

Von algebraischen Datentypen und daten-orientierter Programmierung zu Pattern Matching überall

Falk Sippach

embarc · @sipp sack

Oktober 2026 · betterCode Java



Falk Sippach

- Softwarearchitekt, Berater, Trainer bei embarc
- früher bei Orientation in Objects (OIO), Trivadis

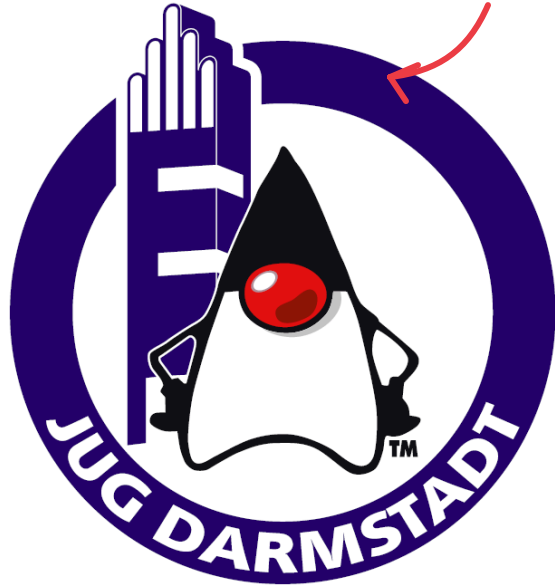
Schwerpunkte

- Architekturberatung und -bewertung
- Cloud- und Java-Technologien



Teil der Java Community

Nächster Termin
am **29.10.2026**



jug-da.de

monatliche Opensource
Code Camps



cyberland.ijug.eu

Agenda

- 01** Einleitung: DOP, Pattern Matching, ADTs
- 02** Features: von Switch Expressions bis Primitive Patterns
- 03** Live-Coding
- 04** Fazit: Was fehlt noch? Was ist in Arbeit?

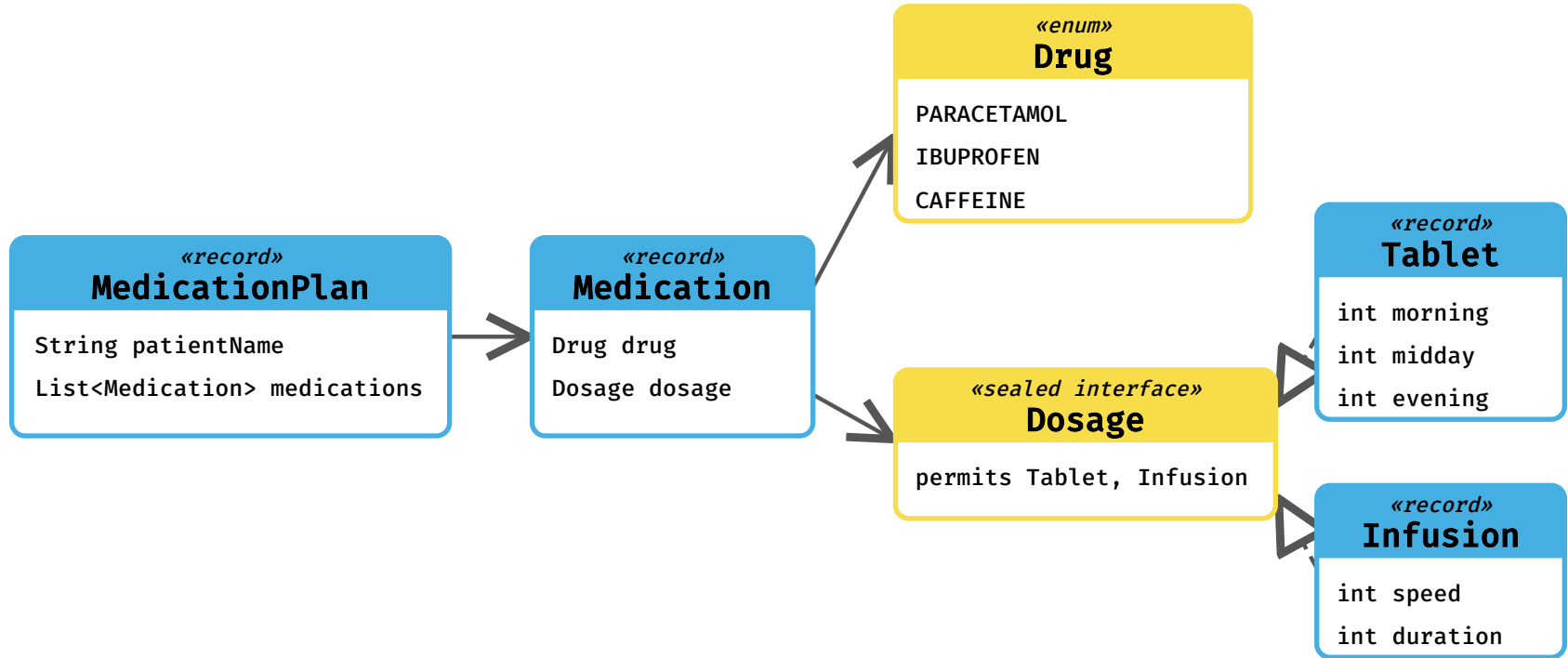
01

Einleitung

DOP, Pattern Matching, ADTs



Unser Beispiel: ein Medikationsplan



Alle Werte frei erfunden, keine medizinische Beratung.

Inspiziert von

Data Modeling with Sums and Products

Michael Sperber und Stefan Wehr · funktionale-
programmierung.de · November 2024

Produkte (*und*) und **Summen** (*oder*): die zwei Grundwerkzeuge für gute Datenmodelle.

Das Modell in Code

```
enum Drug { PARACETAMOL, IBUPROFEN, CAFFEINE }

sealed interface Dosage permits Tablet, Infusion { }
record Tablet(int morning, int midday, int evening) implements Dosage { }
record Infusion(int speed, int duration) implements Dosage { }

record Medication(Drug drug, Dosage dosage) { }
record MedicationPlan(String patientName, List<Medication> medications) { }
```

Modernes Java: **Records, Sealed Types**, ...

Data-Oriented Programming

- **Daten** zuerst: präzise, unveränderlich, transparent modellieren
- **Operationen** danach: getrennte Funktionen von Daten zu Daten
- **Programme** als Produktionsstraße: Daten fließen rein, werden transformiert und fließen wieder raus

Java's Werkzeugkasten für DOP

Records

Daten als Daten
(Produkttypen)

`Tablet`, `Infusion`,
`Medication`

Sealed Types

Alternativen
(Summentypen)

`Dosage`

Pattern Matching

Operationen auf Daten

`describe(Dosage)`

Schauen wir sie uns einzeln an.

Das Problem

Klassische Fallunterscheidung

- `if - else`-Ketten mit `instanceof` und Casts
- `switch` mit `break`, Fall-through und `default`
- Neue Fälle fallen niemandem auf

Ein Modell für alles

- Ein Typ mit vielen optionalen Feldern
- Ungültige Zustände sind darstellbar
- Fehler zeigen sich erst **zur Laufzeit**

Algebraische Datentypen

Produkttypen: *dies und das*

`Tablet = int & int & int`

`Medication = Drug & Dosage`

Records

Summentypen: *dies oder das*

`Dosage = Tablet | Infusion`

`Drug = PARACETAMOL | ...`

Sealed Types und Enums

Records + Sealed Types = algebraische Datentypen

Was ist Pattern Matching?

- Werte **prüfen** *und* nach ihrer **Struktur zerlegen**
- Gefundene Bestandteile direkt **weiterverwenden**
- Compiler prüft **Vollständigkeit** (Exhaustiveness)

```
case Infusion(int speed, int duration) -> "%d ml/h for %d min".formatted(speed, c
```

02

Features

Von Switch Expressions bis Primitive
Patterns



Der klassische switch

```
String categoryOld(Drug drug) {  
    String category;  
    switch (drug) {  
        case PARACETAMOL:  
        case IBUPROFEN:  
            category = "painkiller";  
            break;  
        case CAFFEINE:  
            category = "stimulant";  
            break;  
        default:  
            category = "unknown";  
    }  
    return category;  
}
```

- Statement, veränderliche Variable, **break**, Fall-through
- **default** versteckt später hinzugefügte Fälle

Switch Expressions

Java 14

```
String category(Drug drug) {  
    return switch (drug) {  
        case PARACETAMOL, IBUPROFEN -> "painkiller";  
        case CAFFEINE -> "stimulant";  
    };  
}
```

- Ein **Ausdruck**: liefert einen Wert
- Kein Fall-through, kein **break**
- Vollständig geprüft: kein **default** nötig ➡ neues **Drug**? Compile-Fehler!

Records: Data Carriers

Java 16

```
var tablet = new Tablet(1, 0, 1);

tablet.morning();           // 1
tablet.equals(new Tablet(1, 0, 1)); // true
tablet.toString();         // Tablet[morning=1, midday=0, evening=1]

// immutable: "changing" means creating a new one
var withMidday = new Tablet(tablet.morning(), 1, tablet.evening());
```

- **Transparent:** der Zustand *ist* die Komponentenliste
- **Unveränderlich:** keine Setter, finale Felder
- Konstruktor, Accessors, `equals`, `hashCode`, `toString` gratis

Records: Validierung und Invarianten

```
record Tablet(int morning, int midday, int evening) implements Dosage {  
    Tablet {  
        if (morning < 0 || midday < 0 || evening < 0)  
            throw new IllegalArgumentException("Counts must not be negative");  
    }  
}  
  
record MedicationPlan(String patientName, List<Medication> medications) {  
    MedicationPlan {  
        Objects.requireNonNull(patientName);  
        medications = List.copyOf(medications);  
    }  
}
```

- Compact Constructor ➡ ungültige Daten können nicht existieren
- Records sind nur *flach* unveränderlich ➡ `List.copyOf`

Welche Dosierungen sind gültig?

```
// Which of these combinations make sense?  
record Dosage(  
    Integer morning, Integer midday, Integer evening,  
    Integer speed, Integer duration) { }  
  
new Dosage(1, 0, 1, 5, 2);           // tablets *and* infusion?  
new Dosage(null, null, null, null, null); // nothing at all?
```

Ein Typ für alles ➡ ungültige Zustände sind **darstellbar**

Sealed Types

Java 17

```
sealed interface Dosage permits Tablet, Infusion { }  
record Tablet(int morning, int midday, int evening) implements Dosage { }  
record Infusion(int speed, int duration) implements Dosage { }
```

- **Geschlossene** Menge von Alternativen: `permits` nennt alle Subtypen
- Compiler und Leser kennen *alle* Varianten
- Records als Varianten $\rightsquigarrow$ transparente Alternativen

Type Patterns: instanceof

Java 16

Vorher

```
int morningPillsOld(Dosage dosage) {  
    if (dosage instanceof Tablet) {  
        Tablet tablet = (Tablet) dosage;  
        return tablet.morning();  
    }  
    return 0;  
}
```

Nachher

```
int morningPills(Dosage dosage) {  
    if (dosage instanceof Tablet tablet) {  
        return tablet.morning();  
    }  
    return 0;  
}
```

Prüfen, casten und binden in einem Schritt: `dosage instanceof Tablet tablet`

Type Patterns: Scope

```
boolean hasMorningPills(Dosage dosage) {  
    return dosage instanceof Tablet t && t.morning() > 0;  
}  
  
int middayPills(Dosage dosage) {  
    if (!(dosage instanceof Tablet tablet)) {  
        return 0;  
    }  
    // `tablet` is in scope here  
    return tablet.midday();  
}
```

Die Variable gilt dort, wo der Test **wahr** ist (Flow Scoping).

Pattern Matching für switch

Java 21

```
String describe(Dosage dosage) {  
    return switch (dosage) {  
        case Tablet tablet -> "%d-%d-%d tablets".formatted(  
            tablet.morning(), tablet.midday(), tablet.evening());  
        case Infusion infusion -> "%d ml/h for %d min".formatted(  
            infusion.speed(), infusion.duration());  
    };  
}
```

- Type Patterns als `case`-Labels, ein Zweig pro Alternative
- Keine `if-else`-Kette, keine Casts

Exhaustiveness

```
String describe(Dosage dosage) {  
    return switch (dosage) {  
        case Tablet tablet -> "%d-%d-%d tablets".formatted(  
            tablet.morning(), tablet.midday(), tablet.evening());  
        // error: does not cover all possible values  
    };  
}
```

- Sealed Types ↗ der Compiler kennt alle Fälle
- Fehlt einer? **Compile-Fehler**
- Kein `default` ↗ ein neuer Subtyp bricht *jeden* betroffenen `switch`

Guards (when)

```
String warn(Dosage dosage) {  
    return switch (dosage) {  
        case Tablet t when t.morning() > 3 -> "Too many tablets";  
        case Tablet t -> "OK";  
        case Infusion i when i.speed() > 100 -> "Too fast";  
        case Infusion i -> "OK";  
    };  
}
```

`case Type t when condition` ↗ ein Pattern verfeinern

Dominance und null

Allgemein vor speziell

```
switch (dosage) {  
    case Tablet t -> "OK";  
    // error: dominated by a  
    // preceding case label  
    case Tablet t when t.morning() > 6  
        -> "Too many tablets";  
    ...  
}
```

Reihenfolgefehler fallen auf.

null im Switch

```
return switch (dosage) {  
    case null -> "no dosage";  
    case Tablet t -> "tablets";  
    case Infusion i -> "infusion";  
};  
  
case null, default -> "something else";
```

Ohne `case null`: `NullPointerException`

Record Patterns

Java 21

```
String format(Dosage dosage) {  
    return switch (dosage) {  
        case Tablet(int morning, int midday, int evening) ->  
            "%d-%d-%d tablets".formatted(morning, midday, evening);  
        case Infusion(int speed, int duration) ->  
            "%d ml/h for %d min".formatted(speed, duration);  
    };  
}
```

- Typ prüfen **und** zerlegen
- Komponenten direkt an Variablen gebunden
- Konstruktion und Dekonstruktion sind symmetrisch

Verschachtelte Record Patterns

```
String format(Medication medication) {  
    return switch (medication) {  
        case Medication(var drug, Tablet(var morning, var midday, var evening))  
            -> "%s: %d-%d-%d".formatted(drug, morning, midday, evening);  
        case Medication(var drug, Infusion(var speed, var duration))  
            -> "%s: %d ml/h for %d min".formatted(drug, speed, duration);  
    };  
}
```

- Patterns verschachteln sich wie die Daten
- Ein `switch` statt geschachtelter `if`s, weiterhin vollständig geprüft!

Verschachtelte Patterns und Guards

```
String warn(Medication medication) {  
    return switch (medication) {  
        case Medication(var drug, Infusion(var speed, _))  
            when drug == Drug.CAFFEINE && speed > 10  
            -> "Caffeine: too fast";  
        case Medication(var drug, Tablet(var m, var d, var e))  
            when drug == Drug.IBUPROFEN && m + d + e > 3  
            -> "Ibuprofen: too many tablets";  
        case Medication _ -> "OK";  
    };  
}
```

Erst die Form matchen, dann mit `when` verfeinern.

Unnamed Patterns und Variables

Java 22

```
int morningPills(Dosage dosage) {  
    return switch (dosage) {  
        case Tablet(var morning, _, _)  
            -> morning;  
        case Infusion(_, _) -> 0;  
    };  
}
```

```
medication instanceof Medication(_, Tablet _)  
  
case Tablet _ -> "swallow";  
  
for (var _ : plan.medications()) {  
    count++;  
}
```

- `_` ➡ „brauche ich nicht“: zeigt die Absicht, keine ungenutzten Variablen
- Funktioniert bei Komponenten, Type Patterns, Variablen, Schleifen, Lambdas, `catch`

Primitive Type Patterns

```
// the pump only accepts a byte
void setPumpSpeed(byte speed) { /* ... */ }

void start(Infusion infusion) {
    if (infusion.speed() instanceof byte speed) {
        setPumpSpeed(speed);
    } else {
        throw new IllegalArgumentException(
            "Speed does not fit the pump: " + infusion.speed());
    }
}
```

- Patterns auch für `byte`, `int`, `double`, ...
- Treffer nur, wenn der Wert **verlustfrei** konvertierbar ist
- Ersetzt manuelle Bereichsprüfungen und Casts

Primitive Patterns in Records

```
void start(Dosage dosage) {  
    switch (dosage) {  
        case Infusion(byte speed, _) -> setPumpSpeed(speed);  
        case Infusion _ -> throw new IllegalArgumentException(  
            "Speed does not fit the pump");  
        case Tablet _ -> { }  
    }  
}
```

- Record-Komponenten mit engerem primitiven Pattern
- Kein Treffer ➡ der nächste `case` wird probiert

Goetz' vier Prinzipien

Brian Goetz, Juni 2022:

- **Daten sind unveränderlich.**
- **Modelliere die Daten, die ganzen Daten und nichts als die Daten.**
- **Mach ungültige Zustände unrepräsentierbar.**
- **Validiere an der Grenze.**

Überarbeitet: Parlogs DOP v1.1

Nicolai Parlog, 2024:

- **Daten unveränderlich und transparent modellieren.**
- **Modelliere die Daten, die ganzen Daten und nichts als die Daten.**
- **Mach ungültige Zustände unrepräsentierbar.**
- **Trenne Operationen von Daten.**

Von 1.0 zu 1.1

- **Unveränderlich** ↗ **unveränderlich und transparent**: die Daten müssen lesbar sein, nicht nur konstant
- **Validiere an der Grenze** ↗ Teil von *ungültige Zustände*: der Konstruktor ist die Grenze
- **Neu: Trenne Operationen von Daten**: Funktionen neben den Typen, nicht darin

Die beiden „Model the data“-Prinzipien bleiben unverändert.

Die Prinzipien im Dosage-Beispiel

- Unveränderlich, transparent: `Tablet`, `Infusion`, `List.copyOf`
- Die ganzen Daten, nichts sonst: `Medication` = `Drug` + `Dosage`
- Ungültige Zustände: sealed `Dosage` + Compact Constructors
- Trenne Operationen: `describe`, `format`, `dailyPills`

Eine Operation ergänzen

```
int dailyPills(Dosage dosage) {  
    return switch (dosage) {  
        case Tablet(var morning, var midday, var evening)  
            -> morning + midday + evening;  
        case Infusion _ -> 0;  
    };  
}
```

- Eine neue Funktion ↗ **kein** bestehender Typ ändert sich
- Der `switch` ist vollständig: kein `default` nötig

Eine Alternative ergänzen

```
sealed interface Dosage permits Tablet, Infusion, Patch { }
record Patch(int changeEveryHours) implements Dosage { }

String describe(Dosage dosage) {
    return switch (dosage) {
        case Tablet tablet -> "...";
        case Infusion infusion -> "...";
        // error: does not cover all possible values
    };
}
```

- Der Compiler zeigt **jeden** `switch`, der einen neuen Fall braucht
- Ohne `default` wird nichts stillschweigend übersprungen
- Trade-off: neue Alternativen sind der teure Teil

OOP und DOP

OOP	DOP
Zustand + Verhalten zusammen	Daten und Verhalten getrennt
Kapselung versteckt Details	Transparenz zeigt die Daten
offen für neue Subtypen	geschlossene Menge von Alternativen
sich entwickelnde Abläufe	Daten, die herumgereicht werden

Ergänzende Werkzeuge, keine Rivalen.

Wann DOP?

- **Daten-Sub-Systeme:** Batch Jobs, Analyse, Event Processing
- **Kleine Probleme**, die keine weitere Modularisierung brauchen
- **Daten von außen:** JSON, Nachrichten, ASTs, Rückgabewerte

Klein anfangen: schon ein Record und ein `switch` helfen.

Was zählt

- **Records** ↗ Daten als Daten
- **Sealed Types** ↗ Alternativen, die der Compiler kennt
- **Pattern Matching** ↗ vollständige Operationen, kein default
- **DOP** ↗ ein Stil, der alle drei nutzt

Zusammenfassung

Feature	Final in	JEP
Switch Expressions	Java 14	JEP 361
Records	Java 16	JEP 395
Type Patterns für <code>instanceof</code>	Java 16	JEP 394
Sealed Classes und Interfaces	Java 17	JEP 409
Pattern Matching für <code>switch</code>	Java 21	JEP 441
Record Patterns	Java 21	JEP 440
Unnamed Patterns und Variables	Java 22	JEP 456
Primitive Types in Patterns	Preview (Java 27)	JEP 532
Data-Oriented Programming	Java 21+	–

03

Live-Coding

Dosage in IntelliJ



Live-Coding: Code zum Mitmachen

Den Code gibt es auf GitHub

github.com/sippsack/java-dop-pattern-matching

04

Fazit

Was fehlt noch? Was ist in Arbeit?



Ausblick: Was kommt als Nächstes?



**Pattern
Assignment**



**Patterns in
for-each**



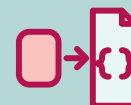
**Dekonstruierbare
Klassen**



**Dekonstruierbare
Interfaces**



with-Ausdrücke



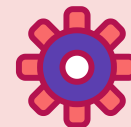
**Bessere
Serialisierung**



**Nominale
Patterns**



Record-Evolution



**Member-
Patterns**

Ausblick: Dekonstruktion überall

Syntax nicht final

```
class Infusion(int speed, int duration) {  
    private final Rate rate;  
}
```

Dekonstruierbare Klasse
State Description im Kopf

```
interface Medication(Drug drug, Dosage dosage) { }
```

Dekonstruierbares Interface
Implementierung verborgen

```
Medication(var drug, Infusion(var speed, _)) = med;
```

Pattern Assignment
Zerlegen ohne if oder switch

```
for (Map.Entry(var drug, var dose) : plan.entrySet()) {  
    IO.println(drug + ": " + dose);  
}
```

Pattern im for-each
Kein getKey() / getValue()

Ausblick: Rekonstruktion

Syntax nicht final

```
class Infusion(int speed, int duration) {  
    public Infusion {  
        if (speed <= 0)  
            throw new IllegalArgumentException();  
    }  
}
```

Kanonischer Konstruktor

Hier liegen die Invarianten

```
var slower = infusion with { speed = speed / 2; };
```

with-Ausdruck

Kopie, Original bleibt

```
String json = serializer.write(infusion);
```

Serialisierung

Zustand ohne Reflection

```
Infusion back = serializer.read(json);
```

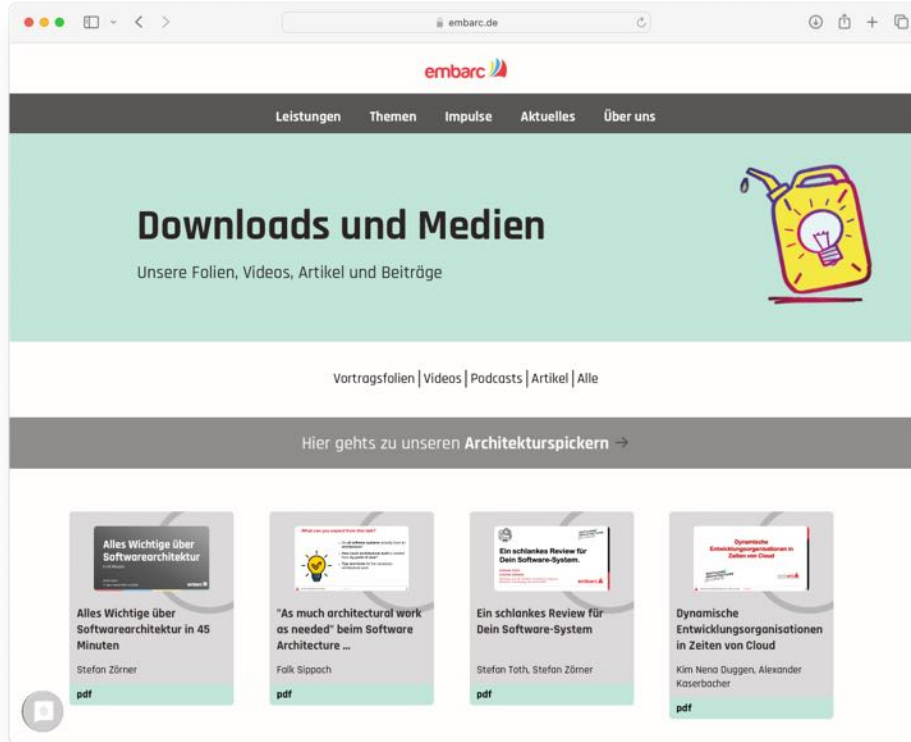
Deserialisierung

Ruft den Konstruktor auf

Take-aways

- **Pattern Matching** macht Fallunterscheidungen explizit, kurz und compilergeprüft
- **Records + Sealed Types** modellieren Daten präzise, ungültige Zustände werden unrepräsentierbar
- **DOP** trennt Daten und Verhalten, ergänzt OOP und ist schon heute mit Java 21+ möglich

Folien als PDF zum Download



embarc.de/download/

Unser Line-Up

Vorträge, Workshops, neue Perspektiven und Plaudern
in lockerer Punsch-Atmosphäre am 7. Dezember 2026!



Ulrich Lehner
(LVM Versicherungen)



Tom Asel
(tangible concepts)



Roman Hecht
(Continental Reifen)



Niklas Trentmann



Falk Sippach



Michael Rabe
(Continental Reifen)



Stefan Zörner



Stefan Toth



Alle Infos & Anmeldung: embarc.de/architektur-punsch/








Der Architektur-überblick

Ein Architekturüberblick macht die zentralen Lösungsansätze Ihrer Softwarearchitektur in kompakter Form nachvollziehbar.

1

NR.

IN DIESER AUSGABE

- Welche Zutaten gehören in einen Architekturüberblick?
- Welche Formen bewähren sich in welchen Situationen?
- Wie fertigen Sie einen an?

Herausforderungen

- Team- oder Projektmitgliedern (z.B. Entwicklern) fehlt ein Überblick über Lösungsansätze auf hoher Ebene, um fokussiert zu arbeiten
- Neue Teammitglieder, die mitentwickeln wollen, finden sich in der Architektur nicht zurecht
- Entscheider und andere Stakeholder haben Unsicherheiten oder geringes Vertrauen in die Lösung
- Teamfremde Kollegen sind an Lösungsansätzen interessiert, finden aber keine oder sehr detaillierte Informationen, die Ihnen einen schnellen Überblick schwierig machen.

Inhalte eines Architekturüberblicks

Arbeiten Sie kleinteilig! Fertigen Sie unabhängige „Zutaten“ an, die Sie zu unterschiedlichen Formen rekombinieren, und bei Bedarf iterativ verfeinern.

Formen

Je nach Zielgruppe und Kommunikationsweg sind für einen Architekturüberblick sehr unterschiedliche Formen denkbar.

- **Architekturwand:** Jedermann zugänglicher, großformatiger, modularer Aushang an einer Wand im Projektraum
- **Architekturflyer oder -poster:** Kleines Handout, z.B. DIN A4 beidseitig bedruckt, 2-3x gefaltet, oder größer produziert (z.B. DIN A1) als Plakat zur weiten Verbreitung
- **Architekturportal im Wiki:** Einstiegsseite(n) im Wiki, die Interessierte durch die Inhalte führen
- **Prägnantes Dokument:** Strukturierter Text, angereichert mit Illustrationen, Umfang maximal 20 Seiten
- **Foliensatz:** 10-15 Folien zur Unterstützung einer Präsentation der Architektur
- **Video:** Aufzeichnung eines Überblicks in Ton und Bild, evtl. kombiniert mit Foliensatz

Zutaten

Was gehört rein? Die Zutaten dieser Abbildung sind auf der nächsten Seite beschrieben. Keine Sorge, Sie brauchen i.d.R. nicht alle.

Problemstellung	Lösungsstrategie	Lösungsdetails
Mission Statement	Lösungsstrategie	Architektur-entscheidungen
Einflussfaktoren	Architekturprinzipien	„Sichten“
Architekturziele		Struktur
Randbedingungen		Verhalten
Risiken		Verteilung
Kontextabgrenzung	Überblicksbild	Übergreifende Konzepte

Architektur-Spicker (1-16)

„Mit unseren Architektur-Spickern beleuchten wir die konzeptionelle Seite der Softwareentwicklung.“

Architektur-Spicker #1 Der Architekturüberblick



PDF, 4 Seiten
Kostenloser Download.

embarc.de/architektur-spicker/

Vielen Dank.

Ich freue mich auf Eure Fragen!

✉ falk.sippach@embarc.de

in [linkedin.com/in/falk-sippach](https://www.linkedin.com/in/falk-sippach)

X [@sipp sack](#)

m [@sipp sack@ijug.social](#)

