Systemintegrator Workshop | 2024.05

BECKHOFF

7. Mai 2024 | Besonderheiten

BECKHOFF

Die USA feieren den **National Teacher Day**, den nationalen Tag der amerikanischen Lehrer.

In USA der nationalen **Tag des Cosmopolitans**

In USA Tag der **Kissenpoesie**

In USA **Tag des Verpackungsdesigns**

Zu Ehren des Physikers Alexander Stepanowitsch Popow feiert Russland **Tag des Radios** bzw. **Tag der Kommunikationsarbeiter**.

Internationaler UNESCO-Tag **der Toleranz**

Welt-Asthma-Tag

Agenda

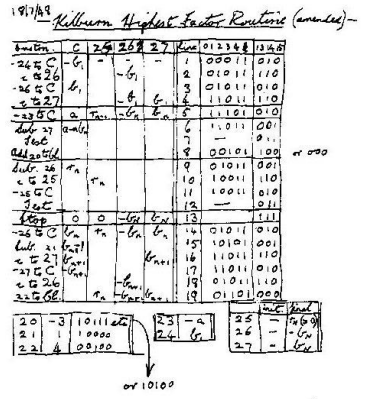
BECKHOFF

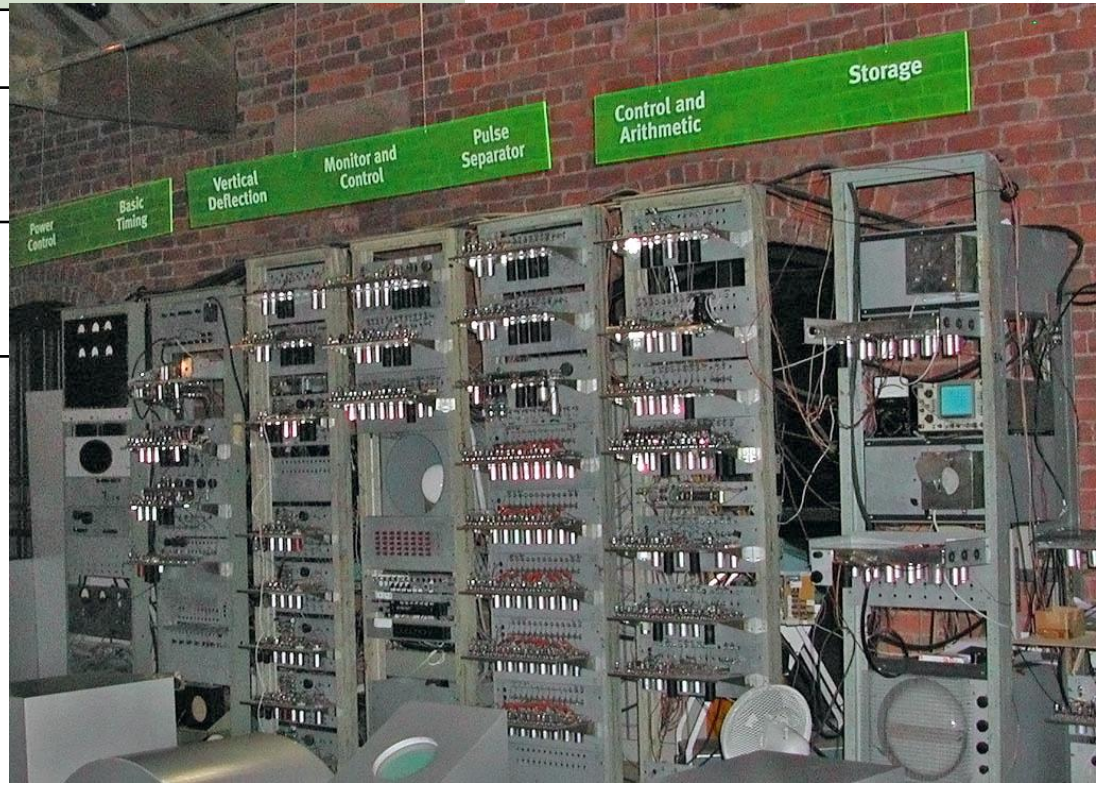
9:00 - 09:15	Begrüßung
9:15 - 09:30	Einführung
9:30 - 10:30	Programmiersprachen / Technische Schulden / Kognitive Psychologie
10:30 - 11:00	Kaffeepause 1
11:00 - 12:30	Zyklenfreiheit / Architekturstile / Programmcode besser verstehen
12:30 - 13:30	Mittag
13:30 - 15:00	Praktische Beispiele
15:00 – 15:30	Kaffeepause 2
15:30 – ca. 17:15	Praktische Beispiele
Ca. 17:15 - 17:30	Verabschiedung

Erstes Softwareprogramm

BECKHOFF

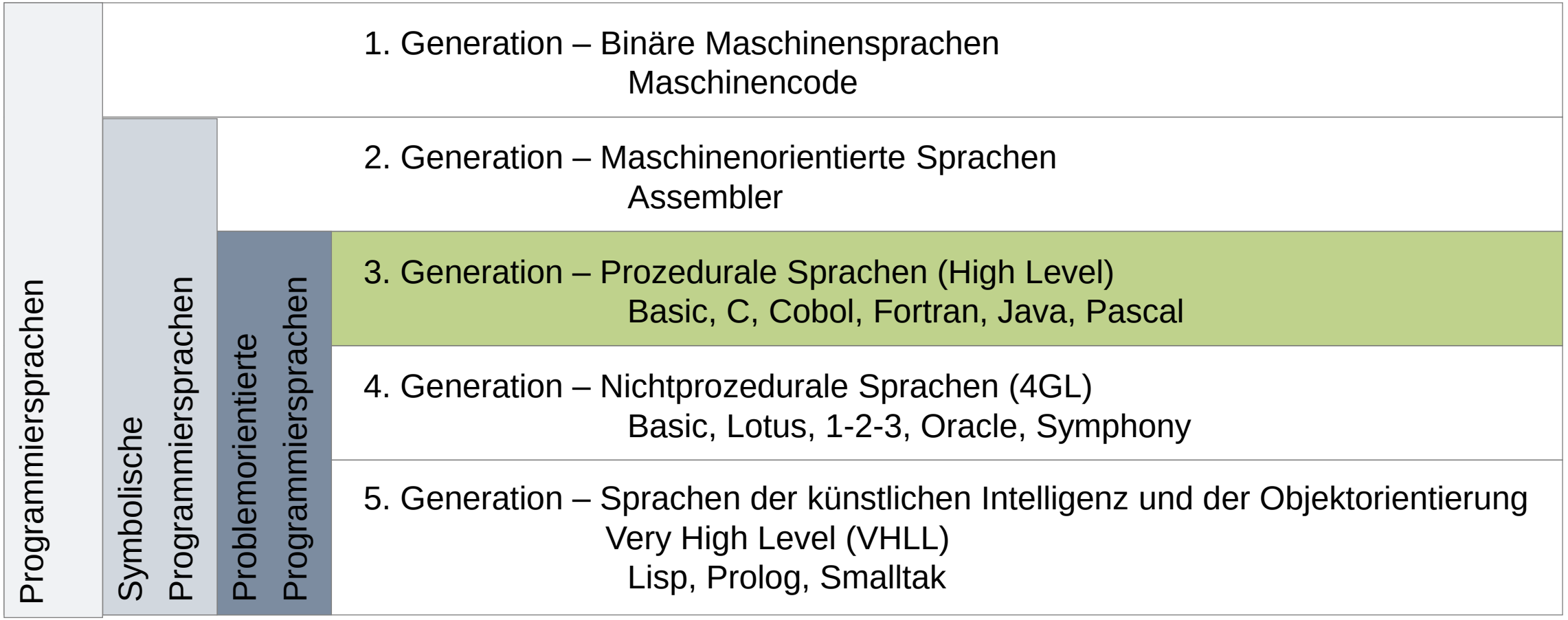
Ziel	Speicherfunktionen des Computers zu testen
Funktion	eine Zahl zu verdoppeln
Wo	Institute for Advanced Study in Princeton, New Jersey, USA
Computer	"Manchester Baby„ (Small-Scale Experimental Machine, SSEM)
Datum	21. Juni 1948 um 11:44 Uhr
Laufzeit	52 Minuten
Anzahl der Befehle	17
Anzahl der Speicherzugriffe	3,5 Millionen
Programmiert von	Tom Kilburn und Freddie Williams





Entwicklung der Programmiersprachen

BECKHOFF



Entwicklung der Programmiersprachen II

BECKHOFF

1. Generation

Maschinensprachen

binär codiert (hexadezimale Darstellung)
Abarbeitung vom Prozessor direkt
für Menschen schwer verständlich

```
3E FF
76
```

2. Generation

Assemblersprachen

Benutzung mnemonische Abkürzungen für binär codierte Befehle
Übersetzung Assembler in Maschinensprache durch Software möglich
für Menschen schwer verständlich; Formeln o. ä. nicht erkennbar

```
LD      TRUE
ANDN    BOOL1
JMPC    marke
LDN     BOOL
ST      ERG
```

3. Generation

problemorientierte imperative Sprachen

Beschreibung der Problemlösung erfolgt algorithmisch
erste Vertreter: FORTRAN (formular translator) und ALGOL (algorithmic language)
typische (heutige) Vertreter: Java, C++, Object Pascal (Delphi), ...

```
WHILE counter >= 10  DO
  Var1 := Var1 * 2;
  counter := counter + 1;
END_WHILE
```

4. Generation

Sprachen zur Kommunikation mit Datenbanken

Besitzen Operationen zur Manipulation von Tabellen und Datenbanken
typischer Vertreter: SQL (structured query language)

```
SELECT *
FROM T_Recipe
WHERE T_Recipe.Name='Product1';
```

5. Generation

Sprachen, die eine deklarative Beschreibung von Problemen ermöglichen

Sprachen unterstützen Schleifenstruktur nicht mehr
Rekursion wird primär eingesetzt
Varianten:
logischer Ansatz z. B. mit PROLOG (programming in logic) oder
funktionaler Ansatz z. B. mit LISP (list processing) oder HASKELL

```
maennlich(paul).
maenlich(jens).
alter(jens,7)kind(X):-alter(X,Z),Z<14.
```

Lebensdauer von Software

BECKHOFF

- Kurzlebige Software
  - Keine Anforderungen an langlebige Architektur
  - z.B. Migration von Daten aus Altsystemen
  - Wegwerf Software wird auch als „Code Kata“ in der Clean-Code-Bewegung bezeichnet.
- Nicht erwartete Langlebige Software
  - Software die länger verwendet werden als gedacht
  - Z.B 2000-Problem: in den 1960/70er war Speicher teuer. Entwickler verwendeten 2-stellige Jahreszahlen.
- Heute wird von Softwaresystemen eine lange Lebensdauer erwartet
  - Investition muss sich rentieren
  - Geringe Wartungs- und Erweiterungskosten

Definition „technische Schulden“

BECKHOFF

Technische Schulden

Code repräsentiert immer das aktuelle Verständnis des durch die Software gelösten Problems. Wenn das Team Neues lernt, dann muss der Code durch Refactorings konsolidiert werden, sodass das Gelernte optimal repräsentiert ist. Wenn die Refactorings unterbleiben, entstehen technische Schulden und der Code ist schwieriger zu ändern.

Ward Cunningham 1992

Die vier Quadranten zur Einteilung technischer Schulden

BECKHOFF

	Rücksichtslos	Umsichtig
Bewust	2 „Wir haben keine Zeit für Design.“	1 „Wir müssen schnell liefern und kümmern uns später um die Konsequenzen.“
Versehendlich	4 „Was ist eine Schichtenarchitektur?“	3 „Jetzt wissen wir, was wir hätten tun sollen.“

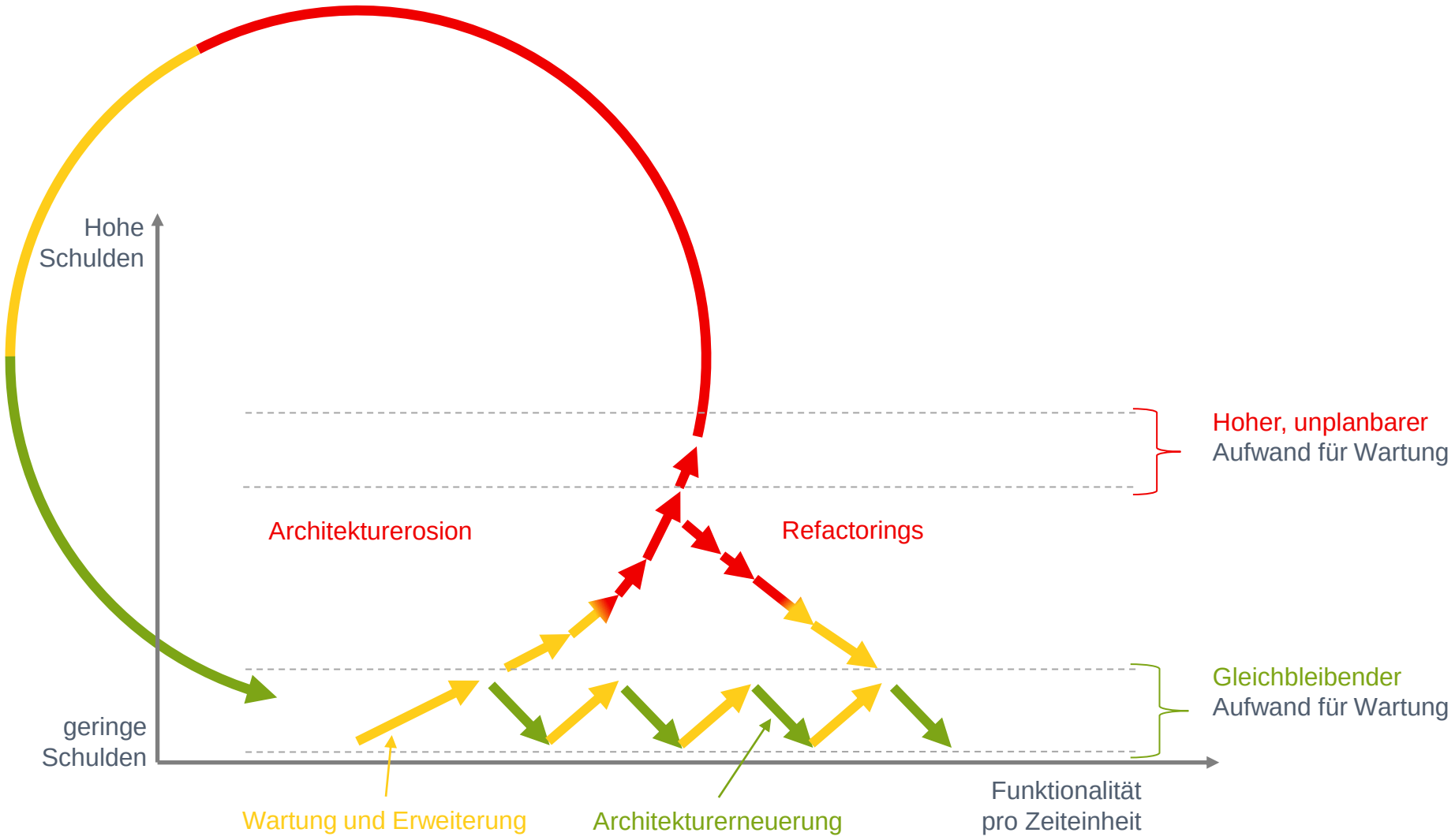
Ursachen von technischen Schulden

BECKHOFF

- Das Phänomen „Programmieren kann jeder“
  - Programmieren ≠ Softwarearchitektur
  - Hohe Warscheinlichkeit für unbrauchbare Software
- Das Unverständnis des Managments und der Kunden für Individualsoftwareentwicklung
  - Architektur kostet Extrageld
  - Software ≠ industriell herstellbares Produkt
- Die Architekturerosion steigt unbemerkt
- Komplexität und Größe von Softwaresystemen

Architekturerosion

BECKHOFF



Komplexität und Größe

BECKHOFF

- Komplexität eines Softwaresystem ist der Zusammenhang von
  - Probleminhärente Komplexität
  - Lösungsabhängige Komplexität

	Essenziell = unvermeidlich	**Akzidentell = überflüssig**
Probleminhärent	• Komplexität der Fachdomäne	• Missverständnisse über die Fachdomäne
Lösungsabhängig	• Komplexität der Technologie und der Architektur	• Missverständnisse über die Technologie • Überflüssige Lösungsanteile

- Große Softwaresysteme sind komplex
- Ist die Lösung Komplexer als das Problem, ist ein Redesign erforderlich
- Architektur reduziert Komplexität

Arten von technischen Schulden

BECKHOFF

- **Implementationsschulden**
  Code-Smells:
  Lange Methoden, Codeduplikate, usw.

- **Desgin- und Architekturschulden**
  Struktur-Smells:
  Design der Klassen, Packete, Module passen nicht zur geplanten Architektur

- **Testschulden**
  Es fehlen Tests bzw. nur der Gut-Fall wird getestet

- **Dokumentationsschulden**
  Es gibt keine, wenig oder veraltetet Dokumentation

- **Anfänge**
  - Erstmalige Erwähung „Softwarearchitektur“ bei einer NATO Konferenz 1969 in Rom
  - In den 1970er / 1980er Jahren wurde „Architektur“ für Systemarchitekturen (Rechner , Server) verwendet
- **Etabliert**
  - Ab den 1990er Jahren wurde die Softwarearchitektur ein Teilgebiet der Softwaretechnik
  - Ab 1995 im industriellen Umfeld zunehmend mehr Bedeutung
  - 1996 erschien das Buch „Pattern-oriented Software Achitecture. Beschreibt Entwurfsmuster zur besseren Wiederwendbarkeit von Software
- **Aktuell**
  - Im Jahre 2000 erschien die *IEEE 1471:2000* Norm „Recommended Practice for Architectural Description of Software-Intensive Systems zur Architekturbeschreibung von Softwaresystemen “

Definition 1

„Softwarearchitektur ist die Struktur eines Software-Produkts. Diese umfasst Elemente, die extren wahrnehmbaren Eigenschaften der Elemente und die Beziehungen zwischen den Elementen

Len Bass, Paul Clements, Rick Kazman: Software Architecture in Practice. Addison-Wesley Professional 2012

Definition 2

„Softwarearchitektur = Σ aller wichtigen Entscheidungen
Wichtige Entscheidungen sind alle Entscheidungen, die im Verlauf der weiteren Entwicklung nur schwer zu ändern sind

Philippe Kruchten: The Rational Unified Process, Third Edition. Addison-Wesley Professional 2004

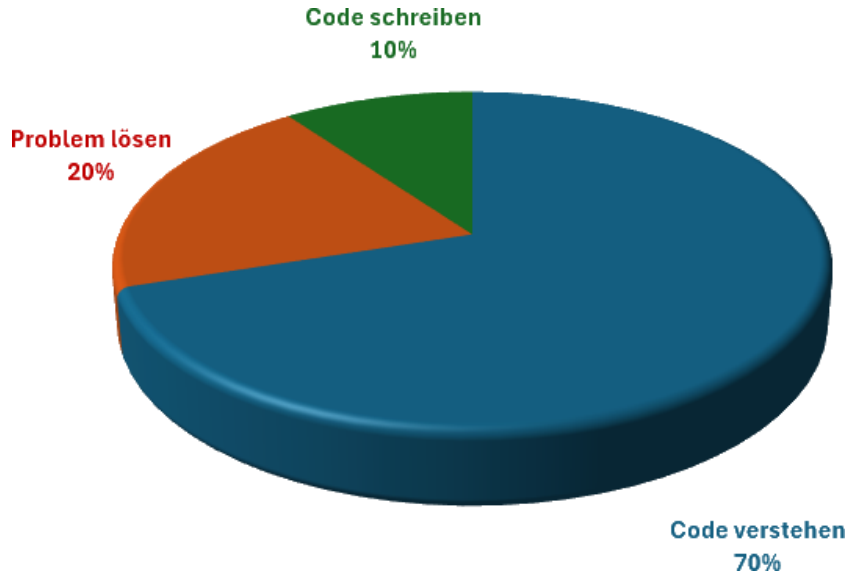
Entscheidung schafft Leitplanken

Schaffen Sie eine Architektur, die den Designraum bei der Entwicklung des Softwaresystems einschränkt und Ihnen dadurch bei Ihrer Arbeit die Richtung weist.

Verhältnis Code lesen / schreiben

BECKHOFF

- Großteil der Zeit wird zum Lesen und Verstehen von Code verwendet
- Analysierbarkeit
- Problemlösung in komplexen Zusammenhängen



Frage:

Mit welchen Strukturen können Softwareentwickler ihre Arbeit effizient erledigen?

Antwort:

Die kognitive Psyhologie

Kognitive Psychologie und Architekturprinzipien

BECKHOFF

- Wissenschaft der menschlichen Informationsverarbeitung
- Die kognitive Psychologie befasst sich unteranderem mit
  - dem Verstehen
  - dem Wissenserwerb
  - der Problemlösung
- Es gibt 3 strukturbildende Prozesse, die für die Softwareentwicklung interessant sind
  - **Chunking**
    Zusammenfassen von kleinen zu größeren Wissenseinheiten
  - **Aufbau von Schemata**
    Bildung von allgemeinen Kategorien
  - **Bildung von Hierarchien**
    Hierarchische Struktur aus Ober- und Unterelementen

Bewusstseinsvorstellung aus dem 17. Jahrhundert von Robert Fudd

Chunking

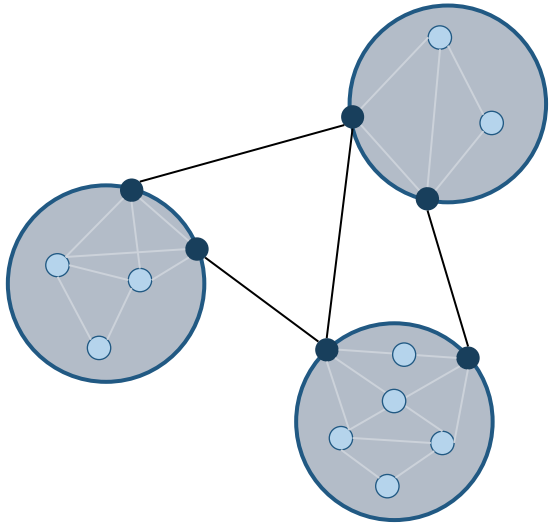
BECKHOFF

- Chunks sind Wissenseinheiten
- Kleiner Einzeleinheiten zu größeren verbinden
  - Im Kurzzeitgedächtnis werden ca. 7 (+/-2) Einheiten verarbeitet
- „Recording“ verdichtet kleiner Wissenseinheiten zu größeren
- Unbekannte Programme werden in **Bottom-up** Verfahren erfasst
  - Programmtext im Detail gelesen
  - kleinen Einheiten werden zu immer größeren Einheiten zusammengefasst
- Bekannte Programme werden eher in **Top-down** Verfahren gelesen
  - Verwendet werden Hierarchien und Schemata
- Experten können deutlich höhere Anzahl an Wissenseinheiten verdichten
  - Es müssen jedoch Sinnvolle zusammenhänge vorhanden sein

Chunking und Modularität

BECKHOFF

- Regeln für eine modulare Architektur
  - **Einheit** als Chunk
    In Ihrem Inneren ein zusammenhängendes, kohärentes Ganzes bilden, das für genau eine Aufgabe zuständig ist
  - **Schnittstelle** als Chunk
    Nach Außen eine explizite, minimale und delegierende Kapsel bilden
  - **Kopplung** zur Chunk Trennung
    Mit anderen Bausteinen minimal und lose gekoppelt sein

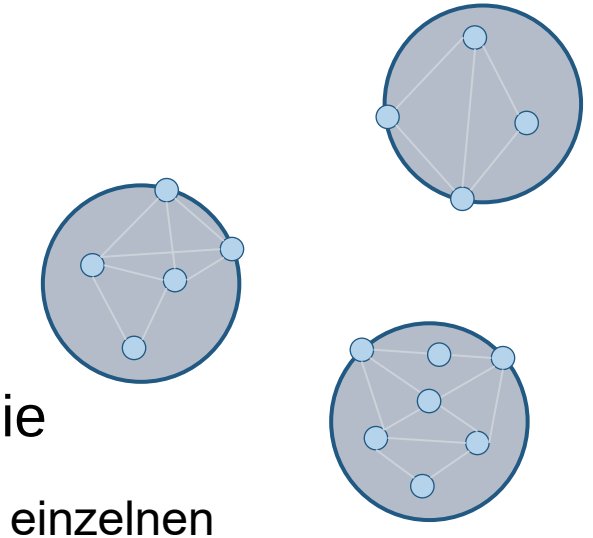


Modularität | Einheit

BECKHOFF

- Sinnvoll zerlegen
  - Separation of Concerns
- Zusammenhalt
  - Kohäsion
- Zuständigkeit
  - Responsibility Driven Design
- Verantwortung für eine Aufgabe
  - Single Responsibility Principle (SRP)
- Lokale Änderung
  - DRY (Don‘t repeat yourself) und SPOT (Single point of truth)
    IEEE-Standard-Glossar für Software-Engineering-Terminologie

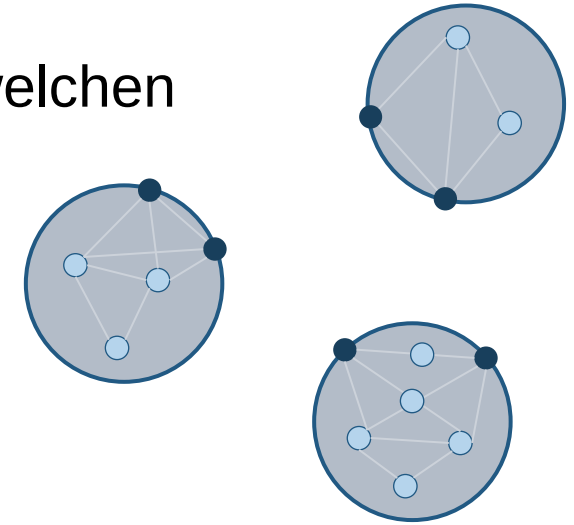
    „Modularität ist der Grad, in dem ein System oder Computerprogramm aus einzelnen Komponenten besteht, so dass eine Änderung an einer Komponente nur minimale Auswirkungen auf andere Komponenten hat.“



Modularität | Schnittstellen

BECKHOFF

- Separate Schnittstellen
  - Trennung von Schnittstelle und Implementierungen
- Minimale Dienste
  - Interface Segregation Principle (ISP). Enthalten im „SOLID“ Prinzip
  - Möglichst minimales Set an Funktionalitäten
- Injizierte Abhängigkeiten
  - Explizite Abhängigkeiten. An der Schnittstelle erkennbar mit welchen anderen Bausteinen kommuniziert werden kann
- Abgeschlossene Dienste
  - Delegierende Schnittstellen und das Low of Demeter
  - Schnittstellen die Interna nach Außen befördern, verstoßen gegen das Prinzip

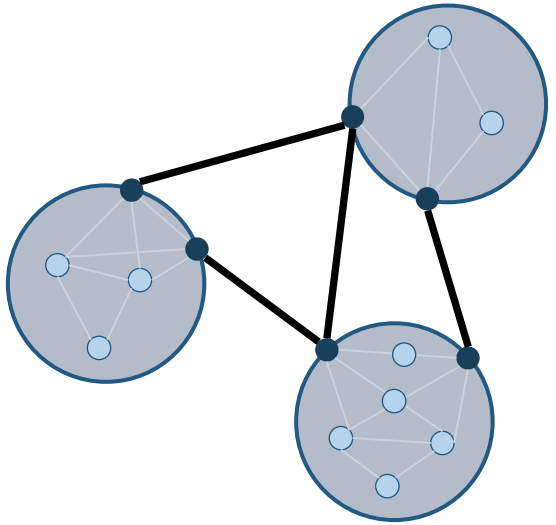


Werden alle diese Grundprinzipien beachtet, können Wissenseinheiten schneller verarbeitet werden

BECKHOFF

- Lose Kopplung
  - Bezeichnet den Grad der Abhängigkeiten zwischen den Bausteinen

Je höher der Grad, umso schwieriger wird es, einzelne Beteiligte Bausteine mit der begrenzten Kapazität des Kurzzeitgedächtnisses zu analysieren!



Schemata

BECKHOFF

- Menschen können komplexe Zusammenhänge mit Schemas strukturieren
- Wissenseinheiten werden hier in Kombination von abstraktem und konkreten Wissen verstanden

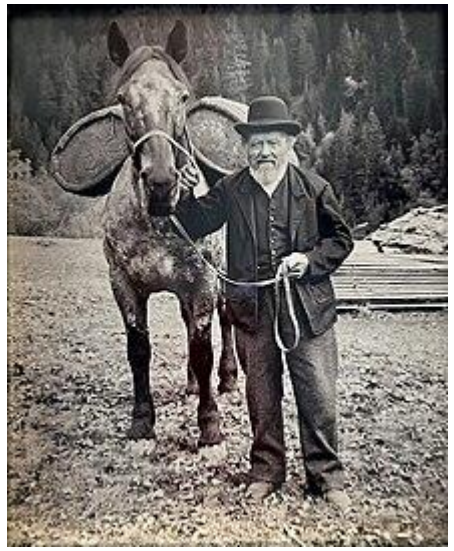
- Beispiel: Schema „Lehrer“
  - Arbeitet an einer Schule
  - …

Schemata

BECKHOFF

- Beispiel: Schema: „Säumer“
  - Was ist ein Säumer ?

Ein historischer Beruf aus dem 18. Jahrhundert
Säumer haben Waren an den Berghängen der Alpen, den „Säumen“
Entlang mit Ihren Eseln von Deutschland nach Italien und zurück transportiert



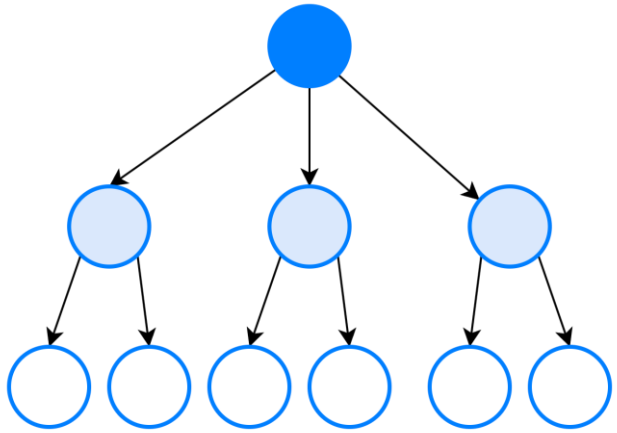
- Wurde das Entwurfsmuster schon mehrfach verwendet, so können Programme und Strukturen schneller erkannt und verstanden werden
- Entwurfsmuster werden nicht erfunden sondern entdeckt und aus Erfahrung gebildet
- Änderung von Schemas ist immer schwierig
- Schemata und Chunking wird von Menschen häufig in Kombination eingesetzt
- In der Softwarearchitektur sind Entwurfsmuster mit Schemas gleichzusetzen

Hierarchisierung

BECKHOFF

- Wissen kann in hierarchischen Strukturen am einfachsten
  - Aufgenommen werden
  - Wiedergeben werden
  - Sich zurechtfinden
- Menschen verwenden Hierarchien und Chunking in Kombination zur Wissensverwaltung im Gedächtnis

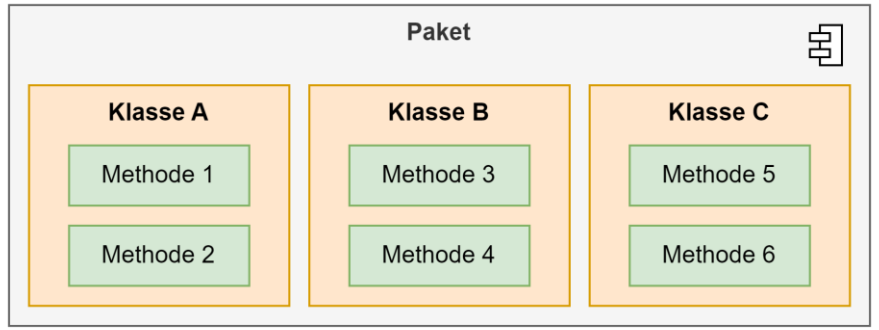
- Hierarchien lassen Top-down Programmverstehen zu
- Ein effizientes Arbeiten am Softwaresystem möglich



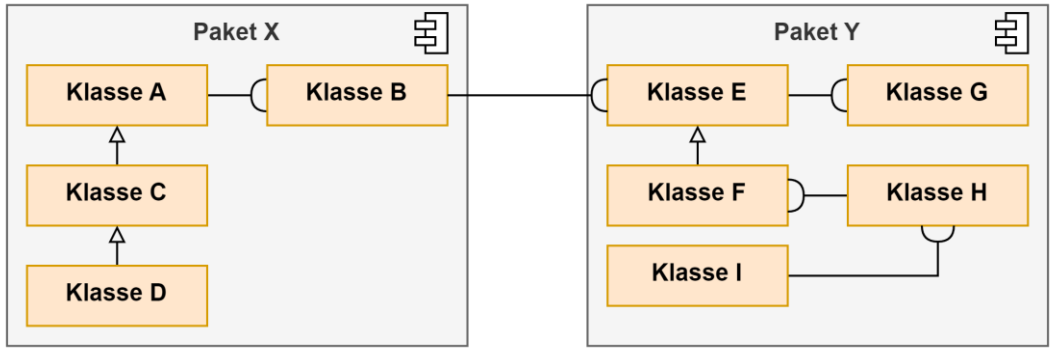
Hierarchisierung | Enthalten, Vererben, Benutzen

BECKHOFF

- Hierarchien werden bei Programmiersprachen bei den „Enthalten-Sein-Beziehungen“ verwendet
  - Methoden
    - Methoden sind in Klassen enthalten
    - Klassen in Paketen (Libraries) enthalten



  - Bei allen anderen Beziehungen gilt das nicht
    - Beliebige Klassen und Interfaces verwenden weitere Klassen.
    - Durch „Enthalten – Vererbung – Benutzen“ entstehen komplexe Beziehungen
    - Sogenannte **Zyklenfreiheit**



Kaffeepause 1 | 10:30 – 11:00

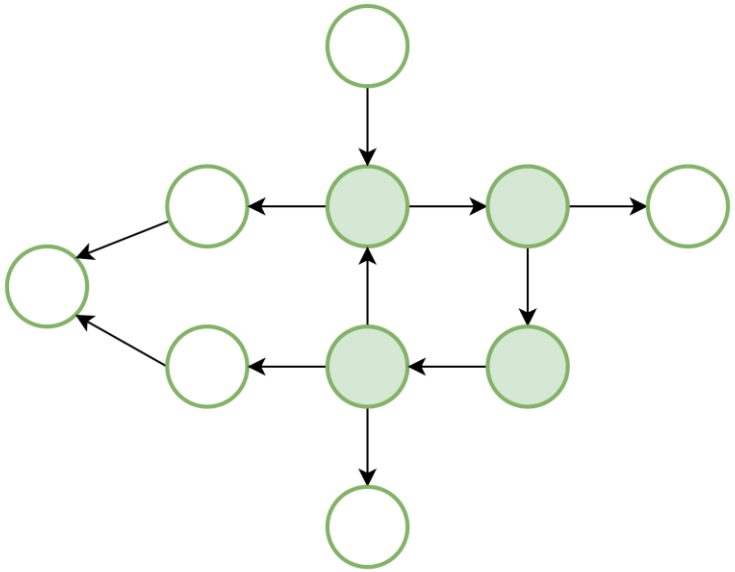
BECKHOFF

Zyklen, Zyklengruppen

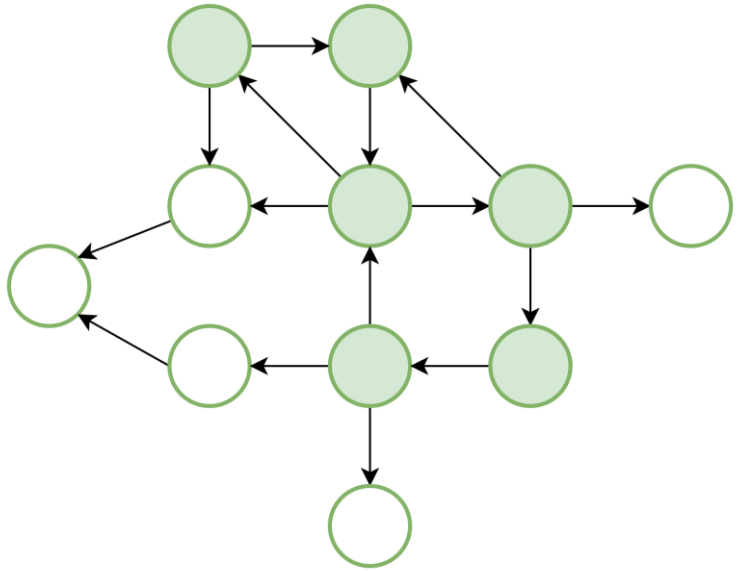
BECKHOFF

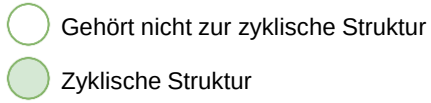
- Eine Architektur ist Zyklen frei, wenn von einem Baustein über Beziehungen zu anderen Bausteinen nicht zu sich selber zurück kehrt
  - Solche Strukturen werden auch **gerichtete azyklische Graphen** genannt

Zyklus mit
vier Knoten



Zyklusgruppe mit
drei einzelnen Zyklen





Argumente gegen Zyklen I

BECKHOFF

- Warum sind geordnete Strukturen den Zyklen vorzuziehen
  - Erweiterbarkeit
    Zyklische Strukturen sind schwer zu erweitern. Änderungen an einem Baustein ziehen Änderungen an den Verknüpften Baustein nach sich
  - Testbarkeit
    Zyklisch verknüpfte Bausteine können nicht isoliert werden. So ein isolierter Test für den einen Baustein nicht möglich ist
  - Evolution
    Im Verhältnis ändern sich Bausteine in Zyklen häufiger und sind fehleranfälliger. Zyklische Strukturen breiten sich aus, weil Erweiterungen neue Klassen an die vorhandenen Zyklen angebaut werden
  - Deployment
    Sollen einzelne Komponenten ausgeliefert werden, so dürfen keine zyklischen Abhängigkeiten vorhanden sein

Argumente gegen Zyklen II

BECKHOFF

- Warum sind geordnete Strukturen den Zyklen vorzuziehen
  - Verständlichkeit
    Bausteine im Zyklus lassen sich nur als Ganzes betrachten.
    Häufig spielen an Zyklen beteiligte Bausteine mehrere Rollen
    Daraus entstehen weitere Abhängigkeiten. Das System wird komplexer und schwer überschaubar und damit schwer verständlich

Regeln für eine zyklenfreie Architektur

Die Beziehungen zwischen den Bausteinen einer zyklenfreien Architektur müssen auf allen Ebenen ohne Rückbeziehungen auskommen

Konsequenzen für die Architektur

BECKHOFF

- Damit der Prozess des Programmverstehens und –änderns möglichst schnell und fehlerfrei durchgeführt werden kann, sollten Softwaresysteme

  - **Modularität**
    Sinnvoll zusammenhängende Module auf verschiedenen Abstraktionslevels, die das Chunking erleichtern

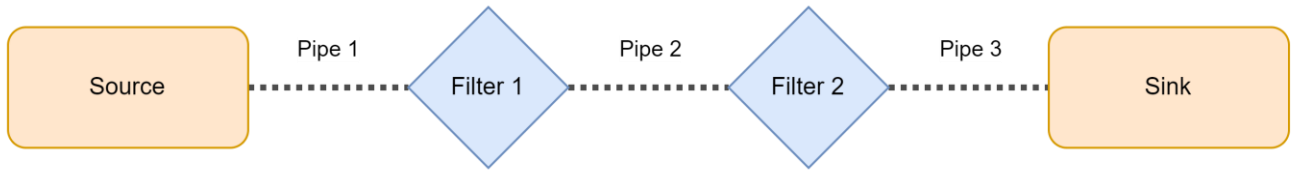
  - **Zyklenfreiheit**
    Zyklen frei aufgebaut sein, um die Bildung von Hierarchien zu unterstützen

  - **Musterkonsistenz**
    Auf den verschiedenen Abstraktionslevels wiederkehrende Muster anbieten, aus denen Schemata gebildet werden können

- Regeln von Architekturstilen
  - Welche Arten von Elementen hat der Architekturstil?
  - Welche Regeln gelten für diesen Stil?
    - **Elementregeln**
      Betreffen nur eine Elementart. Legen die Schnittstelle einer Elementart fest oder spezifizieren, aus welchen Bausteinen sie aufgebaut ist
    - **Gebotsregeln**
      Schreiben Beziehungen zwischen bestimmten Elementarten vor
    - **Verbotsregeln**
      Verbieten die Beziehungen zwischen bestimmten Elementarten

- Pipes & Filter
  Enthält 2 Elementarten:
  - Filter als ausführende Elemente
  - Pipes als Verbindungsströme

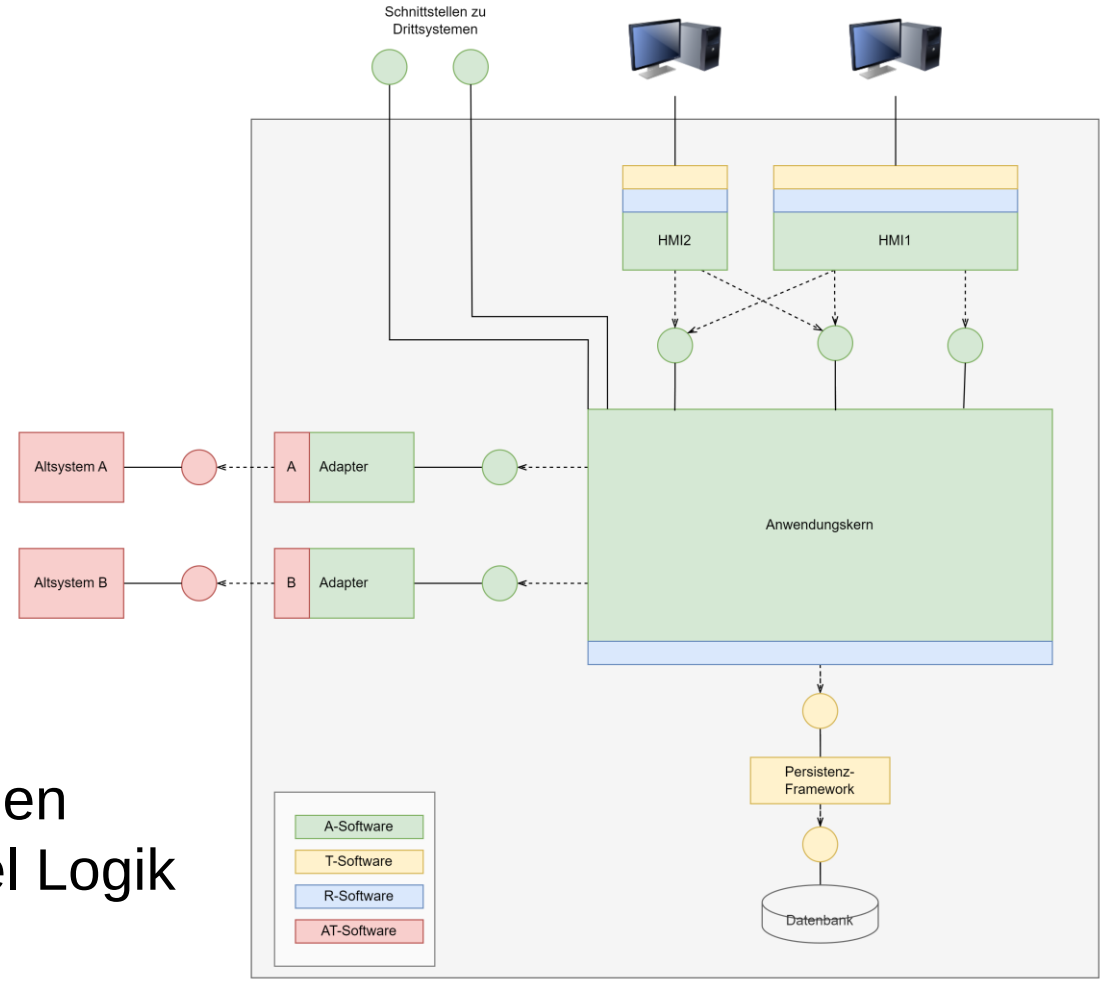
  Eine wichtige Regel dieses Stiles ist, das Pipes nur mit Filtern verknüpft werden können



Architekturstil | Trennung von fachlichen und technischen Bausteinen

BECKHOFF

- Quasar – System (Blutgruppen für Software)
  - Blutgruppe **A**
    Fachliche Softwarebausteine
  - Blutgruppe **T**
    Technische Softwarebausteine
  - Blutgruppe **AT**
    Mischung von A und T.
    Sollten vermieden werden.
    Im Notfall abgrenzen vom Rest
  - Blutgruppe **R**
    Transformation zwischen A und T Bausteinen
    Sind in der Regel klein und haben nicht viel Logik



Schichtenarchitekturen

BECKHOFF

- Eine der am häufigsten verwendeten Architektur Stile
- Jede Schicht kann als Baustein gesehen werden

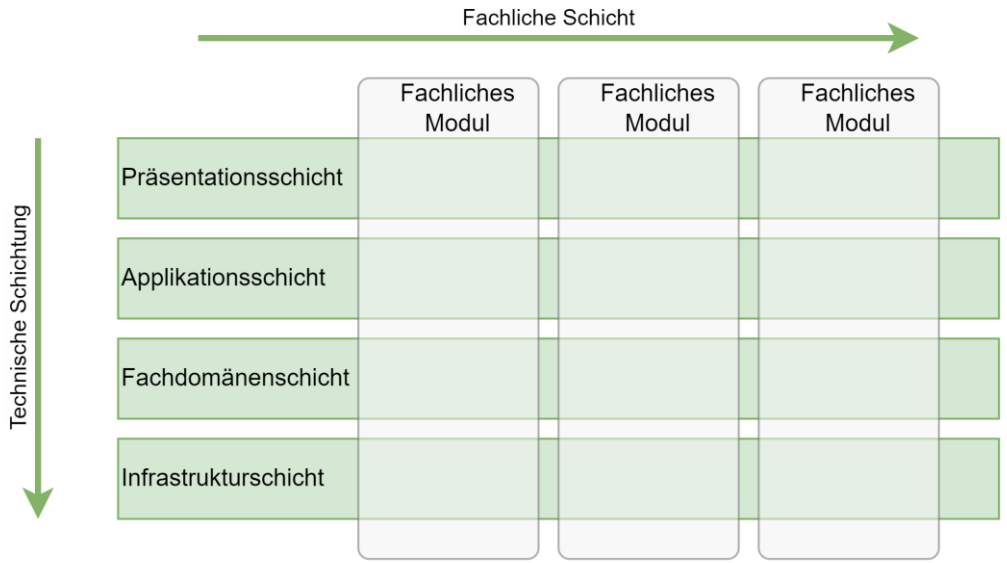
- Architekturregel besagt:
  Tiefere Schichten dürfen nicht auf höhere Schichten zugreifen

- Schichtarten
  - Technische Schichtung
  - Fachliche Schichtung
  - Infrastrukturschicht

Schichtenarchitekturen | Technische Schichtung

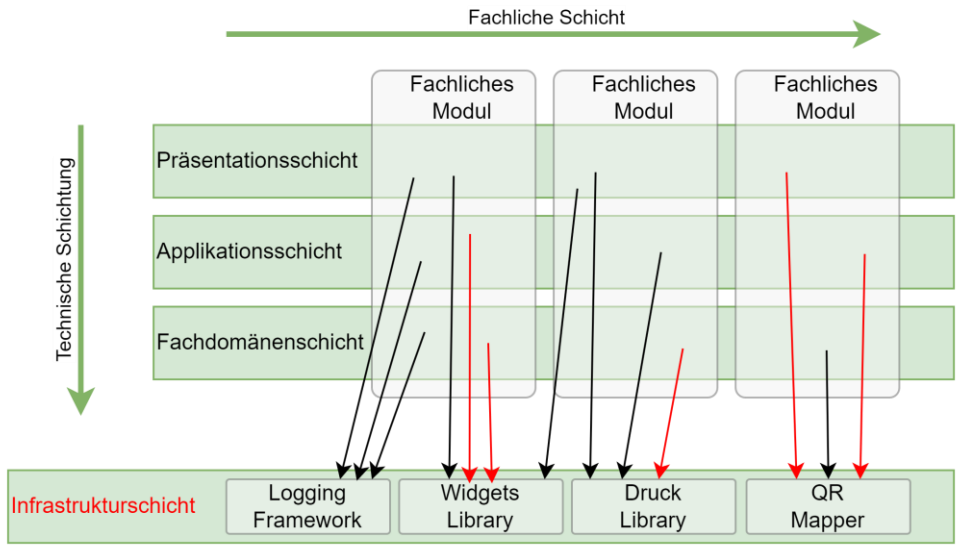
BECKHOFF

BECKHOFF

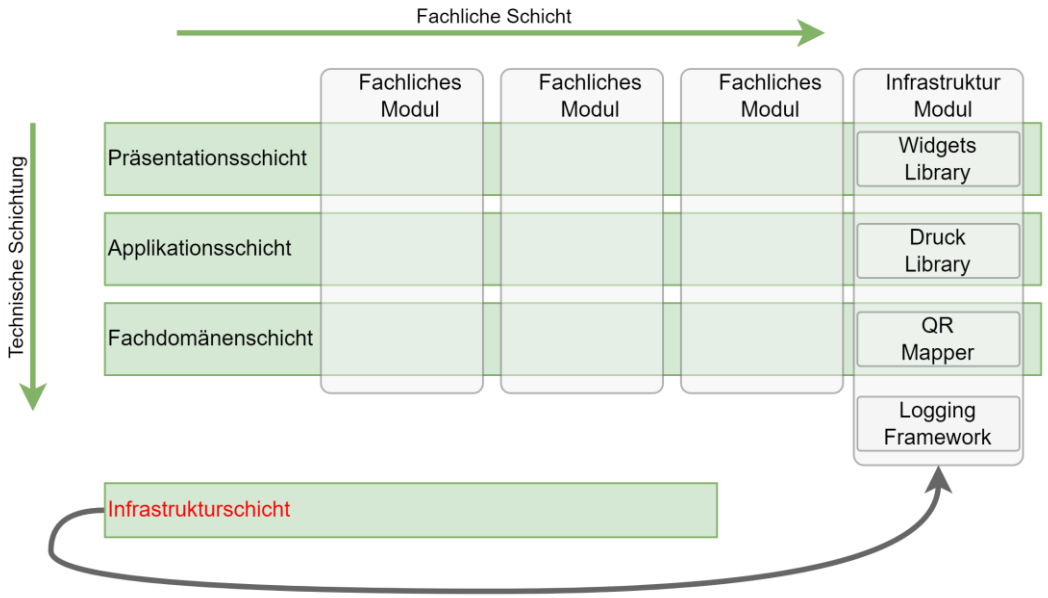

Fachliche Schicht
Technische Schichtung
Fachliches Modul
Fachliches Modul
Fachliches Modul
Präsentationsschicht
Applikationsschicht
Fachdomänenschicht
Infrastrukturschicht

BECKHOFF

Probleme der Infrastruktur



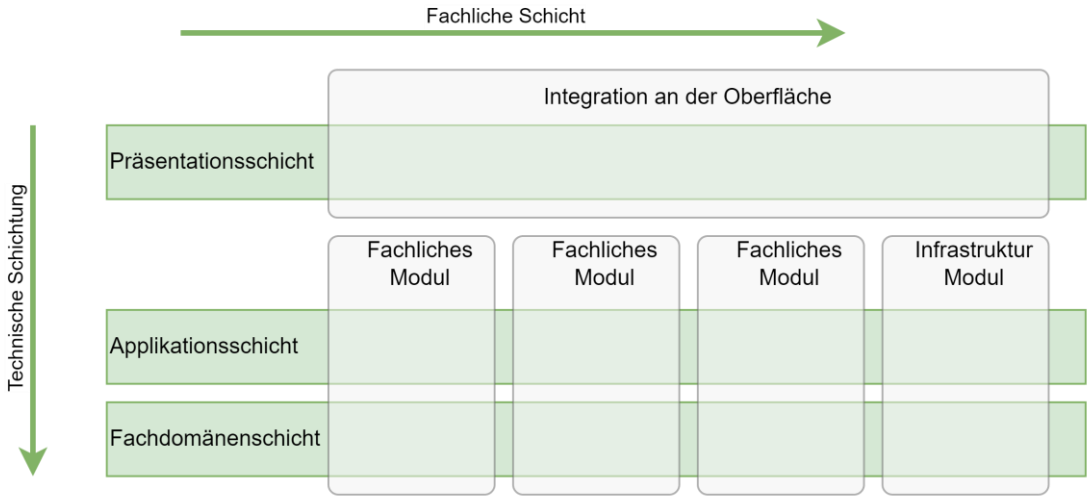
Sinnvolle Anordnung der Infrastrukturschicht



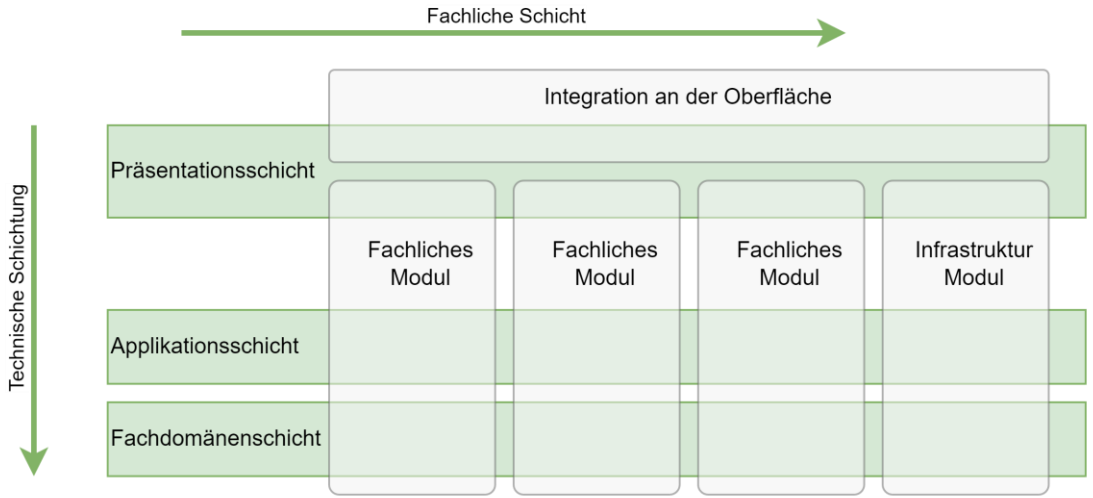
Schichtenarchitekturen | Integration von fachlichen Schichten

BECKHOFF

Integration der fachlichen Module In der Oberfläche

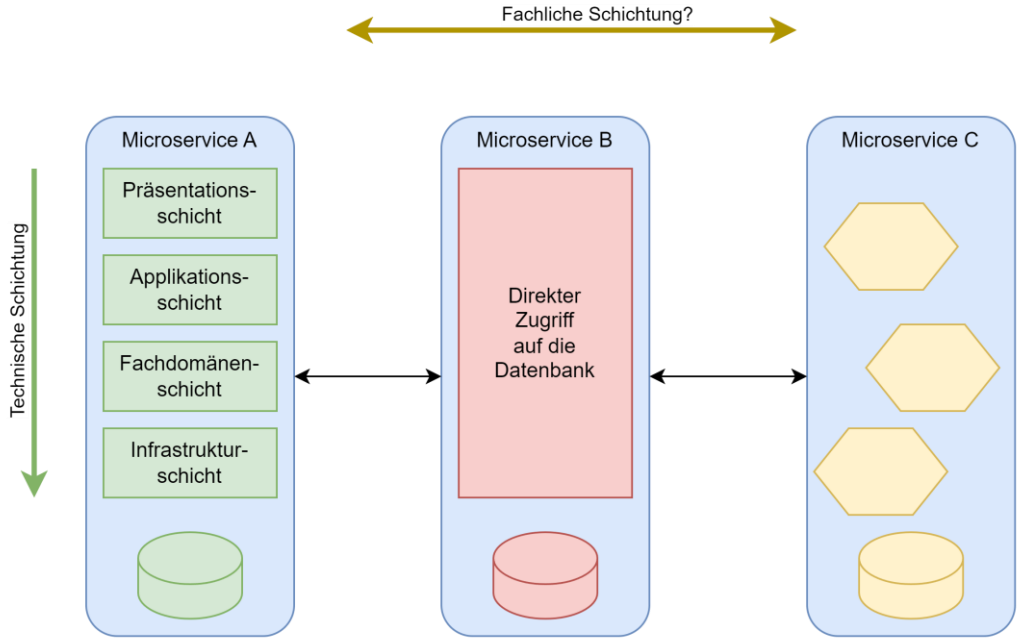


Schmale Integration von fachlichen Modulen In der Oberfläche



BECKHOFF

- Um das Chaos an Verbindungen zwischen den Services konzeptionell zu erfassen, wird das strategische Design von Domain-Driven-Design DDD vorgeschlagen
  - Shared Kernel
  - Customer / Supplier
  - Published Language
  - Open-Host-Service
  - Anticorruption Layer
  - Conformist
  - Separate Ways



Kommunikation:
Die Logik muss in den Services implementiert werden

Code der verstanden wird | Namen

BECKHOFF

- Die Pfadfinder-Regel
  - Der Code muss sauber gehalten werden
    „Hinterlasse den Code sauberer als du Ihn vorgefunden hast“

- Aussagekräftige Namen
  - Namen wählen, die den Zweck ausdrücken

```
d : INT; // abgelaufe Zeit in Tagen
```

```
elapsedTimeInDays : INT;
```

- Aussprechbare Namen verwenden

```
Class DtaRcrd102 {
   private Date genymdhms;
   private String pszqint = „102“;

 }
```

```
Class Customer {
   private Date generationTimestamp;
   private String recordId = „102“;

 }
```

Code der verstanden wird | Codierung

BECKHOFF

- Codierungen vermeiden
  - Keine Verschlüsselungssprache verwenden

- Ungarische Notation
  - Der Datentyp wir als Präfix verwendet.
    Ist das erforderlich?

```
fGrenzwertOben : REAL;
```

- Mentale Mappings vermeiden
  Unterschied zwischen einem schlauen Programmierer und einem professionellen Programmierer: Er weiß das **Klarheit hat absolut Vorrang.**
  Profis schreiben Code den andere verstehen

Code der verstanden wird | Funktionen

BECKHOFF

- Beschreibende Namen. Keine Angst vor längeren Namen
- Funktionsargumente so gering wie möglich halten
  Funktionen solltenb nur eine Funktionalität haben
- Keine Flag-Argumente verwenden

```
ReadFile(true);
```

- Bei einer größeren Anzahl Argumente, kann eine separate Klasse helfen

Code der verstanden wird | Kommentare

BECKHOFF

- „Kommentieren Sie schlechten Code nicht – schreiben Sie ihn neu“

  Brain W. Kernighan und P.J. Plaugher

- Kommentare sind Zeichen der Unfähigkeit. Wir brauchen Sie, da nicht immer klar ist „Wie soll ich es ausdrücken“
- Erklärung im und durch den Code
  Was ist besser:

```
// Prüfen. ob der Mitarbeiter alle Benefits bekommen soll
If ((employee.flag & HOURLY_FLAG) && ( employee.age > 65) )
```

  oder

```
If (employee.sollAlleBenefitsBekommen() )
```

- Je weniger Kommentare, desto besser

Code der verstanden wird | Formatierungen

BECKHOFF

– Vertikale Offenheit zwischen Konzepten

```
jsonDoc        := fbJson.ParseDocument(sMessage);
jsonIterator   := fbJson.MemberBegin(jsonDoc);
jsonIteratorEnd := fbJson.MemberEnd(jsonDoc);
IF bGetMetaData THEN
  WHILE jsonIterator <> jsonIteratorEnd DO
    sName        := fbJson.GetMemberName(jsonIterator);
    jsonValue    := fbJson.GetMemberValue(jsonIterator);
    IF sName = 'meta' THEN
      fbJson.CopyJson(jsonValue, sMetaData, SIZEOF(sMetaData));
      fbJsonParser.ParseValues(sMetaData, pDomParserMetaHandler);
      bGetMetaData := FALSE;
    END_IF
    jsonIterator  := fbJson.NextMember(jsonIterator);
    END_WHILE
  END_IF
```

```
jsonDoc        := fbJson.ParseDocument(sMessage);
jsonIterator   := fbJson.MemberBegin(jsonDoc);
jsonIteratorEnd := fbJson.MemberEnd(jsonDoc);

IF bGetMetaData THEN
  WHILE jsonIterator <> jsonIteratorEnd DO
    sName        := fbJson.GetMemberName(jsonIterator);
    jsonValue    := fbJson.GetMemberValue(jsonIterator);

    IF sName = 'meta' THEN
      fbJson.CopyJson(jsonValue, sMetaData, SIZEOF(sMetaData));
      fbJsonParser.ParseValues(sMetaData, pDomParserMetaHandler);
      bGetMetaData := FALSE;
      END_IF

    jsonIterator  := fbJson.NextMember(jsonIterator);
    END_WHILE
  END_IF
```

Code der verstanden wird | Formatierungen

BECKHOFF

– Vertikale Dichte

```
CASE data.TypeClass OF
 (* ---------------------------------------------------------------
    BYTE
 ---------------------------------------------------------------  *)
 __SYSTEM.TYPE_CLASS.TYPE_BYTE:
   _anyByte := data.pValue;
   AnyToString := BYTE_TO_STRING(_anyByte^);
 (* ---------------------------------------------------------------
    INT
 ---------------------------------------------------------------  *)
 __SYSTEM.TYPE_CLASS.TYPE_INT:
   _anyInt := data.pValue;
   AnyToString := INT_TO_STRING(_anyInt^);
 (* ---------------------------------------------------------------
    UINT
 ---------------------------------------------------------------  *)

 __SYSTEM.TYPE_CLASS.TYPE_UINT:
   _anyUInt := data.pValue;
   AnyToString := UINT_TO_STRING(_anyUInt^);

 …..
```

```
CASE data.TypeClass OF

__SYSTEM.TYPE_CLASS.TYPE_BYTE:
    _anyByte := data.pValue;
    AnyToString := BYTE_TO_STRING(_anyByte^);

__SYSTEM.TYPE_CLASS.TYPE_INT:
    _anyInt := data.pValue;
    AnyToString := INT_TO_STRING(_anyInt^);

__SYSTEM.TYPE_CLASS.TYPE_UINT:
    _anyUInt := data.pValue;
    AnyToString := UINT_TO_STRING(_anyUInt^);

  …..
```

Code der verstanden wird | Formatierregeln

BECKHOFF

- Jeder Programmierer hat seine Regeln. Wird im Team gearbeitet
  Hat das **Team Vorrang**

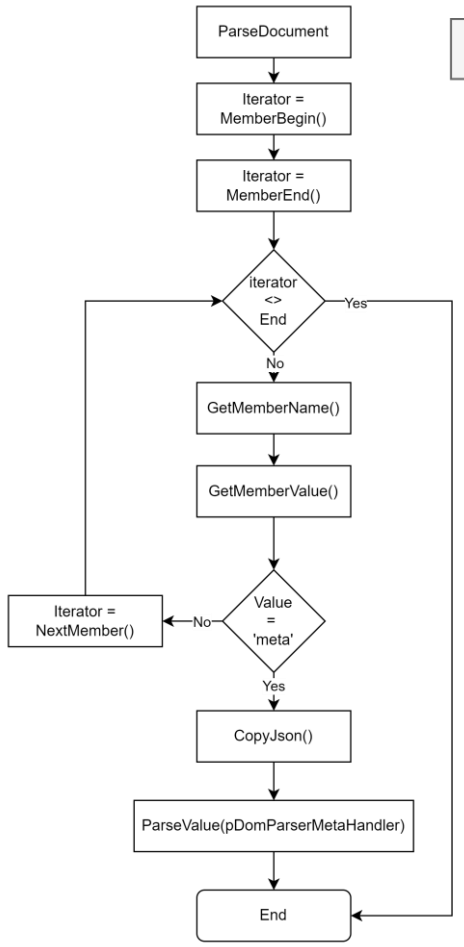
Mittag | 12:30 – 13:30

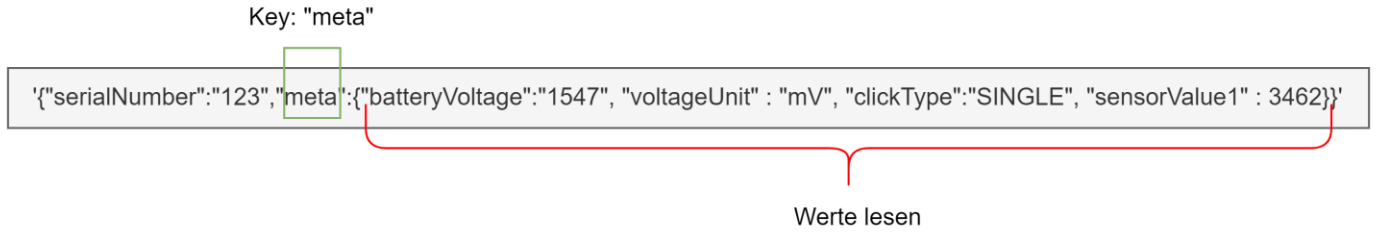
BECKHOFF

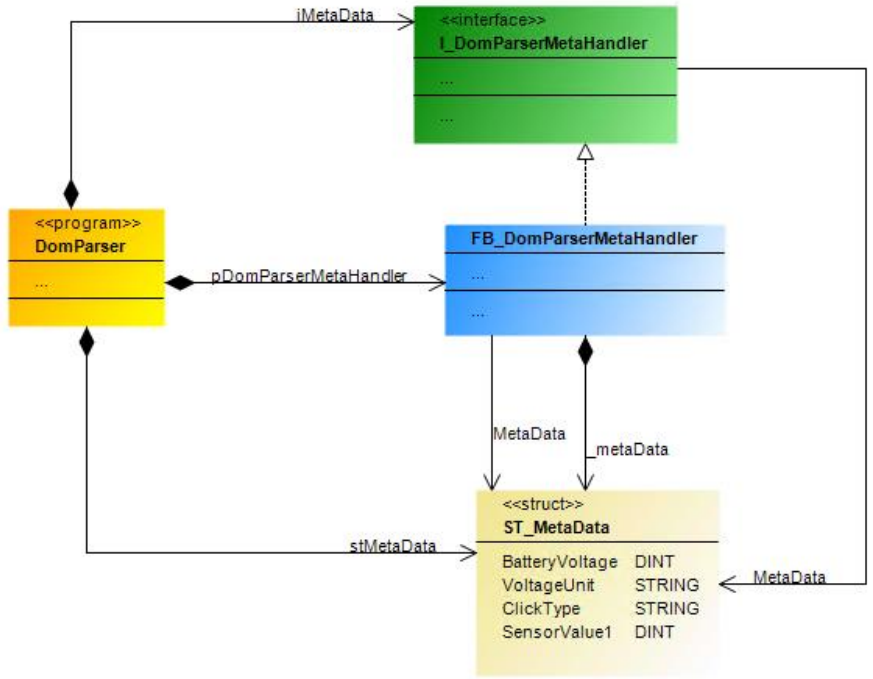
Beispiel | JSON Verarbeiten

BECKHOFF

– Standard Library: Tc3_JsonXml



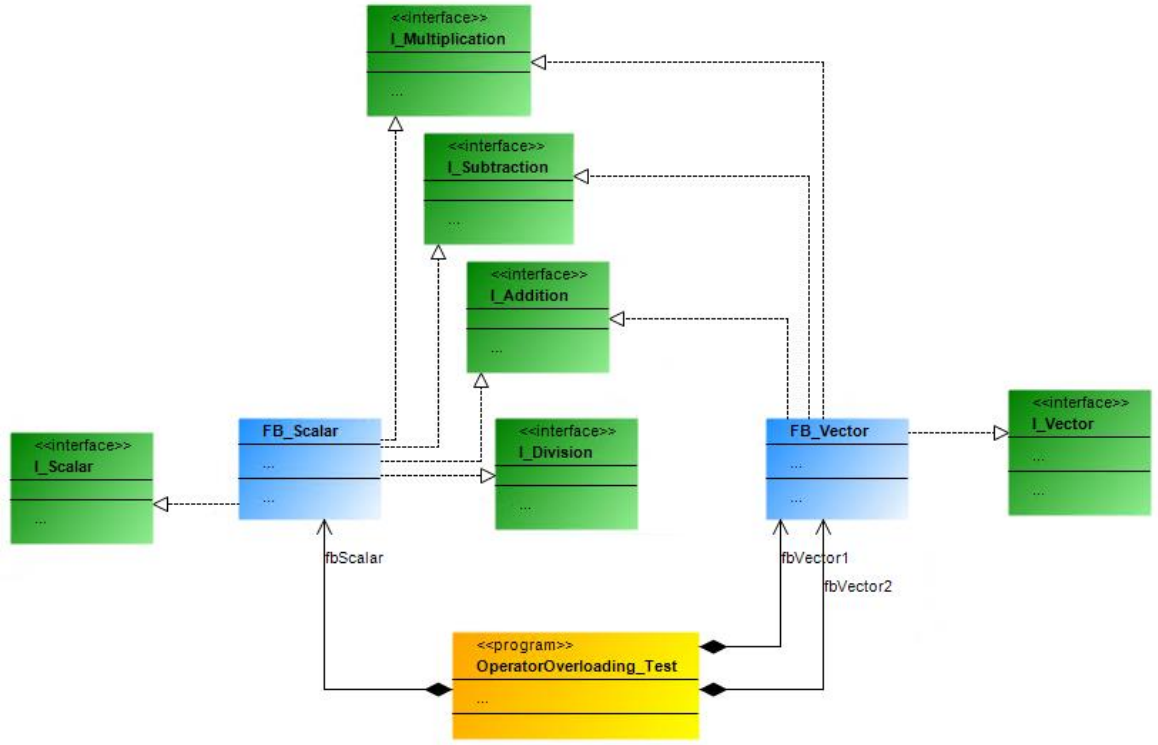




Beispiel | Operatorüberladung in TwinCAT mit OOP

BECKHOFF

- Operatorüberladung mit Interfaces
- FB_Vector, FB_Scalar
- Verschiedene Operation
  - Multiplikation
  - Addition



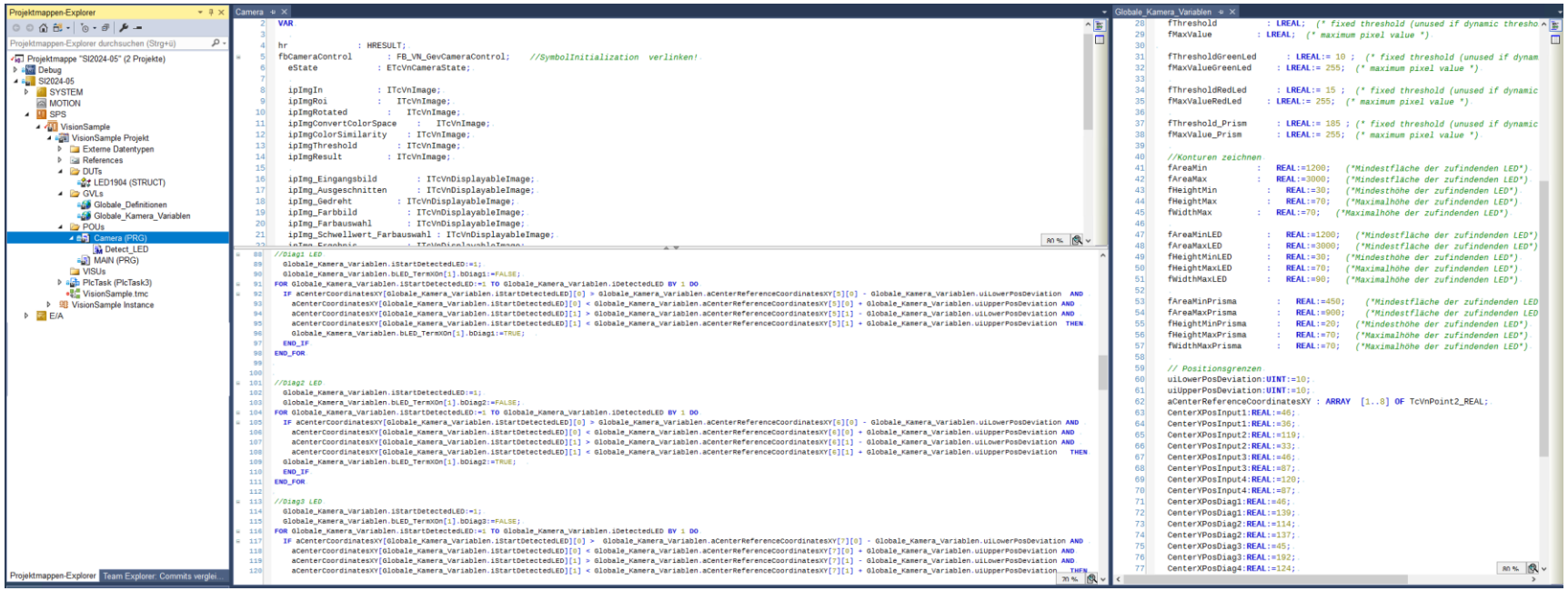
Kaffeepause 2 | 15:00 – 15:30

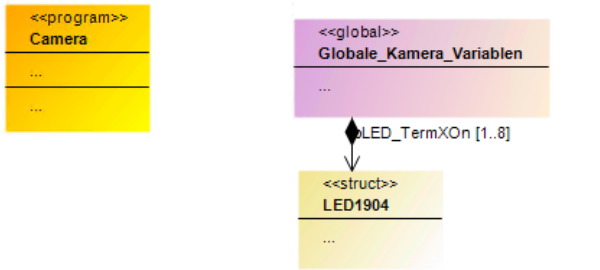
BECKHOFF

Beispiel | Refactoring

BECKHOFF

– Ein toll strukturiertes Programm überarbeiten …





Danke für Ihre Teilnahme