Anwendungsgebiete:
- KI (Sprachverarbeitung)
- Datenbanksuche
- Expertensysteme
- Constraint Programming

deklarative/ Logische Programmierung

Prolog

Syntax

,	AND-Operator
head :- body	Implikation body → Head head genau dann wenn body
;	OR-Operator
:-	If-Operator
=:=	Numerisch gleich
=/=	Ungleich
=	Gleichgestellt, also Unifizierbar
/=	nicht Unifizierbar
==	identisch
/==	nicht identisch

Negation as failure: If unsure or false, assume false

Test ob ein Term in den anderen Term umgewandelt werden kann

Test ob 2 Terme gleich sind

- Closed-World-assumption: Datenbass und sonst nichts gilt

append(operation, adds items of one List into another
Bsp: append([1,2,3], [a,b,c], [1,2,3])

Terme:
- Zahlen 5, 3.5, -5
- Strings "meow"
- Listen [1,2,3], [boj, esra]
- Variablen X, Y, Z, _buch, Person
- Funktoren buch(seite, nummer, heft)
- Individuenkonstanten leia, 'apple', cutie, -->, ^^ (kleinBuchstabe, nur Spezialzeichen oder 'X')

Variablen
- beliebig ersetzbar
- Gültigkeitsbereich (Scope): Klausel in der sie definiert wurde
- Variable und überall gleichdefiniert innerhalb einer Klausel
- Variable die nur einmal vorkommt = Singleton ⇒ meist ein Fehler ⇒ SWI-Prolog gibt Warning aus
- Variablen die mehrfach auftaucht = gebundene Variablen

append/3: append([],L,L). append([H|T],L2,[H|L3]) :- append(T,L2,L3).

Begriffe

- Programm besteht aus Menge von Prädikaten
- Prädikate sind allgemeiner als Funktionen - können durch mehrere Belegungen erfüllt werden (mehrere Ergebnisse)
- Prädikate durch Menge von ~~Literalen~~ Klauseln definiert
- Klauseln bestehen aus Literalen die durch logische Operatoren verbunden sind
- Literale haben als Argumente Terme die die Datenstrukturen darstellen
- Klausel

Fakt (Hornklauseln (Axiome))
- einfachste Form von Klauseln
- besteht aus einem Literal
- Aussage, die 'wahr' ist

father(anakin, luke).
father(anakin, leia).

Regel
- Klausel mit Form (head :- body)
- body besteht aus Literalen verbunden durch Log. Operatoren
- Eine Regel ist eine Implikation aus einer wahren Aussage

grandmother(G,S) :- mother(G,S), (father(G,S); mother(C,S)).

Anfrage (Query, Goal)
- Fakten & Regeln werden an Interpreter zur Lösung gegeben
- ⇒ true = Belegung gefunden

2- father(anakin,luke).
true.

Query Anfragen:
→ Ist Anakin Father von Luke?
?- father(anakin, luke).
true.
→ Hat Anakin Kinder?
?- father(anakin, _).
true.
→ Von wem ist Shmi die Großmutter
?- grandmother(shmi, X).
X = luke; 1.
X = leia; 2.

Funktoren

- Terme die aus anderen Termen besteht → beliebig tief geschachtelt

printList([]) :- true.
printList([H|T]) :- write(H), write(" "), printList(T).

Funktoren ≠ Funktionen denn sie werden nicht ausgewertet

printPlus1(X) :- write(X+1).
?- printPlus1(10)
10+1
true.
(statt 11 wird einfach 10+1 ausgegeben)

← Funktor zur expliziten Argument Benennung

persondata(esra, age(25), adress('Huhnstraße', '35', 'Huhntown')).

↑ Literal mit Individuenkonstanten

Liste wird invertiert
→ rev([], Rev, Result) :- Result = Rev.
rev([H|T], Rev, Result) :- rev(T, [H|Rev], Result).
revertList(L,R) :- rev(L, [], R).

route(X,Y,Visited) :- \+ member(X,Visited), \+ member(Y,Visited), directConnection(X,Y,_), revertList([Y,X|Visited], FinalRoute), printList(FinalRoute).

Verbindungen in richtiger Reihenfolge; in der Liste Visited wird gemerkt, damit keine Doppelungen gibt

route(X,Z,Visited) :- \+ member(X,Visited), directConnection(X,Y,_), route(Y,Z,[X|Visited]).

allRoutes(X,Y,ResultList) :- findall((Route, Distance), route2(X,Y,[],Distance,Route), ResultList).

Alle möglichen Routen mit Längen in ResultList; Variablen werden ausgegeben

nach X sortiert

write(X). Druckt X auf Konsole und gibt immer true aus
findall(X,G,L). alle Variablen die die Anfrage G erfüllen sind in L enthalten
setof(X,G,L). wie findall aber nach X sortiert

shortestRoute(X,Y,Shortest) :- setof((Distance,Route), route2(X,Y,[],Distance,Route), [Shortest|_]).

kürzeste Route mit Länge wird ermittelt

greater than 1 ... [T], Start, End, SubList ...

Listen

vordefinierte Rekursivstruktur

Bsp. [1,2,3]   Beispiel Leere Liste []

| trennt Head (Elemente) und Tail (Liste)

Bsp → (H|T) →Bsp [first, second | Rest]

Bsp → ~~[1,2]~~ [1,2|[3,4,5]]

Listen Operationen

is_list(L). erfüllt wenn L eine Liste ist

length(List, Length) Length erhält die Länge von List

member(E, List). erfüllt wenn ein Element == E in List enthalten ist

delete(List1, E, List2). List2 entspricht List1 ohne das Element E

append(L1, L2, L1_2). L1_2 entspricht L1 konkateniert mit L2 (2 Listen zusammen gefügt ohne Reihenfolge zu verändern)

Code Smells

Maßnahmen zur Erkennung
- Code Reviews
- Konventionen: Metriken, Code Style und Linting
- Pair Programming

Code Smells Eigenschaften
- Datenklumpen - gleiche Variablen treten an vielen Stellen gemeinsam auf
- Neid (falsche Zuständigkeit) - Eine Methode in einer Klasse verwendet viele Attribute einer anderen Klasse
- Viele Bedingungen im Kontrollfluss, die durch eine Switch Anweisung entstehen, können schwer verständlich sein
- Lange Methoden & Parameterlisten
- Große Klassen
- Viele Kommentare
- Duplizierter Code
- Neigung zu elementaren Datentypen
- Unangebrachte Intimität - Eine Klasse verwendet viele Teile einer weiteren Klasse, bei denen Details der Implementierung eine Rolle spielen

Achtung: → Namen von Funktoren können keine Variablen sein

X(a,b)
↑ falsch

Arithmetik

Grundrechenarten: +, -, *, mod

Zuweisung eines numerischen Wertes zu einer Variable mit is

Bsp: X1 is X+1

Bsp im Interpreter:
?- X is (5*5-3) mod 2.
X = 0

ansonsten — is wertet aus

?- X = (5*5-3) mod 2.
X = (5*5-3) mod 2.

Unifikation

zweier Terme: Ersetzung (Substitution) der Variablen in den Termen, sodass die entstandenen Zeichenfolgen gleich sind

Unifizierbar wenn:
→ gleiche Individuenkonstante
→ einer davon eine freie Variable
→ Beide komplexe Terme:
  → Funktor der selbe
  → Anzahl der Parameter gleich
  → Argumente paarweise unifizierbar

Bsp: adress(street(X,135),Y)
= adress(street("17,uni"),Z), plz(berlin, 10587)).

Resolution

→ Ableiten von Fakten aus Datenbasis wird durch Resolution ermöglicht → Beweisverfahren aus Logik

→ Anfragen werden schrittweise durch Unifikation substituiert bis leere Behauptung übrig bleibt → True

- mehrere gültige Eingaben erzeugen Lösungsbaum

Falsch beispiele:

Prolog

Resolution Beispiel

Backtracking is the repeated searching for additional solutions, goes back to find solution ⇒ Resolution

Fakten
man(luke).
man(anakin).
woman(shmi).
woman(leia).
father(anakin, luke).
father(anakin, leia).

2 gültige Belegungen von X

?- woman(X), father(anakin, X).

X/shmi → woman(shmi), father(anakin, shmi). → woman(shmi), father(anakin, shmi). → father(anakin, shmi) → false

X/leia → woman(leia), father(anakin, leia). → father(anakin, leia). → true, X = leia

backtracking

Rekursive Berechnungen

fak(0,1).
fak(N,F):-
  N1 is N-1,
  fak(N1,F1),
  F is N*F1.

Rekursionsschritt

Aktualisieren Ergebnis nach Rekursionsschritt

?- fak(5,F).
F = 120

Rekursion

Prädikate können Rekursiv definiert werden, dh in einem Prädikat kann es selbst wieder auftauchen

ancestor(A,B):- parent(A,B).
ancestor(A,B):- parent(A,A_B), ancestor(A_B,B).

Prädikat so oft in sich selbst ersetzen, bis gültige Belegung gefunden wird.

Rekursionsschritt: Verfahren können beliebig viele Generationen entfernt sein

Alternative 1

cntAncestor(A,B,Cnt,Erg):- parent(A,B), Erg = Cnt.
cntAncestor(A,B,Cnt,Erg):- parent(A,A_B),

In jedem Rekursionsschritt wird Zähler inkrementiert → Cnt1 is Cnt+1,
cntAncestor(A_B,B,Cnt1,Erg).

?- cntAncestor(shmi, luke, 1, Erg).
Erg = 2

Alternative 2

Startwert von Erg im Rekursionsanker

cntAncestor2(A,B,1):- parent(A,B).

→ Cnt initialisiert mit Startwert (1)
→ für Erg soll eine Belegung gefunden werden

cntAncestor2(A,B,Erg):- parent(A,A_B),
  cntAncestor2(A_B,B,Erg1),
  Erg is Erg1+1

?- cntAncestor2(shmi, luke, Erg).
Erg=2

Code Smells wird von Refactoring gelöst.

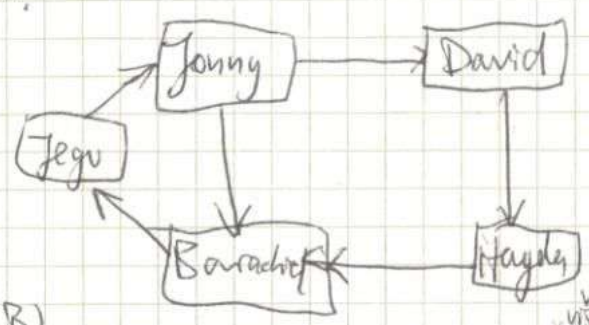
→ Verbesserung interne Struktur
→ verständlicher und wartbarer schreiben
→ Äquivalentes Verhalten desselben muss gewährleistet werden
→ Changeimpact analysis
Funktionalität muss neu getestet werden

Vorgehen:
- Klasse extrahieren
  Smells: duplizierter Code, Datenklumpen, große Klasse
  Code: Neue Klasse, relevante in die neue Klasse
- Methode extrahieren:
  Smells: Lange Methode, duplizierter Code, Kommentare
  TODO: Aus Fragment eine Methode, deren Namen die Methode erklärt
- geschachtelte Bedingungen durch Wächterbedingungen ersetzen
  Smells: Lange Methode
  TODO: Wächterbedingungen für Spezialfälle
- Parameter durch explizite Methode ersetzen
  Smells: Lange Parameterliste, switch befehl
  TODO: separate Methode für jeden Parameter

Zyklische Daten

Über wie viele Ecken kennen sich folgende Personen?

friend(Jonny, David).
friend(David, Hayden).
friend(Hayden, Barachiel).
friend(Barachiel, Jegu).
friend(Jegu, Jonny).
friend(Jonny, Barachiel).
knows(A,B):- friend(A,B).
knows(A,B):- friend(A,C), knows(C,B).



Zyklische Abhängigkeiten ⇒ unendliche Rekursion möglich

Lösung:
"Merken" von verwendeten Fakten (Daten)

bereits visited liste

Abfrage ob schon visited wurde

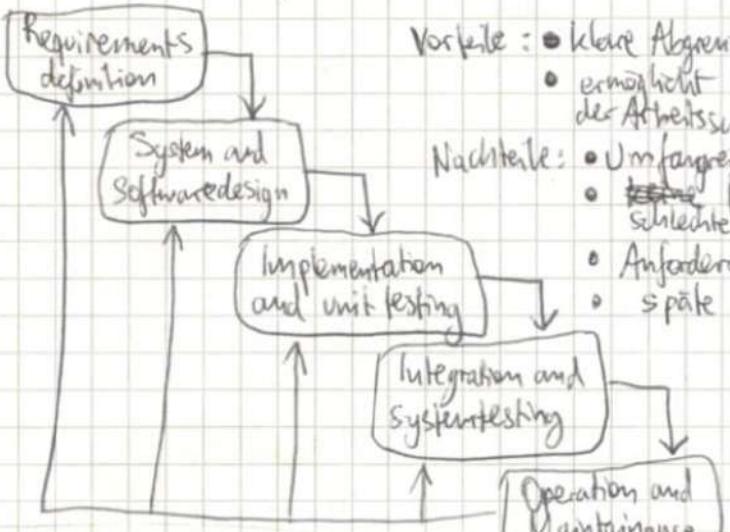
knows(A,B,_):- friend(A,B).
knows(A,B,Visited):- friend(A,C), \+ member(C,Visited),

Anforderung → **Spezifikation** → **Entwicklung** → **Validierung** → **Evolution**

- **Spezifikation**: • Definition der zu entwickelnden Software und ihrer Einsatzbedingungen
- **Entwicklung**: • Entwurf und Programmierung der Software
- **Validierung**: • Überprüfung der Software auf Erfüllung der Anforderungen
- **Evolution**: • Weiterentwicklung und Anpassung an geänderte Anforderungen

Plangesteuerte Prozesse

Wasserfallmodell

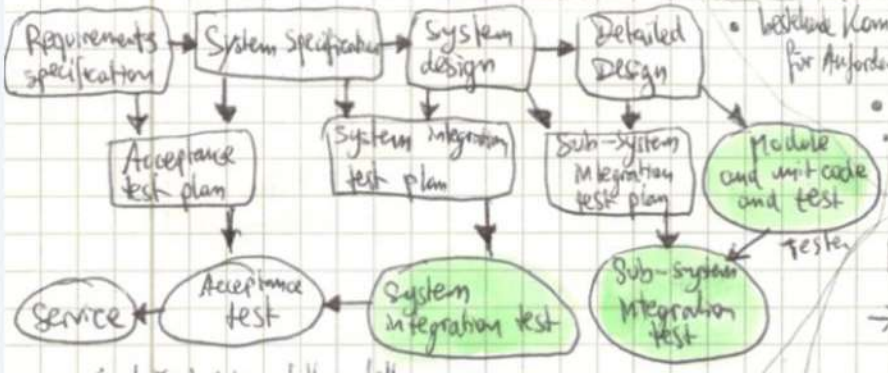


Vorteile: • klare Abgrenzung der Phasen
• ermöglicht einfache Koordinierung der Arbeitsschritte

Nachteile: • Umfangreiche Planung nötig
• ~~testing~~ schlechte Flexibilität bei Veränderungen
• Anforderungen müssen stabil sein
• späte Integration und Systemtests (Fehler spät erkannt)

⇒ geeignet für stabile Großprojekte mit mehreren Entwicklungsteams

V-Modell



- Erweitert Wasserfallmodell
  ↳ starkerer Fokus auf Validierungsphasen
  - Entwicklertests (Programmierfehler)
  - Systemtests (Schnittstellen)
  - Abnahmetests (Alpha-tests) Kunden

Wiederverwendungsorientierte Prozesse

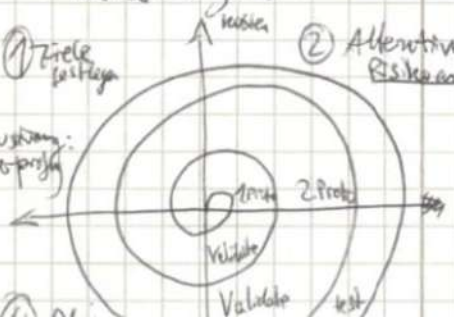
- bestehende Komponenten werden passend gesucht für Anforderungsspezifikation
  - Analyse der Komponenten
  - Anpassung der Anforderungen
  - Ergänzung & Integration der Komponenten

Inkrementelle Entwicklung ⇒ Spezifikation, Entwicklung & Validierung parallel

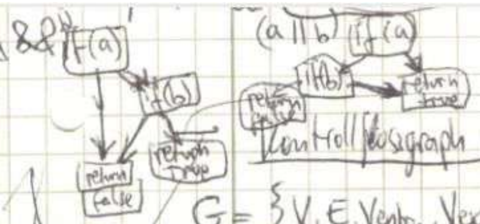
→ Spiralmodell nach Böhm:
- Transparenz des Entwicklungsprozesses ✓
- Risikobewertung → besteht Risiko → Projekt gescheitert

Vorteile
- frühes Testen (von Prototypen)
- iterative Entwicklung reduziert Risiko

Nachteile
- für große Projekte nicht geeignet
- Risikoanalyse teuer



Agil	Plangesteuert
• inkrementelle Auslieferung realistisches, schnelles Kundenfeedback gewünscht	• detaillierte Spezifikation vorab
• kleines Team & informeller Austausch	• großes Entwicklerteam
• Programmierer arbeiten selbständig	• komplexe Anforderungen, hoher Analysebedarf ✓
	• umfangreiche Dokumentation
	• Pflichtenheft
	• externe hohe Vorschriften



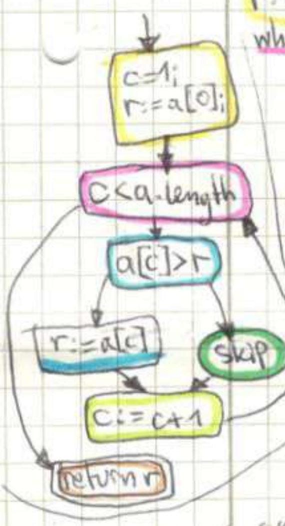
Kontrollflussgraph (CFG)

$G = \{V, E, V_{entry}, V_{exit}\}$

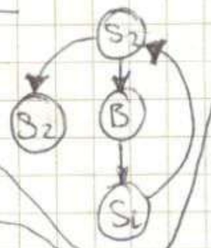
- ◯ V Knoten (Instruktionen bzw. Basisblöcke)
- → $E \subseteq V \times V$ Kanten (Kontrollfluss zwischen Knoten)
- →◯ $V_{entry} \in V$ Startknoten
- ◎ $V_{exit} \in V$ Endknoten

Bsp. Schleife: maxArrayElem(int a[])

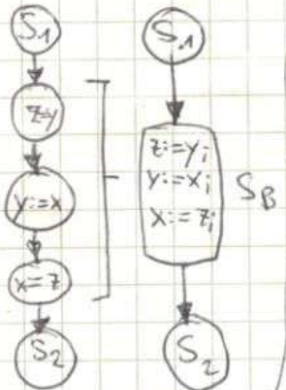
```
c := 1;
r := a[0];
while c < a.length do
  if a[c] > r then
    r := a[c]
  else
    skip
  fi;
  c := c+1
od;
return r
```



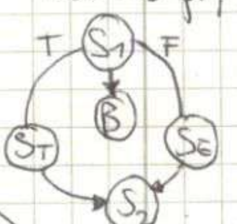
Schleifen: S_1; while B do S_L od; S_2



Basisblöcke: S_1; $z := y$; $y := x$; $x := z$; S_2 (S_B)



Verzweigung: S_1; if B then S_T else S_E fi; S_2



- Prozessqualität: Management, Entwicklungsmodelle, Softwareinfrastruktur (Automatisierung)
- Produktqualität:
  - Konventionen; Codeverbesserung Methoden
  - Einheitliche IDE
  - Codereviews → Code-Smells wird verhindert
  - Kommentare → Erleichterung von Verständnis → Formatierung/Anpassung
  - Statische Analyse → Linten: identifizierung von falschen Codestyle, Fehlerquellen
  - Dokumentation → interne & externe Artefakte

Metriken

- misst Umfang & Komplexität von Software ✓
- → Qualitätsverbesserung & Aufwand & Kostenberechnung

Produktmetriken (Qualität)

- Statisch
  - Programmgröße (Zeilenmetriken)
  - Programmstruktur (Zyklomatische Komplexität)
  - Verhältnisse von Methoden und Klassen zueinander
- Dynamisch
  - Fehlerberichte, Speicherverbrauch, Laufzeit
  - wichtig um Effizienz und Zuverlässigkeit zu bewerten

Prozessmetriken (Qualität)

- benötigte Ressourcen, Absolute Zeiten, Bugs, Änderungen

ohne Metriken?
↳ Alternative: maschinelles Lernen um aus statischen Merkmalen auf dynamisches Verhalten zu schließen.
Erfolgreiche Softwareprojekte ohne Steuerung & Kontrolle:
- Open Source Projekte, Wikipedia, Google Earth, Leo

Größe

Zeilenmetriken
Metriken: misst Umfang der Komplexität

Lines of Code (LOC)
Bsp. LOC = 11 Funktion: 4–40 Zeilen, Datei: 40–400 Zeilen (10–100 Funktionen)
NCLOC = 6 Non-Commenting Lines of Code (NCLOC)
ohne Leerzeilen, reine Kommentarzeilen
↳ Kommentarzeilen zwischen 30% und 75%

Strukturmetriken
Zyklomatische Komplexität (SO) v(G) von McCabe
→ # von konditionellen Zweigen im CFG des Programms
→ # binärer Zweigungen + 1

Zyklomatische Zahl
- $v(G) \leq 10$ einfache Programme
- $v(G) > 10$ Fehler nehmen stark zu
- $v(G) \geq 50$ sehr bzw. zu komplexe Programme, kaum zu testen

je höher desto mehr Testfälle nötig

Zyklomatische Komplexität
- e # der Kanten im Graph (einschließlich)
- n # der Knoten im Graph
- p # Komponenten (zusammenhängende Graphen)

$$v(G) = e - n + 2 \cdot p$$

Bsp. # Kanten = 8 = e, # Knoten = 7 = n, # Komponenten = 1 = p
$v(G) = 8 - 7 + 2 \cdot 1 = 3$ also einfaches

- → Testplanung ✓
- → gute Korrelation ✓
- → leicht zu berechnen ✓
- → Kontrollfluss ✓
- → Datenfluss ✗
- → objektorientierte [illegible]

Halstead Metriken

	Werte
different Operatoren	n1
different Operanden	n2
Operatoren im Programm	N1
Operanden im Programm	N2
Größe des Vokabulars	$n = n1 + n2$
Länge des Programms	$N = N1 + N2$

Operanden: max a len i 0 1 m x
Operatoren: int, [ ] { } ( ) for ; = if < > ++ return

Metriken
- Volumen des Programms $V = N * \log_2(n)$
- Schwierigkeit $D = (n1/2) * (N2/n2)$
- Aufwand von Programm $E = D * V$

Weitere Metriken
- NBD: Verschachtelungstiefe, sollte < 5 sein
- NST: Number of Statements, # Semicolons in Java, sollte < 50 sein
- NFC: Number of Function Calls, sollte < 5 sein
- NOM: Number of Methods

Objektorientierte Metriken
- DIT (Depth of Inheritance Tree): max Abstand von Wurzel der Klassenhierarchie zur Klasse — wenn DIT größer, mehr Fehler
- NOC (Number of Children): # Direkter Subklasse (nicht immer schlecht)
- RFC (Response for a Class): # der Methoden, wenn ein Objekt der Klasse eine eingehende Methode ausführt (mehr Fehler)
- WMC (weighted methods per class): # Methoden einer Klasse, je größer, desto mehr Fehler
- CBO (Coupling between Objects): # Klassen mit denen Klasse gekoppelt ist → niedriges Kopplung spricht für bessere Wiederverwendung