# BKK Atomium Service App – Einheitliche App-Entwicklung im Team

# Inhaltsverzeichnis



Dieses Tutorial zeigt anhand eines konkreten Beispielprojekts, wie ein Entwicklungsteam mit Claude Code eine iOS-/iPadOS-App durchgängig und nachvollziehbar entwickelt: die BKK Atomium Service App einer fiktiven Krankenkasse. Die App verwendet bewusst das Composer-Pattern und trennt View, ViewModel, Interactor und Repository klar voneinander. Die beschriebenen Arbeitsabläufe, Regeln und Qualitätsstandards lassen sich jedoch auch auf Projekte mit anderen Architekturansätzen übertragen.


*BKK Atomium Service App auf iPhone und iPad*

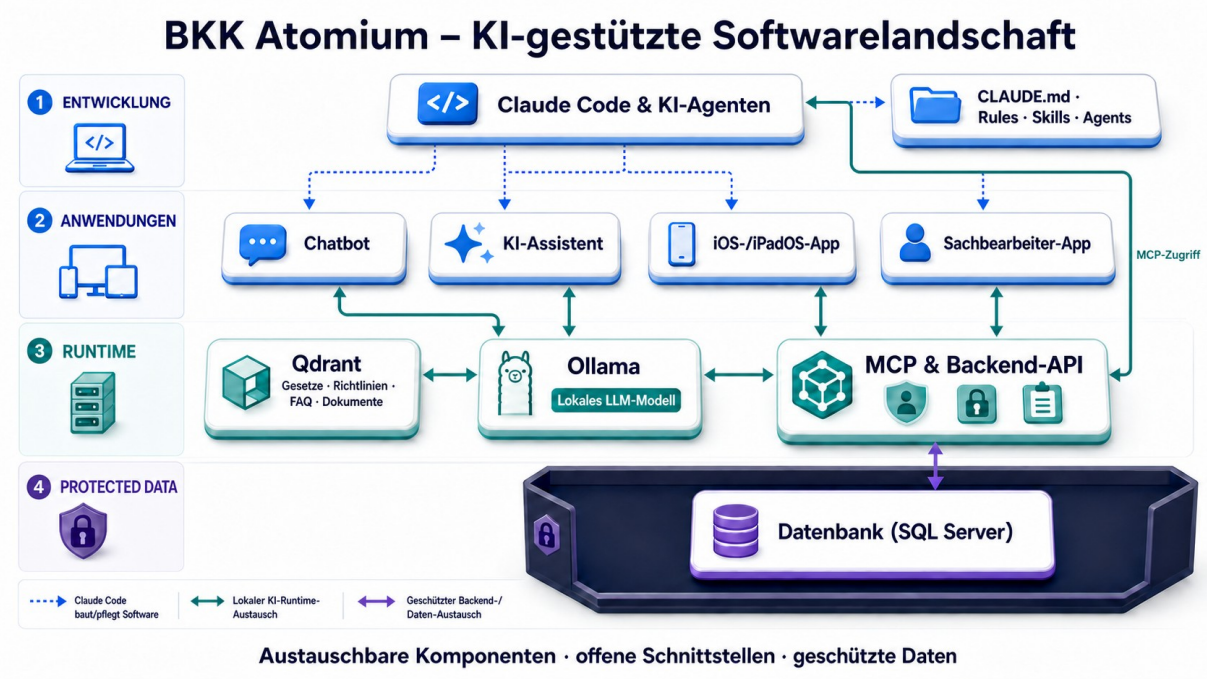
## 1. Projektüberblick

Die BKK Atomium Service App ist eine Demo-App einer fiktiven gesetzlichen Krankenkasse. Sie ist bewusst kein Startup-Prototyp und keine Spielwiese, sondern so aufgebaut, wie ein reales Krankenkassen-Team eine App planen würde: mit klarer Architektur, Testpflicht, Lokalisierung in neun Sprachen und einem Regelwerk, das beschreibt, was gebaut werden soll, bevor es gebaut wird.

Das unterscheidet dieses Projekt von den meisten Tutorial-Apps. Normalerweise entsteht der Funktionsumfang nebenbei, während man programmiert. Hier ist es umgekehrt: `.claude/rules/product-scope.md` legt verbindlich fest, welche Tabs, Screens und Formulare existieren sollen, bevor Claude Code einen Handgriff Code schreibt. Das Tutorial folgt demselben Prinzip. Es beschreibt, was im Code wirklich steht, und kennzeichnet offene Lücken als das, was sie sind: dokumentierte Restarbeit, keine verschwiegene Unvollständigkeit.

Dieses Tutorial steht nicht für sich allein. Es gehört zu einer Reihe weiterer Tutorials desselben Autors, die denselben Claude-Code-Workflow an anderen, ebenfalls bereits existierenden Produkten zeigen:

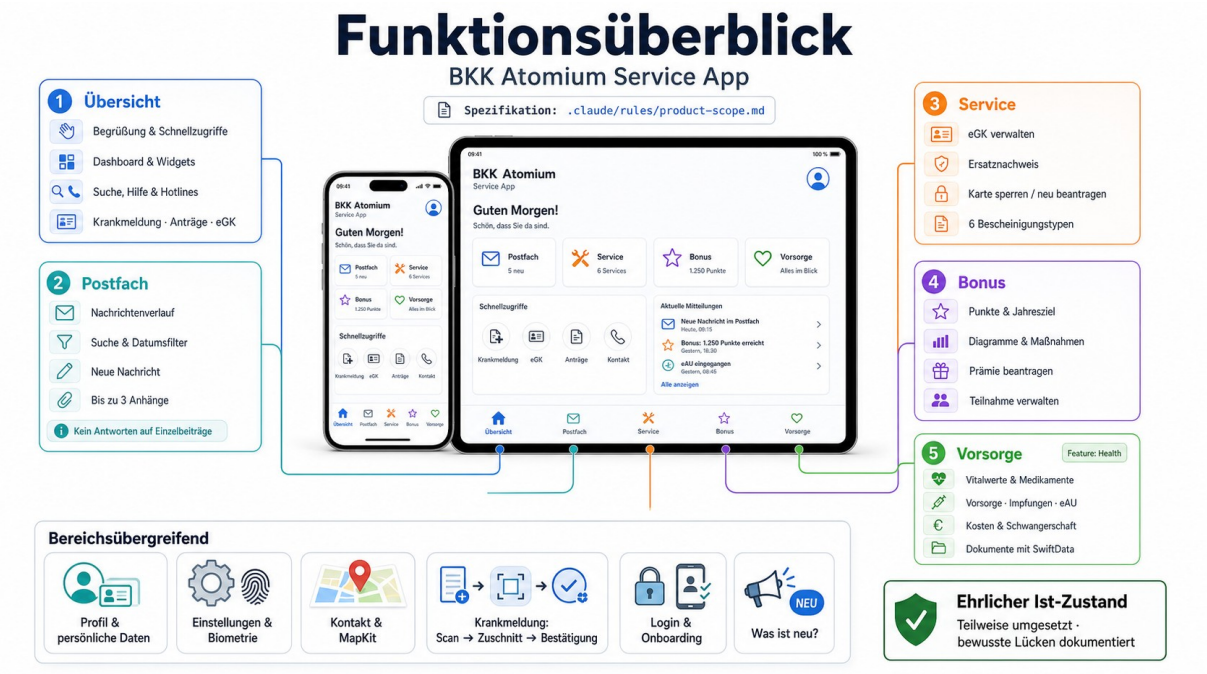
- eine Fachsoftware für die Sachbearbeitung einer Krankenkasse — mit lokaler KI-Anbindung sowie Zugriff auf Gesetze, Richtlinien und FAQs,
- ein vollständig lokaler Chatbot für deutsche Gesetze,
- die datenschutzkonforme Anbindung von Unternehmensdaten an lokale KI-Agenten.




*Architekturüberblick*

## Funktionsüberblick

Bevor es um Claude Code, Rules und Skills geht, lohnt sich ein Blick darauf, was die App eigentlich können soll. Die verbindliche, vollständige Spezifikation steht in `.claude/rules/product-scope.md`; hier die Kurzfassung als Orientierung für den Rest des Tutorials.




*Funktionsüberblick*

Die App gliedert sich in fünf Tabs:

| Tab | Kernfunktionen |
|---|---|
| **Übersicht** | Tageszeitabhängige Begrüßung, Schnellzugriff auf Krankmeldung/Kontakt/Postfach, Bonus- und Postfach-Widgets, Werbe-Karussell, Dashboard-Kacheln (Krankmeldung, Anträge, Krankengeld, eGK), Service-Hotlines, Hilfe-Sektion, projektweite Suche. Der Profilbereich ist über einen Button oben rechts erreichbar. |
| **Postfach** | Chat-artiger Nachrichtenverlauf mit der Kasse, Volltextsuche, Datumsfilter, Verfassen neuer Nachrichten mit bis zu drei Anhängen. Kein Antworten auf einzelne Nachrichten — nur globales Verfassen. |
| **Service** | eGK-Verwaltung (Ersatznachweis anfordern, Karte sperren, neue Karte beantragen) und Anforderung von sechs Bescheinigungstypen (Mitglieds-, Beitrags-, Zuzahlungsbefreiungsbescheinigung u. a.). |
| **Bonus** | Punktestand mit Jahresziel (Ring- und Balkendiagramm), Kategorie-Donut-Chart, Liste eingereichter Maßnahmen, neue Maßnahme melden, Prämie beantragen (Überweisung/Gutschein/Beitragsreduktion), Teilnahme beenden. |
| **Vorsorge** (Feature `Health`) | Gesundheitsdaten mit Vitalwert-Sparklines, Medikamentenliste, Vorsorgeuntersuchungen, Impfstatus, eAU-Verlauf, Schwangerschafts-Leistungsübersicht, Kostenübersicht mit Belastungsgrenze, sowie persistente Rechnungs- und Dokumentenverwaltung (SwiftData, inkl. Filter, Swipe-Actions, Teilen). |

Dazu kommen bereichsübergreifende Bausteine: ein Profilbereich mit Verwaltung persönlicher Daten (Adressen, Telefonnummern, Bankverbindungen, E-Mail, Versicherungsstatus), Einstellungen (Sprache, Darstellung, Benachrichtigungen, Face ID/Touch ID), ein Kontakt-Sheet (Anruf, Kontaktformular, Kassen-Adresse per MapKit), ein mehrstufiger Krankmeldungs-Flow (Scan, Zuschnitt, Bestätigung) sowie Login, Onboarding und ein „Was ist neu”-Sheet.

Nicht jede dieser Funktionen ist bereits vollständig umgesetzt — welche Lücken bewusst offen sind, beschreibt der Abschnitt „Ehrlich über den Ist-Zustand” weiter unten.

## Zielgruppe und Lernziele

Das Tutorial richtet sich an iOS-Entwickler, die mit Claude Code produktiv arbeiten wollen. Gemeint sind nicht Claude-Code-Neulinge, die zuerst die Grundlagen brauchen (dafür gibt es die offizielle Dokumentation), sondern Leute, die verstehen wollen, wie ein Team CLAUDE.md, Rules, Skills, Hooks und Agents zusammensetzt, damit am Ende ein konsistenter Entwicklungsprozess entsteht statt zwanzig widersprüchlicher Einzelkonventionen.

Nach diesem Tutorial solltest du:

- verstehen, wie das Composer-Pattern dieses Projekts aufgebaut ist und warum es so geschnitten ist,
- die Rolle von CLAUDE.md, Rules, Skills, Hooks, Agents und Auto-Memory unterscheiden können: wissen, was davon technisch durchgesetzt wird und was reine Anweisung an Claude ist, nicht nur die Begriffe benennen können,
- an einem realen, im Projekt dokumentierten Beispiel gesehen haben, wie ein neues Feature entsteht, ein bestehendes erweitert wird und ein echter Bug behoben wird,
- eine Einschätzung haben, was Claude Code in diesem Workflow abnimmt und was weiterhin Aufgabe des Entwicklers bleibt.

## Geräte und Umfang

Die App läuft auf iPhone und iPad (`TARGETED_DEVICE_FAMILY: "1,2"` in `project.yml`), Deployment-Ziel ist iOS/iPadOS 18.0. Es gibt keine Watch-App, keine Mac-Version und keine Widgets: Der Funktionsumfang ist bewusst auf die beiden Haupt-Formfaktoren begrenzt. Das komplette, verbindliche Pflichtenheft steht in `.claude/rules/product-scope.md`: fünf Tabs (Übersicht, Postfach, Service, Bonus, Vorsorge), ein Profilbereich, mehrstufige Flows wie die Krankmeldung, und diverse Formulare mit klaren Validierungsregeln.

## Mock-Daten statt Backend — bewusst

Ein Punkt, der im gesamten Tutorial immer wieder auftaucht, deshalb hier vorab klar benannt:

> Die BKK Atomium Service App verwendet in diesem Tutorial ausschließlich Mock-Daten. Es gibt kein echtes Backend, keine echten Versichertendaten und keine produktive Infrastruktur. Ziel ist eine saubere, testbare iOS-/iPadOS-App-Architektur, nicht die Umsetzung einer vollständigen Backend-Integration.

Das ist keine technische Notlösung, sondern eine architektonische Entscheidung: Jedes Repository hat ein Protokoll, eine `Mock`-Implementierung und (bis auf drei Ausnahmen, dazu in Kapitel 3 mehr) eine `Live`-Implementierung, die nur `AppError.notImplemented` wirft. Sobald eine echte API-Spezifikation vorliegt, wird die `Live`-Implementierung ausprogrammiert, ohne die Architektur anzufassen. Bis dahin ist die App vollständig über deterministische Mocks nutz- und testbar.

## Ehrlich über den Ist-Zustand

Dieses Tutorial verschweigt nicht, dass die App noch nicht zu 100 % dem in `product-scope.md` beschriebenen Soll-Zustand entspricht. Das Projekt führt dazu eine eigene, ehrliche Buchhaltung unter `.claude/product/GAPS.md`. Die größten offenen Punkte laut diesem Dokument (Stand 13.07.2026):

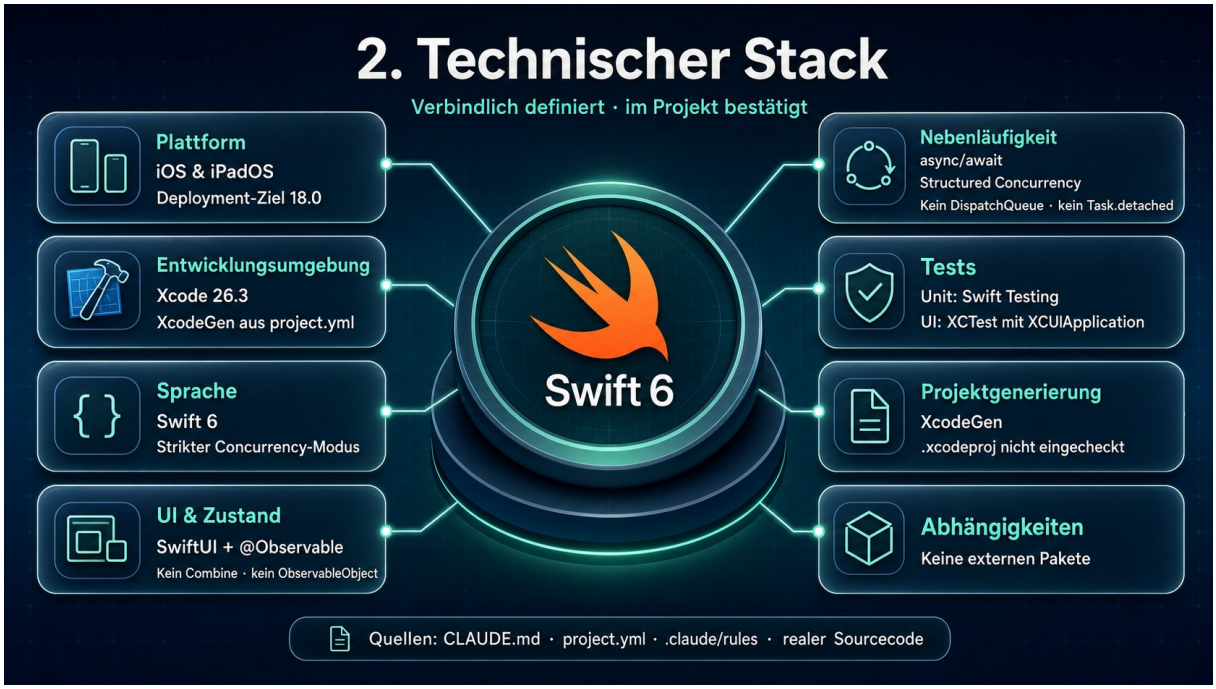
- Onboarding und ein „Was ist neu”-Sheet fehlen komplett, obwohl `product-scope.md` beides verlangt.
- Vier von sechs Dashboard-Kacheln auf dem Start-Screen (Krankmeldung, Anträge, Krankengeld, eGK) zeigen aktuell nur einen `HomeComingSoonView`-Platzhalter statt eines echten Flows.
- Das Kontakt-Sheet deckt bisher nur den Direktanruf ab, nicht das Kontaktformular, die Kartenansicht oder die weiteren im Pflichtenheft genannten Optionen.
- Die Testabdeckung ist in einzelnen Features (Service, Bonus, Profil) lückenhaft.

Diese Lücken sind kein Makel des Tutorials, sondern der Ausgangspunkt für die praktischen Beispiele in den Kapiteln 12 bis 14: Statt erfundener Spielbeispiele arbeitet dieses Tutorial an echten, dokumentierten offenen Punkten.

---

## 2. Technischer Stack

Der Stack ist in `CLAUDE.md` verbindlich festgelegt und wird durch `project.yml` sowie den realen Code bestätigt:




*Technischer Stack*

| Bereich | Festlegung | Beleg |
|---|---|---|
| Plattform | iOS/iPadOS, Deployment-Ziel 18.0 | `project.yml` |
| Xcode | 26.3 | `CLAUDE.md` |
| Sprache | Swift 6, `SWIFT_VERSION: "6.0"`, strikter Concurrency- | `project.yml`, `.claude/rules/concurren` |

| Bereich | Festlegung | Beleg |
|---|---|---|
| | Modus | `cy.md` |
| UI | SwiftUI + Observation (`@Observable`) — kein Combine, kein `ObservableObject` | `CLAUDE.md` |
| Nebenläufigkeit | async/await, Structured Concurrency — kein `DispatchQueue`, kein `Task.detached` | `CLAUDE.md`, `.claude/rules/concurrency.md` |
| Unit-Tests | Swift Testing (`@Suite/@Test/#expect`) | bestätigt an echten Testdateien, z. B. `BonusViewModelTests.swift` |
| UI-Tests | XCTest (`XCUIApplication`) — einziger erlaubter XCTest-Einsatz | `.claude/rules/testing.md` |
| Projektgenerierung | XcodeGen aus `project.yml` — das `.xcodeproj` ist nicht eingecheckt | `.gitignore`, `.claude/product/PROCESS.md` |
| Abhängigkeiten | keine externen Paket-Abhängigkeiten | `CLAUDE.md` |

## Native Apple-Frameworks statt externer Pakete

„Keine externen Abhängigkeiten" heißt nicht „minimaler Funktionsumfang". Für die geforderte Funktionsvielfalt greift das Projekt auf Erstanbieter-Frameworks zurück, die tatsächlich im Code verwendet werden:

- **SwiftData**: für Rechnungen und Dokumente im Gesundheit-Feature. Zwei `@Model`-Entitäten (`InvoiceEntity`, `DocumentEntity`), ein zentraler `ModelContainer` in `AppDependencies`.
- **Swift Charts** (`import Charts`): Sparklines in `HealthDataView`, Donut-Chart in `BonusCategoryChartSection`, Balkendiagramm in `CostOverviewView`.
- **`.fileImporter`/`ShareLink`**: Dokumenten-Upload und Rechnungs-Export im Gesundheit-Feature, Anhänge im Postfach.
- **Local Authentication**: als Protokoll (`AuthService`) für Face ID/Touch ID vorgesehen; die tatsächliche Login-Implementierung nutzt aktuell einen Demo-Auth-Service ohne echte Biometrie-Abfrage.

Eine ehrliche Lücke gegenüber dem Pflichtenheft: **MapKit wird bisher nirgends importiert**, obwohl `product-scope.md` für die Kassen-Adresse im Kontakt-Sheet eine Kartenansicht verlangt. Das

`ContactSheet` deckt laut eigenem Code-Kommentar aktuell nur den Direktanruf ab. Die Karte gehört zu den in Kapitel 1 genannten offenen Punkten.

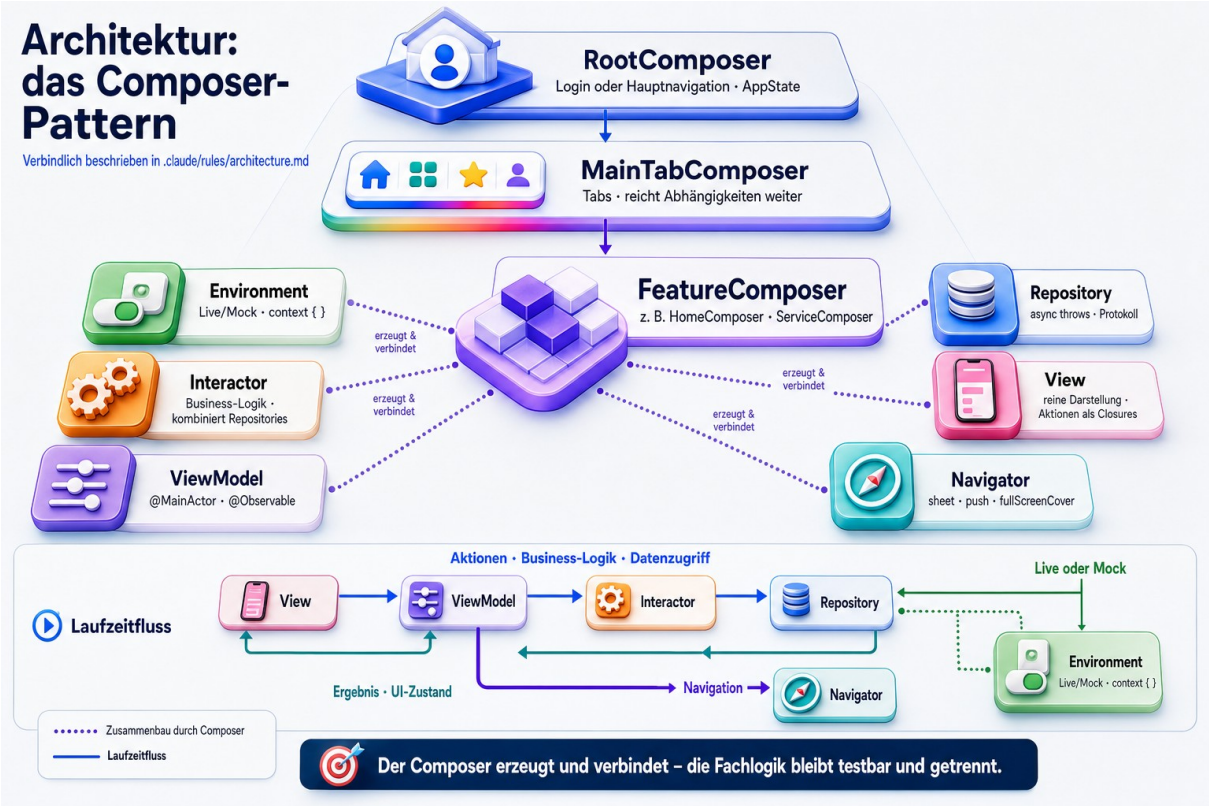
### Warum kein lokales Swift-Framework

Anders als in manchen größeren Projekten üblich, besteht `BKKAtomium` aus einem einzigen Xcode-Projekt mit drei Targets (App, Unit-Tests, UI-Tests): kein separates lokales Swift Package, keine Modul-Grenze zwischen „Core" und „Feature". Die Trennung von Verantwortlichkeiten passiert ausschließlich über Ordnerstruktur und Zugriffsregeln (siehe Kapitel 3), nicht über Compiler-erzwungene Modulgrenzen. Für die Projektgröße reicht das aus, und es vermeidet den Mehraufwand separater Paket-Versionierung.

---

## 3. Architektur: das Composer-Pattern

Die Architektur ist in `.claude/rules/architecture.md` verbindlich beschrieben. Sie kombiniert eine klassische Schichtentrennung (View → ViewModel → Repository) mit einem expliziten Zusammenbau-Mechanismus pro Feature, dem **Composer**.




*Composer-Pattern*

```
RootComposer                          (Login vs. Hauptnavigation, AppState)
└── MainTabComposer                   (Tabs, reicht dependencies weiter)
    └── <Feature>Composer             (z. B. HomeComposer, ServiceComposer)
        ├── Environment               (Abhängigkeiten live/mock, via
context{})
        ├── Interactor                 (Business-Logik, kombiniert
Repositories)
        ├── ViewModel                 (@MainActor, @Observable)
```

```
├── Repository                (async throws Protokoll)
├── View                      (dumm, Aktionen als Closures)
└── Navigator                 (sheet/push/fullScreenCover)
```

### Composer, Environment, `context{}`

Jedes Feature hat genau einen Composer: ein `enum`, damit er keinen eigenen Instanzzustand halten kann. Am Beispiel des real existierenden `HomeComposer` (`BKKAtomium/Features/Home/UI/HomeComposer.swift`): Der Composer baut über den projektweiten `context {}`-Helfer (`Core/Composition/Context.swift`) eine `HomeEnvironment` und reicht sie an eine private Flow-View weiter. Scheitert der Bau der Environment (aktuell praktisch nie, weil es kein Backend gibt, das fehlschlagen könnte), fängt `context{}` den Fehler ab und zeigt eine generische `ContextErrorView` statt abzustürzen.

Die private Flow-View (in der Composer-Datei selbst, nicht als eigene Datei) hält den kompletten Navigationszustand: `@State private var path: [HomeDestination] = []`, das ViewModel als `@State`, sowie Sheet-Flags. Sie implementiert außerdem direkt das feature-eigene Navigator-Protokoll und verdrahtet `NavigationStack(path:)` mit `.navigationDestination(for: HomeDestination.self)`.

### Environment entscheidet Mock vs. Live

Nur an einer einzigen Stelle im gesamten Feature wird zwischen Mock- und Live-Implementierung eines Repositories entschieden: in der `<Feature>Environment`. ViewModels bekommen ausschließlich Protokolltypen (`any BonusRepository`, nicht `BonusRepositoryMock`), sodass diese Entscheidung austauschbar bleibt, ohne dass ein ViewModel geändert werden müsste.

### „Live ohne Backend" — und drei echte Ausnahmen

Weil es kein Backend gibt, existiert zu praktisch jedem Repository eine `…RepositoryLive`-Datei, die absichtlich nur wirft:

```
struct BonusRepositoryLive: BonusRepository {
    func fetchProgress() async throws -> BonusProgress {
        throw AppError.notImplemented
    }
    // … drei weitere Methoden, gleiches Muster
}
```

Das ist kein Zwischenstand, der vergessen wurde: Es ist die bewusste Grenze zwischen „App-Architektur fertig" und „Backend-Anbindung noch offen". Sobald eine echte API-Spezifikation vorliegt, füllt der Skill `adding-backend-endpoint` (Kapitel 6) genau diese Datei mit echtem Netzwerkcode.

Drei Repositories weichen davon ab, weil sie **keine Backend-Anbindung, sondern lokale Persistenz** sind und deshalb bereits echte, funktionierende `Live`-Implementierungen haben, die auch bereits in `AppDependencies` verdrahtet sind: `InvoiceRepositoryLive` und `DocumentRepositoryLive` (beide `@ModelActor actor`, SwiftData-basiert) sowie `SettingsRepositoryLive` (UserDefaults-basiert). Alle übrigen Repositories laufen weiterhin über ihre `…Mock`-Implementierung.

## Schichten und erlaubte Abhängigkeitsrichtungen

| Schicht | darf kennen | darf NICHT kennen |
|---|---|---|
| Domain | Foundation | SwiftUI, Repositories, ViewModels |
| Repository (Protokoll + Live + Mock) | Domain, Core/Mocks, Core/Errors | SwiftUI, ViewModels |
| Interactor | Domain, Repository-Protokolle, Core | SwiftUI, Views |
| ViewModel | Interactor, Domain, Core | SwiftUI-Views, konkrete Repositories |
| View | eigenes ViewModel, Navigator/Closures, DesignSystem | Repositories, Interactor direkt, andere Features |
| Composer | alles im eigenen Feature + `AppDependencies` | andere Features |

Features referenzieren einander nie direkt, nicht aus Prinzip, sondern weil sonst genau die Kopplung entsteht, die diese Architektur vermeiden soll. Auffällig im echten Code: Das `Home`-Feature (Start-Dashboard) hat keinen eigenen `Repository`- oder `Interactor`-Ordner. Es liest nur statische/aggregierte Daten und braucht keine eigene Datenquelle. Das ist keine Abweichung von der Regel, sondern deren korrekte Anwendung: Wer keinen Interactor braucht, bekommt auch keinen künstlichen.

## Vier-Zustände-Rendering: `ViewState<Value>`

Jede asynchrone View-Sektion verwendet ein Zustands-Enum statt loser `isLoading`/`errorMessage`-Paare. Die reale Definition in `Core/State/ViewState.swift`:

```
enum ViewState<Value: Sendable>: Sendable {
    case idle
    case loading
    case loaded(Value)
    case failed(AppError)
}
```

Views rendern konsequent alle vier Fälle: `loading` zeigt eine getintete `ProgressView`, `failed` eine `ErrorStateView` mit Retry-Button, `loaded` mit leerem Ergebnis einen `EmptyStateView`, `loaded` mit Inhalt die eigentliche Ansicht. Sieben der neun Unterseiten im Gesundheit-Feature sind bewusst rein statisch (keine Ladevorgänge). Sie bekommen entsprechend auch kein `ViewState` und kein eigenes ViewModel, sondern lesen direkt statische Domain-Daten. Auch das ist keine Inkonsistenz, sondern eine im Projekt dokumentierte, gewollte Ausnahme (`.claude/product/PROCESS.md`, Abschnitt 5).

### Swift-6-Concurrency in der Praxis

ViewModels sind ausnahmslos `@MainActor @Observable final class`. Mocks mit veränderlichem Zustand sind `actor`: im echten Code alle zwölf `…RepositoryMock`-Typen sowie die beiden SwiftData-`Live`-Repositories. `nonisolated(unsafe)` und `@unchecked Sendable` sind projektweit verboten; wer glaubt, sie zu brauchen, hat laut `.claude/rules/concurrency.md` ein Designproblem, keine Ausnahmesituation.

### iPad als Standard, nicht als Nachgedanke

Weil `TARGETED_DEVICE_FAMILY` beide Geräteklassen einschließt, verlangt `.claude/rules/swiftui.md`, dass jedes neue Layout in allen Größenklassen funktioniert: nicht nachträglich hochskaliert, sondern von Anfang an mit `.frame(maxWidth:)`-Begrenzung oder `NavigationSplitView` bei Master-Detail-Charakter gedacht.

## 4. Claude Code im Entwicklungsalltag

Dieses Kapitel bündelt alles, was für die tägliche Arbeit mit Claude Code selbst wichtig ist, unabhängig vom konkreten Projekt. Alle Aussagen sind Stand 13.07.2026 gegen `code.claude.com/docs` geprüft.

### 4.1 Oberflächen: Terminal, Desktop, IDE, Web

Claude Code ist nicht nur ein CLI-Tool. Offiziell dokumentiert sind mehrere Oberflächen für denselben Claude-Code-Kern (code.claude.com/docs/en/overview):

- **Terminal/CLI**: der vollständige Funktionsumfang, startet mit `claude` im Projektverzeichnis.
- **Desktop-App** (macOS, Windows): grafische Oberfläche über dem gleichen Claude-Code-Kern, für alle, die kein Terminal wollen.
- **IDE-Integrationen**: VS-Code-Extension und JetBrains-Plugins (IntelliJ, PyCharm, WebStorm u. a.) mit Inline-Diffs direkt im Editor.
- **Web** (`claude.ai/code`): Cloud-Sessions ohne lokales Setup, unter anderem für den `/code-review ultra`-Workflow relevant, der in diesem Environment als Cloud-Review verfügbar ist.

Für dieses Tutorial ist das Terminal die Referenz-Oberfläche, weil sich Hooks, Skills und CLI-Flags dort am direktesten beobachten lassen. Die Desktop-App eignet sich, wenn du mehrere Sessions parallel im Blick behalten willst; IDE-Integrationen lohnen sich, wenn du sowieso den Großteil des Tages in Xcode beziehungsweise VS Code verbringst und nicht zwischen Fenstern wechseln willst.

### 4.2 Abonnement und Kosten

Claude Code wird pro API-Token abgerechnet, auch im Rahmen eines Abos. Für Pro-, Max-, Team- und Enterprise-Pläne ist die Nutzung in der Seat-Allokation enthalten; auf der Claude Console (API) sowie bei Amazon Bedrock, Google Cloud oder Microsoft Foundry wird direkt pro Token abgerechnet (code.claude.com/docs/en/costs).

Konkrete, offiziell genannte Anhaltspunkte für Unternehmenseinsatz (Stand 13.07.2026): durchschnittlich rund 13 US-Dollar pro Entwickler und aktivem Tag, 150 bis 250 US-Dollar pro Entwickler und Monat, 90 % der Nutzer bleiben unter 30 US-Dollar pro aktivem Tag. Für die konkreten Preise der Abo-Stufen selbst

verweist die Dokumentation ausdrücklich auf claude.com/pricing. Diese Zahlen ändern sich zu häufig, um sie hier sinnvoll fest einzutragen.

Zwei Befehle helfen beim Kostenüberblick direkt in der Session: `/usage` zeigt Token-Verbrauch und (bei Team/Enterprise) eine Aufschlüsselung nach Skills/Subagenten/Plugins; auf Pro/Max lässt sich über `/usage-credits` ein monatliches Ausgabenlimit setzen.

## 4.3 Welches Modell für welche Aufgabe?

Aktuell in Claude Code wählbar sind vier Modelle:

| Modell | ID | Einsatz |
|---|---|---|
| Claude Fable 5 | `claude-fable-5` | längste, komplexeste Agentenläufe |
| Claude Opus 4.8 | `claude-opus-4-8` | komplexe Architekturentscheidungen, Multi-Step-Reasoning |
| Claude Sonnet 5 | `claude-sonnet-5` | Standard-Modell für die meisten Coding-Aufgaben |
| Claude Haiku 4.5 | `claude-haiku-4-5-20251001` | schnelle, günstige Aufgaben, ideal für Subagenten |

Die offizielle Empfehlung ist unaufgeregt: Sonnet erledigt die meisten Coding-Aufgaben gut und kostet weniger als Opus; Opus lohnt sich für komplexe Architekturentscheidungen oder mehrstufiges Reasoning; für einfache Subagenten-Aufgaben reicht `model: haiku` in der Agent-Konfiguration. Umschalten geht jederzeit über `/model` (Session) oder dauerhaft über `/config`. In diesem Projekt sind die beiden Review-Agents (`concurrency-reviewer`, `swiftui-reviewer`, Kapitel 11) bewusst nicht auf ein bestimmtes Modell festgelegt: Sie erben das Modell der Hauptsitzung.

## 4.4 Plan-Modus

Der Plan-Modus lässt Claude eine Codebasis analysieren und einen Plan vorschlagen, bevor irgendetwas geändert wird: kein Schreibzugriff, bis du zustimmst. Aktivierung über `Shift+Tab` (schaltet zwischen den Permission-Modi `default`, `acceptEdits`, `plan` und, falls aktiviert, `auto`/`bypassPermissions` durch), über den Start-Flag `claude --permission-mode plan`, oder dauerhaft über `"permissions": { "defaultMode": "plan" }` in `settings.json`.

Für ein Projekt mit so viel vorab definiertem Soll-Zustand wie `product-scope.md` ist der Plan-Modus mehr als eine Vorsichtsmaßnahme: Er zwingt dazu, vor der Implementierung explizit zu sagen, welche Rule- und Skill-Dateien überhaupt relevant sind, genau der Schritt, der in Kapitel 12 bis 14 an echten Beispielen vorgeführt wird.

## 4.5 Installation

Die offiziell empfohlene Installation (macOS/Linux/WSL) ist ein einzeiliger Installer mit automatischen Updates im Hintergrund (code.claude.com/docs/en/setup):

```
curl -fsSL https://claude.ai/install.sh | bash
```

Windows PowerShell: `irm https://claude.ai/install.ps1 | iex`. Alternativ: Homebrew (`brew install --cask claude-code`, kein automatisches Update), WinGet (`winget install Anthropic.ClaudeCode`) oder npm (`npm install -g @anthropic-ai/claude-code`, benötigt Node.js 22+). Nach der Installation reicht `claude` im Projektverzeichnis; `claude doctor` prüft die Installation, `claude --version` zeigt die aktuelle Version. Systemvoraussetzungen: macOS 13+, Windows 10 1809+, Ubuntu 20.04+/Debian 10+/Alpine 3.19+, mindestens 4 GB RAM.

## 4.6 Projektstruktur für Claude Code in BKKAtomium

So sieht die tatsächliche, im Projekt eingecheckte Claude-Code-Konfiguration aus (Kapitel 5 bis 11 gehen auf jede Datei einzeln ein):

```
BKKAtomium/
├── CLAUDE.md                          — Projektgedächtnis (Kapitel 5)
├── project.yml                        — XcodeGen-Konfiguration
├── .swiftlint.yml                     — SwiftLint-Konfiguration inkl. 5
Custom Rules
├── Scripts/
│   ├── check_project.sh               — Qualitäts-Gate P1–P8 (Kapitel 16)
│   └── generate_feature.sh            — Feature-Gerüst-Generator (Kapitel
6, 12)
├── Templates/                         — *.template-Vorlagen für
generate_feature.sh
└── .claude/
    ├── settings.json                  — geteilt, eingecheckt (Kapitel 4.7)
    ├── settings.local.json            — persönlich, gitignored (Kapitel
4.7)
    ├── rules/                         — 10 Regeldateien (Kapitel 7)
    ├── skills/                        — 8 Skills (Kapitel 6)
    ├── agents/                        — 2 Subagenten (Kapitel 11)
    ├── hooks/
    │   └── swiftlint-on-write.sh      — PostToolUse-Hook (Kapitel 9)
    └── product/                       — Ist-Zustand pro Feature (Kapitel
8)
```

Auto-Memory (Kapitel 10) gehört bewusst **nicht** in diese Liste: Sie liegt außerhalb des Projekt-Repos unter `~/.claude/projects/…` und wird nie eingecheckt.

## 4.7 `settings.json` und `settings.local.json`

`settings.json` unterstützt eine sehr lange Liste dokumentierter Top-Level-Schlüssel (`permissions`, `hooks`, `model`, `env`, `autoMemoryEnabled`, `autoUpdatesChannel` und viele weitere, code.claude.com/docs/en/settings). Die meisten davon braucht ein einzelnes Projekt nie. `BKKAtomium` nutzt bewusst nur zwei Bereiche:

```
{
  "permissions": {
    "allow": [
      "Bash(xcodegen*)", "Bash(xcodebuild*)", "Bash(swiftlint*)",
      "Bash(xcrun simctl*)", "Bash(plutil*)",
      "Bash(./Scripts/check_project.sh*)",
"Bash(./Scripts/generate_feature.sh*)"
    ],
    "deny": ["Bash(rm -rf*)", "Bash(git push*)", "Bash(git reset --hard*)"]
  },
```

```
  "hooks": { "PostToolUse": [ /* siehe Kapitel 9 */ ] }
}
```

Diese Datei ist eingecheckt (nicht in `.gitignore`) und gilt für jeden, der am Projekt arbeitet. Die `deny`-Liste ist keine Empfehlung an Claude, sondern eine von der Harness selbst durchgesetzte Sperre: Ein `git push` oder `rm -rf` wird gar nicht erst zur Ausführung angeboten.

`.claude/settings.local.json` liegt daneben und ist über `.gitignore` explizit ausgeschlossen:

```
{ "permissions": { "allow": ["Skill(run)", "Read(//Users/…/**)"] } }
```

Persönliche, maschinenspezifische Ergänzungen: hier zusätzliche Leserechte und die Freigabe des `run`-Skills. Beide Dateien werden zusammengeführt; `settings.local.json` überschreibt bei Konflikten. Das Muster dahinter ist immer dasselbe: **geteilt = Projektregel, lokal = persönliche Maschine.**

## 4.8 Eingebaute Slash-Commands, CLI-Flags und Skills-vs.-Commands

Ein wichtiger, oft übersehener Punkt zuerst: **Custom Slash Commands (`.claude/commands/*.md`) sind offiziell in Skills aufgegangen.** Zitat aus der aktuellen Dokumentation: „Custom commands have been merged into skills." Eine Datei unter `.claude/commands/deploy.md` und ein Skill unter `.claude/skills/deploy/SKILL.md` erzeugen beide denselben `/deploy`-Befehl. Alte `.claude/commands/`-Dateien funktionieren zwar weiter, aber Skills sind der aktuelle, empfohlene Mechanismus: Sie erlauben zusätzlich ein eigenes Verzeichnis für Begleitdateien, Frontmatter-Steuerung (wer darf den Skill auslösen: Nutzer, Claude, oder beide) und automatisches Laden durch Claude, wenn es passt. `BKKAtomium` hat konsequenterweise **keinen** `.claude/commands/`-Ordner, sondern ausschließlich Skills (Kapitel 6).

Eingebaute Slash-Commands (Auszug, kein Projekt-spezifisches): `/model` (Modellwahl), `/config` (dauerhafte Einstellungen), `/plan` (Plan-Modus für eine Anfrage), `/clear` (Konversation zurücksetzen), `/compact` (Kontext zusammenfassen, optional mit Anweisung was erhalten bleiben soll), `/usage` und `/usage-credits` (Kostenüberblick), `/agents` (Subagenten-Übersicht), `/hooks` (Hook-Browser), `/context` (Kontextfenster-Visualisierung), `/mcp` (MCP-Server verwalten), `/doctor` (Diagnose), `/keybindings` (Tastenkürzel anpassen, Kapitel 4.9).

CLI-Flags, offiziell dokumentiert (code.claude.com/docs/en/cli-reference), Auszug der für den Alltag relevantesten:

| Flag | Zweck |
|---|---|
| `-p, --print` | Antwort ausgeben, nicht interaktiv — für Skripte/CI |
| `-c, --continue` | letzte Konversation im aktuellen Verzeichnis fortsetzen |
| `-r, --resume` | bestimmte Session per ID/Name fortsetzen |
| `--model` | Modell für diesen Start festlegen |
| `--permission-mode` | Start-Modus: `default`, `acceptEdits`, `plan`, `auto`, `dontAsk`, `bypassPermissions`, |

| Flag | Zweck |
|---|---|
| | `manual` |
| `--add-dir` | zusätzliche Arbeitsverzeichnisse freigeben |
| `--settings` | Pfad zu einer alternativen Settings-Datei |
| `--allowedTools` / `--disallowedTools` | Tool-Freigaben für diesen Start |
| `-v, --version` | installierte Version anzeigen |

### 4.9 Tastenkürzel

Die offiziell dokumentierten Standard-Tastenkürzel (code.claude.com/docs/en/keybindings), Auszug:

| Kürzel | Aktion |
|---|---|
| `Shift+Tab` | Permission-Modus durchschalten (default → acceptEdits → plan → …) |
| `Ctrl+C` | laufende Aktion abbrechen |
| `Ctrl+D` | Claude Code beenden |
| `Escape` | Eingabe abbrechen; zweimal bei leerer Eingabe öffnet das Rewind-Menü |
| `Ctrl+R` | Verlauf durchsuchen |
| `↑ / ↓` | vorheriger/nächster Verlaufseintrag |
| `Ctrl+T` | Claudes To-do-Checkliste ein-/ausblenden |
| `Ctrl+O` | ausführliches Transkript ein-/ausblenden |
| `Tab` | Autovervollständigung übernehmen |

Alle Tastenkürzel lassen sich über `/keybindings` in einer eigenen `~/.claude/keybindings.json` anpassen oder entbinden. Für dieses Projekt gibt es dafür keinen Sonderbedarf, deshalb bleibt die Konfiguration beim Standard.

## 5. CLAUDE.md — das Projektgedächtnis

CLAUDE.md wird bei jedem Sitzungsstart automatisch geladen, kein manueller Aufruf nötig. Offiziell lädt Claude Code sie hierarchisch über mehrere Ebenen additiv (code.claude.com/docs/en/memory):

| Ebene | Pfad | Geltungsbereich | Geteilt? |
|---|---|---|---|
| Organisationsweit | z. B. `/Library/Application Support/ClaudeCode/CLAUDE.md` (macOS) | alle Projekte der Organisation | zentral verwaltet |
| Nutzer-global | `~/.claude/CLAUDE.md` | alle Projekte dieses Nutzers | nein, privat |
| Projekt | `./CLAUDE.md` | dieses Projekt, ganzes Team | ja, eingecheckt |
| Lokal | `./CLAUDE.local.md` | dieses Projekt, nur dieser Nutzer | nein (gehört in `.gitignore`) |

In `BKKAtomium` existieren zwei Ebenen tatsächlich: eine private, nutzerglobale `~/.claude/CLAUDE.md` mit allgemeinen Swift-/Apple-Konventionen (die ausdrücklich zurücktritt, sobald projekteigene Rules existieren, genau der Fall hier), und die projekteigene `CLAUDE.md` im Repo-Root. Eine `CLAUDE.local.md` gibt es in diesem Projekt nicht, obwohl `.gitignore` bereits einen Eintrag dafür reserviert. Sie würde greifen, sobald jemand persönliche, nicht zu teilende Projektnotizen braucht.

## Was tatsächlich in der Projekt-CLAUDE.md steht

Das ist keine Erfindung, sondern der reale Aufbau: Zweck der App, verbindlicher Tech-Stack, eine Kurzfassung des Composer-Patterns, die Verzeichnisstruktur, erlaubte Abhängigkeitsrichtungen, eine Verweisliste auf alle zehn Rule-Dateien („wann welche Datei zuerst lesen"), die vollständige Neun-Sprachen-Lokalisierungstabelle, die Standard-Befehle (`xcodegen generate`, Build-/Testkommandos, `check_project.sh`, `generate_feature.sh`) und eine siebenteilige Definition of Done.

Bemerkenswert für die Praxis: Die CLAUDE.md nennt selbst einen veralteten Zwischenstand („`CFBundleLocalizations` aktuell noch `[de, en]`"). Im echten `project.yml` stehen inzwischen bereits alle neun Sprachen. Das ist keine Ausnahme, sondern der Normalfall: CLAUDE.md beschreibt einen Zustand zum Zeitpunkt des letzten Updates, keinen live nachgeführten Datenbankauszug. Wer eine Abweichung zwischen CLAUDE.md und Code bemerkt, aktualisiert die Datei im selben Arbeitsschritt, genau das listet Kapitel 8 als eine der bekannten, dokumentierten Lücken.

## Grenzen von CLAUDE.md

CLAUDE.md ist für dauerhaftes, kompaktes Grundwissen gedacht: Die offizielle Empfehlung liegt bei unter 200 Zeilen, weil sie bei jedem Start vollständig in den Kontext geladen wird und übermäßige Länge die Befolgung verschlechtern kann. Alles, was eher eine mehrstufige Anleitung als eine Tatsache ist („wie lege ich ein neues Feature an"), gehört nicht in CLAUDE.md, sondern in einen Skill (Kapitel 6): Skills laden ihren Inhalt nur bei tatsächlicher Nutzung, CLAUDE.md-Inhalt ist immer präsent, ob gebraucht oder nicht.

---

## 6. Skills — wiederverwendbare Arbeitsabläufe

Ein Skill ist eine `SKILL.md`-Datei mit Frontmatter (`name`, `description`) und einer Anleitung im Markdown-Body. Anders als das ältere Vorgänger-Tutorial behauptet hatte, ist das kein reiner Namenskonvention-Trick ohne technische Unterstützung: Claude Code hat ein natives Skill-System. Claude kann einen passenden Skill automatisch laden, wenn die Beschreibung zur Aufgabe passt, oder er wird explizit per `/skill-name` aufgerufen. `BKKAtomium` hat acht projekteigene Skills unter `.claude/skills/`, alle rein Markdown, ohne eigenen Code:

| Skill | Zweck | Auslöser |
|---|---|---|
| `creating-feature` | komplettes neues Feature nach Composer-Pattern erzeugen (Domain, Repository+Mock, Environment, Composer, ViewModel, View, Destination, Lokalisierung, Tests) | „neues Feature X" |
| `creating-swiftui-screen` | einzelnen Screen in einem *bestehenden* Feature ergänzen | „neue Seite/Formular im Feature X" |
| `creating-repository` | neues Repository-Protokoll + Mock erzeugen, in `AppDependencies`/Environment verdrahten | „Repository für X" |
| `creating-mock-service` | konfigurierbaren Mock (App- oder Test-Double) für ein bestehendes Protokoll bauen | „Mock für X" |
| `adding-backend-endpoint` | echten Backend-Endpunkt anbinden (API-Client, Mapper, `RepositoryLive`) | nur wenn eine echte API-Spezifikation vorliegt — sonst bricht der Skill bewusst ab |
| `fixing-ios-bug` | Bug reproduzierbar beheben: erst Ursache belegen, dann Regressionstest, dann minimaler Fix | Fehlerbericht |
| `writing-swift-tests` | Swift-Testing-Unit-Tests (bzw. XCTest-UI-Tests) nach Projektstandard schreiben | „schreibe Tests für X" |
| `reviewing-architecture` | read-only Architektur-Review eines oder aller Features gegen das Composer-Pattern | vor Feature-Abschluss, „prüfe die Architektur" |

Alle acht sind reine Anleitungen: Sie können nichts blockieren, sie steuern nur, wie Claude vorgeht. Kapitel 12 bis 14 zeigen `creating-feature`, `writing-swift-tests` und `fixing-ios-bug` an echten, dokumentierten Beispielen im Einsatz.

Neben den Projekt-Skills gibt es eine große Zahl global installierter Skills (unter `~/.claude/skills/`) für Themen, die projektübergreifend gebraucht werden, etwa App-Store-Prüfungen, Sicherheitsaudits oder Lokalisierungs-Checks. Wo sich Inhalte überschneiden (zum Beispiel ein globaler Architektur-Check-Skill gegenüber `reviewing-architecture`), gilt: Die projekteigene Regel- beziehungsweise Skill-Definition hat Vorrang vor dem generischen Standardverhalten eines globalen Skills.

---

## 7. Rules — projektbezogene Regeln

Zehn Dateien unter `.claude/rules/`, jede zu einem Themenbereich, alle in `CLAUDE.md` verlinkt mit dem Hinweis, wann sie zu lesen sind:

| Datei | Themenbereich |
|---|---|
| `product-scope.md` | verbindliche Feature-Checkliste — Soll-Zustand von Tabs, Screens, Formularen |
| `architecture.md` | Composer-Pattern, Schichten, DI, „Live ohne Backend”-Regel |
| `swift.md` | Naming, Fehlerbehandlung, Doku-Pflicht, Stil |
| `swiftui.md` | dumme Views, State Ownership, Design-Tokens, iPad-Pflicht |
| `concurrency.md` | Swift-6-Isolation, Structured Concurrency, verbotene Muster |
| `mocks.md` | deterministischer Seed, `MockBehavior`, CallCount/Capture-Bauplan |
| `networking.md` | Backend-Grenze — keine erfundenen Endpunkte ohne Spezifikation |
| `security-and-privacy.md` | Logging-Verbote, Keychain-Pflicht, Demo-Login ohne Klartext-Credentials |
| `accessibility.md` | Label-/Identifier-/Hint-Pflicht, Mindesttappfläche, Dynamic Type |
| `testing.md` | Swift Testing für Unit-, XCTest nur für UI-Tests, Pflichtfälle pro Ladepfad |

### Anweisung oder technisch erzwungen? Eine ehrliche Einordnung

Eine Rule-Datei ist zunächst nichts anderes als CLAUDE.md: Text, den Claude beim Arbeiten berücksichtigen soll. Ob eine Vorgabe darüber hinaus **technisch durchgesetzt** wird, hängt einzig davon ab, ob ein Skript oder Linter sie tatsächlich prüft. Für `BKKAtomium` lässt sich das an zwei Werkzeugen konkret festmachen:

- **SwiftLint** (`.swiftlint.yml`) erzwingt automatisch: keine `DispatchQueue-asyncAfter`-Aufrufe, keine fixen Font-Größen, kein `@EnvironmentObject`, `@MainActor`+`@Observable` in der richtigen Reihenfolge vor jedem `…ViewModel`, sowie Standardregeln wie verbotenes `force_cast`/`force_try`/`force_unwrapping`.
- **`Scripts/check_project.sh`** prüft in acht Stufen (P1–P8, Details in Kapitel 16): Farbliterale außerhalb `DesignSystem/`, fixe Fontgrößen, verbotene Concurrency-Muster, Force-Konstrukte, hartcodierte UI-Strings (heuristisch), Lokalisierungs-Vollständigkeit de/en, und Singletons. Sieben dieser acht Stufen sind echte Gates (Exit-Code ungleich null bei Fund).

Alles andere (die komplette Composer-Struktur, die Interactor-Pflicht, das Navigator-Muster, die Pflicht zu vier Testfällen pro Ladepfad, die deutsche `///`-Dokumentationspflicht, die Vollständigkeit von Accessibility-Labels) wird von keinem Skript geprüft. Es bleibt reine Anweisung an Claude, ergänzt durch die beiden Review-Agents aus Kapitel 11, die stichprobenartig, aber ohne Gate-Wirkung, dagegen prüfen. Diese Unterscheidung ist wichtig: „Steht in einer Rule" heißt nicht „kann nicht verletzt werden", es heißt „sollte nicht verletzt werden, und ein Teil davon fällt technisch auf, ein anderer Teil nur im Review."

---

## 8. Produktdokumentation (`.claude/product/`) — Soll- und Ist-Zustand

Ein Baustein, den viele Claude-Code-Projekte nicht haben, der sich in `BKKAtomium` aber als sehr wirksam erwiesen hat: ein eigener Ordner, der dokumentiert, was der Code **tatsächlich** tut, nicht was er tun soll.

`.claude/rules/product-scope.md` beschreibt den **Soll-Zustand**: was die App leisten soll, unabhängig vom aktuellen Implementierungsstand. `.claude/product/` beschreibt den **Ist-Zustand**: was tatsächlich im Code steht, inklusive bekannter Lücken. Bei einem Widerspruch gilt ausdrücklich: `rules/` ist die Vorgabe, `product/` ist die Bestandsaufnahme. Eine Lücke zwischen beiden ist kein Fehler der Dokumentation, sondern eine dokumentierte Restaufgabe.

Der Ordner enthält sieben Feature-Dateien (`Home.md`, `Postfach.md`, `Service.md`, `Bonus.md`, `Health.md`, `Profile.md`, `Login.md`, jeweils mit Screens, Domain-Modellen, Repository-Signaturen und Validierungsregeln), dazu `GLOSSARY.md` (Deutsch-Englisch-Begriffszuordnung, damit neue Identifier konsistent bleiben), `PROCESS.md` (wann `xcodegen generate`/Build/`check_project.sh` laufen müssen, siehe Kapitel 16) und `GAPS.md`, die in Kapitel 1 bereits zitierte, konsolidierte Lückenliste.

Diese Dateien wurden per Agent aus dem realen Code generiert und sind laut eigenem Änderungsvermerk noch nicht manuell nachgeprüft. Das ist eine bewusste Zwischenstufe, keine Endgültigkeit: Bei der nächsten Änderung an einem Feature wird der zugehörige Abschnitt gegen den Code gegengelesen und korrigiert, im selben Arbeitsschritt, nicht als nachgelagerte Aufgabe. Eine veraltete Produktdokumentation gilt im Projekt ausdrücklich als schlechter als gar keine, weil sie Vollständigkeit vortäuscht, die nicht mehr da ist.

## 9. Hooks — automatische Aktionen

Hooks sind Shell-Kommandos, die Claude Code an festgelegten Punkten im Lebenszyklus einer Sitzung automatisch ausführt: deterministisch, unabhängig davon, ob das Modell in diesem Moment daran denkt. Offiziell dokumentierte Ereignisse (code.claude.com/docs/en/hooks-guide): `SessionStart`, `UserPromptSubmit`, `PreToolUse`, `PostToolUse`, `PostToolUseFailure`, `PermissionRequest`, `PermissionDenied`, `Notification`, `SubagentStop`, `Stop`, `StopFailure`, `PreCompact`, `SessionEnd`.

`BKKAtomium` nutzt genau einen Hook, konfiguriert in `.claude/settings.json`:

```
{
  "hooks": {
    "PostToolUse": [
      { "matcher": "Write|Edit",
        "hooks": [{ "type": "command", "command":
"\"$CLAUDE_PROJECT_DIR\"/.claude/hooks/swiftlint-on-write.sh" }] }
    ]
  }
}
```

`.claude/hooks/swiftlint-on-write.sh` läuft nach jedem `Write`- oder `Edit`-Aufruf. Es liest den geschriebenen Dateipfad aus dem JSON auf stdin, prüft nur `.swift`-Dateien, und ruft `swiftlint lint --quiet <Datei>` auf. Entscheidend für das Verständnis von Hooks: Das Skript endet immer mit `exit 0`. Es meldet SwiftLint-Verstöße nur informativ in der Ausgabe, **blockiert aber nichts**. Ein `PreToolUse`-Hook könnte das (per Exit-Code 2 einen Tool-Aufruf ablehnen), dieser `PostToolUse`-Hook kann es grundsätzlich nicht mehr, weil die Datei zu diesem Zeitpunkt bereits geschrieben ist.

Das ist die einzige Stelle im Projekt, an der nach jeder Code-Änderung automatisch (ohne expliziten Skill- oder Prompt-Aufruf) etwas passiert. Trotzdem hat sie keine Gate-Wirkung: Wer eine SwiftLint-Warnung ignoriert, wird nicht aufgehalten. Das eigentliche Gate ist `check_project.sh` (Kapitel 16), das explizit vor Abschluss einer Aufgabe laufen muss.

---

## 10. Auto-Memory

Auto-Memory ist keine Datei innerhalb des Projekts: Sie ist eine globale, sitzungsübergreifende Claude-Code-Funktion, die außerhalb jedes Repos unter `~/.claude/projects/<projekt-pfad>/memory/` liegt. Ein `MEMORY.md` dient als Index (die ersten 200 Zeilen werden bei jeder Sitzung geladen), einzelne Themendateien werden bei Bedarf nachgeladen. Anders als das ältere Vorgänger-Tutorial es beschrieb, ist das kein rein manuell per „Merke dir" getriggerter Mechanismus. Claude legt projektbezogene Erkenntnisse eigenständig an, wenn sie über die aktuelle Sitzung hinaus nützlich sind.

Vier Kategorien sind vorgesehen: **user** (Rolle, Vorlieben, Wissensstand des Nutzers), **feedback** (was Claude beim nächsten Mal anders oder genauso wieder machen soll — sowohl Korrekturen als auch bestätigte Vorgehensweisen), **project** (laufende Vorhaben, Entscheidungen, Deadlines) und **reference** (Verweise auf externe Systeme wie Ticket-Tracker oder Dashboards). Für `BKKAtomium` existiert im globalen Memory-Index bereits ein realer Eintrag zur Entstehung der Architektur dieses Projekts: ein Beispiel dafür, dass Auto-Memory nicht spekulativ ist, sondern in diesem Tutorial-Kontext auch wirklich genutzt wird.

Wichtige Abgrenzung zu CLAUDE.md: CLAUDE.md wird von Menschen geschrieben und eingecheckt, gilt für das ganze Team. Auto-Memory wird von Claude geschrieben, liegt außerhalb des Repos, ist nicht versioniert und in der Regel personenbezogen (an den jeweiligen Entwickler-Rechner gebunden). Was ins Team gehört, gehört in CLAUDE.md oder eine Rule-Datei. Was eine persönliche Beobachtung über den Verlauf der Zusammenarbeit ist, gehört ins Memory. Aus Datenschutzsicht folgt daraus eine klare Regel für dieses Projekt (vertieft in Kapitel 21): Auto-Memory darf keine Versichertendaten, keine Patientendaten und keine sonstigen sensiblen Projektinhalte enthalten: Es ist für Arbeitskontext gedacht, nicht für fachliche Daten.

---

## 11. Agents — spezialisierte Sub-Instanzen

Ein Subagent ist eine `.md`-Datei unter `.claude/agents/` (projektweit) oder `~/.claude/agents/` (nutzerweit) mit YAML-Frontmatter. Pflichtfelder sind nur `name` und `description`; optional unter anderem `tools` (Werkzeug-Allowlist), `model`, `disallowedTools`, `permissionMode` (code.claude.com/docs/en/sub-agents). Ein Subagent läuft in einem eigenen Kontextfenster mit eigenem Systemprompt und eigenen Tool-Rechten: Ergebnisse kommen als Zusammenfassung zurück in die Hauptsitzung, ohne dass Zwischenschritte (Suchergebnisse, gelesene Dateien) den Hauptkontext belasten. Claude delegiert automatisch an einen passenden Subagenten, wenn dessen `description` zur Aufgabe passt, oder er wird explizit mit `@agent-name` angesprochen.

`BKKAtomium` hat zwei projekteigene Subagenten, beide mit `tools: Read, Grep, Glob`. Sie können also strukturell nicht schreiben, nicht bauen, keine Bash-Befehle ausführen:

| Agent | Prüft gegen | Einsatz |
|---|---|---|
| `concurrency-reviewer` | `.claude/rules/concurrency.md` — MainActor-Isolation, Sendable, Cancellation, verbotene Muster (`DispatchQueue`, `Task.detached`, `nonisolated(unsafe)`) | nach Arbeit an ViewModels/Datenschicht, vor Feature-Abschluss |
| `swiftui-reviewer` | `.claude/rules/swiftui.md` + `.claude/rules/accessibility.md` — dumme Views, Theme-Tokens, ViewState-Vollständigkeit, Accessibility, iPad | nach UI-Arbeit, vor Feature-Abschluss |

Beide liefern einen Befund mit Datei- und Zeilenangabe, keine automatische Korrektur. Die einzige technisch erzwungene Grenze ist die Tool-Beschränkung selbst: Inhaltlich bleiben es Meinungsberichte, kein Gate. Kapitel 12 und 13 zeigen beide Agents an echten Beispielen im Einsatz.

### Abgrenzung: Skill, Rule, Hook, Agent

Vier Mechanismen, die leicht verwechselt werden, weil sie alle „Verhalten steuern”:

- **Rule**: Kontext/Verbot, das für *jede* Aufgabe im Themenbereich gilt, unabhängig davon ob explizit aufgerufen.
- **Skill**: eine *abrufbare* Anleitung für einen wiederkehrenden, mehrschrittigen Arbeitsablauf.
- **Hook**: ein *deterministisch* ausgeführtes Shell-Kommando an einem festen Lebenszyklus-Punkt, unabhängig vom Modell.
- **Agent**: eine *isolierte* Sub-Instanz mit eigenem Kontext und eigenen Tool-Rechten für eine klar abgegrenzte Teilaufgabe.

## 12. Beispiel A: Ein neues Feature entwickeln

**Wichtiger Hinweis zu diesem Beispiel:** Dieses Tutorial ändert selbst keinen Swift-Code im Projekt: Das war eine ausdrückliche Vorgabe für seine Erstellung. Das folgende Beispiel ist deshalb ein vollständig durchgespielter, realistischer Workflow, kein tatsächlich vorgenommener Commit. Es ist bewusst kein erfundenes Spielbeispiel, sondern die konkrete Umsetzung einer echten, in `.claude/product/GAPS.md` dokumentierten Lücke: Vier der sechs Dashboard-Kacheln im Start-Tab zeigen aktuell nur `HomeComingSoonView` statt eines echten Flows, darunter die Kachel „Krankengeld”.

In den Dialog-Abschnitten steht **Du** für den Prompt des Entwicklers, **Claude Code** für die Reaktion, **Prüfung** für den Schritt, den der Entwickler danach selbst noch macht. Claude Code liefert nie ein „fertig, ungeprüft übernehmen”.

### Schritt 1: Anforderung verstehen

**Du:** „Die Dashboard-Kachel ‚Krankengeld’ zeigt aktuell `HomeComingSoonView`. Baue eine echte Krankengeld-Übersicht: Liste laufender und abgeschlossener Krankengeld-Fälle mit Status, Tagessatz und Zeitraum. Nutze Mock-Daten, kein Backend.”

Das ist präzise genug für den nächsten Schritt: Ein konkretes Ziel, ein Verweis auf eine reale Codestelle, eine explizite Absage an erfundene Backend-Anbindung.

### Schritt 2: Relevante Dateien und Rules ermitteln

**Claude Code** liest zuerst `.claude/rules/product-scope.md` (Abschnitt „Dashboard-Kachel-Grid”) und `.claude/product/Home.md` sowie `.claude/product/GAPS.md`, um zu bestätigen, dass „Krankengeld” wirklich als offene Kachel dokumentiert ist und nicht mit einem anderen Feature verwechselt wird. Danach `.claude/rules/architecture.md` (Composer-Pattern-Struktur), `.claude/rules/mocks.md` (Mock-Bauplan) und `.claude/rules/testing.md` (Pflichtfälle).

### Schritt 3: Plan-Modus verwenden

**Du** aktivierst Plan-Modus (`Shift+Tab`). **Claude Code** schlägt vor: neues Feature `SickPay` nach dem Composer-Pattern, Domain-Modell `SickPayCase` (Status `active`/`completed`/`pending`, Zeitraum, Tagessatz), ein `SickPayRepository`-Protokoll mit `fetchAll() async throws -> [SickPayCase]`, eine `SickPayRepositoryMock` mit deterministischem Seed über `MockBehavior`, ein `SickPayViewModel` mit `ViewState<[SickPayCase]>`, eine Liste mit Status-Badges, Einbindung

als `HomeDestination`-Fall statt des bisherigen `HomeComingSoonView`-Aufrufs für diese Kachel, plus Tests. **Prüfung:** Der Plan wird bestätigt: kein Repository-Zugriff auf ein echtes Backend, kein Overengineering für eine Demo-Funktion.

### Schritt 4: Architektur prüfen

**Claude Code** hält sich an das in Kapitel 3 beschriebene Schema: `SickPayRepository` kennt nur Domain und `Core/Errors`, das ViewModel kennt nur den Interactor beziehungsweise (weil hier keine Kombination mehrerer Repositories nötig ist) direkt das Repository-Protokoll, die View kennt nur ihr ViewModel und Closures für Navigation.

### Schritt 5: Feature implementieren — mit `creating-feature`

**Du:** „/creating-feature SickPay SickPayCase" (der Skill ruft dabei intern `Scripts/generate_feature.sh SickPay SickPayCase` auf, das die zehn Gerüst-Dateien aus `Templates/*.template` erzeugt).

Domain-Modell:

```
/// Ein einzelner Krankengeld-Fall mit Zeitraum, Tagessatz und Status.
struct SickPayCase: Sendable, Hashable, Identifiable {
    let id: UUID
    /// Beginn des Krankengeld-Zeitraums.
    let startDate: Date
    /// Ende des Zeitraums, `nil` solange der Fall noch läuft.
    let endDate: Date?
    /// Tagessatz in Cent, um Rundungsfehler bei Fließkommazahlen zu
vermeiden.
    let dailyRateCents: Int
    let status: SickPayStatus
}

enum SickPayStatus: String, Sendable, Hashable {
    case pending, active, completed
}
```

Repository-Protokoll und Mock (Auszug, nach dem in `.claude/rules/mocks.md` vorgeschriebenen Bauplan):

```
protocol SickPayRepository: AnyObject, Sendable {
    func fetchAll() async throws -> [SickPayCase]
}

actor SickPayRepositoryMock: SickPayRepository {
    private var behavior: MockBehavior
    private(set) var fetchAllCallCount = 0
    private let seed: [SickPayCase]

    init(behavior: MockBehavior = .realistic, now: @escaping @Sendable ()
-> Date) {
        self.behavior = behavior
        self.seed = Self.makeSeed(now: now)
    }

    func fetchAll() async throws -> [SickPayCase] {
        fetchAllCallCount += 1
        try await behavior.simulate()
        return seed
```

```
    }

    func setBehavior(_ behavior: MockBehavior) { self.behavior = behavior }
}
```

ViewModel mit dem projektweiten Vier-Zustände-Muster:

```
@MainActor @Observable final class SickPayViewModel {
    private(set) var state: ViewState<[SickPayCase]> = .idle
    private let repository: any SickPayRepository

    init(repository: any SickPayRepository) { self.repository =
repository }

    func load() async {
        state = .loading
        do {
            state = .loaded(try await repository.fetchAll())
        } catch is CancellationError {
            // Zustand bleibt unverändert — Abbruch ist kein Fehler.
        } catch let error as AppError {
            state = .failed(error)
        } catch {
            state = .failed(.unknown(underlying: error))
        }
    }
}
```

Einbindung in die Navigation: In der `HomeFlowView` (siehe Kapitel 3) wird der bisherige Aufruf von `HomeComingSoonView` für den Fall `.sickPay` in `HomeDestination` durch `SickPayComposer.compose(dependencies:)` ersetzt.

### Schritt 6: Mock-Daten ergänzen

Der Seed liefert deterministische Werte: feste `Date(timeIntervalSince1970:)`-Zeitstempel statt `Date()`, feste UUIDs, ein aktiver und ein abgeschlossener Fall, damit sich beide Status-Darstellungen in Vorschau und Test beobachten lassen.

### Schritt 7: Tests schreiben — mit `writing-swift-tests`

```
@Suite("SickPayViewModel")
@MainActor
struct SickPayViewModelTests {
    @Test("load setzt loaded mit den Fällen aus dem Repository")
    func load_success_setsLoaded() async {
        // Given
        let repository = SickPayRepositoryMock(now:
{ .init(timeIntervalSince1970: 1_752_000_000) })
        let sut = SickPayViewModel(repository: repository)

        // When
        await sut.load()

        // Then
        #expect(sut.state.isLoaded)
        #expect(await repository.fetchAllCallCount == 1)
    }

    @Test("load setzt failed bei einem Repository-Fehler")
    func load_failure_setsFailed() async {
```

```
        // Given
        let repository =
SickPayRepositoryMock(behavior: .failing(.network), now:
{ .init(timeIntervalSince1970: 0) })
        let sut = SickPayViewModel(repository: repository)

        // When
        await sut.load()

        // Then
        #expect(sut.state.hasFailed)
    }
}
```

Nach `.claude/rules/testing.md` fehlen hier noch die Pflichtfälle „leeres Ergebnis" und „Abbruch". Im echten Arbeitsschritt würde Claude Code beide ergänzen, bevor der Skill die Aufgabe als abgeschlossen meldet.

### Schritt 8: Accessibility prüfen

Jede Status-Zeile bekommt `accessibilityLabel` (lokalisiert über `LocalizationKeys.Accessibility.*`) und `accessibilityIdentifier` nach Schema `sickpay.caseRow`, Pflicht laut `.claude/rules/accessibility.md`, nicht optional.

### Schritt 9: Security und Datenschutz prüfen

Ein Krankengeld-Fall ist ein sensibler Gesundheitsbezug. `.claude/rules/security-and-privacy.md` verbietet Logging von Diagnosen, Behandlungen und Antragsinhalten. Hier greift das indirekt, weil `os.Logger`-Aufrufe in diesem Feature ausschließlich mit anonymen technischen IDs arbeiten dürfen, nie mit Fall-Details.

### Schritt 10: Review-Agent ausführen

**Du:** „@swiftui-reviewer prüfe SickPayListView" und „@concurrency-reviewer prüfe SickPayRepositoryMock". Beide Agents laufen read-only und liefern einen Befund mit Datei- und Zeilenangabe, keine automatische Korrektur, siehe Kapitel 11.

### Schritt 11: Build und Tests ausführen

```
xcodegen generate
xcodebuild -project BKKAtomium.xcodeproj -scheme BKKAtomium \
  -destination 'platform=iOS Simulator,name=iPhone 16 Pro' build test
swiftlint lint --quiet
./Scripts/check_project.sh
```

`xcodegen generate` ist Pflicht vor dem ersten Build nach neuen Dateien, sonst kennt Xcode die neuen Swift-Dateien nicht und meldet „Cannot find type in scope" für Symbole, die tatsächlich existieren (siehe `.claude/product/PROCESS.md`).

### Schritt 12: Dokumentation aktualisieren

`.claude/product/Home.md` verliert den Eintrag „Krankengeld ist Platzhalter", `.claude/product/GAPS.md` wird um diesen Punkt gekürzt, im selben Arbeitsschritt, nicht als

Nachtrag. Das ist keine Kür: `.claude/product/README.md` benennt eine veraltete Produktdokumentation ausdrücklich als schlechter als gar keine.

## 13. Beispiel B: Ein bestehendes Feature erweitern

Auch dieses Beispiel ist ein durchgespielter Workflow, kein tatsächlich vorgenommener Commit, aber wieder an einer echten, dokumentierten Lücke: `.claude/product/GAPS.md` nennt für das Bonus-Feature, dass das IBAN-Feld beim „Prämie beantragen"-Formular nur auf „nicht leer" prüft, keine Format- oder Prüfziffernvalidierung, eine dokumentierte Abweichung von der allgemeinen Validierungsregel in `.claude/rules/security-and-privacy.md` („Alle Nutzereingaben validieren: Länge, Format, Bereich").

### Bestandsanalyse

**Du:** „`ApplyBonusViewModel.canSubmit` prüft die IBAN nur auf `!isEmpty`. Ergänze eine echte IBAN-Validierung (Länderformat, Prüfziffer nach ISO 7064 Mod 97-10), ohne das bestehende Formular-Layout zu ändern."

**Claude Code** liest zuerst die bestehende `ApplyBonusView`/`ApplyBonusViewModel` sowie ein bereits vorhandenes, ähnlich aufgebautes Validator-Muster im Projekt (`BankAccountValidatorTests.swift` existiert bereits als Testdatei, ein Hinweis darauf, dass es analoge Validierungslogik zumindest für Bankverbindungen im Profil-Feature bereits gibt, an der sich die neue Regel orientieren sollte, statt einen abweichenden Stil zu erfinden).

### Schutz vorhandenen Verhaltens

Vor der Änderung wird die aktuelle, bewusst schwache Prüfung durch einen Test explizit dokumentiert, damit die anschließende Verschärfung sichtbar eine Verhaltensänderung ist, kein stiller Nebeneffekt:

```
@Test("canSubmit ist aktuell true bei rein nicht-leerer, ungültiger IBAN
(zu behebende Lücke)")
func canSubmit_nonEmptyInvalidIban_isTrueBeforeFix() {
    let sut = ApplyBonusViewModel(repository: BonusRepositoryMock(now:
{ .init(timeIntervalSince1970: 0) }))
    sut.iban = "DE00INVALID"
    #expect(sut.canSubmit) // dokumentiert den Ist-Zustand vor der Änderung
}
```

Dieser Test wird nach der Änderung durch das erwartete neue Verhalten ersetzt, nicht einfach gelöscht: Die Lücke soll sichtbar bleiben, bis sie geschlossen ist.

### Kleine, kontrollierte Änderung

Statt Validierungslogik im ViewModel zu verstreuen, entsteht ein `IBANValidator` in `Features/Bonus/Domain/` (Validierung gehört laut `.claude/rules/architecture.md` in Domain-`…Validator`-Typen, nie ins ViewModel selbst):

```
/// Prüft IBAN-Länderformat und Prüfziffer nach ISO 7064 Mod 97-10.
enum IBANValidator {
    static func isValid(_ raw: String) -> Bool {
        let iban = raw.replacingOccurrences(of: " ", with: "").uppercased()
        guard iban.count >= 15, iban.count <= 34 else { return false }
```

```
        let rearranged = String(iban.dropFirst(4) + iban.prefix(4))
        let numeric = rearranged.compactMap { char -> String? in
            guard let ascii = char.asciiValue else { return nil }
            return char.isNumber ? String(char) : String(ascii - 55)
        }.joined()
        guard let remainder = numeric.reduce(0 as UInt64) { partial, digit
in
            (partial * 10 + UInt64(String(digit))!) % 97
        } as UInt64? else { return false }
        return remainder == 1
    }
}
```

`ApplyBonusViewModel.canSubmit` ruft jetzt `IBANValidator.isValid(iban)` statt `!iban.isEmpty`. Layout und restliche Formularlogik bleiben unverändert. Das ist der Kern einer kontrollierten Erweiterung: eine Regel wird strenger, ohne dass die Umgebung mitverändert wird.

### Review

**Du:** „@concurrency-reviewer prüfe `IBANValidator`" (reine Wertfunktion ohne Nebenläufigkeit, sollte ohne Befund durchlaufen) und ein regulärer `writing-swift-tests`-Durchlauf für Grenzwerte: leere IBAN, zu kurze IBAN, falsche Prüfziffer, gültige Test-IBAN.

### Dokumentation

`.claude/product/GAPS.md` und `.claude/product/Bonus.md` verlieren den Eintrag zur IBAN-Lücke, wieder im selben Arbeitsschritt wie die Code-Änderung, nicht danach.

---

## 14. Beispiel C: Einen Fehler beheben

Dieses dritte Beispiel folgt dem Skill `fixing-ios-bug` an einem weiteren echten, in `.claude/product/GAPS.md` dokumentierten Fund im Postfach-Feature: „`sendMessage(...)` im Mock fügt die gesendete Nachricht nicht in die interne Liste ein; nach dem Verfassen erscheint sie nicht automatisch im Verlauf, solange kein erneuter `fetchMessages()`-Aufruf erfolgt."

### Fehlerbeschreibung

**Du:** „Nach dem Senden einer neuen Nachricht im Postfach bleibt der Nachrichtenverlauf unverändert, bis man die Liste manuell neu lädt. Erwartet: Die gesendete Nachricht erscheint sofort."

### Reproduktion

**Claude Code** reproduziert den Fehler nicht im Simulator, sondern am direktesten über einen Test gegen `PostfachRepositoryMock`: `sendMessage(...)` aufrufen, danach `fetchMessages()`: Die neue Nachricht fehlt im Ergebnis.

### Betroffene Architekturkomponente

Die Ursache liegt ausschließlich im Mock, nicht im ViewModel oder in der View: `PostfachRepositoryMock.sendMessage(...)` simuliert zwar Latenz und Erfolg über

`MockBehavior`, schreibt das übergebene `PostfachMessage` aber nicht in den internen `storage`-Array: Der Speicherzustand des Mocks und das, was er zurückgibt, laufen auseinander.

### Test vor dem Fix (rot)

```
@Test("sendMessage fügt die Nachricht sofort zum internen Verlauf hinzu")
func sendMessage_success_appearsInSubsequentFetch() async throws {
    // Given
    let sut = PostfachRepositoryMock(now: { .init(timeIntervalSince1970:
1_752_000_000) })
    let newMessage = PostfachMessage.fixture(subject: "Rückfrage zur
Erstattung")

    // When
    try await sut.sendMessage(newMessage)
    let messagesAfterSend = try await sut.fetchMessages()

    // Then
    #expect(messagesAfterSend.contains { $0.id == newMessage.id }) //
schlägt vor dem Fix fehl
}
```

### Kleinster sinnvoller Fix

```
func sendMessage(_ message: PostfachMessage) async throws {
    sendMessageCallCount += 1
    lastSentMessage = message
    try await behavior.simulate()
    storage.append(message) // fehlte — Ursache des Bugs
}
```

Eine einzelne Zeile, an der einzigen Stelle, die den beobachteten Fehler tatsächlich erklärt, kein zusätzliches Refactoring, keine Änderung an `PostfachViewModel` oder `ComposeMessageView`, obwohl beide am eigentlichen Symptom „beteiligt" wirkten.

### Test nach dem Fix (grün)

Derselbe Test aus dem vorigen Abschnitt läuft jetzt durch, ohne verändert zu werden: Das ist das Kriterium für einen minimalen Fix. Der Test beschreibt das gewünschte Verhalten, nicht die Implementierung.

### Review und Dokumentation

**Du:** „@concurrency-reviewer prüfe die Änderung an `PostfachRepositoryMock`": bestätigt, dass `storage.append` innerhalb der `actor`-Isolation bleibt und keine neue Nebenläufigkeitsannahme einführt. `.claude/product/GAPS.md` und `.claude/product/Postfach.md` verlieren den entsprechenden Eintrag.

## 15. Teststrategie

`.claude/rules/testing.md` schreibt Swift Testing (`@Suite`/`@Test`/`#expect`/`#require`) für Unit-Tests vor, XCTest ausschließlich für UI-Tests (`XCUIApplication`): kein gemischter Einsatz. Das lässt sich am realen Code bestätigen:
`BKKAtomiumTests/Features/Bonus/BonusViewModelTests.swift` nutzt

`@Suite("BonusViewModel")` mit `@Test("…")`-Titeln und `#expect(...)`; `BKKAtomiumUITests/BonusFlowUITests.swift` ist eine `XCTestCase`.

### Was getestet wird

ViewModels ausschließlich gegen Mock-Repositories, nie gegen `Live`-Implementierungen. Für jeden Ladepfad sind vier Fälle Pflicht: Erfolg, leeres Ergebnis, Fehler, Abbruch (`CancellationError`, der den Zustand unverändert lässt statt `.failed` zu setzen). Für Aktionen zusätzlich: Zustand während der Aktion (`isSubmitting`), CallCount und Parameter-Capture am Mock danach. Domain-Validatoren werden als reine Ein-/Ausgabe-Funktionen getestet, inklusive Randwerten — siehe `IBANValidator` in Kapitel 13. Views und Composer werden nicht unit-getestet; Kernflüsse laufen über UI-Tests.

### Determinismus

Kein `Task.sleep` als Synchronisation, kein `Date()`/`Date.now` in Fixtures oder Erwartungswerten: feste Zeitstempel, Zeit über injizierte `now`-Closure (siehe `SickPayRepositoryMock` und `PostfachRepositoryMock` in den Beispielen). `MockBehavior` (`Core/Mocks/MockBehavior.swift`) kapselt Latenz, injizierten Fehler und `Task.checkCancellation()` vor und nach der Latenz — jeder Mock ruft `try await behavior.simulate()` als ersten produktiven Schritt jeder Methode auf.

### Ehrlicher Blick auf den Ist-Zustand

Die Testabdeckung ist nicht überall gleich weit: `BKKAtomiumTests/` deckt 16 der Features/Komponenten ab, aber für das komplette Service-Feature (drei ViewModels: `CertificateRequestViewModel`, `EGKLostViewModel`, `EGKMissingViewModel`) existiert bislang keine einzige Testdatei: eine reale, im Projekt so dokumentierte Lücke, kein Widerspruch zur Regel selbst. Für eine Demo-/Übungs-App ist das eine bewusste Priorisierung, kein Freibrief: Neuer oder geänderter Code bekommt in jedem Fall Tests, aber es wird nicht rückwirkend jede bestehende Lücke in einer Aufgabe mitgeschlossen, die eigentlich woanders hingehört.

Eine Continuous-Integration-Pipeline (GitHub Actions, Fastlane o. Ä.) existiert für dieses Projekt aktuell nicht. Das ist eine gezielte Auslassung dieses Tutorials: der komplette Build-/Test-/Lint-Zyklus läuft manuell, über die in Kapitel 16 gezeigten Befehle.

---

## 16. Qualitätssicherung im Alltag

### Der tägliche Zyklus

```
xcodegen generate
xcodebuild … build test
swiftlint lint --quiet
./Scripts/check_project.sh
```

Diese Reihenfolge steht so in `.claude/product/PROCESS.md`: nicht nach jeder einzelnen Datei, sondern nach jedem in sich abgeschlossenen Änderungsblock. `check_project.sh` prüft acht Stufen in einem Durchlauf, ohne beim ersten Fund abzubrechen, und meldet am Ende `BESTANDEN` oder `NICHT BESTANDEN`:

| Stufe | Prüfung | Gate? |
|---|---|---|
| P1 | Farb-/RGB-Literale außerhalb `DesignSystem/` | ja |
| P2 | fixe Font-Größen (`.font(.system(size:)`) | ja |
| P3 | verbotene Concurrency-Muster (`DispatchQueue`, `Task.detached`, `nonisolated(unsafe)`, `@unchecked Sendable`) | ja |
| P4 | Force-Konstrukte (`try!`, `as!`, `fatalError()`) | ja |
| P5 | hartcodierte UI-Strings (heuristisch, Debug-Ausnahmen) | ja |
| P6 | `LocalizationKeys` ↔ `Localizable.xcstrings`, de+en vollständig | ja (Waisen-Keys nur Warnung) |
| P7 | `try?`-Nutzung | nein, nur Warnung |
| P8 | Singletons (`static let/var shared`) | ja |

Ergänzend erzwingt `.swiftlint.yml` automatisch fünf projekteigene Regeln (`no_dispatch_after`, `no_print_in_production`, `no_fixed_font_size`, `no_environment_object`, `viewmodel_main_actor`) sowie die Standardregeln `force_cast`/`force_try`/`force_unwrapping` als Fehler.

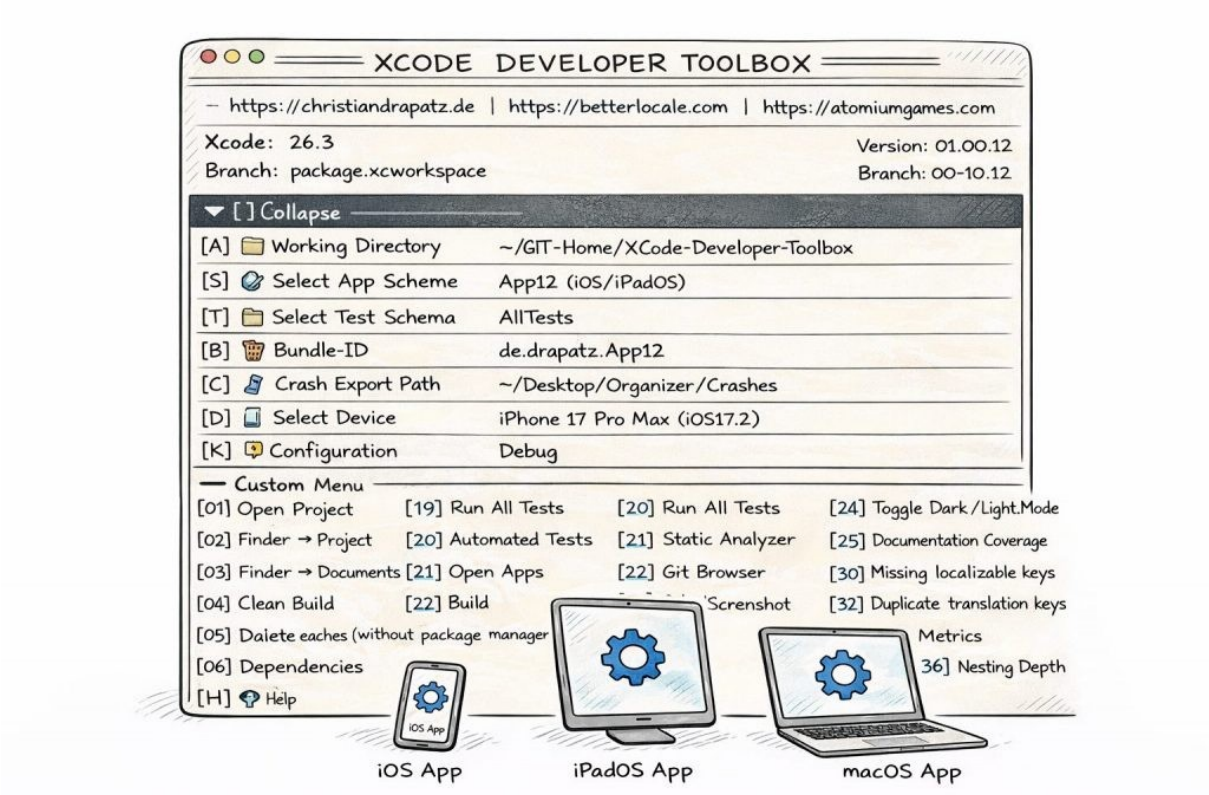
## Wann welches Werkzeug

Die drei Review-Mechanismen aus den vorherigen Kapiteln haben unterschiedliche Einsatzzeitpunkte: Der SwiftLint-Hook (Kapitel 9) läuft automatisch nach jeder Datei: informativ, ohne zu blockieren. `check_project.sh` läuft manuell, aber verbindlich vor Abschluss jeder Aufgabe: ein echtes Gate. Die Review-Agents (Kapitel 11) und der Skill `reviewing-architecture` laufen gezielt nach abgeschlossener Feature- oder UI-Arbeit, wenn strukturelle statt rein syntaktische Fragen im Vordergrund stehen.

## Command-Line-Werkzeuge jenseits von Claude Code

Claude Code deckt den Build-/Test-/Review-Zyklus ab, aber nicht jede Xcode-nahe Aufgabe lässt sich sinnvoll über Prompts erledigen: Simulator-Verwaltung und wiederkehrende Build-Housekeeping-Aufgaben gehören oft eher ins Terminal. Ein Werkzeug, das genau diese Lücke abdeckt, ist **xcodex** von

Christian Drapatz (xcodexcli.com): ein terminalbasiertes Kommandozeilenwerkzeug für Xcode-Projekte, das Build, Tests, Simulatorsteuerung, Geräteverwaltung, Archive, TestFlight-/App-Store-Auslieferung, Lokalisierung, Code Coverage, Crash-Analyse und Projektpflege bündelt. Es steht in keiner Verbindung zu Apple oder Xcode selbst, ist aber für mit Xcode entwickelte Projekte gedacht.




*xcodex — terminalbasiertes Kommandozeilenwerkzeug für Xcode-Projekte*

Für den in diesem Kapitel gezeigten Zyklus ist vor allem relevant, dass xcodex die App bauen und testen, Build- und Ableitungsverzeichnisse löschen und den Simulator steuern kann, inklusive der Verwaltung mehrerer Simulatoren gleichzeitig, etwa um dieselbe App parallel unter mehreren iOS-Versionen zu testen. Es gibt ein Tutorial zu xcodex auf Deutsch und Englisch; das Werkzeug ist kostenlos und ohne Lizenz nutzbar, eine Spende an den Autor ist möglich, aber keine Voraussetzung.

---

## 17. Dokumentationsstandard

`.claude/rules/swift.md` verlangt `///`-Dokumentation auf Deutsch, während Bezeichner (Typen, Methoden, Properties) englisch bleiben: Apple- und Technik-Begriffe wie SwiftData, `@MainActor` oder `Sendable` werden dabei nicht übersetzt. Dokumentationspflicht besteht für jede `struct`/`class`/`actor`/`enum`/`protocol`, jede Property (inklusive `nil`-Bedeutung und Einheit, etwa Cent statt Euro bei `dailyRateCents` in Kapitel 12), jede Funktion mit nicht offensichtlichem Verhalten, jeden `enum case` und jede Concurrency-Annotation mit Begründung. **Nicht** dokumentiert werden SwiftUI-`body`-Properties und selbsterklärende private Hilfsmethoden.

Die Leitregel lautet „Warum, nicht Was": Ein Kommentar, der nur den Namen wiederholt, ist streng genommen ein Verstoß, keine Nullnummer. Ein Beispiel aus Kapitel 12: Der Kommentar zu

`dailyRateCents` erklärt, *warum* Cent statt Euro verwendet wird (Rundungsfehler bei Fließkommazahlen vermeiden), nicht, *dass* es ein Tagessatz ist, das sagt der Name bereits.

---

## 18. Code-Stil und Formatierung

Verbindlich nach `.claude/rules/swift.md`: maximal 120 Zeichen pro Zeile, vier Leerzeichen Einrückung, keine Tabs, alphabetisch sortierte Imports, kein `self.` außerhalb von `init` (außer zur Disambiguierung), keine Force-Unwraps (`guard let`/`if let`/`??` statt `!`).

`.swiftlint.yml` erzwingt konkrete Schwellenwerte automatisch: Zeilenlänge Warnung ab 120, Fehler ab 150 Zeichen; Funktionslänge Warnung ab 40, Fehler ab 60 Zeilen; Dateilänge Warnung ab 400, Fehler ab 600 Zeilen; Typlänge Warnung ab 300, Fehler ab 450 Zeilen; Bezeichner-Mindestlänge zwei Zeichen (mit Ausnahmen wie `id`, `vm`, `x`, `y`). Ergänzend sind `sorted_imports`, `closure_spacing` und `empty_count` als Opt-in-Regeln aktiv.

Die Namenskonvention aus `.claude/rules/swift.md` ist bewusst eng an das Composer-Pattern gekoppelt: nicht generische Swift-Konventionen, sondern feste Baustein-Namen: `<Feature>Composer`, `<Feature>Environment`, `<Feature>ViewModel`, `<Feature>View`/`<Feature>FlowView`, `<Feature>Destination`, `<Entity>Repository`/`<Entity>RepositoryMock`/`<Entity>RepositoryLive`, `<Feature>Interactor`, `<Feature>Navigator`, `<Entity>Validator`. Wer eine dieser Bezeichnungen abwandelt, ohne dass ein Skill sie erzeugt hat, weicht damit implizit von der erwarteten Struktur ab, ein guter Anhaltspunkt für den Architektur-Review-Agent aus Kapitel 11, gezielt danach zu suchen.

## 19. Automatisierung

Zwei Ebenen sind zu unterscheiden: was tatsächlich automatisch läuft, und was Claude Code auf Zuruf erledigt.

**Tatsächlich automatisch** läuft in diesem Projekt nur der SwiftLint-Hook aus Kapitel 9: nach jedem `Write`/`Edit`, informativ, ohne Gate-Wirkung. Eine Continuous-Integration-Pipeline, die Build, Tests und `check_project.sh` bei jedem Push automatisch ausführt, existiert für `BKKAtomium` aktuell nicht. Das ist eine **empfohlene Erweiterung, aktuell nicht Bestandteil des Projekts**: Der komplette Zyklus aus Kapitel 16 läuft manuell.

**Auf Zuruf** automatisiert Claude Code wiederkehrende, mehrschrittige Aufgaben, die sonst jedes Mal neu erklärt werden müssten: `Scripts/generate_feature.sh` über den Skill `creating-feature` (Kapitel 6, 12), `check_project.sh` plus SwiftLint als fester Abschluss jeder Aufgabe (Kapitel 16), die beiden Review-Agents nach UI- beziehungsweise Datenschicht-Arbeit (Kapitel 11), und der Skill `reviewing-architecture` als gelegentlicher, projektweiter Gesundheitscheck, sinnvoll zum Beispiel vor einem größeren Merge, nicht bei jeder einzelnen Datei.

Der Unterschied ist wichtig: Ein Hook läuft garantiert, ein Skill nur, wenn er passend erscheint oder explizit aufgerufen wird. Wer sich auf „Claude Code merkt das schon” statt auf ein echtes Gate verlässt, verwechselt beide Ebenen.

## 20. Debug- und Mock-Architektur

`BKKAtomium/Debug/` ist laut `CLAUDE.md` ausschließlich hinter `#if DEBUG` erreichbar und bündelt Debug-Menü und Mock-Steuerung: Code, der in einem Release-Build nicht einmal kompiliert wird, nicht nur zur Laufzeit versteckt ist. Das ist ein wichtiger Unterschied: Ein reines Laufzeit-Flag könnte versehentlich in Produktion aktiv bleiben, ein `#if DEBUG`-Block kann es nicht.

Statt eines eigenen `FakeLoginComposer` (der laut `.claude/rules/architecture.md` grundsätzlich möglich wäre) nutzt dieses Projekt einen schlankeren Mechanismus: ein `--skip-login`-Launch-Argument, das `BKKAtomiumApp.init()` auswertet und direkt mit einem Test-Profil in den Hauptbereich der App springt. Für UI-Tests ist das der Normalfall: `.claude/rules/testing.md` verlangt definierte Startzustände über Launch-Argumente, die die App nachweislich auswertet, und lehnt ein Launch-Argument ohne echte App-Auswertung als Bug ab.

Mock-Steuerung zur Laufzeit läuft über `MockBehavior.setBehavior(_:)` (Kapitel 15): jeder Mock kann während der Entwicklung testweise auf Fehler, hohe Latenz oder leere Ergebnisse umgeschaltet werden, ohne den Code zu ändern. Das ist der eigentliche Wert der strikten Mock-Architektur aus Kapitel 3: Fehlerzustände lassen sich gezielt provozieren, statt auf einen echten Backend-Ausfall warten zu müssen.

## 21. Security und Datenschutz

Weil es sich um eine Krankenkassen-App handelt (auch wenn sie nur mit Mock-Daten arbeitet), behandelt `.claude/rules/security-and-privacy.md` das Thema nicht als Anhängsel, sondern als eigenen, ausführlichen Regelblock. Für die Arbeit mit Claude Code ergeben sich daraus mehrere konkrete Konsequenzen:

### Logging-Verbote gelten auch für Mock-Daten

Nie geloggt werden darf (auch nicht in Debug-Hilfsausgaben, die versehentlich stehen bleiben): die Versicherungsnummer, Name, Geburtsdatum, Adresse, E-Mail, Telefonnummer, Diagnosen, Medikamente, Behandlungen, Antragsinhalte, Auth-Tokens, Session-IDs, Passwörter, PINs, biometrische Daten, Kontodaten oder IBAN. Erlaubt sind ausschließlich anonyme technische IDs, Fehlertypen und HTTP-Statuscodes, und ausschließlich über `os.Logger.print()` ist nur hinter `#if DEBUG` zulässig. Das gilt ausdrücklich auch für Mock-Implementierungen: Ein `SickPayRepositoryMock` (Kapitel 12) darf beim Simulieren eines Fehlers dessen Typ loggen, aber niemals den fiktiven Fall-Inhalt.

### Keine echten Versichertendaten in Mock- oder Seed-Dateien

Alle Mock-Seeds in diesem Projekt sind erfunden: Namen, Adressen, IBANs, Diagnosen. Das bleibt so, auch wenn ein Mock „realistischer" wirken soll. Realitätsnähe entsteht über Vielfalt und Plausibilität der Testfälle (siehe `.claude/rules/mocks.md`), nicht über tatsächliche, und sei es anonymisierte, Personendaten Dritter.

## Keine Zugangsdaten in Markdown-Dateien

CLAUDE.md, Rules, Skills und die Produktdokumentation sind Klartext-Markdown, eingecheckt in Git. Kein API-Schlüssel, kein Passwort, kein Token gehört dort hinein, auch nicht „nur zu Testzwecken". Der Demo-Login des Projekts validiert laut `.claude/rules/security-and-privacy.md` bewusst nur Regeln (Format, Länge), vergleicht aber nie gegen ein eingebautes Geheimnis: Es gibt schlicht kein Klartext-Passwort, das versehentlich landen könnte.

## Secrets und lokale Settings

`.claude/settings.local.json` ist der vorgesehene Ort für alles, was persönlich und nicht teilbar ist, in diesem Projekt aktuell nur zusätzliche Leserechte, aber genau hierhin gehören auch persönliche API-Schlüssel oder Zugangsdaten zu MCP-Servern, falls das Projekt einmal welche braucht. `.gitignore` schließt diese Datei bereits jetzt explizit aus.

## Datenminimierung, auch beim Prompten

Ein Prompt an Claude Code ist kein geschützter Raum. Wer über ein reales Datenschutz-Problem mit echten Nutzerdaten spricht, sollte das nicht mit echten Namen oder Versichertennummern tun, auch nicht in einer internen Sitzung. Für dieses Projekt ist das ohnehin durchgehend gegeben, weil ausschließlich mit Mock-Daten gearbeitet wird; es bleibt aber die richtige Grundhaltung für den Tag, an dem eine echte Backend-Anbindung entsteht.

## Berechtigungen als technische Grenze

`.claude/settings.json` verbietet `git push`, `git reset --hard` und `rm -rf` projektweit (Kapitel 4.7): eine von der Harness selbst durchgesetzte Grenze, kein Vertrauensvorschuss an das Modell. Das ist ein einfaches, aber wirksames Muster: Wo eine Aktion irreversibel oder für andere sichtbar ist, gehört sie in die `deny`-Liste, nicht in eine Bitte im Prompt.

## Review sicherheitsrelevanter Änderungen

Änderungen an `security-and-privacy.md` selbst, an Auth-Code oder an irgendetwas, das mit Keychain-Zugriff zu tun hat, verdienen ein normales Code-Review durch einen Menschen: Die beiden Review-Agents aus Kapitel 11 sind auf Concurrency und SwiftUI-Konventionen spezialisiert, nicht auf Sicherheitsanalyse. Für eine gezielte Sicherheitsprüfung des gesamten Projekts steht unabhängig von diesem Tutorial ein eigener, allgemeiner Skill zur Verfügung (`security-attack-check`), der Apple-Review-Perspektive und Angreifer-Perspektive kombiniert.

## Claude Code im Unternehmenskontext

Für ein Team, nicht nur einen Einzelentwickler, kommen organisationsweite Einstellungen hinzu (Kapitel 4.7): eine zentral verwaltete `managed-settings.json`, die Projekt- und Nutzereinstellungen nicht überschreiben können, sowie zentrale Kostenkontrolle über Spend-Limits auf Team-/Enterprise-Plänen. Für dieses Tutorial-Projekt ist das nicht relevant, weil es kein reales Team-Deployment ist, aber es ist der Mechanismus, über den ein echtes Krankenkassen-Entwicklungsteam durchsetzen würde, dass zum Beispiel die `deny`-Liste aus `settings.json` von keinem einzelnen Entwickler lokal aufgeweicht werden kann.

# 22. Zusammenarbeit im Team

## Was geteilt wird, was lokal bleibt

| Datei/Ordner | Geteilt (Git) | Lokal/persönlich |
|---|---|---|
| `CLAUDE.md` | ja | — |
| `CLAUDE.local.md` | — | ja (aktuell nicht angelegt, aber in `.gitignore` reserviert) |
| `.claude/rules/*.md`, `.claude/skills/*`, `.claude/agents/*`, `.claude/product/*` | ja | — |
| `.claude/hooks/*.sh` | ja | — |
| `.claude/settings.json` | ja | — |
| `.claude/settings.local.json` | — | ja (gitignored) |
| `project.yml`, `.swiftlint.yml`, `Scripts/`, `Templates/` | ja | — |
| `BKKAtomium.xcodeproj` | — | generiert, gitignored (aus `project.yml` via `xcodegen generate`) |
| Auto-Memory (`~/.claude/projects/…`) | — | immer persönlich, liegt außerhalb des Repos |

Die Faustregel: Alles, was das Verhalten von Claude Code für **jeden** definiert, der am Projekt arbeitet, gehört eingecheckt. Alles, was von der lokalen Maschine, den persönlichen Zugriffsrechten oder der individuellen Arbeitsweise abhängt, bleibt lokal.

## Widersprüche vermeiden

Zwei Mechanismen im Projekt verhindern gezielt Doppelpflege: `.claude/product/README.md` legt fest, dass `rules/` den Soll-Zustand und `product/` den Ist-Zustand beschreibt: bei Widerspruch ist klar, welche Datei recht hat. Und `.claude/rules/product-scope.md` ist ausdrücklich die *einzige* verbindliche Quelle für den Funktionsumfang; sie verbietet explizit, Funktionen „nachzubessern" oder Lücken stillschweigend zu ergänzen, ohne nachzufragen. Wo eine Änderung an einer Rule- oder Skill-Datei nötig wird, ändert sie sich an genau einer Stelle, nicht an einer Kopie in einem anderen Ordner.

## Änderungen an `.claude/` reviewen

Eine geänderte Rule-Datei oder ein neuer Skill ist kein Konfigurationsdetail, das nebenbei durchgeht: sie verändert, wie jeder im Team künftig mit Claude Code arbeitet. Im Projekt-Alltag heißt das: Änderungen

an `.claude/rules/`, `.claude/skills/` oder `.claude/agents/` durchlaufen dasselbe Pull-Request-Review wie Produktivcode, nicht weniger.

### Gemeinsame Definition of Done

Die siebenteilige Definition of Done aus `CLAUDE.md` gilt für jede Aufgabe, unabhängig davon, wer sie umsetzt, Mensch oder mit Unterstützung von Claude Code: Build ohne Warnungen und alle Tests grün, `check_project.sh` und SwiftLint bestanden, neue Strings in allen neun Sprachen übersetzt, neue ViewModels/Repositories mit Tests, interaktive Elemente mit Accessibility-Label und -Identifier, kein toter Code, und Abgleich gegen `product-scope.md`: nichts fehlt, nichts wurde dazuerfunden.

---

## 23. Onboarding neuer Entwickler

Ein praktischer Ablauf für den ersten Tag mit diesem Projekt:

1. **Repository einrichten**: Projekt klonen, `xcodegen` installieren (`brew install xcodegen`, falls nicht vorhanden).
2. **Xcode-Version prüfen**: laut `CLAUDE.md` wird Xcode 26.3 vorausgesetzt; eine abweichende Version zuerst gegen `xcodebuild -version` verifizieren, bevor an anderer Stelle nach Fehlern gesucht wird.
3. **Projekt bauen**:

   ```
   xcodegen generate
   xcodebuild -project BKKAtomium.xcodeproj -scheme BKKAtomium \
     -destination 'platform=iOS Simulator,name=iPhone 16 Pro' build
   ```
4. **Tests ausführen**: vollständig oder nur Unit-Tests:

   ```
   xcodebuild -project BKKAtomium.xcodeproj -scheme BKKAtomium \
     -destination 'platform=iOS Simulator,name=iPhone 16 Pro' test
   -only-testing:BKKAtomiumTests
   ```
5. **`CLAUDE.md` lesen**: nicht überfliegen. Sie ist unter 200 Zeilen genau deshalb, weil sie jeder wirklich lesen soll, nicht nur laden lässt.
6. **`.claude`-Struktur verstehen**: Kapitel 4.6 dieses Tutorials als Karte nutzen, welche Datei beantwortet welche Frage.
7. **Ersten Plan erstellen**: eine kleine, echte Aufgabe im Plan-Modus durchspielen, zum Beispiel eine der in Kapitel 1 genannten `GAPS.md`-Lücken, ohne sie sofort umzusetzen. Das zeigt, ob die relevanten Rules und Skills auch tatsächlich gefunden werden.
8. **Kleine eigene Änderung implementieren**: ein Bug aus `GAPS.md` nach dem Muster aus Kapitel 14, mit Regressionstest.
9. **Review durchführen**: passenden Review-Agent aus Kapitel 11 aufrufen, dazu `check_project.sh` und SwiftLint.
10. **Teamkonventionen prüfen**: Ergebnis gegen die Definition of Done aus Kapitel 22 abgleichen, bevor ein Pull Request entsteht.

Dieser Ablauf ist bewusst iterativ statt linear: Schritt 7 und 8 lassen sich gut am selben, kleinen `GAPS.md`-Eintrag durchspielen, sodass der erste eigene Beitrag von Anfang an dem echten Projektstandard entspricht statt ihn nachträglich anzugleichen.

---

## 24. Weitere Werkzeuge des Autors

Neben xcodex (Kapitel 16) pflegt Christian Drapatz unter der Marke **BetterLocale** (betterlocale.com) mehrere weitere Werkzeuge rund ums Schreiben, Dokumentieren, Lokalisieren und Analysieren von Crash-Logs, jeweils mit optionaler KI-Anbindung über einen selbst mitgebrachten API-Schlüssel:


*Weitere Werkzeuge des Autors*

| Tool | Kurzbeschreibung | Homepage | App Store |
|---|---|---|---|
| BetterLocale Crash | KI-gestützte Analyse von Crash- und Logdateien (iOS/macOS) | betterlocale.com/en-crash | App Store |
| BetterLocale MarkDown | KI-unterstütztes Schreiben und Bearbeiten in Markdown | betterlocale.com/en-markdown | App Store |
| BetterLocale Code | KI-gestützte | betterlocale.com/en- | App Store |

| Tool | Kurzbeschreibung | Homepage | App Store |
|---|---|---|---|
| | Lokalisierung für Xcode-Projekte | code | |
| BetterLocale Store | KI-gestützte Lokalisierung für App-Store-Connect-Metadaten | betterlocale.com/en-store | App Store |
| BetterLocale Doc | KI-gestützte Übersetzung für Webseiten, Dokumente und Texte | betterlocale.com/en-doc | App Store |
| AI Text Editor | KI-gestützter Schreibassistent für iPhone und iPad | — | App Store |
| AI Markdown Editor | KI-gestützter Markdown-Editor für iPhone und iPad | — | App Store |

Für die Lokalisierungsarbeit an diesem Projekt (neun Sprachen, ein String Catalog) ist vor allem BetterLocale Code thematisch naheliegend. Dieses Tutorial verwendet es an keiner Stelle selbst, es ist als Hinweis gedacht, nicht als Bestandteil des beschriebenen Workflows.

---

## 25. Fazit

Claude Code nimmt in diesem Projekt viel Routinearbeit ab: das Gerüst eines neuen Features nach einem festen Muster, das Erzeugen von Tests nach demselben Given/When/Then-Schema, das Nachschlagen und Anwenden von zehn Rule-Dateien, die kein Mensch bei jeder Aufgabe im Kopf haben will. Skills, Rules, Hooks und Agents vereinheitlichen dabei genau das, was sonst zwischen Entwicklern auseinanderdriftet: Wo landet eine Validierung, wie heißt ein Composer, wann ist ein Test „vollständig".

Was nicht vereinheitlicht wird, weil es niemand automatisieren sollte: die Entscheidung, *ob* eine Kachel wie „Krankengeld" jetzt drankommt oder erst nächstes Quartal, ob eine IBAN-Prüfziffer-Validierung reicht oder eine echte Kontoverifikation nötig ist, ob eine Testlücke im Service-Feature tolerierbar ist oder sofort geschlossen werden muss. Das bleiben Produkt- und Architekturentscheidungen von Menschen: `.claude/rules/product-scope.md` verlangt an mehreren Stellen ausdrücklich, bei einer Lücke nachzufragen statt sie eigenmächtig zu schließen oder zu erfinden.

Tests und Reviews werden durch nichts in diesem Setup ersetzt, sie werden nur konsequenter angewendet. Ein SwiftLint-Hook, der jede Datei informativ prüft, ersetzt kein `check_project.sh`, das als echtes Gate vor Abschluss einer Aufgabe läuft. Zwei Review-Agents, die read-only nach Verstößen suchen, ersetzen kein menschliches Review sicherheitsrelevanter Änderungen. Und eine

Produktdokumentation, die den Ist-Zustand ehrlich mit Lücken beschreibt, ersetzt keine Entscheidung darüber, welche dieser Lücken als Nächstes geschlossen wird.

Die Grenzen dieses Tutorials sind bewusst gesetzt: Es beschreibt ein einzelnes, mittelgroßes iOS-Projekt mit Mock-Backend, keine Team-Skalierung über mehrere Repositories, keine Multi-Plattform-Architektur mit Watch- oder Mac-Anteil, keine echte Backend-Integration. Wer diese Themen braucht, findet in den offiziell verlinkten Quellen dieses Dokuments den richtigen Ausgangspunkt. Dieses Tutorial bleibt bei dem, was sich an einem realen, laufenden Projekt tatsächlich zeigen und belegen lässt.