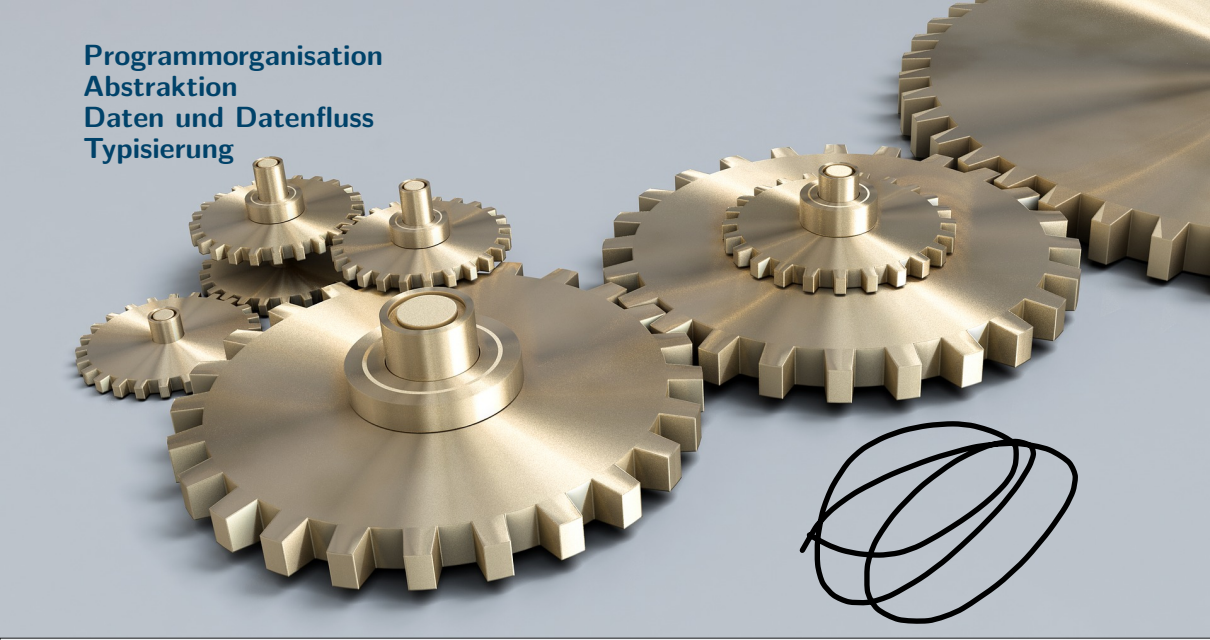
Programmorganisation
Abstraktion
Daten und Datenfluss
Typisierung



Modularisierung

Modularisierung = Faktorisierung = Programme in austauschbare Einheiten zerlegen

Formen von Modularisierungseinheiten:

generell: separater Namensraum, Datenabstraktion, Schnittstelle für Export

- **Objekt** existiert zur Laufzeit, anonym; Identität, Zustand und Verhalten unterscheidbar
- **Klasse** für Objekterzeugung, Klassifizierung; z. B. Java-Klasse/Interface (nicht-`static`)
- **Modul** Übersetzungseinheit, importiert über Namen, zyklenfrei; z. B. Java-Klasse (`static`)
- **Komponente** Übersetzungseinheit, Import anonym, Deployment nötig; z. B. Java-EE-Bean
- **Namensraum** zur Organisation von Modularisierungseinheiten; z. B. Java-Paket

Parametrisierung

Parametrisierung = späteres Befüllen von in Modularisierungseinheiten belassenen Lücken

für Objekte durch Setzen von Objektvariablen, wobei Werte so in Objekte gelangen:

- Konstruktor — übliche Vorgehensweise
- Initialisierungsmethode — z. B. bei zyklischen Abhängigkeiten oder zusammen mit `clone()`
- zentrale Ablage — Objekt holt sich Werte selbst von vorbestimmten Plätzen

für Klassen, Module, Komponenten: — oft keine „normalen“ Werte zum Befüllen der Lücken

- Generizität — Lücken z. B. durch Typparameter gekennzeichnet, später durch Typen ersetzt
- Annotationen — Werte an Programmteile angeheftet, von Werkzeugen explizit ausgelesen
- Aspekte — spezifizieren Modifikationen von Programmteilen (zusätzlich ausgeführter Code)

Problem: starke Abhängigkeit zwischen Lücken und Werten zum Befüllen

Ersetzbarkeit

A durch B ersetzbar wenn Ersatz von A durch B keine Änderung bei Verwendung verlangt

Ersetzbarkeit abhängig von A, B und Erwartungen an A und B

$\rightarrow$ definierte Schnittstellen von A und B nötig — beschreiben erlaubte Erwartungen

$\rightarrow$ Schnittstelle von B impliziert die von A — Schnittstelle von B kann mehr Details festlegen

Schnittstellen auf verschiedene Weisen spezifizierbar:

- Signatur — simpel, automatisch prüfbar, Irrtum möglich (Bedeutung nicht festgelegt)
- Abstraktion realer Welt — zusätzlich zu Signatur, intuitiv, fehlerbehaftet
- Zusicherungen (DbC) — zusätzlich zu Signatur und Abstraktion, aufwändig
- überprüfbare Protokolle — sehr aufwändig, noch nicht ausgereift

Prinzip der Abstraktion

unterschiedliche Semantik, gleiche Abstraktion:

```
// swap a[i] and a[j]; i != j
void swap(int[] a,int i,int j){
    int h = a[i];
    a[i] = a[j];
    a[j] = h;
}
```

```
// swap a[i] and a[j]; i != j
void swap(int[] a,int i,int j){
    a[i] ^= a[j];
    a[j] ^= a[i];
    a[i] ^= a[j];
}
```

gleiche Semantik, unterschiedliche Abstraktionen:

```
// x smaller y if result <  0
// x equals y  if result == 0
// x larger y  if result >  0
int compare(int x, int y) {
    return x - y;
}
```

```
// difference between x and y
int subtract(int x, int y) {
   return x - y;
}
```

Arten von Abstraktionen für Funktionen (Prozeduren, Methoden, . . .)

λ-Abstraktion Abstraktion = Implementierung der Funktion (z. B. wie in λ-Kalkül)
- → Name und Kommentare bedeutungslos
- → inhaltliche Änderung der Implementierung ändert Abstraktion

strukturelle Abstraktion abstrakte Funktion oder Variable, die Funktion enhält
- → Name, Kommentare und Implementierung bedeutungslos
- → nur Änderung der Signatur ändert Abstraktion

nominale Abstraktion Abstraktion durch Name und Kommentare (Funktion oder Variable)
- → Implementierung passt dazu oder ist unbekannt (abstrakte Funktion oder Variable)
- → inhaltliche Änderung von Name oder Kommentaren ändert Abstraktion

Basisabstraktion durch Programmiersprache vorgegebene Bedeutung
- → wie nominale Abstraktion, aber Änderungen nicht möglich

Datenabstraktion

Abstraktionseinheit = Modularisierungseinheit mit Datenkapselung + Data-Hiding

Funktionen und Variablen als untrennbare Einheit / Sichtbarkeit beschränkt

Modularisierungseinheit selbst ist Abstraktion, aber auch jedes enthaltene Element

→ Kommentare bei Modularisierungseinheit selbst und bei enthaltenen Elementen

→ Klassen- oder Funktions-Bibliothek ist auch nur eine Modularisierungseinheit

abstrakter Datentyp = Schnittstelle einer Modularisierungseinheit

Arten abstrakter Datentypen:

struktureller Typ	nur Signatur betrachtet; strukturelle Abstraktion	(selten)
nominaler Typ	Namen und Beschreibungen wesentlich; nominale Abstraktion	(häufig)
algebraischer Typ	Beziehungen zwischen Funktionen und Daten klar festgelegt; nominale Abstraktion oder λ-Abstraktion	(funktionale Programme)

Abstraktionshierarchien

Abstraktionen vage, aber Beziehungen zwischen abstrakten Datentypen klar spezifiziert

Arten von Abstraktionshierarchien:

- **Beziehung in realer Welt** vage Vorstellungen, durch „is-a“ intuitiv verknüpft
  → oft als Ergebnis der objektorientierten Modellierung (Ersetzbarkeit unbekannt)
- **Untertypbeziehung** B ist Untertyp von A wenn jede Instanz von B verwendet werden kann, wo eine Instanz von A erwartet wird (Ersetzbarkeitsprinzip)
  → in objektorientierter Programmierung essentiell (garantiert Ersetzbarkeit)
- **Vererbungsbeziehung** Übernehmen von Programmtexten aus Oberklasse in Unerklasse
  → historisch bedeutsam, aktuell nur mehr als Hilfsmittel (keine Ersetzbarkeit)
- **Untertypbeziehung höherer Ordnung** wenn $B\langle U\rangle$ Untertyp von $A\langle U\rangle$ für alle U
  → für bestimmte Formen der Generizität bedeutend (keine Ersetzbarkeit)
- **Simulation** 1.: Beziehung zwischen Softwareobjekten und Objekten der realen Welt
  2.: Beziehung zwischen formalen Systemen, vergleicht Ausdrucksstärke

Arten von Variablen

lokale Variable in Funktionen, ...; bei Aufruf angelegt, bei Rückkehr freigegeben

Parameter ähnelt lokaler Variable; Aufrufer initialisiert Parameter mit Argument

- **Eingangsparameter** Daten fließen von Aufrufer zu Aufgerufenem;
  gehen bei Rückkehr verloren; Argument ist beliebiger Wert
- **Durchgangsparameter** Daten fließen in beide Richtungen;
  Argument ist initialisierte Variable; oft als *Referenzparameter*
- **Ausgangsparameter** Daten fließen von Aufgerufenem zu Aufrufer;
  Argument ist Variable

Variable in Modularisierungseinheit z. B. Objekt- oder Klassenvariable (gehört zu Modul);
langlebig, meist durch Speicherbereinigung freigegeben

globale Variable 1.: langlebige Variable außerhalb von Modularisierungseinheit;
2.: Variable in statischer Modularisierungseinheit (aber nicht Objektvariable)

Kommunikation über Variablen

an einer Stelle in Variable geschrieben, andere Stelle von gelesenem Wert abhängig

→ gemeinsame Variablen erlauben komplexe Steuerung des Programmablaufs, mächtig
→ unterlaufen durch Kontrollstrukturen gegebenen Programmablauf kaum durchschaubar
→ können scheinbar unabhängige Programmteile verknüpfen Sicherheitsrisiko

Referenzen (Zeiger) können Modifikation von referenzierten Daten ermöglichen

→ Aliase erlauben komplexe Abläufe einfach auszudrücken, noch mächtiger
→ verschleiern Programmabläufe noch schwerer durchschaubar, erhöhtes Sicherheitsrisiko

Kommunikation über Variablen kann Kontrollstrukturen weitgehend ersetzen:

bei *Lazy-Evaluation* bauen Kontrollstrukturen Abhängigkeiten zwischen Daten auf, Reduktion (Ausführung) erst wenn Daten benötigt; Reduktionen wie im λ-Kalkül
ermöglicht flexible Programmierstile Regelauswahl in λ-Kalkül; Beispiel: Iteratoren

Strategien im Umgang mit Kommunikation über Variablen

prozedurale Programmierung: Programmfluss soll Kontrollfluss entsprechen
→ globale Variablen und Aliase erlaubt und nötig, aber unerwünscht → zurückdrängen
→ zur Modularisierung höchstens Module, Prozeduren/Objekte nicht als Daten

funktionale Programmierung: Funktionen als Daten, ersetzen Kontrollstrukturen
→ Seiteneffekte verboten, Aliase stören dadurch nicht (*referentielle Transparenz*)
→ Modularisierung wichtig, λ-Abstraktion, nominale und strukturelle Abstraktion

objektorientierte Programmierung: Prozeduren zusammen mit Objekten als Daten
→ Objekte, Variablen haben nominale abstrakte Datentypen, abstrakt verständlich
→ dynamisches Binden erzwingt abstraktes Verständnis (Kontrollfluss zu unklar)
→ örtlich eingegrenzte Kommunikation über Variablen (`private`)
→ offensiver Umgang mit Aliasen (Identität versus Gleichheit)

Verteilung der Daten

prozedurale Programmierung: globale Daten, jeder Wert nur auf eine Weise zugreifbar

funktionale Programmierung: große Strukturen; änderbare Daten referenzieren stabile

objektorientierte Programmierung: verschiedene Daten zu Objekt zusammengefasst

verteilte Programmierung: Verteilung über viele Rechner (Daten und Berechnungen); Daten oft redundant; Daten und Ort der Berechnungen zusammenbringen

parallele Programmierung: kurze Gesamtlaufzeiten auf vielen Prozessoren angestrebt; Daten meist in Bereiche aufgeteilt, die unabhängig bearbeitbar sind

nebenläufige Programmierung: effiziente Reaktionsfähigkeit auf Ereignisse angestrebt; unterschiedliche Handlungsstränge sollen auf unterschiedliche Daten zugreifen

Echtzeitprogrammierung: garantierte Antwortzeiten; keine zufallsbasierte Strukturen

Metaprogrammierung: Programme sind Daten; mit anderen Paradigmen kombinieren

Typprüfungen

Arten von Typprüfungen hinsichtlich Zeitpunkten:

- **statische Typisierung** — Compiler kennt Typen von Ausdrücken und garantiert Konsistenz
  → statisches Wissen hilft beim Programmverstehen, eingeschränkte Flexibilität
- **starke Typisierung** — Compiler kennt Typen nur teilweise, garantiert dennoch Konsistenz
  → Wissen um Typkonsistenz, darüber hinaus Programmverstehen nicht unterstützt
- **dynamische Typisierung** — Compiler kennt Typen nicht, Konsistenzprüfung zur Laufzeit
  → Programmverstehen nicht unterstützt, Typfehler zur Laufzeit möglich

Mischformen häufig — z. B. ist Java stark typisiert, außer für Arraygrenzen, ... (dynamisch)

Darstellung von Typen im Programm:

- **explizit spezifiziert** — Typinformation als zuverlässige Form der Dokumentation
- **inferiert** — Typinformation ist implizit, eindeutig aus Kontext abgeleitet
  - **global inferiert** — statische Typisierung ohne Typangabe, kein dynamisches Binden
  - **lokal inferiert** — aus expliziten Typen in der Umgebung, meist starke Typisierung

Typen und Entscheidungsprozesse

explizite Typinformation reflektiert Entscheidungen und hält sie fest

je früher Entscheidung getroffen wird, desto stabiler und effizienter

Entscheidungszeitpunkte (mit Beispielen)

Sprachdefinition/Sprachimplementierung	`int` ist 32-bit-Zweierkomplementzahl
Erstellung von Übersetzungseinheit	Variable `x` ist vom Typ `int`
Einbinden statischer Modularisierungseinheiten	Typparameter `T` steht für `Integer`
Compilation	(semantisch unbedeutende Optimierungen)
Initialisierungsphase	`if (!y instanceof Integer) throw ...`
eigentliche Programmausführung	`if (y instanceof Integer) ... else ...`

bei Änderung einer Entscheidung hilft Compiler beim Finden betroffener Stellen

Nominale und strukturelle Typen

nominaler Typ hat einen eindeutigen Namen, struktureller Typ nicht

nominaler Typ	struktureller Typ
je zwei Typen gleich bei gleichem Namen	je zwei Typen gleich bei gleicher Struktur
an nur einer Stelle definiert	an mehreren Stellen definierbar
nominale Abstraktion	strukturelle Abstraktion
hohe Abstaktionsgrade möglich	Abstraktionsgrad stets niedrig
Untertypbeziehungen explizit	Untertypbeziehungen implizit
Spezifikation über Kommentare (DbC)	Kommentare nicht relevant
Programmierer_in sichert Ersetzbarkeit zu	nur strukturelle Ersetzbarkeit
Name stellvertretend für Eigenschaften	man darf keine Eigenschaften annehmen
gut lesbar	weniger gut lesbar
erfordert Programmierdisziplin	einfache Verwendung

Rekursive Datenstrukturen

induktive Konstruktion von Typen rekursiver Datenstrukturen nötig

Beispiel:
$$M_0 = \{\texttt{end}\}$$
$$M_{i+1} = M_i \cup \{\texttt{elem}(n, x) \mid n \in \texttt{Int}; x \in M_i\}$$

$M = \bigcup_{i=0}^{\infty} M_i$ enthält `end`, `elem(1,end)`, `elem(2,elem(1,end))`, ...

kürzere Darstellung (Haskell): `data IntList = end | elem(Int, IntList)`

Fundiertheit: M_0 darf nicht leer sein es muss eine rekursionsfreie Alternative geben

häufig: M_0 vordefiniert, Programm beschreibt Konstruktion von M_{i+1} aus M_i
in Java: M_0 enthält `null`, jede Klasse entspricht Konstruktionsvorschrift von M_{i+1} aus M_i

Statisches Propagieren von Eigenschaften

```
public static int plusZwei(int x) { return x + 2; }
...        int y = plusZwei(3);
```

Typkonsistenz garantiert, dass Typen von `x` und 3 sowie `x + 2` und `y` zusammenpassen

statisch bekannte Eigenschaften werden vom Argument zum Parameter propagiert, auch von der rechten Seite einer Zuweisung zur linken, ...

das gilt nicht nur für Typinformation, sondern für jede statisch verfügbare Information

z. B. kann damit die Information transportiert werden, dass ein Wert ungleich `null` ist

Compiler muss erkennen, welche Information statisch vorliegt, Propagieren ist einfach