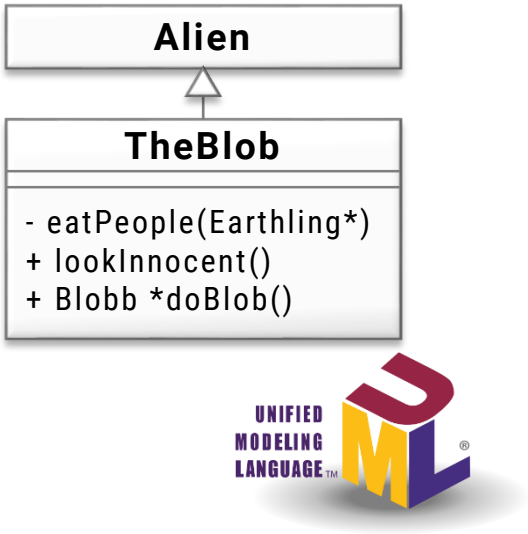
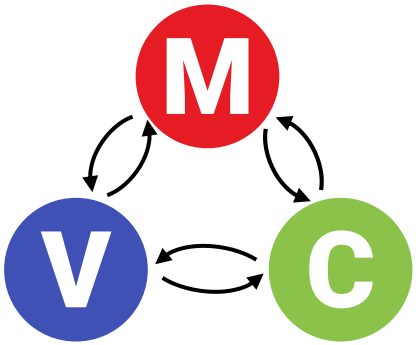
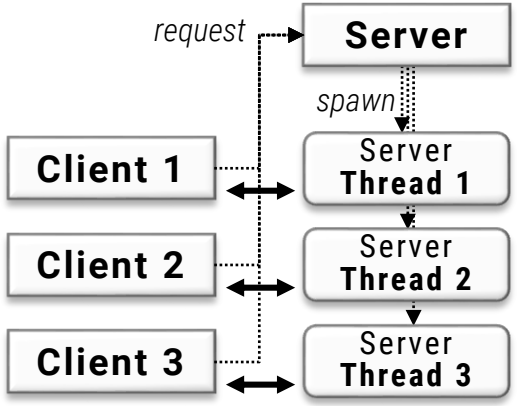
Einführung in die Softwareentwicklung

SOMMERSEMESTER 2020





Design **Patterns**



Architectural **Patterns**

Foliensatz #10

Software Entwurf – Prozesse

Michael Wand · Institut für Informatik · michael.wand@uni-mainz.de

Übersicht

Softwareentwurf

- Entwicklungsprozess
- Design und Architektur
  - Prozedural, objektorientiert, funktional
  - Diagramme als Hilfsmittel (insbes. UML)
  - Designpatterns (-muster)
  - Architekturmuster
- Beispiele
  - „OO-Basisarchitektur“ (JAVA/Smalltalk Style, Value-Types)
  - GUIs: *Typ- und Compositionshierachien*, *Events*, *MVC*, *FPR*
  - Client-Server-Architekturen

Klausurfragen?

Nicht alles ist „Wissenschaft“

- Viele der besprochenen Themen basieren auf
  - Erfahrung
  - Geschmack
  - Soziologischer Evolution (what’s hip, what worked)
- Auswendiglernen macht keine Sinn
  - Keine Wissensfragen in der Klausur
  - Wenn Fragen, dann „warum macht man das“?
- Selbst weiter recherchieren!
  - Gesunde Skepsis ist, wie gesagt, gesund!

Literaturhinweis

Kurze Übersicht zu „Softwareentwurf“

- Bjarn Stroustrup
  „Die C++ Programmiersprache“
  Addison Wesley,
  2. Auflage 1992 (Kapitel fehlen in neuster Auflage)
  - Kapitel 11-13: (insbesondere 11)
    - „11 Programmentwicklung“
    - „12 Design und C++“
    - „13 Bibliotheksdesign“
- Interessante Meinungen zum Thema vom Erfinder von C++
  - Insbesondere Kapitel 11: unabhängig von C++
- Tlw. Basis dieses Abschnitts (aber: C++ und Python/JAVA)

Der Entwiclungsprozess

Was wollen wir erreichen?

Ziele

- Funktion, Robustheit, Testbarkeit
- Flexibilität, Erweiterbarkeit
- Wiederverwendbarkeit
- Verständlichkeit
- Portabilität

(Leicht übersehen)

- (Soziologische Stabilität des Entwicklungsteams)
- (Mentale Stabilität der Anwender des Produktes)

Softwareentwicklung ist schwer

Softwareprojekte

- Hohe Misserfolgsquote
  - Statistiken schwanken
  - Auf Google findet in etwa solche Zahlen:
    - „nur 50% erfolgreich“
    - der Rest scheitert (20%)
    - oder überschreitet Zeit und Budgerahmen deutlich (30%)
- Projektteams sind nicht unbedingt sehr groß
  - Die meisten unter 5 Entwickler
  - Man braucht kein 100-Personen Team für Chaos
- Kampf gegen Komplexität – jede Hilfe nutzen

Kernprinzip

Stroustrup

„*Programm-Design und Programmieren sind menschliche Aktivitäten.*

Wer dies vergisst, hat bereits verloren.“

Struktur eines Entwicklungsprozesses

Aspekte der Softwareentwicklung

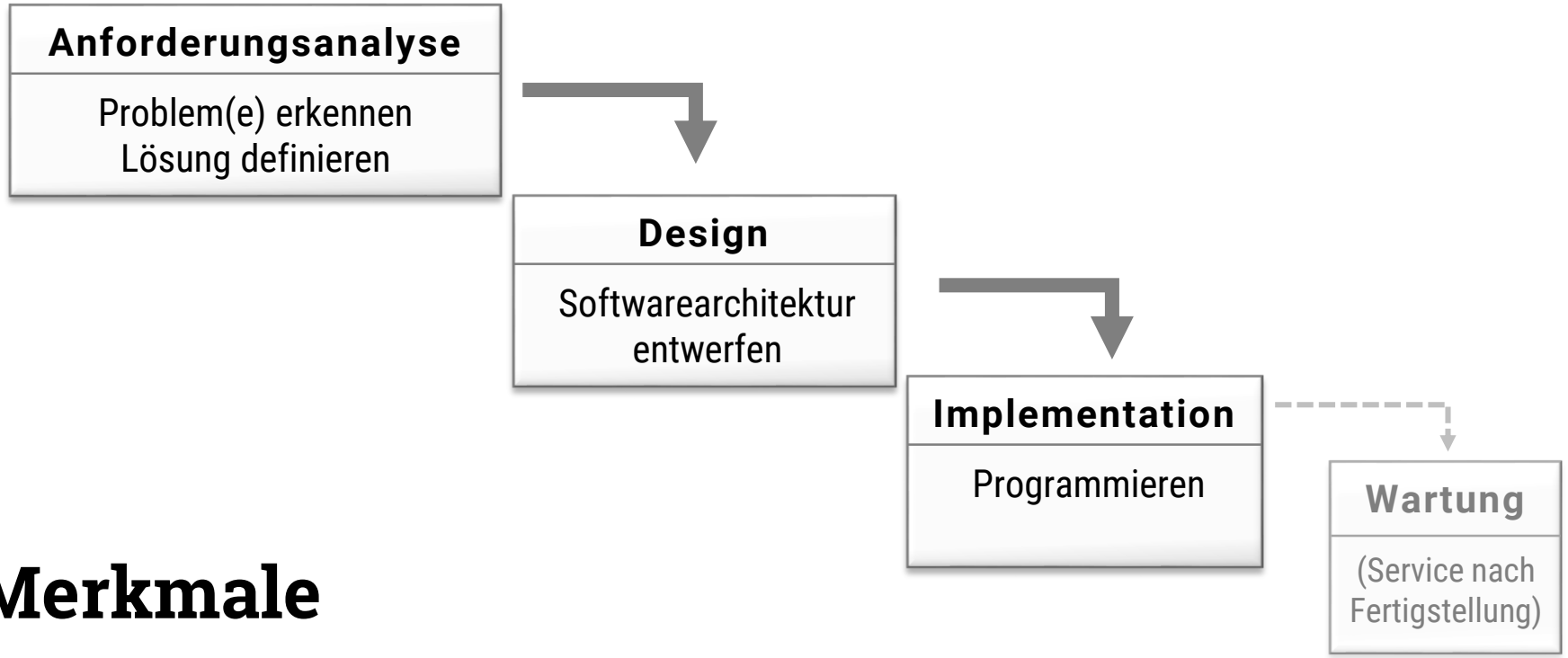
Bestandteile / Schritte

- Analyse (Anforderungsanalyse)
- Design (Softwarearchitektur)
- Implementation („Programmieren“)

Weitere Aspekte

- Management (Meetings, Kommunikation, Entscheidungsprozesse, etc.)
- Dokumentation
- Testen
- Experimentieren / Prototypen

Klassisch: Wasserfallmodel



Merkmale

- Erst Planen, dann ausführen
- Kommunikation (hauptsächlich) abwärts
- Oft an soziologische Hierarchie (auch €€€) gekoppelt
  - Mehr „Macht“ weiter oben (hoffentlich) erfahrenere Entwickler/Innen

Kritik

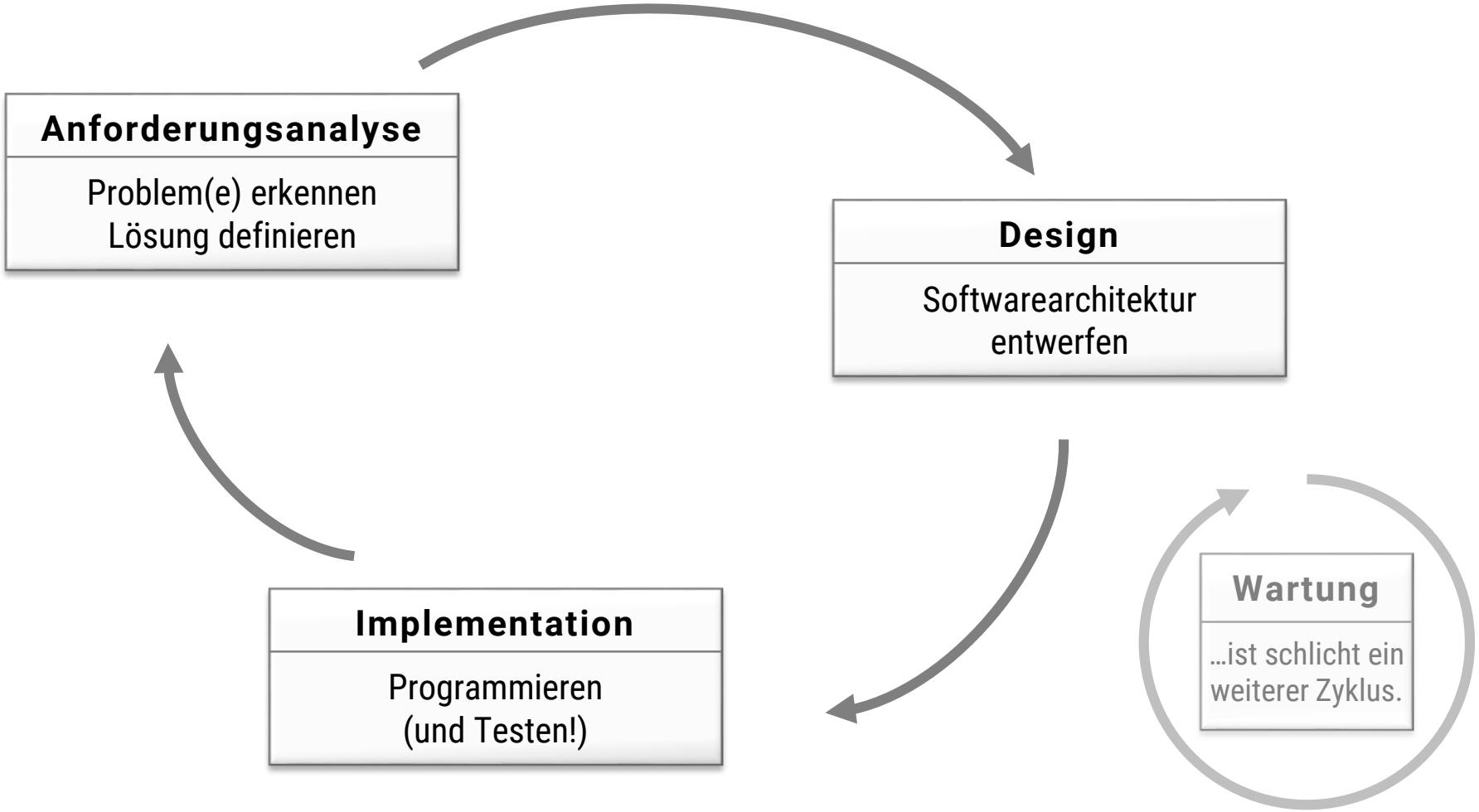
Unbestrittene Vorteile

- Die meisten Fehler werden im ersten Schritte gemacht
  - Die zweitmeisten im Design
  - Explizite Analyse, Design extrem wichtig
- Analyse & Design erfordert mehr Erfahrung

Kritik (Stroustrup)

- Kommunikation hauptsächlich „abwärts“
  - Fehler in Analyse und Design werden spät gefunden
  - Oft „local fixes“ mit „zerstörerischer Struktur“
- Starre Hierarchie hinderlich (Feedback, pers. Entwicklung)

„Moderne“ Sicht: Iterativ



Schritte wiederholen, inkrementelle Verbesserung

Weitere Aspekte

Management

- Verschiedene Vorgehensmodelle
  - z.B. „XP (extreme programming)“, „Agile“, „Scrum“, etc.
  - Wichtiges Thema in Vorl. „Software Engineering“
- Vorgehen abhängig von Projektgröße und Komplexität
  - 2 Entwickler, „triviales“ Tool mit 10.000 LOC*)
    → Reinhacken nach 2h Meeting möglich
  - 7 Entwickler, 1M LOC, 1 Jahr Laufzeit
    → Komplexeres Vorgehensmodell
  - 2 Entwickler, 10.000 LOC, Autopilot für Passagierflugzeug
    → Starke Formalisierung, Zertifizierung, mehrjährig
  - Großprojekt, >10 Personen (z.B. Linux BS, Office Packet): komplexe, oft vielschichtige Struktur

**) LOC = „lines of code“*

Weitere Aspekte

Dokumentation

- Offensichtlich wichtig
  - Oft vernachlässigt
- Zuviel kann auch schädlich sein
  - Unflexibel wg. Änderungsaufwand
    - „dann müssen wir das ganze Handbuch neu schreiben…“
    - Argument für iterative Entwicklung
  - Auch problematisch: „out-of-sync“
    - Doku entspr. nicht „ist-Zustand“
    - Abhilfe: Integration von Dokumentation und Entwurf/Progr.
    - In der Implementationsphase z.B.:
      JAVA-Doc, Python-Doc-Comments, Doxygen (C++)
    - „CASE“-Tools für Analyse, Entwurf (computer aided SE)

Weitere Aspekte

Testen

- Sehr wichtig
- Gleiche Aufmerksamkeit wie programmieren selbst

Diskutiert in EiP

- Unit-Tests, Assertions, Test-Suites

Vorgehensmodell (Bestandteil)

- „Test-Driven-Development“ (TDD)
- Tests parallel zum Code schreiben kontinuierlich testen

Experimentieren / Prototypen

Erfahrung ist wichtig

- Gutes Design für unbekannte Problem kaum möglich (meine Erfahrung)
- Der zweite oder dritte Anlauf funktioniert

Erfahrungen gewinnen

- Experimente sind wichtig!
  - Einfache Lösung „reinhacken“ und schauen, wo es kracht
  - Prototypen bauen
- Gefahr von Prototypen
  - Wenn die Deadline naht, wird daraus schnell das Produkt

Leitlinien

Stoustrup (Auszug)

- Wisse, was Du erreichen willst
- Stecke Dir spezifische und erreichbare Ziele
- Suche nicht nach technischen Lösungen für soziologische Probleme
- Denke langfristig
  - im Design
  - im Umgang mit Menschen
- Verwende [gute*)] Systeme als … Inspiration
- Entwerfe in Hinsicht auf Änderung
  - Flexibilität, Erweiterbarkeit, Portabilität, Wiederverwendung

*) eigene Ergänzung

(Die C++ Programmiersprache, 2. Auflage, Kap.11)

Leitlinien

Stoustrup (Auszug)

- Verwende die besten Werkzeuge…
  - im Design und
  - in der Implementation
- Experimentiere, analysiere und teste so früh und oft wie möglich
- Halte ein der Projektgröße angemessenes Niveau von Formalisierung
- Einfachheit: so einfach wie möglich, aber nicht einfacher

(Die C++ Programmiersprache, 2. Auflage, Kap.11)

Be Careful, but Relax…

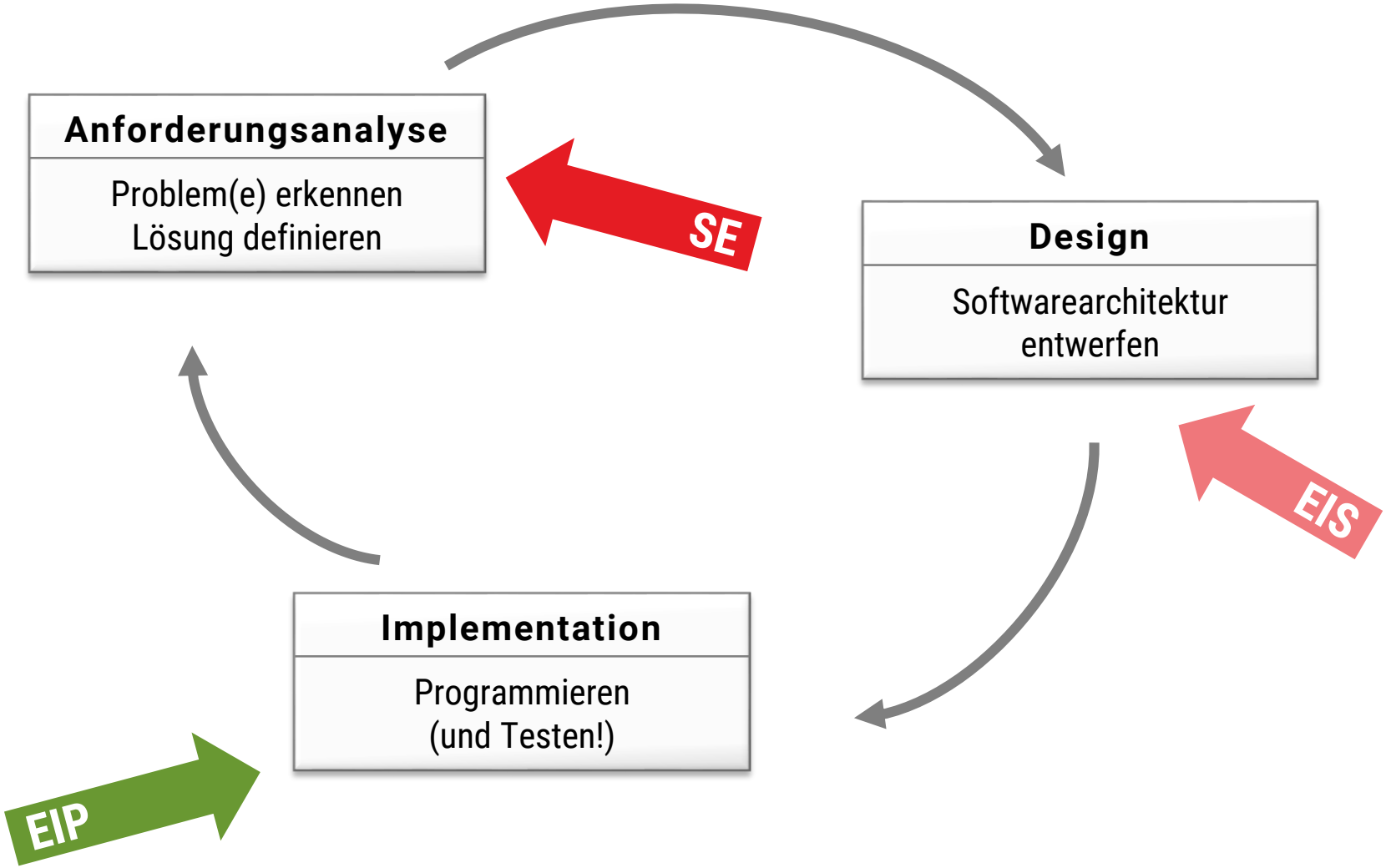
Softwareentwurf

- Menschliche Aktivität
- Erfahrung und „guter Geschmack“ unersetzlich
- „There is no silver bullet“ (Fred Brooks)
  - Skeptisch bei „absoluten“ Versprechen sein

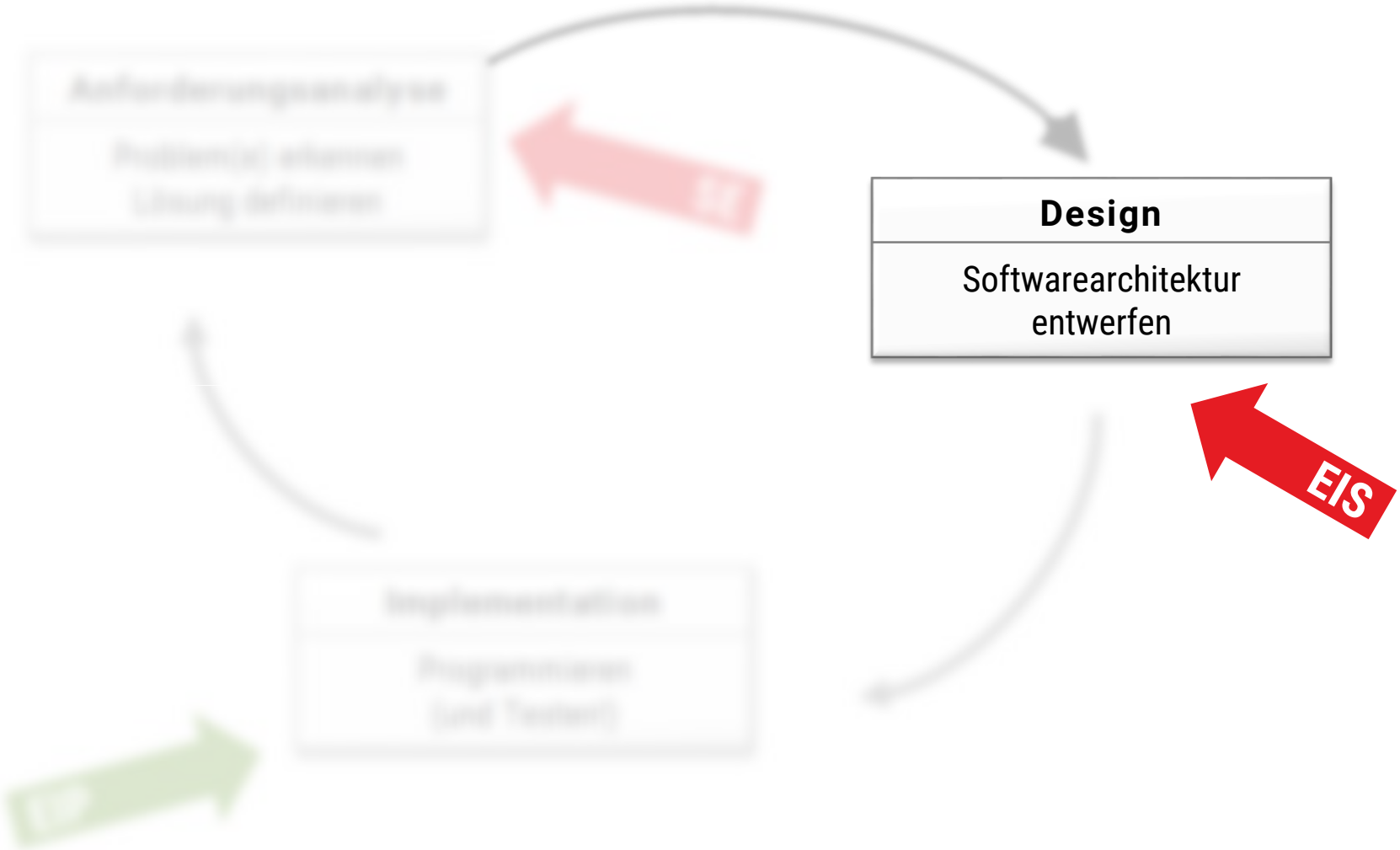
Relax

- Nicht übertreiben
  - Einfaches Design, mit Mitteln, die man vollständig versteht, ist viel besser als „fancy“ aber nicht völlig verstanden
- Auch Perfektionismus kann Projekte töten

Nun: Softwareentwurf (Design)



Nun: Softwareentwurf (Design)



Prozedural, OOP, Funktional

Ziel der Programmiertechniken

Wiederverwendung!

- Alles „nur einmal programmieren“
  - DRY – don’t repeat yourself
  - Fehler vermeiden
  - Arbeit minimieren
  - Strukturierung des Systems
- Alle strukturellen Programmiertechniken (für „Software-Entwurf“) dienen letztlich diesem Zweck

Leitlinie

Softwareentwurf als Bilbliotheksentwurf

- Softwarekomponenten als allgemeine Problemlösungen
- Unabhängig von Kontext / Anwendung nützlich
- Macht es einfacher, Kernidee zu identifizieren
- Nicht zu allgemein
  - So allgemein wie möglich ohne (übertriebenen) Zusatzaufwand
  - Bewährte Muster funktionieren oft am besten (wenn verfügbar / bekannt)

Zusammenfügen

Kernproblem: Schnittstellen

- Schnittstellen sollen abstrahieren
  - z.B. 200 LOC → eine Funktionssignatur
- Schnittstellen sollen einfach sein
  - Einfaches Grundprinzip
  - Klare Annahmen / Invarianten
  - Leicht zu verstehen
  - „As simple as possible but not simpler“
- Schnittstellen gut dokumentieren
- Flexibler Einsatz (z.B. via Polymorphie)

Techniken

Strukturierung von Programmen

- Prozedural
- Funktional
- Objekt-orientiert
- Meta-Programmierung
  - „Statisch“
  - „Dynamisch“

Prozeduraler Entwurf

Prozedurale Programmierung

Prozedurale Programmierung

- Wie in EiP (vor dem OOP-Teil)
- Zwei Strukturelemente
  - Funktionen / Prozeduren
  - Datentypen (`structs`, Arrays, Zeiger)

„Strukturierte Analyse“ (1980er)

- Aufteilen des Programms in Funktionen
- Modellieren der Daten als Structs (bzw. Arrays)
- Visualisierung als *Datenflussdiagramme*

Datenflussdiagramme (DFDs)

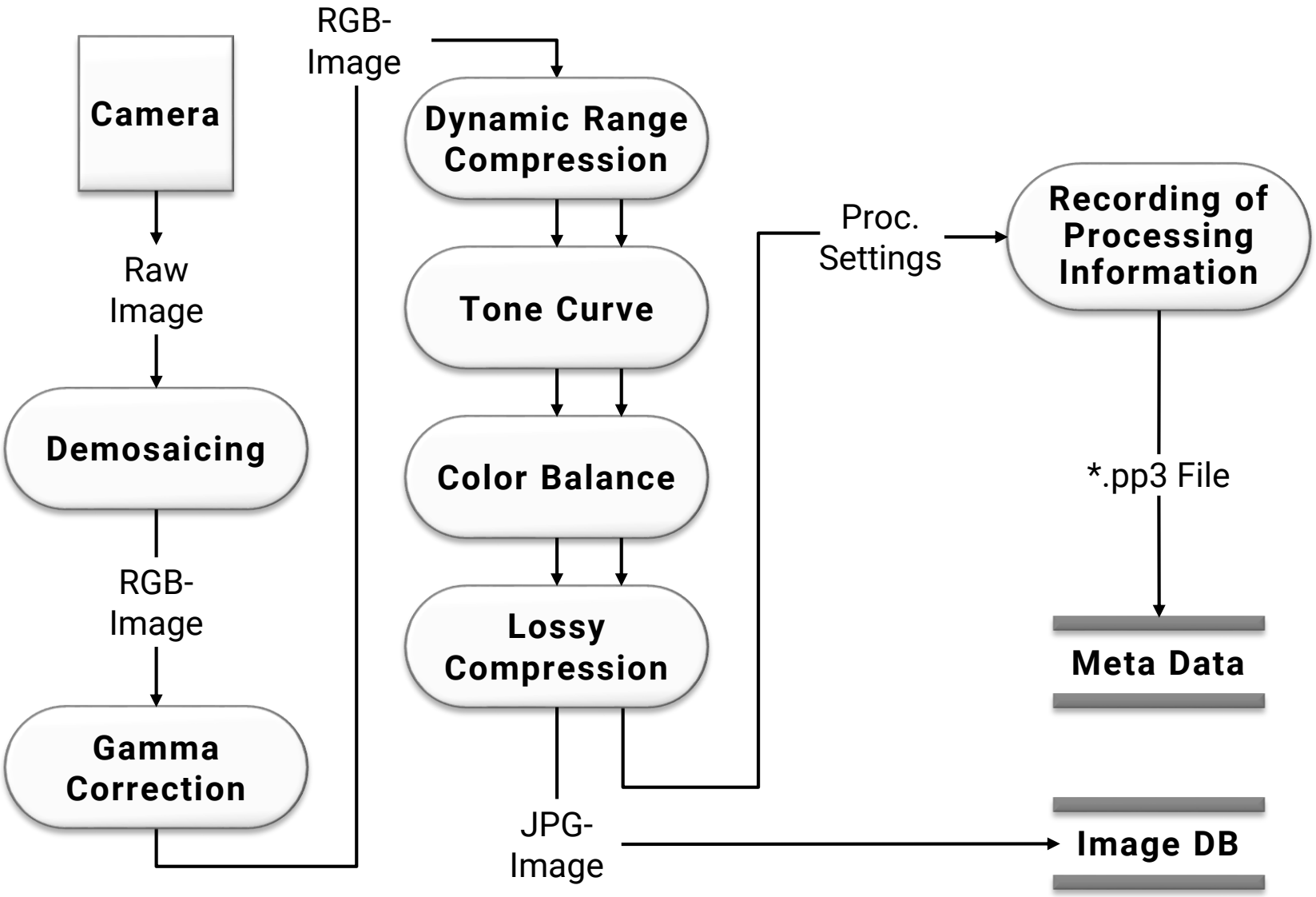
Funktion

Input / Output

Datenspeicher

Datenfluss

Beispiel DFD: RAW-Converter



Vorgehen

Wie gehe ich vor?

- „Divide and Conquer“
- Funktionen identifizieren
  - Vorgänge, Algorithmen
- Daten identifizieren
  - Welche Informationen fließen?
- Von grob nach fein
  - Immer weiter aufteilen
  - Bis Problem lösbar
- Bottom-up auch möglich
  - Bibliotheksdesign

Funktionale Programmierung

Funktionale Programmierung

Grundidee

- Genauso wie prozedural

Also wo ist der Unterschied?

- Erweiterung:
  Funktionen als Datentyp („Code als Daten“)
- Spezielle Einschränkungen:
  Rein funktionale Komposition ohne Seiteneffekte

Funktionale Programmierung

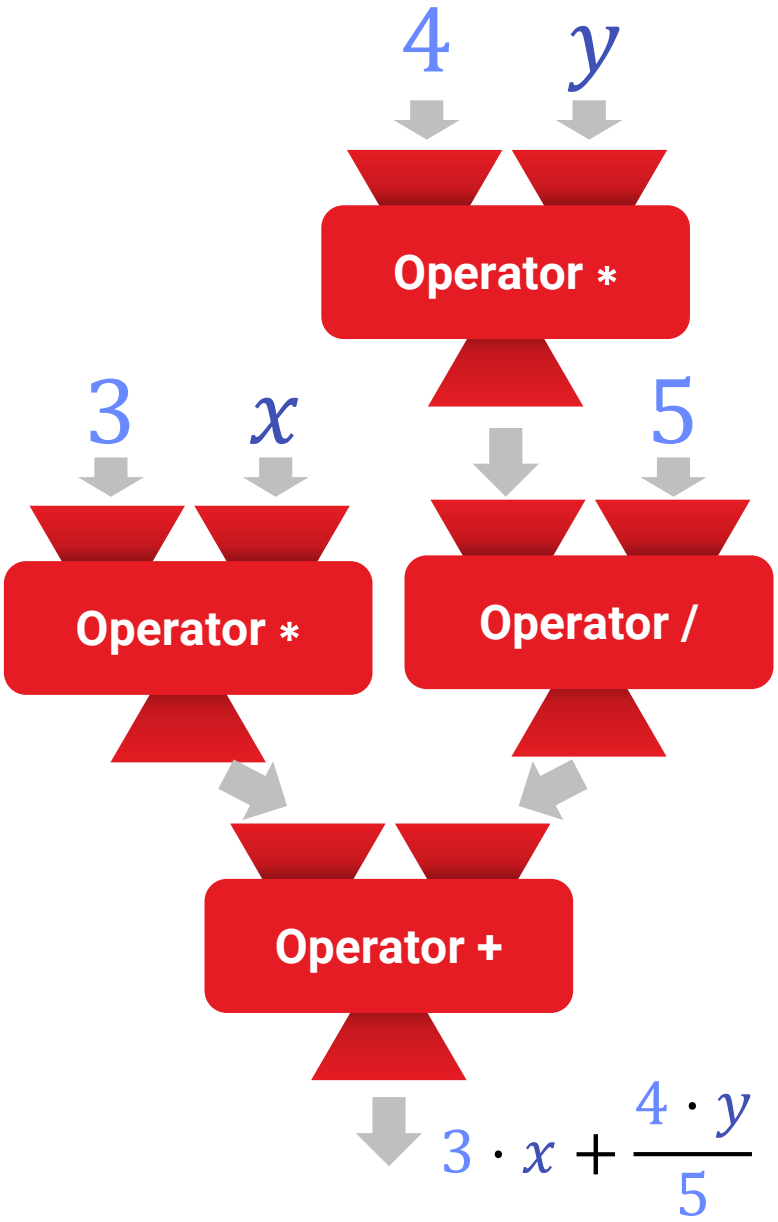
Erweiterung

- „Functions as first-class citizens of the language“
- (Zeiger auf) Funktionen als Datentyp
  - In C/C++, Pascal/Modula/ADA, JAVA/C#, Python problemlos möglich
  - „`typedef int (*IntToIntFunc)(int);`“

Einschränkungen

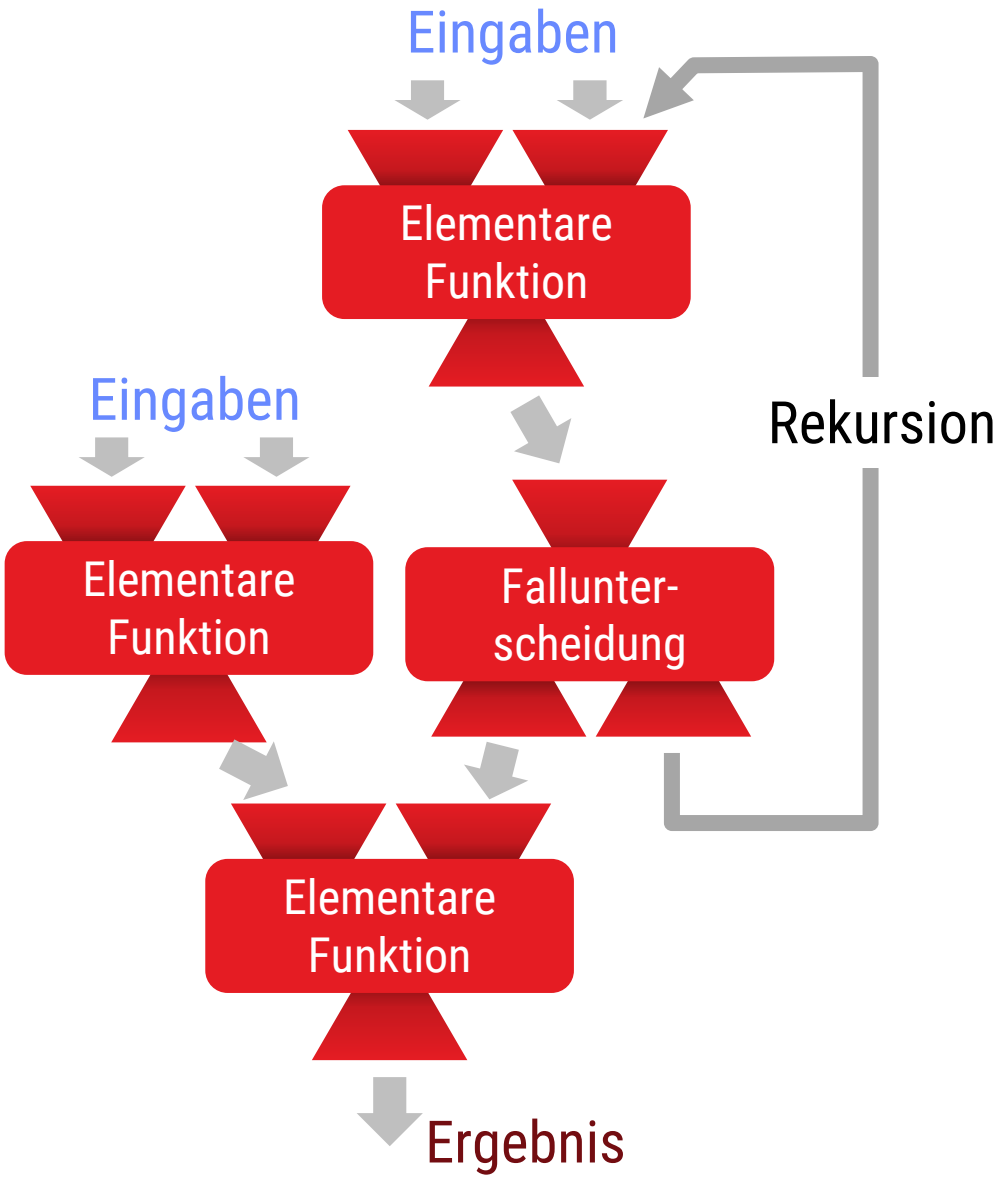
- Keine (oder eingeschränkte) Änderung von Variablen
  - „Avoiding mutable state“
- Dies impliziert: Rekursion statt Schleifen

Einfache Funktionen



“$3 \cdot x + \frac{4 \cdot y}{5}$”

Funktionale Programmierung



Bausteine

- Elementare Funktionen
- Rekursion

Beispiel

$$f(x) = \begin{cases} 1, & \text{falls } x = 0 \\ x \cdot f(x-1), & \text{sonst} \end{cases}$$

Turing-mächtig

- Gleiche Funktionen berechenbar wie C++
- Modell ignoriert Interaktion, I/O u.ä.

Fakultät Funktional

Fakutät – imperativ (prozedural, C++)

```
unsigned factorial(unsigned n) {
    unsigned result = 1;                // mutable state
    for (unsigned i=2; i<=n; i++) {
        result = result * i;            // mutation of state
    }
}
```

Fakutät – „rein“ funktional (C++)

```
unsigned factorial(unsigned n) {
    return (n <= 1) ?                   // no (explicit) mutable state
              1 : factorial(n-1)*n;// “?” wertet nur zutreffenden Fall aus!
}                                       // C++ unterstützt damit “pure functional style”
```

Warum?

Problem von Variablenänderung („mutable state“)

- Problematisch sind globale Variablen, bezogen auf aktuellen Kontext

Erwarteter Schwierigkeitslevel

- Hoch: Variablen außerhalb der Funktion, die implizit geändert werden (keine Parameter)
- Mittel: Referenzen auf äußere Variablen (Prozedural: „**&**“, „*****“ ohne **`const`**, OOP: **`self`**, **`this`**)
- Gering: Variabeln (äußerer) Schleifen

Reasoning about mutable state

Es ist schwierig(er) zu verstehen…

- …wie Code funktioniert, der nicht-lokale Effekte hat
  - Änderungen von Variablen verändern Bedeutung
- „Pure Functions“: kein Schreiben, nur Rückgabe
- Reine Funktionen verhalten sich
  - bei gleichen Eingabeparametern
  - immer gleich (gleiche Ausgabe)
  - „Referentielle Transparanz“
    - Man könnte Werte einsetzen
- Dies erleichtert
  - Verständnis (nicht immer! – es gibt einige Gegenbeispiele)
  - Beweis mit Invarianten (auch nicht immer)

Leitlinie

Gezielte Änderungen

- Reine Funktionen sind oft (nicht immer) einfacher zu verstehen
- Globale Seiteneffekte (state mutation) zu minimieren ist immer eine gute Idee
  - Lokale Seiteneffekte (z.B. Schleifenvariablen) i.d.R. unkritisch
- Prozedurale / OOP Entwürfe profitieren auch davon

Architekturen, die Änderungen kapseln

- Speziell ausgezeichnete Mechanismen, wenn persistente/globale Daten geändert werden

Entwurfsstrategien

Entwurfsstrategie

- Ähnlich wie prozedurale Programmierung
- Datenfluss (ggF. rekursiv)
- Vermeidung von Änderungen von Variablen (die global sind bez. auf aktuelle Funktion)
  - Statt dessen: Parameterübergabe
  - „Streng funktional (pure)“: Rekursion statt Schleifen

Entwurfsmuster

Was ist mit Funktionen als Daten?

- Interessante Muster für „Funktionszeiger“
- Einige Beispiele folgen später
- Mehr funktionales auch in der Vorlesung „Programmiersprachen“

Objekt-Orientierter Entwurf

Kernkonzepte

Klassen

- Typen von Daten, die
  - Gleich strukturiert sind
  - Gleichartig verarbeitet werden
- **Objekte:** Instanzen der Klassen
- **Vererbung:** Beziehungen zwischen Typen

Methoden

- Code, der an Klassen gekoppelt ist
- Auswahl automatisch nach Typ (dynamic dispatch)

Entwurfsstrategie

Klassen finden

- Daten identifizieren
- Oft empfohlen:
  - Ausgangspunkt: Textuelle Problembeschreibung
  - Nomen finden → Nomen werden Klassen
  - Verben finden → Verben werden Methoden

Beispiel

Beispieltext (Kundeninterview)

„Unsere Personalbuchhaltung verwaltet Daten von Personen. Für jede Angestellte werden zusätzlich Abteilung und Gehalt gespeichert. Die Angestellten können befördert werden, dabei erhöht sich das Gehalt. Monatlich fragen wir ab, wer bei uns angestellt ist.

Ach ja, und wir haben auch mehrere Managerinnen; die bekommen einen Bonus. Wenn das Gehalt berechnet wird, wird der Bonus aufgeschlagen.“

Beispiel

Analyse

- Klassen
  - Person
  - Angestellte
  - Managerin
  - (Bonus) → **`double`**
  - (Gehalt) → **`int`**
  - (Abteilung) → **`string`**
- Methoden
  - befördern()
  - berechneGehalt()
  - abfrage() → **`toString()`**

Weiteres Vorgehen

Ähnlichkeiten

- Vererbungshierarchien identifizieren
  - Ähnliches Verhalten (dies ist entscheident!)
  - Manchmal: Ähnliche Datenstrukturen (oft irreführend)
- Oberklassen formulieren
  - Abstrakte Operationen in Basisklasse
  - Implementationen / Spezialisierungen in Nachfahren
  - Gemeinsame Methoden in Basisklassen (DRY)

Kritik

Naives Modell

- „Nomen/Verb“-Feature ist instabil
- Kann in die Irre führen
  - Über-spezifische Designs
  - Natürliche Sprache ist oft wenig abstrakt
  - Datenverarbeitung / Code entscheidend, Namen egal

Oh je – Vererbung!

- Vererbung kann Designs unübersichtlich und schwer wartbar machen (Code über Hierarchie verteilt)
- Oberflächliche Ähnlichkeiten daher gefährlich

Vererbung?

Wann Vererbung benutzen?

- Vererbung ist eine „ist-ein“-Beziehung
- Im Code können alle Unterobjekte die Oberklasse ersetzen (und alles funktioniert wie intendiert)
- „Substitionsprinzip“
  (*Liskov substitution principle* nach Barbara Liskov)

Worauf achten?

- Verhalten!
  - Gleiche Methoden formen Oberklasse
  - Gleiche Invarianten, ähnliche Semantik

Sparsam Erben

Anfängerfehler Nr. 1 in OOP

- Zu viel Vererbung
- Sparsam / gezielt einsetzen

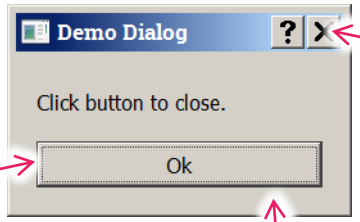
Keine Vererbung

- Delegation: Weiterreichen an andere Funktion
- Aggregation: Objekt besteht aus anderen Objekten

„Composition over Inheritance“

- In solchen Fällen: Objekte anderer Klassen als Datenfelder (Members) statt Oberklassen

Beispiel



Den Ausschalter „×“ vom Allgemeinen Button abstammen zu lassen, macht dagegen Sinn

Jedes Fenster hat genau einen „Ausschalter ×“ Button.

Fenster vom „Ausschalter“-Button abstammen zulassen ist aber eine sehr schlechte Idee.

Fenster „enthalten“ Ausschalter.

viele GUI Frameworks modellieren Buttons als spezielle Fenster, da die Funktionalität gleich ist (etwas unintuitiv, aber gut im Sinne des Substitutionsprinzips)

Wann Erben?

Interfaces / Subtyping für Polymorphie

- Verarbeitung von Datenobjekte
- Gleiche Operationen werden auf Objekte angewandt
- Oberklasse (oder Interface) erlaubt, Algorithmus nur einmal zu schreiben

Subtypes vs. generische Programmierung

- Wann *templates* vs. *dynamic dispatch*?
- Dynamic dispatch erlaubt Erweiterungen zur Laufzeit
- z.B. Laden von Plug-Ins (nicht möglich mit templates)

Wann Erben?

Erben von Code vs. reine Interfaces

- Code erben oder nur reine abstrakte Methoden?
- Nachteile von Vererbung von Code
  - Code-Vererbung koppelt Klassen stärker
  - Strukturelle Änderungen schwieriger
- Vorteil: „Frameworks“
  - Gemeinsame Funktionalität ist konsistent
  - Beispiel: Widget/GUI-Bibliotheken
  - Nur Details müssen ergänzt werden

Wie entwirft man Oberklassen?

Oberklassen finden

- Gemeinsamkeiten extrahieren
- Ziel: Polymorphe Algorithmen

Oberklassen gestalten

- Fokus auf *Methoden*!
- „Was macht ein Objekt dieser Art?“
- Datenfelder möglichst „**`private`**“ machen und über Methoden zugreifen (→ Flexibilität)

Woran erkenne ich gutes Design?

Merkmale, die mit gutem Design korreliert sind

- „Single responsibility“
  - Jede Klasse hat (genau) einen, klar definierten Zweck
  - Jede Methode hat (genau) einen, klar definierten Zweck
- Orthogonalität
  - Minimale Wechselwirkung der Bestandteile
  - Freie Kombinierbarkeit (Lego®-artig)
- Konzeptionelle Klarheit
  - Einfach verständliches Interface
  - Offensichtliche Funktion, Annahmen, Invarianten
  - Wäre auch ohne Kommentare verständlich
    - Trotzdem gut dokumentiert

UML Diagramme

Entwurf von Software

Wie entwirft man Software?

- Diskussion am White-/Black-board
  - „Wandtafel“ in einem ruhigen Besprechungsraum oft wichtigstes Werkzeug
- Problem: Synchronisation der Vorstellungen
  - Jede(r) hat sein eigenes Gehirn
  - Bandbreite der Sprache gering
  - Vorstellungen liegen oft
    - dramatisch auseinander
    - unbemkert!
  - Gemeinsame Sprache nötig

Werkzeug: Diagramme

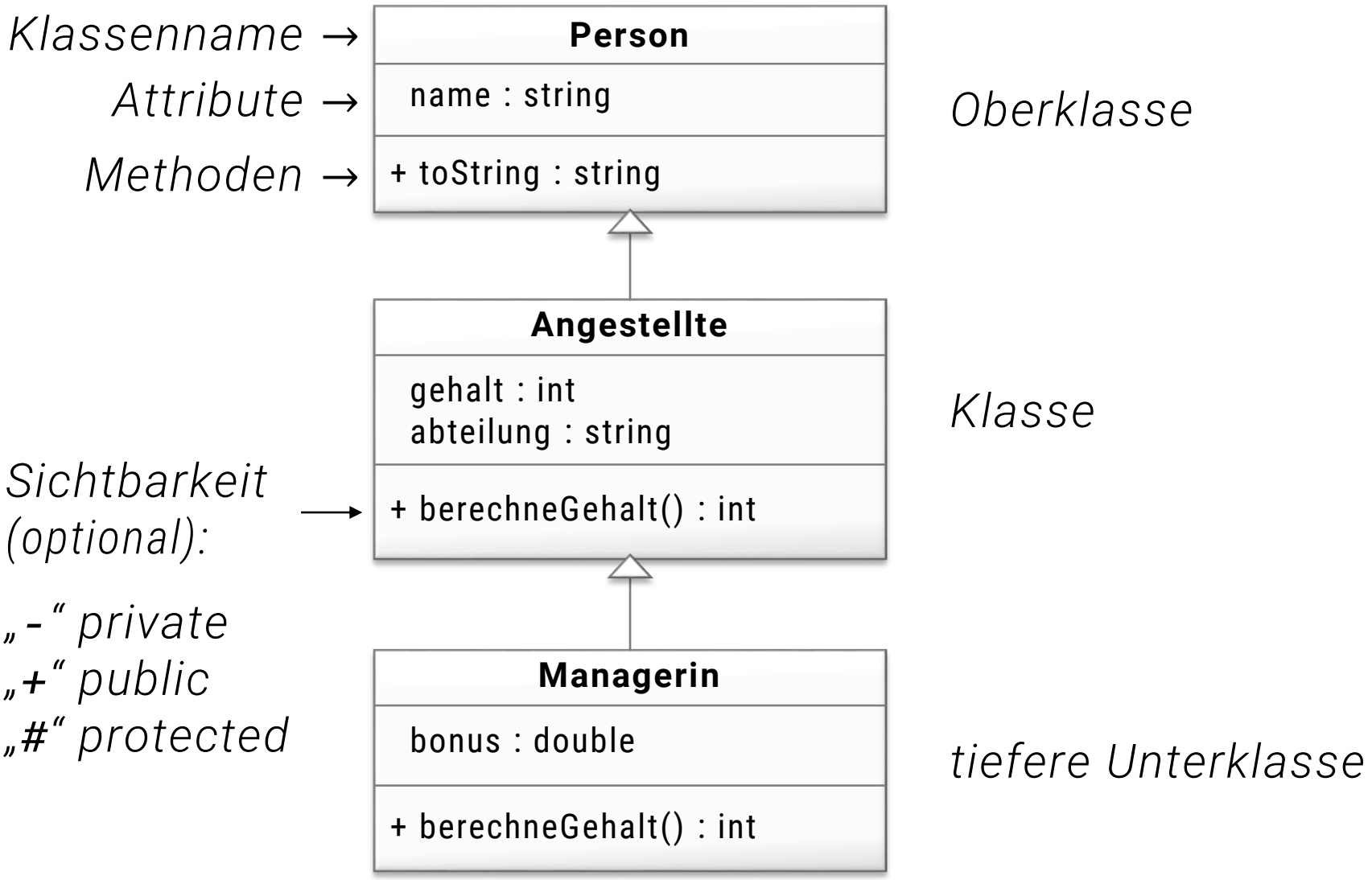
Hilfsmittel: Diagramme

- Visualisierung der *Struktur* und/oder des *Verhaltens* der Software
- Genormte Diagrammsprachen erleichtern klare Kommunikation

UML: „Unified Modeling Language“

- Definition einer Vielzahl von Diagrammtypen
- Recht genaue Definition der Semantik
- Fokus auf objekt-orientierte Entwürfe

Struktur: Klassendiagramme

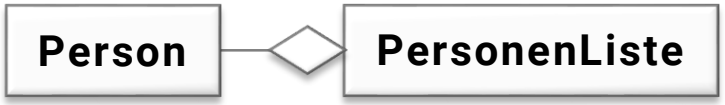


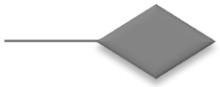
Beziehungen zwischen Klassen

Vererbung
(Pfeil zeigt auf Oberklasse)

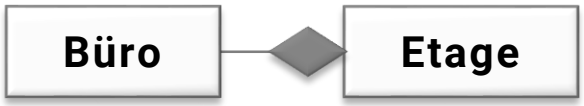
Assoziation
(Pfeil zeigt auf Klasse,
die referenziert wird)

Aggregation
(Raute an Klasse, die
das andere Objekt enhält)



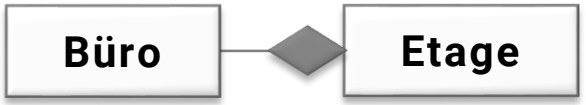


Komposition
(ausgefüllte Raute an Klasse,
die das andere Objekt enhält)



Unterschiedliche Zusammsetzung

Verschiedene Komposition



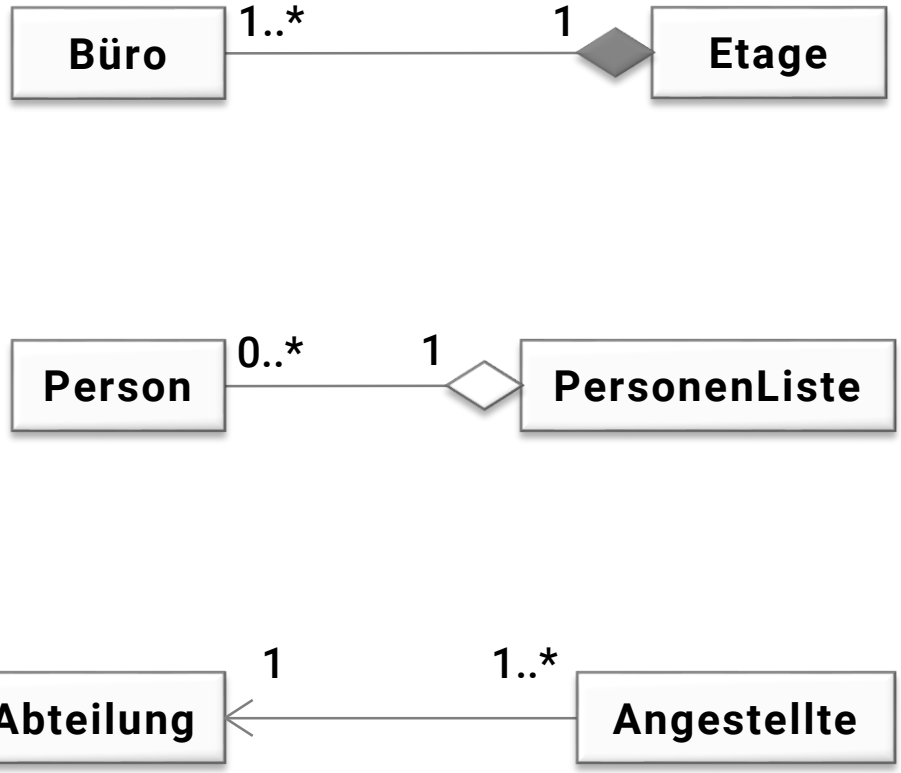
- „Komposition“
  - Klasse besteht aus Objekten anderer Klassen
  - Die Bestandteile müssen immer vorhanden sein

- „Aggregation“
  - Objekte enthalten Objekte anderer Klassen
  - Bestandteile brauchen nicht unbedingt vorhanden zu sein

- „Assoziation“
  - Objekte verweist auf Objekte anderer Klassen
  - Kein Bestandteil, nur eine Referenz

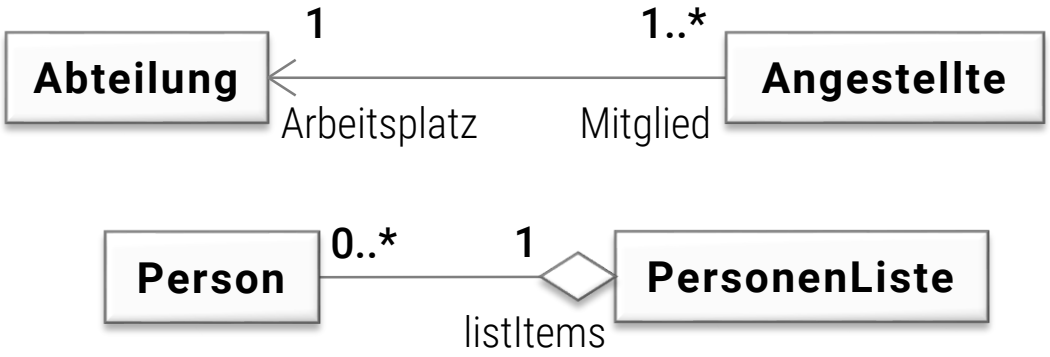
Multiplizitäten

Notation	
0	Keine Instanzen *(wenig Sinn)*
0..1	Keine oder eine Instanz
1 1..1	Genau eine Instanz
0..* *	Null oder mehr Instanzen
1..*	Eine oder mehr Instanzen



Noch zwei Details

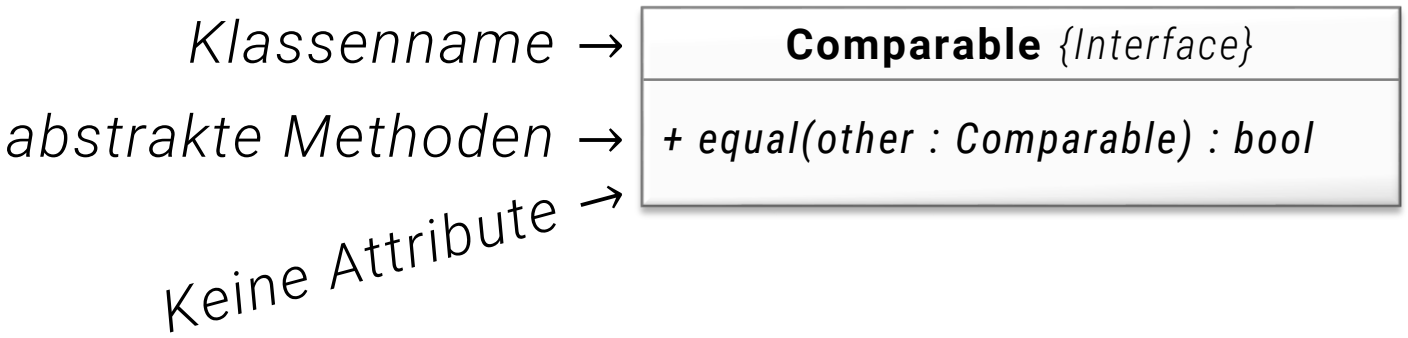
Rollenbezeichnungen
(optional)



Noch zwei Details

Interfaces

(C++: Klassen nur mit abstrakten Methoden)



Praxisbeispiel

Beispiel: GeoX Reflection System