GANZ EINFACH

Mac-Apps entwickeln mit Köpfchen

Swift und SwiftUI gründlich verstehen
Mac-Apps entwickeln und veröffentlichen

Ein unabhängiges Praxisbuch



Danilo Sieren

Danilo Sieren

Mac Apps entwickeln mit Köpfchen

Swift und SwiftUI gründlich verstehen
Mac Apps entwickeln und veröffentlichen

Überarbeitete und erweiterte
Neuauflage 2026

Verstehen · Anwenden · Überprüfen

Vorwort

Eine gute Mac App hilft dir, eine Aufgabe verlässlich zu erledigen. Sie bewahrt deine Arbeit, reagiert auf die Tastatur, erklärt Fehler und fügt sich in Fenster, Menüs und Dateien des Systems ein. Dieses Buch führt dich von den Grundlagen der Sprache Swift bis zu einer überprüfbaren Veröffentlichung. Du lernst dabei auch, wann du eine Funktion besser kleiner hältst, statt ihre Grenzen mit zusätzlichem Code zu verdecken.

Das durchgehende Lernprojekt heißt DenkJournal. Du erfasst Einträge, bearbeitest sie in einer Seitenleiste mit Detailansicht und speicherst sie lokal. Ein offenes Austauschformat ermöglicht Export und kontrollierten Import. Die vollständigen Quelldateien im Praxisteil bilden einen überschaubaren Kern. Netzwerkzugriffe, Cloud, Widgets, Medien und KI werden daran anschließend als eigenständige Erweiterungswege erklärt. Dadurch kannst du das Grundprojekt nachvollziehen, ohne zuerst einen Server, ein Abonnement oder mehrere Apple Geräte einzurichten.

Du brauchst keine Vorkenntnisse in Swift. Ein sicherer Umgang mit Dateien und Fenstern auf dem Mac hilft dir beim Einstieg. Wenn du bereits programmierst, lies die Kapitel zu Isolation, Observation und SwiftData trotzdem aufmerksam: Vertraute Begriffe aus anderen Sprachen haben hier teilweise andere Konsequenzen. Für eine bestehende App ist dieses Buch außerdem ein Arbeitsbuch für Modernisierung und Wartung.

Die Neuauflage bearbeitet sämtliche Themen der bisherigen Fassung und verteilt sie auf 28 Fachkapitel. Sprache, Oberfläche, Zustand, Speicherung und Veröffentlichung sind nun klarer voneinander getrennt. Zusätzliche Werkstätten behandeln Fehlerfälle und Entscheidungen, die beim bloßen Nachbauen einer Oberfläche unsichtbar bleiben. Lies deshalb jeweils auch

die Prüfschritte. Ein erfolgreicher Build ist ein wichtiger Anfang; er beweist weder Datensicherheit noch gute Bedienbarkeit.

Die redaktionelle Prüfung hat den Stand 1. November 2026. Als aktuelle Werkzeugfamilie dient Xcode 27 mit Swift 6; den genauen dokumentierten Stand und die Unterschiede zur lokal verfügbaren Compilerinstallation erläutert Anhang E. Das vollständige Grundprojekt verwendet APIs ab macOS 14. Optionale Erweiterungen können neuere Systeme, passende Hardware, aktivierte Dienste und zusätzliche Berechtigungen benötigen. Eine aktuelle Entwicklungsumgebung und ein ältestes unterstütztes Betriebssystem sind zwei verschiedene Entscheidungen.

Code im Haupttext ist entweder eine vollständige kleine Lerneinheit oder ausdrücklich ein Ausschnitt in einem beschriebenen Kontext. Die zusammengehörigen Dateien des Grundprojekts stehen gesammelt in Kapitel 26. Tippe sie nicht zusätzlich zu gleichnamigen Dateien aus einem anderen Beispiel in dasselbe Target. Du würdest damit Typen doppelt definieren. Lange Dateien sind in beschriftete Abschnitte zerlegt; innerhalb eines abgedruckten Blocks bleibt die Einheit auf einer Seite.

Arbeite zunächst mit erfundenen Einträgen. Teste Löschungen, Migrationen, Import und Wiederherstellung an Kopien, bevor du echte Aufzeichnungen anvertraust. Das ist eine praktische Entwicklungsregel: Ein Fehler soll im Labor erkennbar werden, bevor er eine wichtige Datei betrifft. Im Buch werden keine produktiven Accounts oder Veröffentlichungen vorausgesetzt.

So arbeitest du mit diesem Buch

Lies die Kapitel 1 und 2 für die Produktidee und die Werkzeuge. Kapitel 3 bis 5 liefern das Sprachfundament. In Kapitel 6 bis 11 entstehen Oberfläche, Zustandsmodell und dauerhafte Speicherung. Kapitel 12 bis 20 führen die wichtigsten Mac Erweiterungen ein. Kapitel 21 bis 25 machen aus dem Entwicklungsprojekt ein prüfbares und verteilbares Produkt. Kapitel 26 enthält das vollständige lokale DenkJournal; Kapitel 27 und 28 behandeln Wartung und Abschlussabnahme.

Lege für jedes Experiment einen kleinen eigenen Projektstand an. Notiere Xcode Version, SDK, Deployment Target und Sprachmodus. Wenn sich ein Beispiel anders verhält als erwartet, vergleichst du diese vier Angaben zuerst. Unterschiedliche Standardeinstellungen für Actor Isolation können einen scheinbar identischen Swift Ausdruck anders einordnen. Kopierte Codezeilen ohne ihren Umgebungskontext sind deshalb keine zuverlässige Vergleichsbasis.

Die Quellenkennungen wie [D01] führen zum Quellenverzeichnis. Dort stehen die offiziellen Einstiegsseiten. Die Online Dokumentation entwickelt sich weiter; kontrolliere bei einzelnen Symbolen zusätzlich Plattform, Verfügbarkeit und Hinweise zur Einstellung. Ein Symbol mit einer iOS Beschreibung ist nicht automatisch für eine native macOS App verwendbar. Mac Catalyst ist ebenfalls nicht dasselbe Target wie macOS.

Inhalt

Vorwort 2
So arbeitest du mit diesem Buch 4
1 Eine Mac App als Produkt verstehen 7
2 Xcode und die Entwicklungsumgebung einrichten 12
3 Swift Werte und Sammlungen verstehen 18
4 Typen Fehler und Speicher sauber modellieren 27
5 Nebenläufigkeit mit Swift 6 beherrschen 36
6 Oberflächen mit SwiftUI aufbauen 42
7 Zustand mit Observation verwalten 47
8 Navigation und mehrere Fenster gestalten 53
9 Menüs Fokus und Einstellungen integrieren 57
10 Daten mit SwiftData dauerhaft speichern 62
11 Migrationen Sicherungen und Undo planen 68
12 Dateien Dokumente und Sandbox korrekt nutzen 74
13 Bilder PDF und Audio verarbeiten 79
14 HTTP und APIs zuverlässig ansprechen 87
15 Anmeldung Streaming und Netzwerkfehler behandeln 96
16 iCloud und Synchronisierung realistisch planen 100
17 Statistiken mit Swift Charts verständlich darstellen 105
18 Widgets App Intents und Suche ergänzen 109
19 Maschinelles Lernen und Foundation Models einsetzen 113
20 Gestaltung Barrierefreiheit und Lokalisierung verbinden 118
21 Tests Debugging und Messungen einsetzen 124

22 Architektur Pakete und Zusammenarbeit ordnen 130
23 Datenschutz Sicherheit und Berechtigungen prüfen 139
24 Signierung Notarisierung und direkte Verteilung vorbereiten ... 143
25 Den Mac App Store Auftritt vorbereiten 148
26 Das vollständige lokale DenkJournal bauen 152
27 Wartung Fehleranalyse und sichere Updates organisieren 182
28 Die fertige App fachlich abnehmen 186
Anhang A Tests und nachvollziehbare Prüffälle 190
Anhang B Vollständige eigenständige Erweiterungsbeispiele 197
Anhang C Häufige Fehler systematisch eingrenzen 211
Anhang D Begriffe für die tägliche Arbeit 215
Anhang E Versionsstand und Grenzen der Überprüfung 221
Quellen und offizielle Dokumentation 225
Ausführliches Inhaltsverzeichnis 229
Stichwortverzeichnis 239
Impressum 241

1 Eine Mac App als Produkt verstehen

1.1 Mit einer überprüfbaren Aufgabe beginnen

Beschreibe deine Idee zunächst als Handlung: Du willst einen Eintrag anlegen, später wiederfinden und ohne Datenverlust bearbeiten. Diese Formulierung ist hilfreicher als eine Liste von Frameworks. Aus ihr lassen sich Erfolg und Fehler ableiten. Nach dem Anlegen muss der Eintrag sichtbar sein; nach dem Neustart muss er weiterhin existieren; bei einem abgebrochenen Bearbeitungsdialog darf sich der gespeicherte Text nicht ändern.

Für DenkJournal beschränken wir den ersten Umfang auf Titel, Text, Erstellungszeit und Änderungszeit. Tags, Bilder und Synchronisierung sind Erweiterungen. Sie werden nicht dadurch zuverlässig, dass wir sie früh in dieselbe View schreiben. Eine kleine funktionsfähige Basis lässt sich gezielter prüfen. Du kannst später erkennen, ob ein Fehler aus der Speicherung oder aus einer neu hinzugefügten Cloudfunktion stammt.

Formuliere zusätzlich eine Grenze. Unser Kern ist ein lokales Journal, kein kollaborativer Editor und keine medizinische Auswertung. Ein Feld für Stimmungen wäre eine vom Nutzer gewählte Einordnung, keine Diagnose. Eine KI Zusammenfassung wäre ein Vorschlag, keine autoritative Aussage über den Inhalt. Solche Grenzen beeinflussen Oberfläche, Datenschutz und Tests bereits vor dem ersten Build.

1.2 Fenster und Arbeitskontexte respektieren

Auf dem Mac kann eine Person mehrere Fenster, Apps und Bildschirme gleichzeitig nutzen. Deshalb gehört die Auswahl eines Eintrags normalerweise zum jeweiligen Fenster. Zwei Fenster dürfen unterschiedliche Einträge anzeigen, obwohl beide

dieselbe Datenbank verwenden. Ein globaler Auswahlwert würde das zweite Fenster unerwartet umschalten, sobald du im ersten klickst.

Unterscheide Fensterzustand, gemeinsame Daten und Einstellungen. Fensterzustand enthält zum Beispiel Auswahl und Suchbegriff. Gemeinsame Daten sind die gespeicherten Einträge. Eine Einstellung wie die Darstellung relativer Datumsangaben kann für die ganze App gelten. Diese Aufteilung wird später durch State, ModelContainer und AppStorage ausgedrückt. Sie entsteht aus der Bedienaufgabe, nicht aus der persönlichen Vorliebe für ein Architekturdiagramm.

Teste früh eine kleine und eine große Fenstergröße. Ein überbreites Textfeld erschwert das Lesen; ein zu enges Fenster darf wichtige Schaltflächen nicht unzugänglich machen. Die Sidebar sollte eine brauchbare Mindestbreite haben. Eine Detailansicht braucht auch dann einen klaren Zustand, wenn noch kein Eintrag ausgewählt ist. Ein leeres Fenster ist kein Fehler, muss aber erklären, wie du weiterkommst.

1.3 Menüs und Tastatur als gleichwertige Bedienwege

Wichtige Aktionen sollen über ein passendes Menü erreichbar sein. Die Toolbar ist eine schnelle Abkürzung zu häufigen Handlungen. Sie ersetzt die Menüstruktur nicht vollständig. Eine Aktion wie Neuer Eintrag muss im aktiven Fenster wirken und denselben Arbeitsablauf auslösen wie der entsprechende Button. Zwei unabhängig implementierte Varianten driften später auseinander.

Tastenkürzel brauchen einen nachvollziehbaren Geltungsbereich. Cmd N ist für eine Erzeugungsaktion vertraut, kann aber mit dem von WindowGroup bereitgestellten Fensterbefehl kollidieren. Entscheide deshalb ausdrücklich, ob der Befehl ein neues Fenster oder einen neuen Datensatz anlegt. Im Lernprojekt bekommt der Datensatz ein eigenes

Menükommando. Bestehende Systembefehle ersetzt du nur bewusst und nach einem Test der übrigen Menüeinträge. [D41, D42]

Ein deaktivierter Menüpunkt ist häufig hilfreicher als ein späterer Fehlerdialog. Ohne Auswahl kann Bearbeiten deaktiviert sein. Ein Shortcut darf nicht versehentlich eine Löschung auslösen, während du Text in einem Editor markierst. Fokus und aktives Fenster sind deshalb fachliche Voraussetzungen einer Aktion, keine rein optischen Details.

1.4 Dateien oder eine verwaltete Bibliothek wählen

Eine dokumentbasierte App bearbeitet einzelne Dateien, die der Nutzer im Finder verwaltet. Eine Bibliotheksapp verwaltet ihre Datensammlung intern und bietet gegebenenfalls Export an. Beide Modelle sind sinnvoll, führen aber zu verschiedenen Erwartungen an Speichern, Öffnen, Umbenennen und mehrere Fenster.

DenkJournal verwendet zunächst eine Bibliothek mit SwiftData. Der Export ist eine Kopie in einem offenen Format; er wird nicht automatisch zur aktiven Datenquelle. Für einen Texteditor wäre DocumentGroup häufig die passendere Grundlage. Dort kümmert sich das Dokumentmodell um den Dateilebenszyklus. Eine App gleichzeitig wie eine Bibliothek und wie einen Dateieditor erscheinen zu lassen, ohne die Unterschiede zu erklären, führt zu Missverständnissen über den Speicherort. [D13]

Zeige den Nutzern, welche Daten lokal liegen und was eine angebotene Synchronisierung bedeutet. Ein grünes Wolkensymbol darf nicht eine erfolgreiche Sicherung versprechen, wenn lediglich die iCloud Anmeldung verfügbar ist. Wiederherstellbarkeit braucht einen eigenen Arbeitsablauf mit einer tatsächlich geprüften Kopie.

1.5 SwiftUI und AppKit sinnvoll verbinden

SwiftUI beschreibt die Oberfläche aus dem aktuellen Zustand. AppKit bietet die gewachsene native Mac Infrastruktur, einschließlich spezialisierter Views und detaillierter Ereignisverarbeitung. AppKit ist weiterhin ein regulärer Bestandteil der Plattform. Es ist weder pauschal veraltet noch durch SwiftUI in jeder Aufgabe ersetzt.

Nutze SwiftUI für die Grundstruktur, solange sie deine Anforderungen verständlich abbildet. Eine AppKit Brücke ist sinnvoll, wenn ein konkretes benötigtes Verhalten fehlt oder eine spezialisierte Ansicht bereits existiert. Die Brücke braucht dann einen klaren Vertrag für Daten, Delegates und Lebenszyklus. Ein einzelnes NSViewRepresentable rechtfertigt keine zweite globale Zustandsverwaltung. [D15]

Aufwendige Spezialeffekte werden erst dann produktreif, wenn sie mit Hell und Dunkel, reduziertem Kontrast, unterschiedlichen Fenstergrößen und Tastaturbedienung funktionieren. Ein ansprechender Screenshot ist nur eine mögliche Ansicht eines dynamischen Systems. Die eigentliche Aufgabe bleibt das zuverlässige Bearbeiten der Daten.

1.6 Leistung und Energie ohne Versprechen beurteilen

Apple Silicon stellt verschiedene Recheneinheiten bereit. Ob eine Aufgabe schnell läuft, hängt unter anderem von Algorithmus, Speicherbedarf, Datenmenge und verwendeter API ab. Daraus folgt keine Garantie für lautlosen Betrieb, eine bestimmte Bildrate oder eine bestimmte Laufzeit. Nicht jeder Bildschirm unterstützt dieselbe Aktualisierungsrate.

Für die Journal App bedeutet Effizienz zunächst: keine Bilddekodierung beim Aufbau jeder Listenzeile, keine Volltextanalyse bei jedem Tastendruck und kein synchrones Warten auf Netzwerkantworten im UI Pfad. Miss Startzeit,

Interaktionsverzögerung und Speicher auf einem echten Zielgerät. Vergleiche denselben Datenbestand und dieselbe Buildkonfiguration. Eine Debugmessung ist keine verlässliche Aussage über einen Release Build.

1.7 Werkstatt Einen Produktvertrag schreiben

Lege eine kurze Textdatei im Projekt an. Beschreibe drei Aufgaben, die erste Nutzergruppe und die Daten, die du speicherst. Ergänze für jede Aufgabe einen beobachtbaren Erfolg: Ein neu angelegter Eintrag erscheint nach einem Neustart genau einmal. Ergänze einen negativen Fall: Bei abgebrochenem Import bleibt die Sammlung unverändert.

Skizziere anschließend zwei Fensterzustände: die leere Bibliothek und eine Bibliothek mit ausgewähltem Eintrag. Prüfe, ob beide ohne Erläuterung durch den Entwickler verständlich sind. Wenn die Oberfläche bereits im Entwurf einen Speicherstatus behauptet, notiere, welcher konkrete technische Nachweis ihn auslösen würde.

1.8 Vertiefung Anforderungen begrenzen

Angenommen, du möchtest zusätzlich geteilte Journale. Schreibe auf, welche bisherigen Annahmen dadurch ungültig werden: Ein Eintrag kann von außen verändert werden, eine Berechtigung kann entzogen werden und ein Netzwerkfehler kann die Aktualität einschränken. Diese Änderung betrifft viel mehr als einen Teilen Button.

Entscheide deshalb vor der Implementierung, ob gemeinsame Bearbeitung zum ersten Produkt gehört. Falls du sie verschiebst, formuliere diese Grenze sichtbar. Eine bewusst lokale App kann verlässlich sein. Eine scheinbar kollaborative App mit ungeklärten Konflikten gefährdet dagegen gerade die Arbeit, die sie bewahren soll.

2 Xcode und die Entwicklungsumgebung einrichten

2.1 Werkzeugversionen auseinanderhalten

Xcode enthält Editor, Buildsystem, Debugger und SDKs. Der Swift Compiler hat eine eigene Version; der Sprachmodus kann trotzdem Swift 5 oder Swift 6 sein. Das SDK beschreibt die APIs, gegen die du baust. Das Deployment Target legt fest, auf welchem ältesten Betriebssystem die App laufen soll. Diese vier Angaben gehören in die Dokumentation jedes Projekts. [D01, D03]

Zum Redaktionszeitpunkt listet Apple Xcode 27.2 vom 29. Oktober 2026. Die offizielle Systemtabelle nennt dafür Swift 6.4 und macOS Tahoe 26.6 oder neuer als Hostvoraussetzung. Das heißt nicht, dass deine App nur auf dem Hostsystem laufen kann. Unser lokaler Kern zielt auf macOS 14 oder neuer; neuere Funktionen müssen entsprechend abgesichert sein. Prüfe vor dem Installieren die aktuelle Tabelle, statt die Hardwareanforderung aus einer alten Buchausgabe abzuleiten. [D01, D02]

Eine separat installierte Swift Toolchain ersetzt kein vollständiges Xcode für Archive, Signing, Interfacewerkzeuge und Storeverteilung. Command Line Tools können für Sprachübungen und Teile der Compilerprüfung ausreichen. Wenn xcodebuild meldet, dass das aktive Developer Directory nur die Command Line Tools enthält, ist das eine konkrete Werkzeuggrenze und kein Fehler im SwiftUI Code.

Tabelle 2.1 Vier unterschiedliche Versionsangaben

Angabe	Beschreibt	Prüfung
Xcode Version	Entwicklungswerkzeuge	Help und Versionsinfo
Swift Compiler	Übersetzerstand	Toolchain und Buildlog
SDK	Verfügbare Schnittstellen	Target und Buildstand
Deployment Target	Ältestes Zielsystem	Target Build Settings

2.2 Xcode sicher beziehen und vorbereiten

Beziehe Xcode über Apples offizielle Bezugswege. Plane ausreichend freien Platz für Download, Entpacken, SDKs, Buildprodukte und Archive. Eine feste Speicherzahl wäre schnell überholt; prüfe die tatsächliche Paketgröße und halte zusätzliche Arbeitsreserve frei. Installiere zunächst die Plattform, die du wirklich brauchst.

Starte Xcode, erledige die erforderliche Ersteinrichtung und kontrolliere unter Xcode > Settings die verfügbaren Komponenten. Menübezeichnungen können mit Version und Sprache leicht variieren. Für eine native Mac App brauchst du kein iPhone Simulatorgerät. Als Laufziel verwendest du deinen Mac beziehungsweise die passende Mac Architektur. iOS Simulatoren testen kein natives macOS Target.

Erstelle über File > New > Project ein Projekt aus der macOS App Vorlage. Verwende Swift als Sprache und SwiftUI als Oberfläche. Gib dem Projekt einen überschaubaren Namen wie DenkJournal. Die genaue Auswahl der Testsysteme hängt vom Vorlagendialog ab; Swift Testing eignet sich für neue Unit Tests, XCTest bleibt für UI Tests wichtig. [D22, D54]

2.3 Identität und Targets festlegen

Der Bundle Identifier ist die technische Identität des App Targets. Wähle einen stabilen, von dir kontrollierten Namensraum. Ein Muster wie de.beispiel.DenkJournal erklärt die Struktur, reserviert aber keinen echten Namen. Verwende für die Veröffentlichung deine tatsächliche Kennung und halte sie konsistent mit Signing und App Store Connect.

Ein Target beschreibt ein zu bauendes Produkt. Die Hauptapp, ein Widget und ein Testbundle haben getrennte Targets und unter Umständen unterschiedliche Entitlements. Eine Quelldatei muss zum richtigen Target gehören. Wenn ein Typ scheinbar fehlt, kontrolliere seine Target Membership, bevor du ihn kopierst und dadurch doppelte Definitionen erzeugst.

Für lokale Lernschritte ist nicht automatisch eine kostenpflichtige Mitgliedschaft erforderlich. Distribution und Dienste wie bestimmte Cloudfähigkeiten haben eigene Voraussetzungen. Prüfe Apples aktuelle Mitgliedschaftsübersicht. Das Buch verspricht weder einen bestimmten Preis noch, dass jede persönliche Teamkonfiguration alle benötigten Capabilities bereitstellt. [D49]

2.4 Die Oberfläche gezielt benutzen

Links organisiert der Navigator Dateien, Symbole, Probleme und Tests. Im Editor bearbeitest du den ausgewählten Inhalt. Der Inspektor zeigt Eigenschaften der Auswahl; der Debugbereich enthält Konsole und Variablenansicht. Blende Bereiche bei Bedarf ein, statt möglichst viele gleichzeitig sichtbar zu halten.

Lege zuerst die beiden Dateien der App Vorlage frei und identifiziere Einstiegspunkt und ContentView. Baue den unveränderten Stand mit Product > Build. Starte anschließend mit Product > Run. Damit prüfst du zunächst die Umgebung,

bevor eigener Code hinzukommt. Ein fehlendes Zertifikat und ein Syntaxfehler benötigen verschiedene Lösungen.

SwiftUI Previews sind eine zusätzliche Rückmeldung beim Entwickeln. Sie ersetzen keinen Start der App. Eine Vorschau kann Beispieldaten und eine andere Umgebung verwenden; Dateizugriffe, Berechtigungen, mehrere Fenster und Releaseverhalten müssen separat geprüft werden. Halte Previewdaten klein und unabhängig von echten Nutzerdateien.

2.5 Build Settings nachvollziehbar setzen

Notiere für das Lernprojekt Swift 6 als Sprachmodus. Verwende für die Beispiele eine nicht global auf MainActor voreingestellte Isolation; UI Besitzer bekommen ausdrücklich @MainActor. Wenn ein neues Xcode Projekt bereits MainActor als Default Actor Isolation verwendet, kontrolliere diese Einstellung im Target unter Build Settings. Die Beispiele sind dann anhand der ausdrücklichen Isolation zu beurteilen, statt Warnungen durch zufällige Schalterkombinationen zu verdrängen. [D06, D07]

Debug und Release unterscheiden sich unter anderem bei Optimierung und Diagnose. Eine Releasekonfiguration ist kein Ersatz für Tests; sie kann Fehler sichtbar machen, die im Debuglauf verborgen bleiben. Verändere Einstellungen einzeln und dokumentiere den Grund. Ein pauschales Ausschalten der Concurrencyprüfung macht ein Datenrennen nicht korrekt.

Schemes legen fest, welches Target für Run, Test und Archive verwendet wird. Eine App kann lokal richtig starten, während ein falsch geteiltes Scheme auf einem Buildrechner kein Testtarget enthält. Teile für die Zusammenarbeit die benötigten Schemes bewusst und kontrolliere die Aktionen.

2.6 Git und Projektdateien ordentlich halten

Erstelle nach dem ersten funktionierenden Build einen Git Commit. Der Commit soll eine verständliche Änderung

enthalten. Gute Nachrichten beschreiben ein Ergebnis wie Titelvalidierung ergänzen. Ein sehr großer Commit aus Formatierung, Datenschemaänderung und Signingwechsel erschwert die Fehlersuche.

Bewahre Quellcode, Projektkonfiguration, Assets und benötigte Ressourcen auf. Persönliche Zugangsdaten, private Schlüssel und echte Journaldaten gehören nicht ins Repository. Ein privates Repository macht versehentlich eingecheckte Geheimnisse nicht unbedenklich. Bereits veröffentlichte Zugangsdaten müssen gegebenenfalls ersetzt werden; bloßes Löschen aus dem letzten Stand entfernt sie nicht aus der Historie.

Kontrolliere Änderungen an der Projektdatei im Diff. Nicht jede automatisch erzeugte Änderung ist fachlich notwendig. Bei Zusammenarbeit sollten alle wissen, welche Xcode Hauptversion und welche Paketauflösung verwendet werden. Halte den Zustand so fest, dass ein neuer Checkout möglichst ohne manuelles Rätselraten gebaut werden kann.

2.7 Assets und Dokumentation einordnen

Farben und Bilder gehören in einen Asset Catalog, wenn dieser ihre Varianten sinnvoll verwaltet. Verwende semantische Systemfarben für Standardoberflächen. Prüfe Symbole auf dem ältesten unterstützten Betriebssystem. Ein Symbolname, der in der aktuellen Vorschau existiert, kann auf einem älteren Ziel fehlen.

Neue Iconwerkzeuge wie Icon Composer sind ein zusätzlicher Gestaltungsweg. Sie ersetzen weder die Prüfung der gelieferten Appressourcen noch die Darstellung auf älteren Systemen. Prüfe das fertige Icon im Finder, im Dock und in kleinen Größen. Ein ausdrucksstarkes großes Bild kann als kleines Appsymbol unlesbar sein. [D50]

Öffne Symbolinformationen über Xcodes Dokumentationsfunktionen und lies die Plattformangaben.

Wenn ein Tutorial UIKit importiert, ist es nicht unmittelbar eine Anleitung für dein natives Mac Target. Suche nach der Mac Variante oder baue eine passende AppKit Integration.

2.8 Werkstatt Einen reproduzierbaren Start prüfen

Erzeuge einen neuen leeren Projektstand, baue ihn und notiere die vier Versionsangaben. Füge einen bewusst falsch geschriebenen Typnamen ein. Lies die erste konkrete Fehlermeldung und behebe genau diesen Fehler. Danach kontrollierst du, ob Folgefehler verschwinden. Das trainiert einen wichtigen Ablauf: nicht alle roten Hinweise gleichzeitig ändern.

Schließe Xcode und öffne das Projekt erneut. Kontrolliere Laufziel, Scheme und Dateien. Wenn du mit Git arbeitest, prüfe außerdem einen sauberen Checkout in einem zweiten Ordner. Eine Umgebung ist erst dann gut beschrieben, wenn sie nicht von einem zufällig offenen Editorfenster abhängt.

3 Swift Werte und Sammlungen verstehen

3.1 Konstanten und Variablen unterscheiden

Swift verlangt eine bewusste Entscheidung zwischen einem Wert, den du nach seiner Initialisierung nicht neu zuweist, und einem veränderbaren Wert. let kennzeichnet die erste, var die zweite Absicht. Beginne mit let und wechsle zu var, wenn die Aufgabe tatsächlich eine Änderung verlangt. Eine unveränderliche Referenz auf eine Klasse macht deren innere Eigenschaften allerdings nicht automatisch unveränderlich. [D04]

Für die ersten Sprachübungen genügt eine Swift Datei oder ein kleines Kommandozeilenprojekt. Hier sind vollständige Top Level Anweisungen. Sie gehören nicht unverändert zwischen die Eigenschaften einer SwiftUI View.

```
import Foundation

let titel = "Erster Entwurf"
var anzahl = 1
anzahl += 1
print("\(titel): \(anzahl)")
```

Der Ausdruck in der Zeichenkette ist Stringinterpolation. Er ist für kleine Ausgaben praktisch. Eine lokalisierbare Oberfläche braucht später zusätzlich übersetzbare Texte und Formatierung. Baue Datumswerte oder Währungen nicht dauerhaft durch selbst erfundene Trennzeichen zusammen.

3.2 Typen als Vertrag lesen

Ein String ist Text, ein Int eine ganze Zahl und ein Bool ein Wahrheitswert. Ein Datum sollte als Date beziehungsweise als passend modellierter Datumswert gespeichert werden, nicht als bereits deutsch formatierter Bildschirmtext. Sonst wird

Sortierung nach dem Kalender leicht mit alphabetischer Sortierung verwechselt.

Ein numerischer Typ erzwingt nicht automatisch fachliche Gültigkeit. Eine Zahl für eine Bewertung kann weiterhin außerhalb des erlaubten Bereichs liegen. Definiere deshalb die erlaubten Werte an einer klaren Grenze. Validierung gehört in eine Funktion, die du unabhängig von der Oberfläche testen kannst.

Bei Geldbeträgen sind binäre Fließkommazahlen häufig nicht das passende fachliche Modell. Decimal oder ein ganzzahliger Betrag in einer festgelegten kleinsten Einheit können geeigneter sein. Die Entscheidung hängt von Rundungsregeln und Geschäftsanforderungen ab. DenkJournal benötigt keine Geldarithmetik; wir führen sie nicht beiläufig mit einer ungenauen Beispielrechnung ein.

3.3 Optionals als echte Abwesenheit verwenden

Ein Optional beschreibt, dass ein Wert vorhanden sein kann oder fehlen kann. nil ist nicht dasselbe wie ein leerer String, null als Zahl oder ein unbekannter Fehler. Eine fehlende Auswahl in der Sidebar lässt sich zum Beispiel durch UUID? ausdrücken. Ein leerer Titel ist dagegen ein vorhandener, aber fachlich ungültiger Text.

```
import Foundation

let eingabe = "42"
let zahl = Int(eingabe)
if let zahl {
    print(zahl + 1)
} else {
    print("Keine ganze Zahl")
}
```

Das if let bindet den vorhandenen Wert innerhalb des Zweigs. Nutze ?? nur, wenn der Ersatz fachlich korrekt ist. Ein fehlender

Geburtstag darf nicht stillschweigend heute werden. Ein fehlender Suchbegriff kann dagegen sinnvoll als leerer Filter behandelt werden.

Vermeide erzwungenes Entpacken mit ! bei veränderlichen Daten und Nutzereingaben. Ein Absturz ist keine Eingabevalidierung. Ein bewusst nicht erfüllbarer Entwicklungszustand kann eine Assertion rechtfertigen, aber eine fremde Datei kann immer unerwartete Daten enthalten.

3.4 Funktionen mit verständlichen Namen bauen

Eine Funktion kapselt eine Aufgabe und erhält ihre Eingaben ausdrücklich. Der Name sollte die Wirkung erklären. Eine Validierungsfunktion darf nicht gleichzeitig eine Datei löschen oder eine Cloudanfrage starten. Diese Trennung macht sowohl Fehler als auch Tests verständlicher.

```
import Foundation

func bereinigterTitel(_ text: String) -> String {
    text.trimmingCharacters(
        in: .whitespacesAndNewlines)
}

let ergebnis = bereinigterTitel("  Notiz  ")
print(ergebnis)
```

Ein Parameterlabel kann den Aufruf verständlicher machen. Vermeide aber Namen, die eine Fachregel verstecken. speichern klingt einfach, kann jedoch Validierung, Persistierung und UI Rückmeldung umfassen. Später trennen wir deshalb Entwurf prüfen und Modelländerung übernehmen.

Eine Funktion kann einen Wert zurückgeben, einen Fehler werfen oder beides. Ein Bool als Rückgabe ist für einfache Ja Nein Fragen passend. Wenn mehrere Fehlerursachen für die Bedienung wichtig sind, ist ein eigener Fehlertyp verständlicher als false ohne weitere Information.