

Programmierwerkzeuge

Vom Gedanken zum Code – systematisch



Warum Werkzeuge vor Code?



Denken sichtbar machen

Logik, Struktur, Anforderungen und UI transparent gestalten – lange bevor die erste Zeile Code entsteht

- *Fehler früh erkennen*
- *Kommunikation im Team vereinfachen*
- *Implementierung beschleunigen*

Prüfungswissen AP1

Werkzeuge benennen

Struktogramm, PAP, Pseudocode, UML – präzise unterscheiden können

Passend einsetzen

Situationsgerecht entscheiden: Wann welches Werkzeug optimal ist

② QUIZ

Teste dein Wissen

1 Entscheidungskriterien

Woran erkennst du, ob Struktogramm, PAP oder Pseudocode besser passt?

3 Klassendiagramm-Mindestanforderung

Was muss enthalten sein, damit es für die Implementierung taugt?

2 Use-Case-Qualität

Welche Information macht ein Use-Case-Diagramm wirklich nützlich – nicht nur Dekoration?

4 UI-Bewertung

Nenne 4 Kriterien zur fachlichen Bewertung einer Eingabemaske

Drei didaktische Hilfsmittel

Struktogramm, PAP, Pseudocode – gezielt einsetzen

Struktogramm (Nassi-Shneiderman)

Stärken

Zeigt Programmstruktur kompakt: Sequenz, Verzweigung, Schleife auf einen Blick

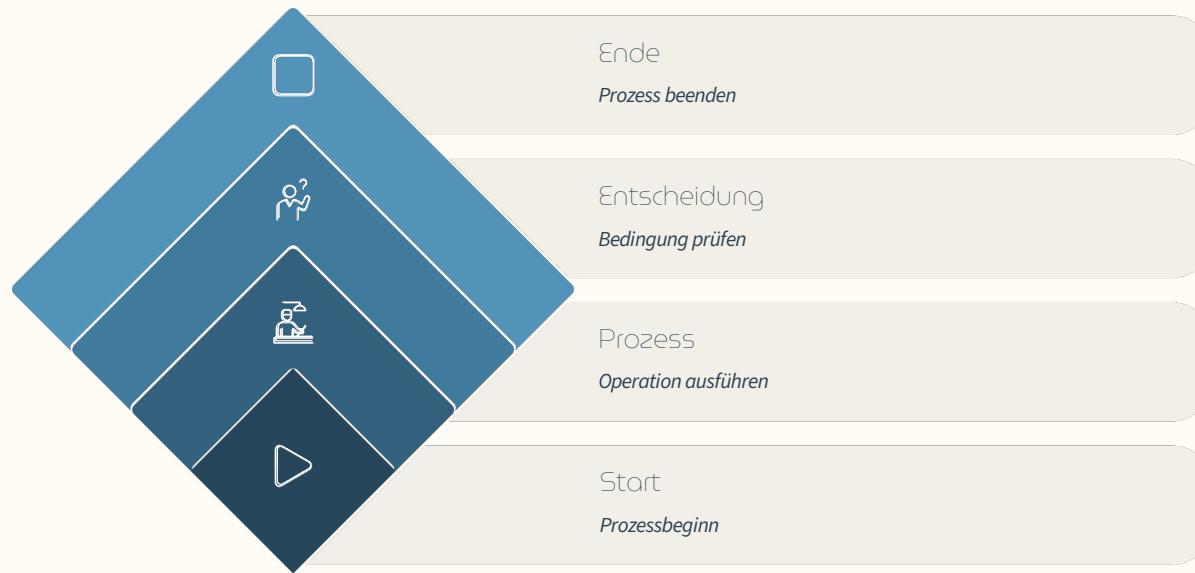
Optimal für

- Klare Logik visualisieren
- Kleine, überschaubare Algorithmen
- Prüfungsnahe Abläufe darstellen



Strukturierte Blöcke verhindern Spaghetti-Code bereits im Entwurf

PAP (Programmablaufplan)



Der klassische Ablaufplan für Prozessvisualisierung

Charakteristik

Zeigt Ablauf als Fluss mit Start, Ende, Entscheidungen und Prozessschritten

Besonders geeignet

- *Wenn der Weg wichtiger ist als Code-Struktur*
- *Für Prozessdokumentation*
- *Bei mehreren Entscheidungspfaden*

Pseudocode

Eigenschaften

Textnah, sprachneutral, schnell geschrieben – die flexible Alternative

Ideal für

- *Algorithmus-Ideen entwickeln*
- *Vor der Syntax-Bindung*
- *Schnelle Prototypen*



Tipp: Präzise genug zum Umsetzen, aber frei von Syntax-Details

Beispiel

```
INPUT alter
IF alter >= 18 THEN
  OUTPUT "Volljährig"
ELSE
  OUTPUT "Minderjährig"
ENDIF
```

Typische Fallen vermeiden

Zu detailliert

1:1 Code abbilden → verliert didaktischen Nutzen und Übersichtlichkeit

Zu ungenau

"Mach irgendwas" → nicht implementierbar, keine klaren Anforderungen

Inkonsistent

Notation wechselt → erschwert Verständnis und Wartbarkeit

Quiz: Werkzeugwahl

01

Beispielaufgaben

Nenne je 1 Situation: Wann ist Struktogramm besser als PAP – und umgekehrt?

03

Häufige Fehler

Welche 3 Fehler passieren bei Verzweigungen/Schleifen in Diagrammen?

02

Qualitätskriterien

Wie unterscheidest du "guten Pseudocode" (präzise, testbar) von "Wischiwaschi"?

04

Akzeptanzkriterien

Wie leitest du aus einem Diagramm Akzeptanzkriterien ab – ohne Code?

Struktogramm: Die Bausteine

Drei Grundelemente für jede Programmlogik

Struktogramm-Elemente



Sequenz

*Schritte nacheinander: Input → Verarbeitung
→ Output*



Verzweigung

Bedingung prüfen → dann/sonst ausführen



Schleife

*Wiederhole bis Bedingung erfüllt oder
solange sie gilt*

Mit diesen drei Elementen lassen sich Off-by-one-Fehler und Endlosschleifen gedanklich verhindern

Quiz: Struktogramm-Praxis

Eingabeprüfung

Wie modellierst du: ungültige Eingabe → erneut fragen?

Endlosschleifen

Welche Bedingung führt dazu – und wie vermeidest du sie?

Schleifentest

Wie testest du gedanklich: Startwert, Abbruch, Schrittweite?

Minimal-Doku

Welche Informationen brauchst du zur Implementierung?



PAP-Symbole und Bedeutung



Start/Ende

Abgerundetes Rechteck markiert Beginn und Abschluss des Prozesses



Prozess

Rechteck für Aktionen und Verarbeitungsschritte



Entscheidung

Raute für Ja/Nein-Abzweigung mit klarer Bedingung



Ein-/Ausgabe

Parallelogramm für Nutzerinput oder Ausgabe



Typische Falle: Entscheidungen ohne eindeutige Bedingung ("wenn ok") bleiben unklar

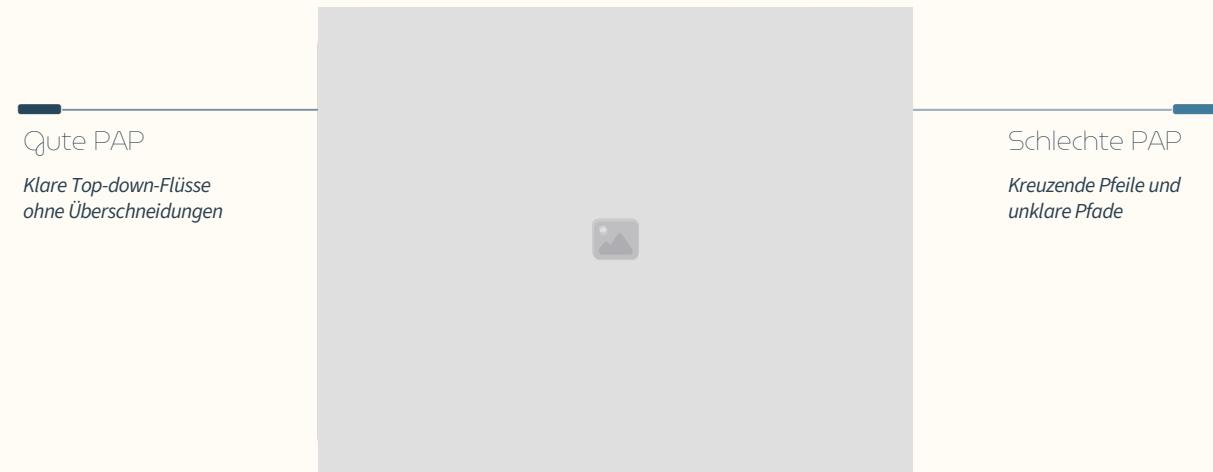
PAP-Qualitätsregeln

Lesbarkeit sichern

- *Eindeutige Bedingungen formulieren*
- *Konsistente Pfeilrichtung (meist top-down)*
- *Übersichtliche Anordnung*

Spaghetti-Flows vermeiden

Bei vielen Sonderfällen: Aufteilung in Teilprozesse oder Strukturierung durch Subprozesse



Klare Struktur erleichtert Testfall-Ableitung: Happy Path und Edge Cases direkt erkennbar

Pseudocode-Qualität

Muss-Kriterien

- *Klare Schritte definieren*
- *Eindeutige Bedingungen*
- *Benannte Variablen/Strukturen*

Soll-Kriterien

- *Sprachneutral bleiben*
- *Keine Java/Python-Syntax*
- *Aber: eindeutig umsetzbar*

Randfälle beachten

- *Leere Eingaben*
- *null/None-Werte*
- *Ungültige Daten*

UML – Standardsprache fürs Design

Anforderungen und Struktur sichtbar machen

UML-Fokus in AP1

Use-Case-Diagramm

Beantwortet die Frage: *Was soll das System leisten?*

- Akteure (Rollen, nicht Personen)
- Use Cases (Nutzerziele)
- Systemgrenze (Scope)

Fehler vermeiden: Use Cases als Nutzerziel formulieren, nicht als technische Details

Klassendiagramm

Zeigt die Struktur: *Wie ist das System aufgebaut?*

- Klassen mit Attributen
- Methoden (optional in AP1)
- Beziehungen (Assoziation, Vererbung)

Ziel: Verantwortlichkeiten klären, Wiederverwendung ermöglichen

Use-Case richtig formulieren



Akteur = Rolle

*Nicht "Max Müller", sondern "Kunde" oder
"Administrator"*

Use Case = Nutzerziel

"Auftrag erfassen" statt "DB speichern"

Systemgrenze

*Klar trennen: Was ist im System, was
außerhalb?*

Aus Use Cases lassen sich Muss-, Soll- und Kann-Anforderungen ableiten

UI-Maske: Ergonomie & Barrierefreiheit



Ergonomie-Basis

Schnell erfassbar, fehlerarm, konsistent – klare Gruppen, sinnvolle Tab-Reihenfolge, Defaults setzen



Barrierefreiheit

Kontrast, Schriftgröße, Tastaturbedienung, sichtbarer Fokus – Labels für alle Felder, keine Infos nur über Farbe



Fehlermeldungen

Konkret formulieren: Was ist falsch + Wie korrigieren – Validierung nahe am Feld platzieren