Markus Schlegel, Active Group, 2025-05-07

Decoupled by Default

Funktionale Softwarearchitektur

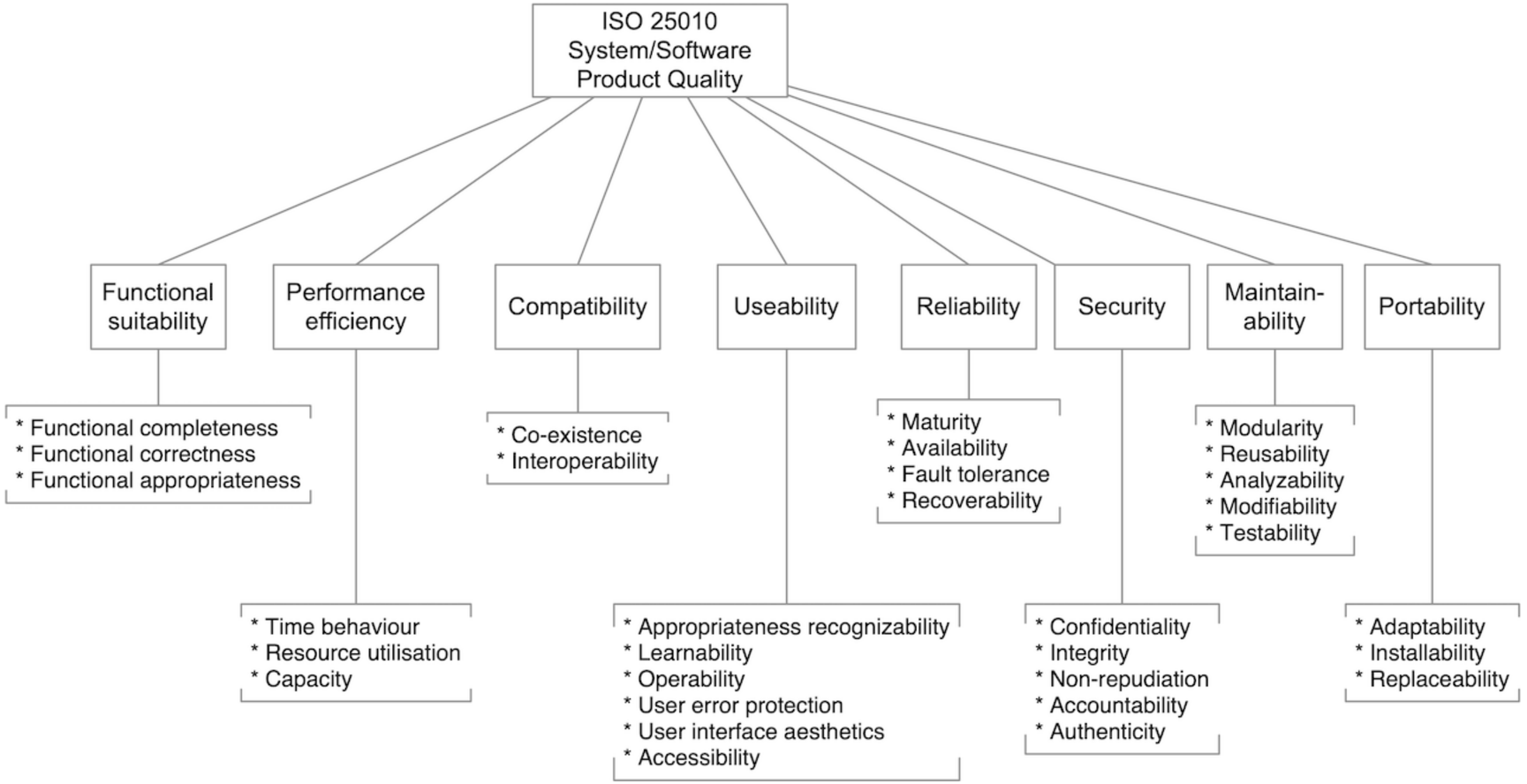
Funktionale Softwarearchitektur

"Functional programming in the large"

Functional Software Architecture refers to methods of construction and structure of large and long-lived software projects that are implemented in functional languages and released to real users, typically in industry.

https://functional-architecture.org/

Warum Wartbarkeit?




Ohne Wartbarkeit ist alles doof.
Funktionalität: doof
Zuverlässigkeit: doof
Sicherheit doof
Compatibility: doof
Portability doof
Krabbelkäfer: doof
Zum Ausfüllen und Verschenken für deinen Lieblingsarchitekten
Performance doof
riva

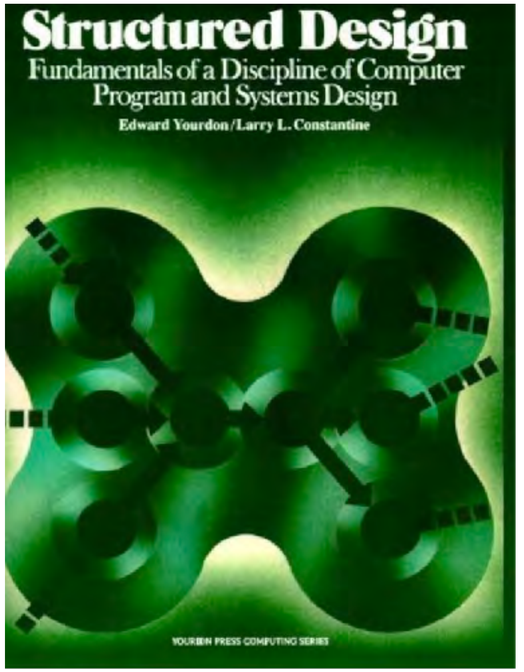
Decoupled (Coupling)

Dt. Kopplung

Kopplung ist das Maß der Abhängigkeiten zwischen zwei Modulen.

“Je mehr wir über Modul B wissen müssen, um Modul A zu verstehen, desto enger gebunden ist A an B.”

– Yourdon & Constantine, Structured Design

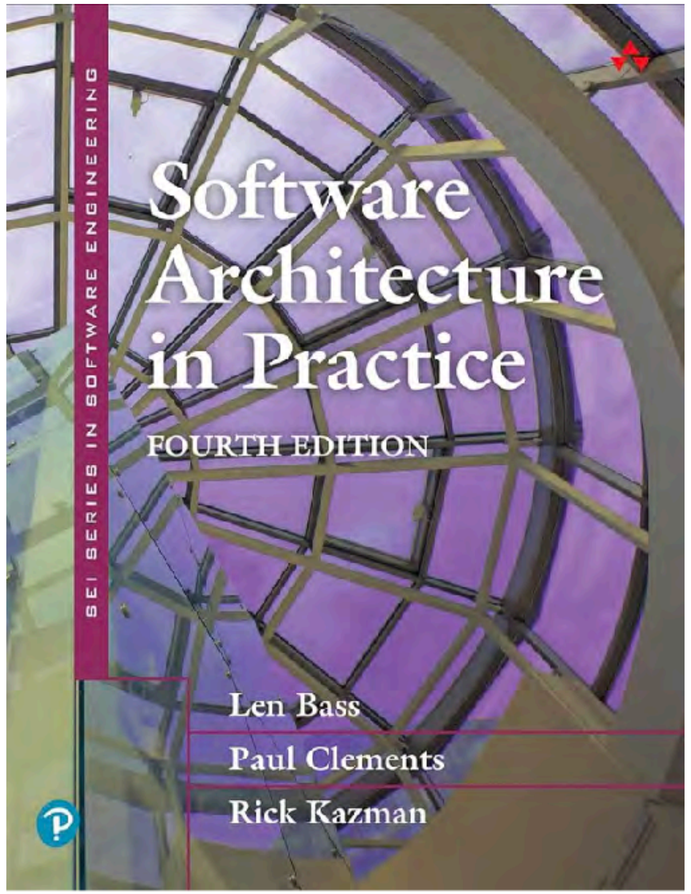


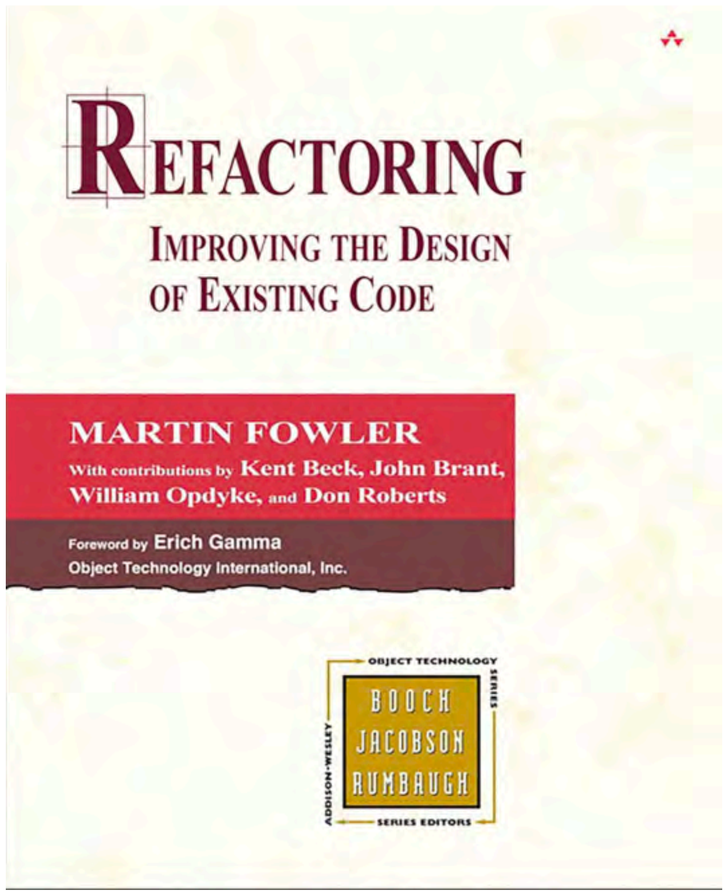
Decoupled (Coupling)

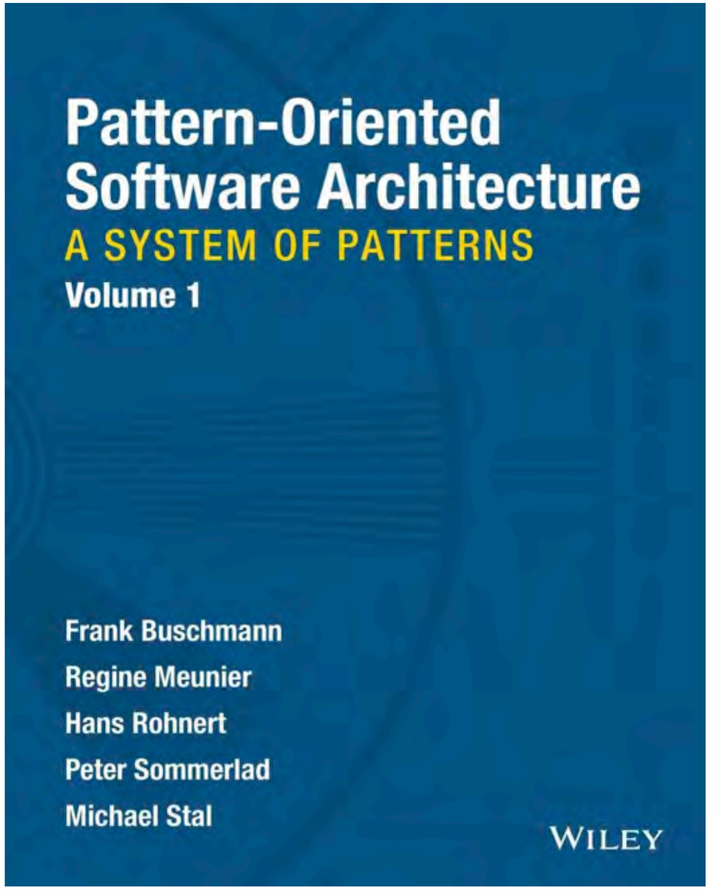
Dt. Kopplung

Kopplung ist damit das entscheidende Maß für die Wartbarkeit (Formbarkeit) eines Softwaresystems.

Decoupled **by Default**







Arten von Kopplung

- Common Coupling: geteilter Zustand
- External Coupling: Kopplung an die Plattform
- Content Coupling: Verletzung des Information-Hiding-Prinzips
- …

Common Coupling

Programmieren mit veränderlichem Zustand

```java
public class Person {
    private String name;
    private String address;
    ...
    public void setAddress(String newAddress) {
        this.address = newAddress;
    }
}

public class A {
    private Person myPerson;
    ...
    public void receivePerson(Person p) {
        myPerson = p;
    }
}

public class B {
    private Person myPerson;
    ...
    public void transferPerson(A other) {
        // Transfer to other department
        other.receivePerson(myPerson);
    }

    public void backToTheOffice() {
        myPerson.setAddress("Seattle");
    }
}
```

Imperativ: Data

Funktional: Data - Mutation

Imperativ: Data

Funktional: Data - Mutation

Während *Daten* in der Umgangssprache Gegebenheiten, Tatsachen oder Ereignisse sind, sind *Daten* in der Fachsprache Zeichen, die eine Information darstellen.

Imperativ: Data

Funktional: Data - Mutation

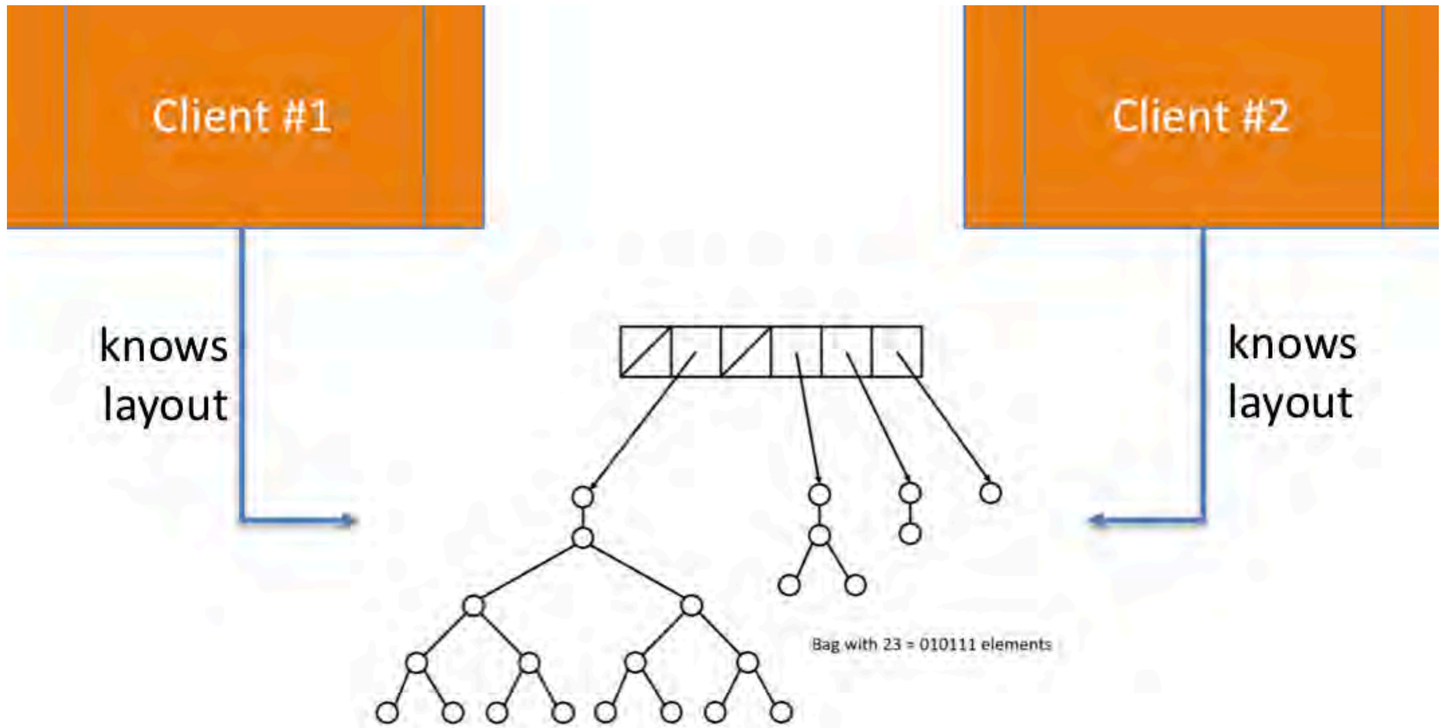
Während *Daten* in der Umgangssprache Gegebenheiten, Tatsachen oder Ereignisse sind, sind *Daten* in der Fachsprache Zeichen, die eine Information darstellen.

Imperativ: Data + Place

Funktional: Data

Content Coupling

Verletzung des Information-Hiding-Prinzips



Zeitreihen

```
interface TimeSeriesServiceLegacy {
    public List<Pair<Time, Double>> getTimeSeriesData(Time from, Time to);
}
```

```
var ts1 = listOf(new Pair<>(t1, 5.0),
                 new Pair<>(t2, 6.5),
                 new Pair<>(t3, 7.3));

var ts2 = listOf(new Pair<>(t2, 6.5),
                 new Pair<>(t1, 5.0),
                 new Pair<>(t3, 7.3));

var ts3 = listOf(new Pair<>(t1, 6.5),
                 new Pair<>(t1, 6.5));

var ts4 = listOf(new Pair<>(t1, 6.5),
                 new Pair<>(t1, 9.9));
```

Make Illegal States Unrepresentable

Effective ML video

AUG 21, 2010 | 1 MIN READ

By: Yaron Minsky

A while back I mentioned that I'd given a guest lecture at classes at Harvard and Northeastern, and that the Harvard class had been taped. I finally got and uploaded that video:

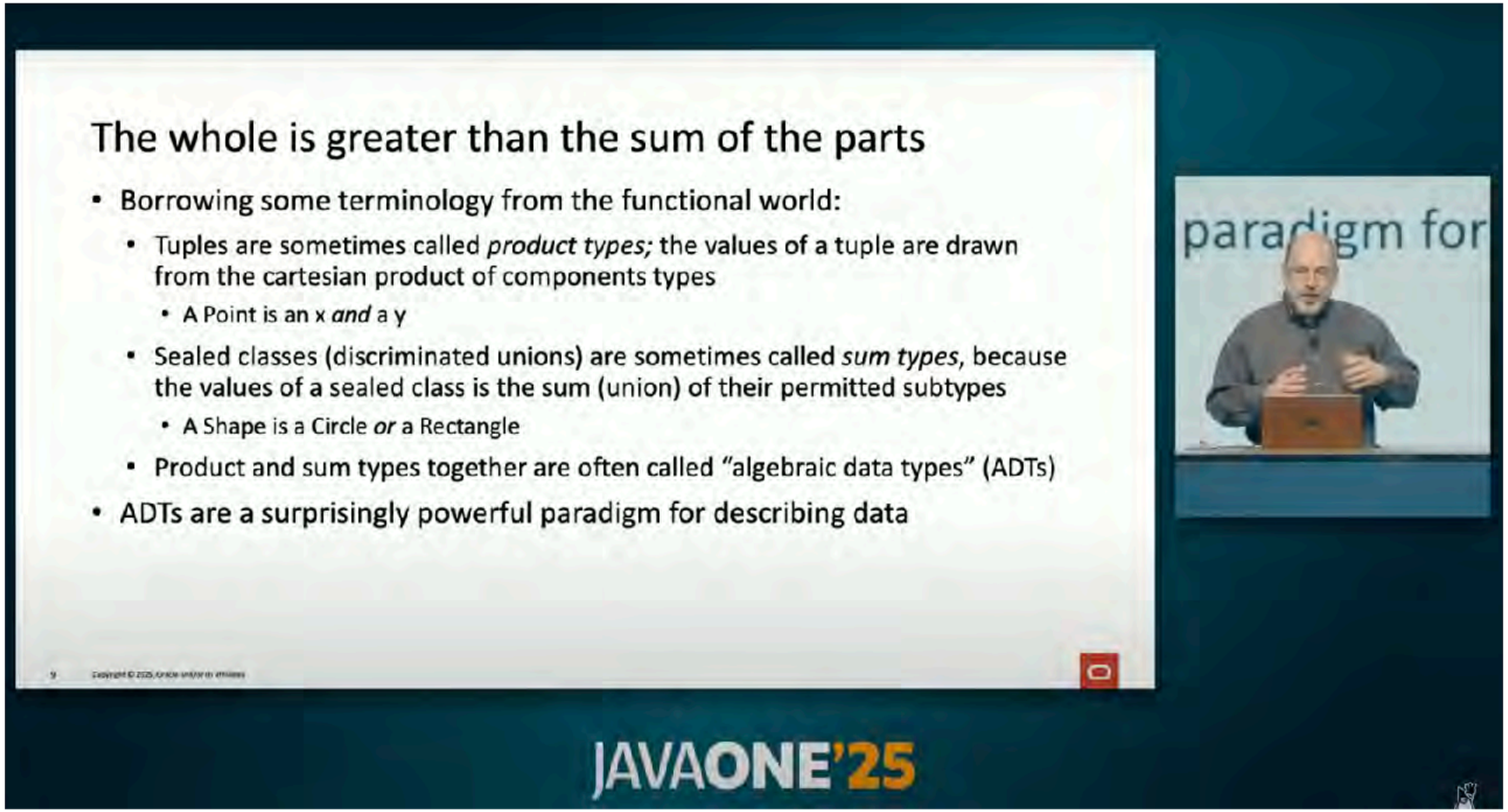
Make Illegal States Unrepresentable

Aber wie?

- **Smart Constructors**
- Parsing mit nominell frischen Typen (**“Parse, don’t validate”**, https://lexi-lambda.github.io/blog/2019/11/05/parse-don-t-validate/)
- Flache Produkttypen mit nullables zu **Summentypen** machen (https://funktionale-programmierung.de/2024/11/25/sums-products.html)
- Assoziative **Maps** nutzen
- **Funktionen** nutzen
- …

```
type connection_state =
| Connecting
| Connected
| Disconnected

type connection_info = {
  state: connection_state;
  server: Inet_addr.t;
  last_ping_time: Time.t option;
  last_ping_id: int option;
  session_id: string option;
  when_initiated: Time.t option;
  when_disconnected: Time.t option;
}
```

```
type connecting = { when_initiated: Time.t; }
type connected = { last_ping : (Time.t * int) option;
                   session_id: string; }
type disconnected = { when_disconnected: Time.t; }

type connection_state =
| Connecting of connecting
| Connected of connected
| Disconnected of disconnected

type connection_info = {
  state : connection_state;
  server: Inet_addr.t;
}
```

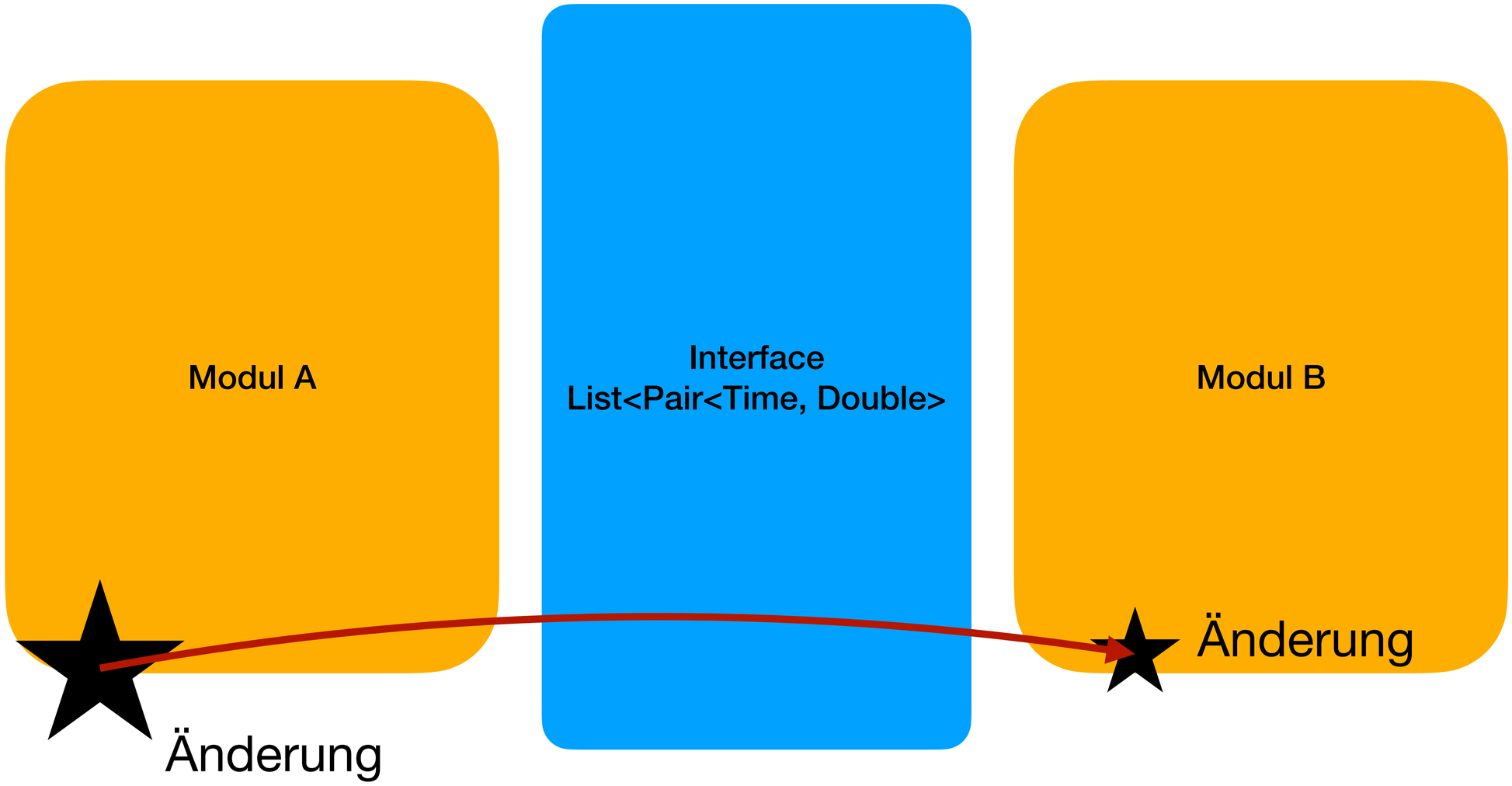


https://www.youtube.com/watch?v=1dY57CDxR14

```java
public class TimeSeriesServiceAntiCorruption {
    private TimeSeriesServiceLegacy legacyService;

    public TimeSeriesServiceAntiCorruption(TimeSeriesServiceLegacy ls) {
        legacyService = ls;
    }

    private Map<Time, Double> parseList(List<Pair<Time, Double>> l, Map<Time, Double> acc) {
        if (l instanceof Nil) {
            return acc;
        } else if (l instanceof Cons<Pair<Time, Double>> cns) {
            return parseList(cns.cdr(),
                             acc.assoc(cns.car().left(),
                                       cns.car().right(),
                                       acc));
        } else {
            return null;
        }
    }

    public Map<Time, Double> getTimeSeriesData(Time from, Time to) {
        return parseList(legacyService.getTimeSeriesData(from, to), EmptyMap.getInstance());
    }
}
```

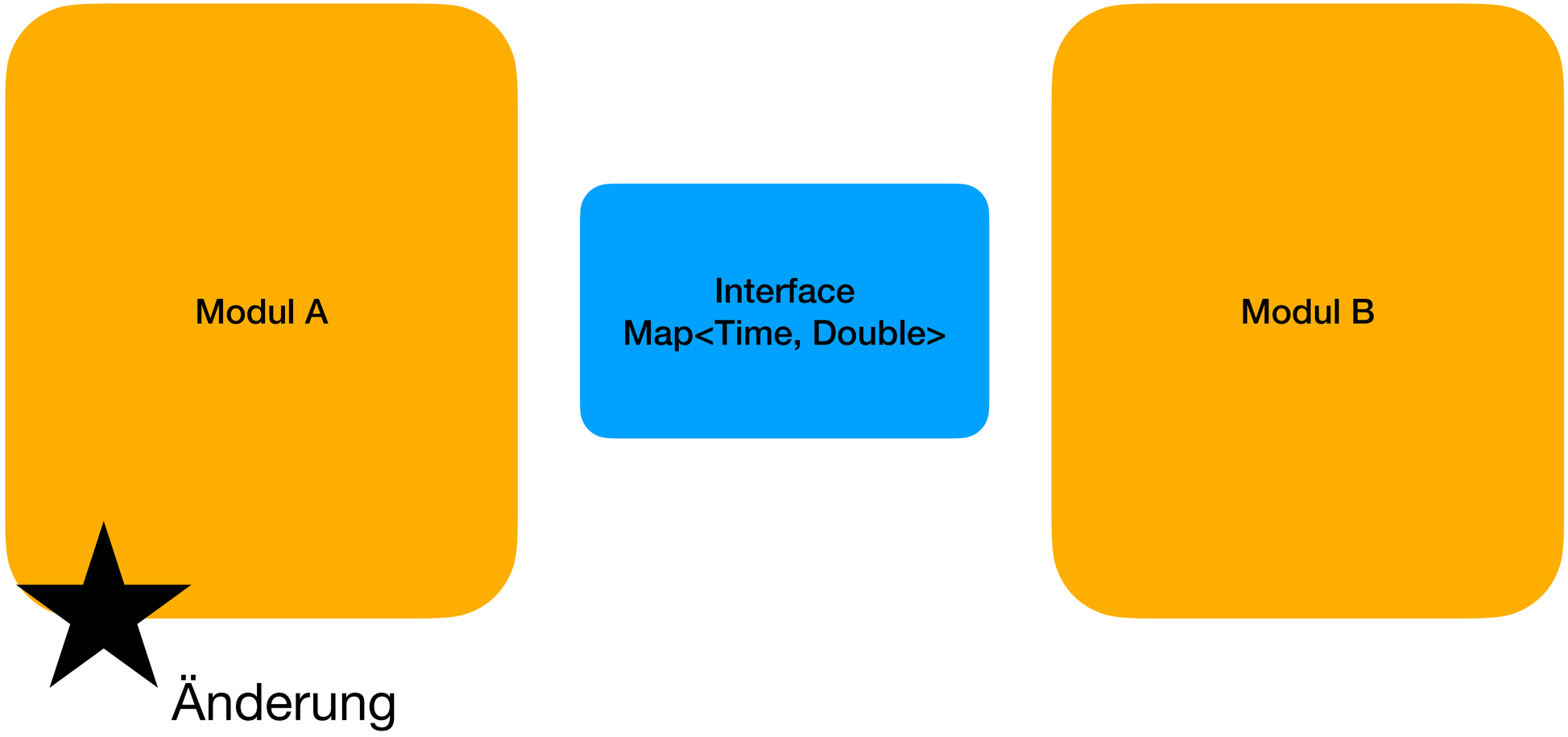
```java
public class TimeSeriesServiceAntiCorruption {
    private TimeSeriesServiceLegacy legacyService;

    public TimeSeriesServiceAntiCorruption(TimeSeriesServiceLegacy ls) {
        legacyService = ls;
    }

    private Map<Time, Double> parseList(List<Pair<Time, Double>> l, Map<Time, Double> acc) {
        if (l instanceof Nil) {
            return acc;
        } else if (l instanceof Cons<Pair<Time, Double>> cns) {
            return parseList(cns.cdr(),
                             acc.assoc(cns.car().left(),
                                       cns.car().right(),
                                       acc));
        } else {
            return null;
        }
    }

    public Map<Time, Double> getTimeSeriesData(Time from, Time to) {
        return parseList(legacyService.getTimeSeriesData(from, to), EmptyMap.getInstance());
    }
}
```

Zu offenes Interface



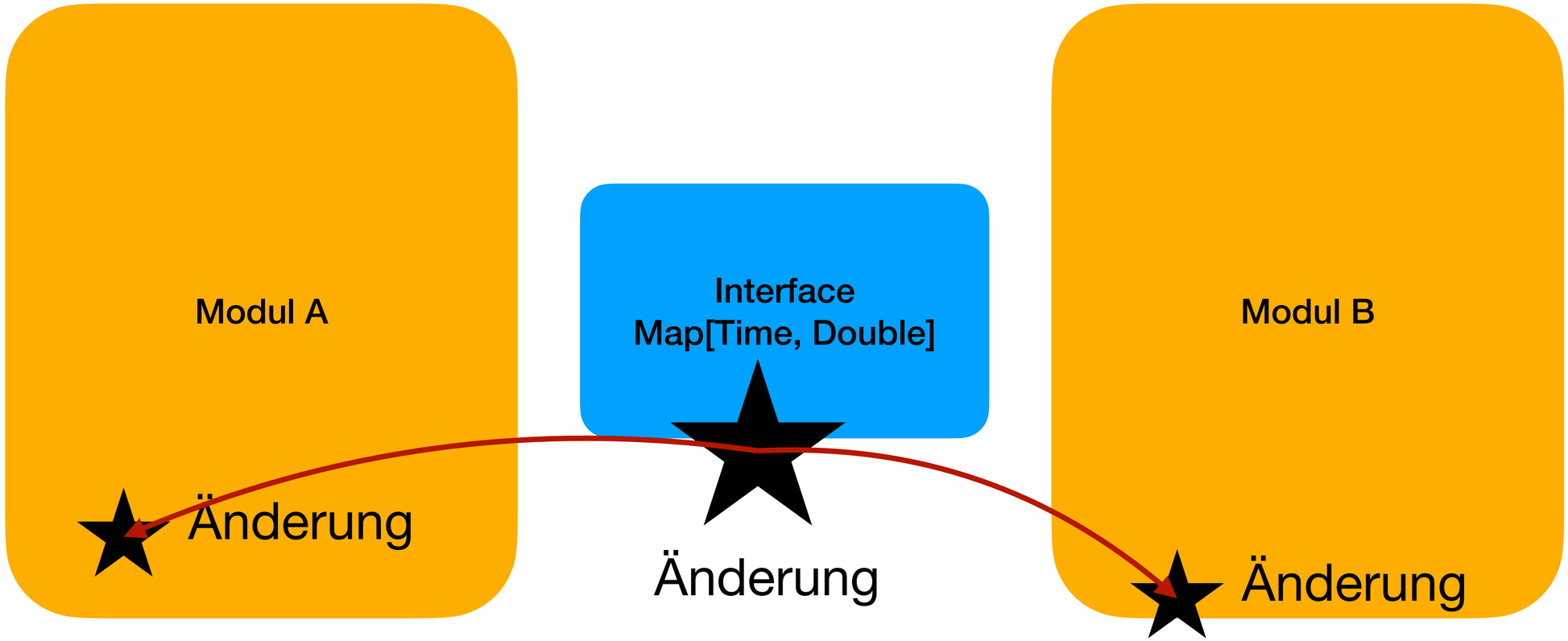
Zu beschränktes Interface

(Make Illegal States Unrepresentable)



Zu beschränktes Interface

(Make Illegal States Unrepresentable)



Make Legal States Representable

Denotational Design

Denotational design with type class morphisms (extended version)

Conal Elliott

LambdaPix

conal@conal.net

Abstract

Type classes provide a mechanism for varied implementations of standard interfaces. Many of these interfaces are founded in mathematical tradition and so have regularity not only of *types* but also of *properties* (laws) that must hold. Types and properties give strong guidance to the library implementor, while leaving freedom as well. Some of this remaining freedom is in *how* the implementation works, and some is in *what* it accomplishes.

To give additional guidance to the *what*, without impinging on the *how*, this paper proposes a principle of *type class morphisms* (TCMs), which further refines the compositional style of denotational semantics. The TCM idea is simply that *the instance's meaning follows the meaning's instance*. This principle determines the

little. It is form without *essence*. Users of this abstraction care about what *functionality* these names have. They care what the names *mean*.

Which leads to the question: what do we mean by "mean"? What is the *essence* of a type, enlivening its form? One useful answer is given by denotational semantics. The meaning of a program data type is a type of mathematical object (e.g., numbers, sets, functions, tuples). The meaning of each operation is defined as a function from the meanings of its arguments to the meaning of its result. For instance, the meaning the type $Map\ k\ v$ could be partial functions from k to v. In this model, *empty* is the completely undefined function, *insert* extends a partial function, and *lookup* is just function application, yielding $\perp$ where undefined. The meaning of the

Time Series = Map = Partial + Function ?

Time Series = ~~Map = Partial~~ + Function

```java
interface TimeFunction<A> extends Function<Time, A> {}

record TFList<A>(List<Pair<Time, A>> list) implements TimeFunction<Maybe<A>> {
    public Maybe<A> apply(Time t) {
        if (list instanceof Nil) {
            return Maybe.nothing();
        } else if (list instanceof Cons<Pair<Time, A>> cns) {
            if (cns.car().left().equals(t)) {
                return Maybe.just(cns.car().right());
            } else {
                return new TFList<>(cns.cdr()).apply(t);
            }
        } else {
            return null;
        }
    }
}
```

```java
public class TimeSeriesService2 {
    private TimeSeriesServiceLegacy legacyService;

    public TimeSeriesService2(TimeSeriesServiceLegacy ls) {
        legacyService = ls;
    }

    public TimeFunction<Maybe<Double>> getTimeSeriesData(Time from, Time to) {
        return new TFList<Double>(legacyService.getTimeSeriesData(from ,to));
    }
}
```

```
record TFFun<A>(Function<Time, A> f) implements TimeFunction<A> {
    public A apply(Time t) {
        return f.apply(t);
    }
}
```

```
var immerPi = new TFFun<Double>(t -> 3.1415);
```

Zusammenfassung

Decoupled by Default

- Wartbarkeit/Formbarkeit ist der Enabler für alle weiteren Softwarequalitäten
- Kopplung ist das wichtigste Maß für Wartbarkeit
- Funktionale Architekturen sind per Werkseinstellung wartbarer als klassische Architekturen
- Mit Denotational Design entstehen ganz beiläufig lose gekoppelte Systeme
- Mehr gibt’s im iSAQB-Modul FUNAR: https://active-group.de/schulung/FUNAR/

Getypte funktionale Programmierung auf der Java-Plattform mit Scala
Scala – woher, wohin, warum?

Funktionale Programmierung ist besser
Mächtigere Modelle, weniger Kopplung, aussagekräftige Typen

Einführung in die funktionale Programmierung mit Scala
Von null auf Scala

Softwarearchitektur: Warum Kombinatormodelle die Domäne revolutionieren können
Kombiniere … das Modell passt

Property-based Testing in Scala
Don't Write Tests: Write Properties!

Funktionale Programmierung in der Softwarearchitektur
Decoupled by Default

Mailand oder Neubauten, Hauptsache S(k)cala
Editorial