tests

Der Mapper hat keine Abhängigkeiten. Er nimmt ein API-Modell entgegen und gibt ein Domain-Modell zurück. Das ist reine Eingabe/Ausgabe – kein Mock, kein async, kein setUp nötig.

Was getestet wird:

- Werden alle Felder korrekt übertragen?
- Werden Enum-Strings korrekt geparst?
- Was passiert bei ungültigen Werten (z.B. unbekannter Typ-String)?

```swift
final class CertificateMapperTests: XCTestCase {

    // setUp und tearDown nicht nötig – kein State, kein Mock

    func test_map_validResponse_returnsCorrectDomainModel() {
        // Given – ein typisches API-Response-Objekt
        let apiModel = CertificateAPIModel(
            id: "abc-123",
            title: "Mitgliedsbescheinigung",
            issuedAt: "2025-01-15T10:00:00Z",
            type: "MEMBERSHIP_PROOF",
            downloadURL: "https://api.pkk.de/docs/abc-123"
        )

        // When
        let result = CertificateMapper.map(apiModel)

        // Then – alle Felder stimmen
        XCTAssertEqual(result.id, "abc-123")
        XCTAssertEqual(result.title, "Mitgliedsbescheinigung")
        XCTAssertEqual(result.type, .membershipProof)   // String wurde zu Enum geparst
        XCTAssertEqual(result.downloadURL.absoluteString, "https://api.pkk.de/docs/abc-123")
    }

    func test_map_unknownType_fallsBackToDefault() {
        // Given – API schickt einen unbekannten Typ-String
        let apiModel = CertificateAPIModel(
```

```swift
        id: "xyz",
        title: "Unbekannt",
        issuedAt: "2025-01-15T10:00:00Z",
        type: "UNKNOWN_FUTURE_TYPE",
        downloadURL: "https://api.pkk.de/docs/xyz"
    )

    // When
    let result = CertificateMapper.map(apiModel)

    // Then – App crasht nicht, Fallback-Wert gesetzt
    XCTAssertEqual(result.id, "xyz")
    XCTAssertEqual(result.type, .other)   // oder was immer der Fallback ist
  }
}
```

10.6 Fixtures – warum und wie

Was sind Fixtures?

Fixtures sind vorgefertigte Testinstanzen eines Modells mit sinnvollen Standardwerten. Sie lösen ein konkretes Problem: Wenn du im Test ein Certificate-Objekt brauchst, willst du nicht jedes Mal alle Pflichtfelder hinschreiben.

Ohne Fixture – nervig und unlesbar:

```swift
let cert = Certificate(
  id: "1",
  title: "Test",
  issuedAt: Date(),
  type: .membershipProof,
  downloadURL: URL(string: "https://test.de")!
)
```

Mit Fixture – klar und fokussiert:

```swift
// Nur das ändern was für den Test relevant ist:
let cert = Certificate.fixture(id: "cert-123")
```

Wo Fixtures liegen:

```
Atom/Features/Certificate/Tests/Fixtures/Certificate+Fixture.swift
```

Fixtures sind nur für Tests. Sie liegen im Tests-Ordner, nie im Production-Code.

Wie eine Fixture-Datei aussieht:

```swift
// Certificate+Fixture.swift
extension Certificate {
  // Alle Parameter haben sinnvolle Defaults.
  // Im Test überschreibst du nur, was für den Test relevant ist.
  static func fixture(
```

```
    id: String = "fixture-id",
    title: String = "Mitgliedsbescheinigung",
    type: CertificateType = .membershipProof,
    issuedAt: Date = Date(timeIntervalSince1970: 1_700_000_000)  // fester Wert, kein Date.now
!
  ) -> Certificate {
    Certificate(
      id: id,
      title: title,
      issuedAt: issuedAt,
      type: type,
      downloadURL: URL(string: "https://api.pkk.de/test/\(id)")!
    )
  }

  // Fertige Liste für Tests die mehrere Elemente brauchen
  static let testList: [Certificate] = [
    .fixture(id: "1", type: .membershipProof),
    .fixture(id: "2", type: .taxStatement),
    .fixture(id: "3", type: .treatmentRecord)
  ]
}
```

Warum kein Date.now in Fixtures? Date.now liefert bei jedem Test-Lauf einen anderen Wert. Das macht Tests flaky – sie schlagen sporadisch fehl ohne erkennbaren Grund. Immer einen festen Zeitstempel verwenden.

10.7 Tests mit Claude Code erstellen

Hier ist der vollständige Workflow: was du eingibst, was Claude Code macht, was du prüfst.

Variante A: Tests für ein neues Feature (zusammen mit dem Feature)

Wenn du ein neues Feature mit dem Prompt aus Kapitel 8 erstellst, einfach ergänzen:

► **Du gibst ein** (Ergänzung zum Feature-Prompt):

```
... und erstelle vollständige UnitTests für:
- CertificateRepositoryLive (mit CertificateAPIClientMock)
- CertificateViewModel (mit CertificateRepositoryMock)
- CertificateMapper

Verwende unser Standard-Pattern aus .claude/skills/unittest.md:
- setUp/tearDown async throws
- @MainActor auf ViewModel-Testklasse
- Given/When/Then Kommentare
- Naming: test_[methode]_[bedingung]_[ergebnis]
- Fixtures in Tests/Fixtures/Certificate+Fixture.swift
```

◀ **Claude Code erstellt** alle Testdateien plus die Fixture-Datei in einem Durchgang.

Variante B: Tests für bestehenden Code nachträglich erstellen

▶ **Du gibst ein:**

```
Erstelle vollständige UnitTests für CertificateViewModel.
Die Datei liegt in Atom/Features/Certificate/UI/CertificateViewModel.swift.
Der Mock liegt in Atom/Features/Certificate/Repository/CertificateRepositoryMock.swift.

Anforderungen:
- Alle öffentlichen Methoden testen
- Jeden Methoden-Aufruf: Success-Pfad UND Error-Pfad
- isLoading-State testen (true während Laden, false danach)
- Fixtures falls noch keine existieren anlegen
- Pattern aus .claude/skills/unittest.md
```

◀ **Claude Code:**

- Liest CertificateViewModel.swift und identifiziert alle öffentlichen Methoden
- Liest CertificateRepositoryMock.swift um die verfügbaren Mock-Properties zu kennen
- Erstellt Tests/CertificateViewModelTests.swift mit allen Tests
- Erstellt Tests/Fixtures/Certificate+Fixture.swift falls noch nicht vorhanden

---

Variante C: Fehlende Tests für ein ganzes Feature finden

▶ **Du gibst ein:**

```
/check-tests Certificate
```

◀ **Claude Code** analysiert alle Methoden in CertificateViewModel.swift, CertificateRepositoryLive.swift und CertificateMapper.swift und vergleicht sie mit den vorhandenen Tests in Tests/. Es zeigt dir:

- Welche Methoden bereits getestet sind ✅
- Welche Methoden noch keine Tests haben ❌
- Welche Fehlerszenarien fehlen ⚠️

▶ **Du gibst ein** wenn du die fehlenden Tests haben willst:

```
Erstelle die fehlenden Tests aus dem letzten Check.
```

◀ **Claude Code** schreibt nur die fehlenden Tests – ohne die vorhandenen anzufassen.

Variante D: Test-Mock erstellen (wenn noch kein Mock existiert)

▶ **Du gibst ein:**

```
/make-mock CertificateRepository
```

◀ **Claude Code** liest das Protokoll CertificateRepository.swift und erstellt CertificateRepositoryMock.swift mit:

- Result<T, Error> Property für jede async throws Methode
- callCount-Zähler für jede Methode
- lastParameter-Capture für jeden Methodenaufruf

✎ **Du prüfst** ob der Mock alle Methoden hat und ob die Standardwerte sinnvoll sind.

10.8 Zusammenfassung: Was du tust, was Claude Code tut

Überblick: wer was tut.

Schritt	Wer	Was
Tests beauftragen	**Du**	Prompt mit ViewModel/Repository-Name und Anforderungen
Methoden lesen	**Claude Code**	Swift-Datei analysieren, alle public/internal Methoden finden
Mock lesen	**Claude Code**	Mock-Datei lesen, verfügbare Result-Properties kennen
Tests schreiben	**Claude Code**	Testdatei erstellen: alle Methoden, Success + Error, Given/When/Then
Fixtures erstellen	**Claude Code**	Falls nicht vorhanden, Fixture-Datei anlegen
Tests prüfen	**Du**	In Xcode ausführen: alle grün? Deckt es ab was du erwartest?
Fehlende Tests finden	**/check-tests**	Command analysiert Gap zwischen Methoden und vorhandenen Tests
Mock erstellen	**/make-mock**	Command generiert Mock aus Protokoll-Definition

---

15. Qualitätssicherung im Alltag

Kein neuer Inhalt – nur die Frage: wann setzt du welchen Command ein? Hier kommt die Antwort.

12.1 Vor einem Merge-Request

► **Im Chat eingeben:**

```
/review-feature [FeatureName]
```

Das ist der einzige Command der vor jedem Merge läuft. Er fasst alle anderen Checks zusammen.

◄ **Claude Code gibt einen Report zurück.** Alle ❌-Punkte müssen vor dem Merge behoben werden.

12.2 Während der Feature-Entwicklung

Welchen Command in welcher Situation – direkt während du baust, nicht erst kurz vor dem Merge.

Situation	Command
Neuen async Code geschrieben	/check-async Atom/Features/[Feature]

Situation	Command
ViewModel erweitert	/check-threading [ViewModelName]
API-Anbindung fertig	/check-dsgvo Atom/Features/[Feature]/Repository
Feature-Entwicklung abgeschlossen	/check-tests [FeatureName]
Wöchentlicher Gesundheitscheck	/check-all

12.3 Pre-Merge-Workflow – Schritt für Schritt

Obligatorisch vor jedem PR. Fasst alle 7 Einzelchecks in einem Aufruf zusammen.

▶ **Du gibst ein:**

```
/review-feature Certificate
```

◀ **Claude Code führt alle 7 Checks aus** und gibt den Report zurück (ca. 1-3 Minuten je nach Feature-Größe).

✎ **Du liest den Report.** Für jeden ❌-Punkt:

▶ **Du gibst ein:**

```
Bitte behebe den DSGVO-Fehler in CertificateRepositoryLive.swift Zeile 45.
```

◀ **Claude Code ändert** nur die betroffene Stelle und zeigt den Diff.

▶ **Du gibst ein** nach allen Fixes:

```
/review-feature Certificate
```

◀ **Claude Code prüft erneut.** Alle Punkte grün → Merge freigeben.

12.4 Commands anlegen – einmalige Einrichtung

Wenn du alle Commands auf einmal anlegen willst:

▶ **Im Chat eingeben:**

```
Erstelle alle Command-Dateien aus Kapitel 11.3 dieser Anleitung
unter .claude/commands/. Jede Datei soll den vollständigen
Prompt-Inhalt aus der Dokumentation enthalten.
```

◀ **Claude Code erstellt** alle 14 Command-Dateien direkt ins Projekt.

✎ **Danach:** Claude Code neu starten (Ctrl+C im Terminal, dann claude). Alle Commands sind dann per Tab-Completion verfügbar.

---

16. Dokumentationsstandard

Was wird dokumentiert, was nicht, wie sieht korrekte Apple-Dokumentation (///) aus – und wie generiert Claude Code sie. Kein manuelles Schreiben mehr.

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

13.1 Was wird dokumentiert – und was nicht?

Bevor du Dokumentation erzeugst: klar sein was überhaupt dokumentiert werden soll. Alles andere ist Boilerplate.

Diese Deklarationen immer dokumentieren:

- Protokoll-Definitionen (jeder der sie implementiert muss sie verstehen)
- public und internal Typen und Methoden
- Parameter die nicht selbsterklärend sind
- Fehlerszenarien die der Aufrufer kennen muss

Diese Deklarationen nie dokumentieren:

- private Implementierungsdetails
- Selbsterklärende Properties: var isLoading: Bool braucht keinen Kommentar
- Standard SwiftUI body-Implementierungen

13.2 Wie sieht korrekter Apple-Stil aus?

Dokumentation in Atomium folgt dem Apple-Stil mit ///. Keine /* */ Blöcke, kein // MARK: für Dokumentation.

So sieht es richtig aus:

```
/// Fetches all available certificates for the current user, ordered by issue date.
///
/// - Returns: An array of ``Certificate`` objects, empty if none exist.
/// - Throws: ``CertificateError/unauthorized`` if the session has expired.
func fetchCertificates() async throws -> [Certificate]

/// Downloads the raw PDF data for a certificate.
///
/// - Parameter id: The unique certificate identifier from ``Certificate/id``.
/// - Throws: ``CertificateError/notFound`` if the certificate no longer exists.
func downloadCertificate(id: String) async throws -> Data
```

So sieht es falsch aus – vermeide das:

```
// Falsch – beschreibt das Was (offensichtlich), nicht das Warum:
/// This function loads the data from the repository and updates the view model.

// Falsch – reines Boilerplate, sagt nichts:
/// CertificateViewModel is a view model for the Certificate feature.

// Richtig – erklärt einen nicht-offensichtlichen Aspekt:
/// Caches the result for 5 minutes to avoid redundant API calls during
/// rapid back-and-forth navigation in the certificate list.
```

Faustregel: Wenn das Löschen des Kommentars einen informierten Leser nicht verwirren würde, gehört er nicht hin.

13.3 Wie Claude Code Dokumentation erzeugt

Du gibst an was dokumentiert werden soll – Claude Code generiert es nach dem richtigen Stil.

Einzelne Datei dokumentieren:

► **Im Chat eingeben:**

```
/add-docs Atom/Features/Certificate/Repository/CertificateRepository.swift
```

◄ **Claude Code liest** die Datei, identifiziert alle public und internal Deklarationen und gibt die vollständige Datei mit eingefügten Kommentaren zurück.

✎ **Du prüfst** den generierten Inhalt in Xcode und übernimmst ihn. Claude Code ändert die Datei nicht automatisch – du siehst sie zuerst im Chat, dann speicherst du sie selbst oder sagst „Schreib die Datei direkt."

Feature komplett dokumentieren:

► **Im Chat eingeben:**

```
Dokumentiere alle public und internal Deklarationen in Atom/Features/Certificate/.
Verwende Apple-Stil (/// Kommentare). Überspringe private Deklarationen und
selbsterklärende Properties wie isLoading oder certificates.
Schreib jede geänderte Datei direkt.
```

◄ **Claude Code geht** alle Dateien im Feature-Ordner durch und fügt Kommentare ein, Datei für Datei.

✎ **Du öffnest** danach Xcode und prüfst ob die Kommentare korrekt sind und keinen Inhalt verfälscht haben.

Vorhandene Dokumentation auf Qualität prüfen:

► **Im Chat eingeben:**

```
Prüfe die Dokumentation in Atom/Features/BonusProgram/ auf:
- Fehlende Kommentare bei public/internal Deklarationen
- Boilerplate-Kommentare die nichts sagen
- Falsche Beschreibungen (beschreibt das Was statt das Warum)
Liste jeden Fund mit Datei und Zeile.
```

◄ **Claude Code gibt** eine priorisierte Liste aller Dokumentationsprobleme zurück – du entscheidest welche du behebst.

---

17. Code-Stil und Formatierung

Legende: ► Du gibst ein = Eingabe im Claude Code Chat · ◄ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

Was automatisch läuft und was Claude Code macht – das ist hier der Schlüssel.

14.1 Werkzeuge und wer sie ausführt

Was läuft automatisch – und was stößt du selbst an:

Werkzeug	Wer führt es aus	Wann
SwiftLint	CI automatisch	Bei jedem Pull Request
SwiftFormat	CI automatisch (check-only)	Bei jedem Pull Request
Claude Code	Du im Chat	Wenn du tiefer analysieren willst
Xcode	Du manuell	Beim Entwickeln

SwiftLint (im Projektroot: .swiftlint.yml) prüft Stil-Regeln – Zeilenlänge, Funktionsgröße, Naming. Es läuft automatisch in CI und schlägt fehl wenn Regeln verletzt sind. Du musst nichts manuell starten.

SwiftFormat formatiert Code automatisch. In CI läuft es im „check only" Modus – es formatiert nicht selbst, sondern meldet nur Abweichungen.

Die wichtigsten konfigurierten Grenzen:

```yaml
line_length:
  warning: 120   # Warnung ab 120 Zeichen
  error: 150     # Fehler ab 150 Zeichen

function_body_length:
  warning: 30    # Funktion zu lang ab 30 Zeilen
  error: 50

file_length:
  warning: 300   # Datei zu lang ab 300 Zeilen
  error: 500
```

14.2 Pflicht-Struktur jeder Swift-Datei

Alle Dateien in Atom/ folgen dieser MARK-Struktur. Claude Code generiert sie automatisch.

```swift
struct MeineKlasse {

   // MARK: - Types
   // (verschachtelte Typen, Enums die nur hier gebraucht werden)

   // MARK: - Properties
   // (alle Properties zusammen, erst public dann private)

   // MARK: - Lifecycle
   // (init, deinit)

   // MARK: - Public Interface
   // (öffentliche Methoden, semantisch beschriftet z.B. "// MARK: - Data Loading")
```

```swift
    // MARK: - Private
    // (private Hilfsmethoden)
}
```

Große Views aufteilen: Wenn eine View-Datei über 150 Zeilen geht, werden Subviews in Extensions ausgelagert:

```swift
// CertificateView+Subviews.swift ← eigene Datei für die View-Teile
private extension CertificateView {
    var certificateList: some View { ... }
    var emptyState: some View { ... }
}

// CertificateView+Accessibility.swift ← Accessibility separat
private extension CertificateView {
    func accessibilityLabel(for certificate: Certificate) -> String { ... }
}
```

14.3 Code-Stil mit Claude Code prüfen

SwiftLint prüft Regeln – aber erklärt nicht warum etwas ein Problem ist oder wie man es am besten aufteilt. Das macht Claude Code.

► **Im Chat eingeben:**

```
Prüfe Atom/Features/BonusProgram/UI/BonusProgramView.swift auf:
- Funktionen länger als 30 Zeilen
- Dateilänge (über 300 Zeilen)
- Korrekte MARK-Struktur
- Auskommentierte Code-Blöcke
- Kein dead code

Schlage konkrete Aufteilungen in Extensions vor wenn nötig.
```

◄ **Claude Code liest** die Datei und gibt dir eine Liste der Probleme mit konkreten Vorschlägen: welche Subview-Extensions du anlegen solltest, welche Methoden ausgelagert werden können.

✎ **Du entscheidest** welche Vorschläge du übernimmst.

► **Du gibst ein** wenn du die Aufteilung haben willst:

```
Führe die vorgeschlagene Aufteilung durch. Erstelle die Extensions-Dateien direkt.
```

◄ **Claude Code erstellt** die neuen Dateien und passt die Hauptdatei an.

---

18. Automatisierung

Legende: ► Du gibst ein = Eingabe im Claude Code Chat · ◄ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

Zwei Arten von Automatisierung: was der CI-Server bei jedem Pull Request selbst ausführt – und was du selbst in Claude Code anstößt.

15.1 CI-Pipeline – automatische Prüfungen bei jedem Pull Request

Was ist CI? Wenn du einen Pull Request pushst, startet automatisch ein Server der folgende Befehle ausführt. Ergebnis: grünes Häkchen am PR = alles okay, rotes X = irgendetwas schlägt fehl.

Du musst diese Befehle nicht selbst starten und nicht im Terminal eintippen.

```
# SwiftLint – Stil-Prüfung (schlägt fehl bei Verstößen)
swiftlint lint --strict

# SwiftFormat – Format-Check (meldet nur, ändert nichts)
swiftformat --lint .

# Unit Tests (schlägt fehl wenn Tests rot sind)
xcodebuild test \
 -project Atomium.xcodeproj \
 -scheme AtomiumTests \
 -destination 'platform=iOS Simulator,name=iPhone 15'
```

Wenn CI schlägt fehl: Benachrichtigung, Fehler lokal beheben, erneut pushen.

Build lokal testen (im Terminal, nicht im Chat):

```
xcodebuild -project Atomium.xcodeproj \
      -scheme Atomium \
      -destination 'platform=iOS Simulator,name=iPhone 15' \
      build
```

15.2 Wiederkehrende Claude-Code-Workflows

Diese Workflows startest du selbst: claude im Terminal → Enter → /command direkt in den Chat tippen. Kein Bash-Befehl.

Wöchentlicher Architektur-Check (jeden Montag):

► Im Chat eingeben:

```
/check-all
```

◄ **Claude Code scannt** alle Feature-Ordner in Atom/Features/ und gibt eine Tabelle zurück: welche Features vollständig sind und wo es Lücken gibt (fehlendes Mock, fehlende Tests, Legacy-Imports).

✎ **Du siehst auf einen Blick** welche Features Aufmerksamkeit brauchen und kannst gezielt nachbessern.

Nach Abschluss eines Sprint-Features:

► Im Chat eingeben:

```
/review-feature [FeatureName]
```

◀ **Claude Code führt** alle 7 Checks durch (Architektur, Async, Speicher, DSGVO, Security, Tests, Code-Qualität) und gibt einen strukturierten Report zurück.

✎ **Du gehst** die ❌-Punkte durch und behebst sie bevor du den PR öffnest.

Wenn ein neuer Entwickler anfängt (Onboarding-Automatisierung):

▶ **Im Chat eingeben:**

```
Ich bin neu im Team. Analysiere Atom/Features/ und erstelle mir
einen Überblick aller Features mit:
- Was jedes Feature macht (1 Satz)
- Welche Features vollständig sind (Composer, Tests, Mock)
- Welche Features noch Legacy-Abhängigkeiten haben
- Wo ich als Nächstes sinnvoll mitarbeiten kann
```

◀ **Claude Code liest** die gesamte Feature-Struktur und gibt eine priorisierte Übersicht zurück. Kein manuelles Lesen von 30 Dateien.

---

19. Debug-Architektur

Legende: ▶ Du gibst ein = Eingabe im Claude Code Chat · ◀ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

Das Debug-Menü ist ein eigenständiger Feature-Tree ausschließlich für #if DEBUG-Builds. Es erscheint nie in einem App-Store-Release.

16.1 Wo das Debug-Menü liegt und was es enthält

Wo jede Komponente liegt und welche Rolle sie hat – Claude Code liest diese Struktur automatisch wenn du ein Debug-Feature beauftragst.

```
Atom/Debug/
├── DebugMenuComposer.swift      ← Einstiegspunkt, verbindet alles
├── DebugMenuNavigator.swift    ← Navigation zwischen Debug-Features
├── DebugMenuDestination.swift   ← Alle möglichen Zielseiten
├── Overview/
│   └── DebugOverviewView.swift  ← Die Liste aller Debug-Optionen
└── Features/
    ├── LoginDebug/         ← FakeLogin, Session ohne echte Credentials
    ├── NetworkDebug/          ← Request-Logs, Mock-Endpoints aktivieren
    ├── DateDebug/          ← Systemdatum überschreiben (z.B. für Ablauf-Tests)
    └── FeatureFlags/         ← Features aktivieren/deaktivieren ohne Build
```

Wichtig: Jede Datei in Atom/Debug/ ist entweder mit #if DEBUG umschlossen oder liegt in einem Debug-only Target. Kein Debug-Code landet im Release-Build.

16.2 Neues Debug-Feature hinzufügen – mit Claude Code

Beispiel: Du willst eine Möglichkeit hinzufügen, für einzelne API-Endpoints Mock-Antworten zu erzwingen statt echte Netzwerkanfragen zu schicken.

► Im Chat eingeben:

```
Füge ein neues Debug-Feature "API-Response-Override" hinzu.
Es soll ermöglichen, für einzelne API-Endpoints eine vordefinierte
Mock-Antwort zu aktivieren, statt echte Netzwerkanfragen zu machen.

Erstelle:
1. NetworkDebugEnvironment mit Override-Logik (UserDefaults als Storage)
2. NetworkDebugView mit Liste der Endpoints und Toggle pro Endpoint
3. Neuer Case .networkDebug in DebugMenuDestination
4. Navigations-Eintrag in DebugMenuNavigator
5. View-Registrierung im DebugMenuComposer

Alle Dateien müssen mit #if DEBUG abgesichert sein.
```

◄ Claude Code liest die bestehende Debug-Struktur (DebugMenuDestination, DebugMenuNavigator, DebugMenuComposer), versteht das Pattern und erstellt alle 5 Dateien konsistent dazu.

✎ Du prüfst in Xcode:

- Kompiliert der Debug-Build ohne Fehler?
- Erscheint der neue Eintrag im Debug-Menü?
- Ist der neue Code nirgendwo im Release-Build sichtbar (Scheme: Release bauen und prüfen)?

16.3 Welche Debug-Komponenten existieren

Vorgefertigte Bausteine aus ATUI – instanziieren, nicht selbst schreiben. Claude Code kennt sie und setzt sie automatisch ein.

ActionTableCells – vorgefertigte SwiftUI-Komponenten aus ATUI für das Debug-Menü:

```
// Navigation zu einem Debug-Feature:
DebugActionCell(
  title: "Datum überschreiben",
  subtitle: currentDate.map { ISO8601DateFormatter().string(from: $0) },
  action: { navigator.navigateToDateOverride() }
)

// An/Aus-Schalter für ein Debug-Feature:
DebugToggleCell(
  title: "FakeLogin aktiviert",
  isOn: $environment.isFakeLoginEnabled
)

// Auswahl aus mehreren Optionen:
DebugPickerCell(
```

```
    title: "Aktiver Nutzer",
    options: FakeProfile.allCases,
    selection: $environment.activeProfile
)
```

Diese Komponenten kommen aus ATUI – du instanziierst sie in deiner DebugView, du schreibst sie nicht selbst.

16.4 FakeLogin – was es ist und wie man es nutzt

FakeLogin überspringt den echten IAM-Login und injiziert direkt ein Testprofil in die Session. Nur in Debug-Builds verfügbar.

► Im Chat eingeben wenn du FakeLogin für ein neues Profil erweitern willst:

```
Füge ein neues FakeProfile "TestFamilyAdmin" zum FakeLogin hinzu.
Es soll ein Profil mit Familien-Administratorrechten simulieren:
- 2 Familienmitglieder
- aktive Krankenversicherung
- offene Nachrichten im Postfach

Passe FakeProfile enum und die entsprechenden Fixture-Daten an.
```

◄ **Claude Code liest** die bestehende FakeLogin-Implementierung und ergänzt das neue Profil konsistent.

---

20. Onboarding neuer Entwickler

Drei Tage, klare Schritte – für wer gerade anfängt.

Legende: ► Du gibst ein = Eingabe im Claude Code Chat · ◄ Claude Code = was Claude Code tut · ✎ Du prüfst = manueller Schritt

---

Tag 1 – Setup und erstes Gespräch mit Claude Code

Ziel: Claude Code läuft im Projekt, du hast ein erstes Gefühl wie es antwortet und was es über Atomium weiß.

✎ **Schritt 1: Repo klonen und ins Verzeichnis wechseln** (im Terminal):

```
git clone [repo-url]
cd Atomium
```

✎ **Schritt 2: Claude Code starten** (im Terminal):

```
claude
```

Claude Code liest automatisch CLAUDE.md im Projektroot. Du musst nichts konfigurieren.

► **Schritt 3: Ersten Überblick holen** (im Claude Code Chat):

```
Gib mir einen Überblick über die Projektarchitektur von Atomium.
Erkläre mir:
```

1. Was sind die vier Frameworks und wofür ist jedes zuständig?
2. Was ist das Composer-Pattern und warum verwenden wir es?
3. Zeige mir ein konkretes Beispiel anhand des Bonusprogramm-Features.

◀ **Claude Code erklärt** die Architektur anhand der tatsächlichen Projektdateien – nicht aus der Theorie, sondern aus dem Code den es gerade liest.

✎ **Du liest** die Erklärung. Wenn etwas unklar ist, stelle direkt Rückfragen im selben Chat.

Tag 2 – Ein bestehendes Feature vollständig verstehen

Ziel: Du kannst ein bestehendes Feature vollständig lesen – Composer, ViewModel, Repository, Tests – und weißt wie Claude Code dir dabei hilft.

► **Im Chat eingeben:**

Ich bin neu im Team und möchte das Postfach-Feature vollständig verstehen.
Analysiere Atom/Features/Mailbox/ und erkläre mir:
1. Was macht dieses Feature aus Benutzersicht?
2. Welche Composer, ViewModels und Repositories gibt es?
3. Wie läuft die Navigation (sheet, push, TabBar)?
4. Welche Tests existieren und was decken sie ab?
5. Gibt es Abweichungen vom Standard-Pattern – und wenn ja warum?

◀ **Claude Code liest** alle Dateien in Atom/Features/Mailbox/ und gibt eine strukturierte Erklärung zurück. Du musst keine Datei selbst öffnen um zu verstehen wie es aufgebaut ist.

✎ **Du öffnest danach** ein oder zwei der erklärten Dateien in Xcode um das Gelesene mit dem tatsächlichen Code zu verknüpfen.

Tag 3 – Erste eigene kleine Änderung

Ziel: Erste echte Änderung am Code – mit Analyse, Plan und Review durch Claude Code. Der Ablauf ist derselbe wie für jede spätere Aufgabe.

► **Im Chat eingeben:**

Ich soll dem Postfach-Feature einen Ungelesen-Badge auf dem TabBar-Icon hinzufügen.
Der Badge soll die Anzahl ungelesener Nachrichten zeigen.

Analysiere das bestehende Postfach-Feature und erkläre mir:
1. Wo wird die ungelesene-Nachrichten-Anzahl aktuell verwaltet?
2. Wie kommuniziert das Postfach-Feature mit dem TabBar?
3. Was ist der minimale und sicherste Weg diese Änderung umzusetzen?

Noch keinen Code schreiben – erst den Plan zeigen.

◀ **Claude Code analysiert** die bestehende Struktur und gibt dir einen Plan mit konkreten Dateien und Schritten.

✎ **Du liest den Plan** und prüfst ob er zu deinem Verständnis aus Tag 2 passt.

► **Du gibst ein:**

Plan sieht gut aus. Setze Schritt 1 um.

◀ **Claude Code ändert** nur den spezifischen Teil – keine großflächigen Refactorings, kein unrequested Code.

✎ **Du prüfst in Xcode:** Kompiliert es? Erscheint der Badge korrekt?

Was Claude Code im Alltag ersetzt – und was nicht

Was Claude Code wirklich abnimmt – und wo du weiterhin selbst urteilen musst.

Claude Code macht:

- Neue Features von Grund auf generieren (Composer, Repository, Mock, Tests)
- Bestehenden Code erklären und analysieren
- Qualitätsprüfungen (DSGVO, Security, Async, Tests)
- Dokumentation schreiben
- Refactoring-Pläne erstellen und umsetzen

Claude Code macht nicht:

- Produktentscheidungen treffen (was gebaut wird)
- Architekturentscheidungen ohne Kontext aus CLAUDE.md
- PR-Reviews zwischen Menschen ersetzen
- DSGVO-Verantwortung übernehmen – das bleibt beim Team

Wenn Claude Code etwas falsch macht: Direkt im selben Chat korrigieren: „Das ist falsch weil..." Claude Code ändert nur den angesprochenen Teil.

Fazit

Atomium ist eine App mit klaren Verantwortlichkeiten: Atom für neue Features, ATCore für Infrastruktur, ATUI für Design. Das Composer-Pattern gibt jedem Feature eine vorhersehbare Struktur. Claude Code ist nützlich, wenn es diese Struktur kennt – und das setzt voraus, dass CLAUDE.md, Skills und Rules gepflegt werden.

Die wichtigste Regel: **Claude Code ist so gut wie das Wissen, das du ihm gibst.** Eine veraltete CLAUDE.md produziert veralteten Code. Eine fehlende DSGVO-Rule produziert unsicheren Code. Halte die Konfiguration aktuell, und Claude Code wird konsistenten, projektkonformen Code liefern.

Interne Dokumentation · Atomium iOS Team · Nicht für externe Weitergabe