Einfachheit und Flexibilität

Strukturierte Programmierung

Programm durch wenige einfache Strukturen beschrieben:

- Sequenz (Hintereinanderausführung)
- Alternative (Verzweigung)
- Wiederholung (Iteration oder Rekursion)

Ziel: **ein Anfangspunkt** und **ein Endpunkt** um

- Programmablauf einfach zu erfassen,
- Strukturen uneingeschränkt zu kombinieren

Verletzung der strukt. Programmierung

`goto` (Sprung an beliebige Programmstelle, nicht in Java)

Fall-through (`case`-Klausel ohne `break` am Ende)

`break` zum Verlassen von Schleifen

`continue` in Schleifen

`return` an beliebigen Programmstellen

Werfen und Abfangen von Ausnahmen

Dangling-`else`: `if(a) if(b) ...; else ...;`
(Anfangs- oder Endpunkt der Kontrollstruktur unklar)

Fallen objektorientierter Sprachen

Ersetzbarkeit angenommen, aber nicht gegeben

unnötig komplexe Zusicherungen

kovariante Probleme (und schlechte Lösungsversuche)

Casts auf Referenztypen

Generizität statt Ersetzbarkeit

unnötig weite Sichtbarkeit

Vererbung auf wenig stabilen Klassen

Simulation unnötiger Funktionalität

Spezielle Fallen in Java

Klasse versus Interface

überladene Methoden statisch aufgelöst (diffizile Regeln)

Überladen statt Überschreiben (durch `@Override` erkennbar)

Dateiverzeichnisse als Pakete

Raw-Types bei fehlenden Ersetzungen von Typparametern

falsche Sicherheit

Gefährliche Typkompatibilitäten (Java)

Ersetzbarkeit von Arrays gefährlich:

```
B[] bs = new B[10];
A[] as = bs;       // erlaubt wenn B Untertyp von A
as[0] = new A();   // Ausnahme beim Einfügen
```

Generizität bei richtiger Verwendung sicher:

```
Liste<B> bs = new Liste<B>();
Liste<A> as = bs; // Compiler meldet Fehler
as.add(new A());  // daher nicht compilerbar
```

Raw-Types (bei Generizität) gefährlich:

```
Liste<B> bs = new Liste<B>();
Liste as = bs;    // Liste ist Raw-Type, erlaubt
as.add(new A());  // Ausnahme beim Einfügen
```

Aufgabe: Gefährliche Sprachkonzepte

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Frage:

Warum unterstützen Programmiersprachen gefährliche Konzepte?

Zeit: 2 Minuten

PK

Vertrauen und Kontrolle

Programmierstile

defensiv: zur Laufzeit prüfen, ob Bedingungen eingehalten

> „Vertrauen ist gut, Kontrolle ist besser“
> klingt nach hoher Qualität, muss aber nicht so sein;
> Problem: unnötige Prüfungen, hoher Aufwand

offensiv: darauf verlassen, dass Bedingungen eingehalten

> „Solange sich niemand beschwert, wird es schon passen“
> riecht nach Schlampigkeit, aber oft vorteilhaft;
> Problem: erfordert gut eingespieltes Team

Stil ist richtig wenn an jeweilige Situation angepasst

Defensiv vs. offensiv nach Situation

Validierung: Überprüfungen zur Laufzeit in jedem Stil nötig

man braucht klare Regeln, wo Überprüfungen erfolgen,
Überprüfungen dürfen nicht zu streng oder schwach sein,
mehrfache inkonsistente Prüfungen ganz schlecht

Zusicherungen: Bedingungen sind einzuhalten (auch ohne Überprüfungen)

Überprüfungen oft nicht möglich wo Bedingungen einzuhalten sind,
systematische mehrfache Überprüfungen unzweckmäßig,
`assert`-Anweisung für „unnötige“ Überprüfung sinnvoll

Wiederverwendbarkeit

Zusicherungen oft für **einen** Anwendungsfall ausgelegt

Zusicherungen zu restriktiv $\longrightarrow$ andere Anwendungsfälle behindert

statische Analyse lässt unnötige Zusicherungen erkennen

`assert`-Anweisungen helfen beim Erkennen

sorgfältige defensive Programmierung unterstützt Analyse

unbedachte defensive Programmierung behindert Wiederverwendung

Programmierstil und Vertrauen

einige Einflussfaktoren auf Vertrauen in Programmcode:

- Klarheit der Schnittstellen
- Verständlichkeit des Codes
- Zusammenpassen von Code und Entwicklungsprozess
- Intensität der Wartung
- Standardkonformität

Mangelndes Vertrauen im Team

Eigenbrötelei und mangelnde Kommunikation,
Mehrgleisigkeiten,
viel Code für sinnlose Überprüfungen,
Fehler durch Widersprüche und Inkonsistenzen,
hoher Ressourcenverbrauch,
schließlich Scheitern eines Projektes

Team-Regeln für einheitlichen Stil

Form der Dokumentation und Art der Kommentare,
Vorschriften für Einrückung, Klammerung, ...
wer was wie unter welchen Bedingungen ändern darf,
Vorgehen bei Entwurf, Testen, Code-Review, ...
...

fördern auch Teamzusammengehörigkeit