Softwaretechnik 2 (ST2)

Informatik Bachelor, SoSe 2025

Prof. Dr.-Ing. Stefan Bente

Workshop M2:

Domain Primitives
(und Generelles zum Code)

Technology
Arts Sciences
TH Köln

Domain Primitives

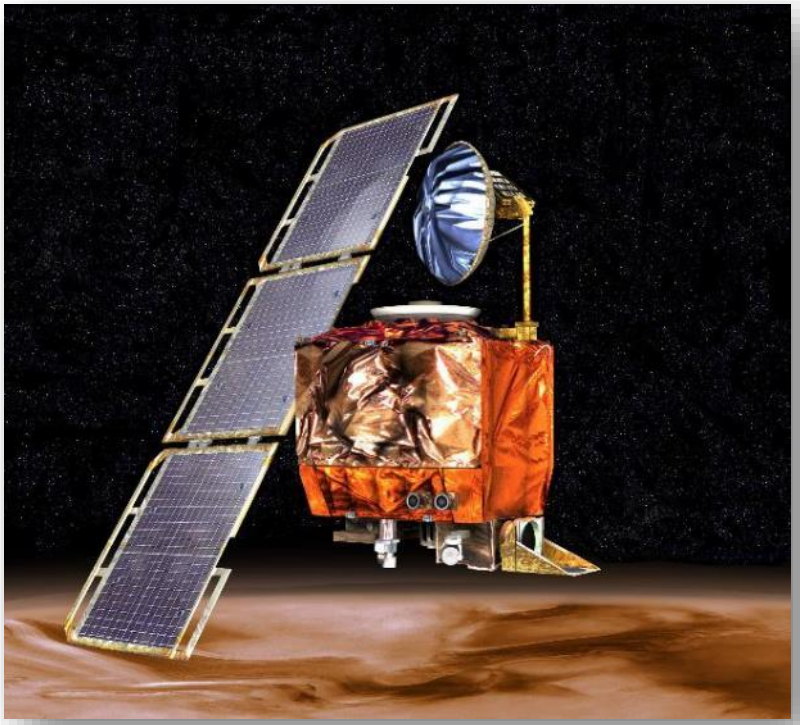
Zwei prominente (sehr teure) Fehler mit Basistypen

Ariadne-5-Absturz, 1996

“However, problems began to occur when the software attempted to stuff this 64-bit variable (…) into a 16-bit integer. (…) For the first few seconds of flight, the rocket’s acceleration was low, so the conversion between these two values was successful. However, as the rocket’s velocity increased, the 64-bit variable exceeded 65k, and became too large to fit in a 16-bit variable.“

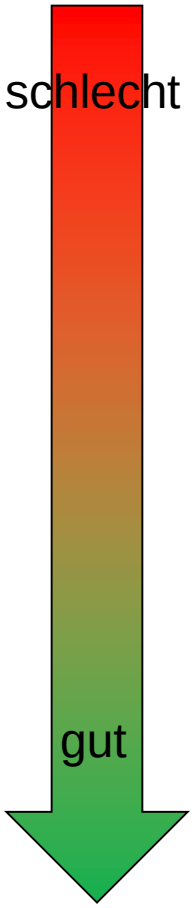
Absturz Mars Climate Orbiter, 1999

“Quality Assurance had not found the use of an imperial unit in external software, despite the fact that NASA’s coding standards at the time mandated use of metric units..“



Quellen: https://www.bugsnag.com/blog/bug-day-mars-climate-orbiter , https://www.bugsnag.com/blog/bug-day-ariane-5-disaster,
Bilder: NASA/JPL/Corby Waste - Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=390903,
DLR German Aerospace Center - Raumfrachter ATV-4 "Albert Einstein" Ariane 5ES Rollout_4, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26533952

Die Rangfolge von „am schlechtesten“ bis „am besten“



1. ~~Attribute als primitive Datentypen (int, float, double, …)~~

Sollte man immer vermeiden (ist ja auch leicht umzusetzen)

2. Attribute als Wrapperklassen (Integer, Float, Double, …)

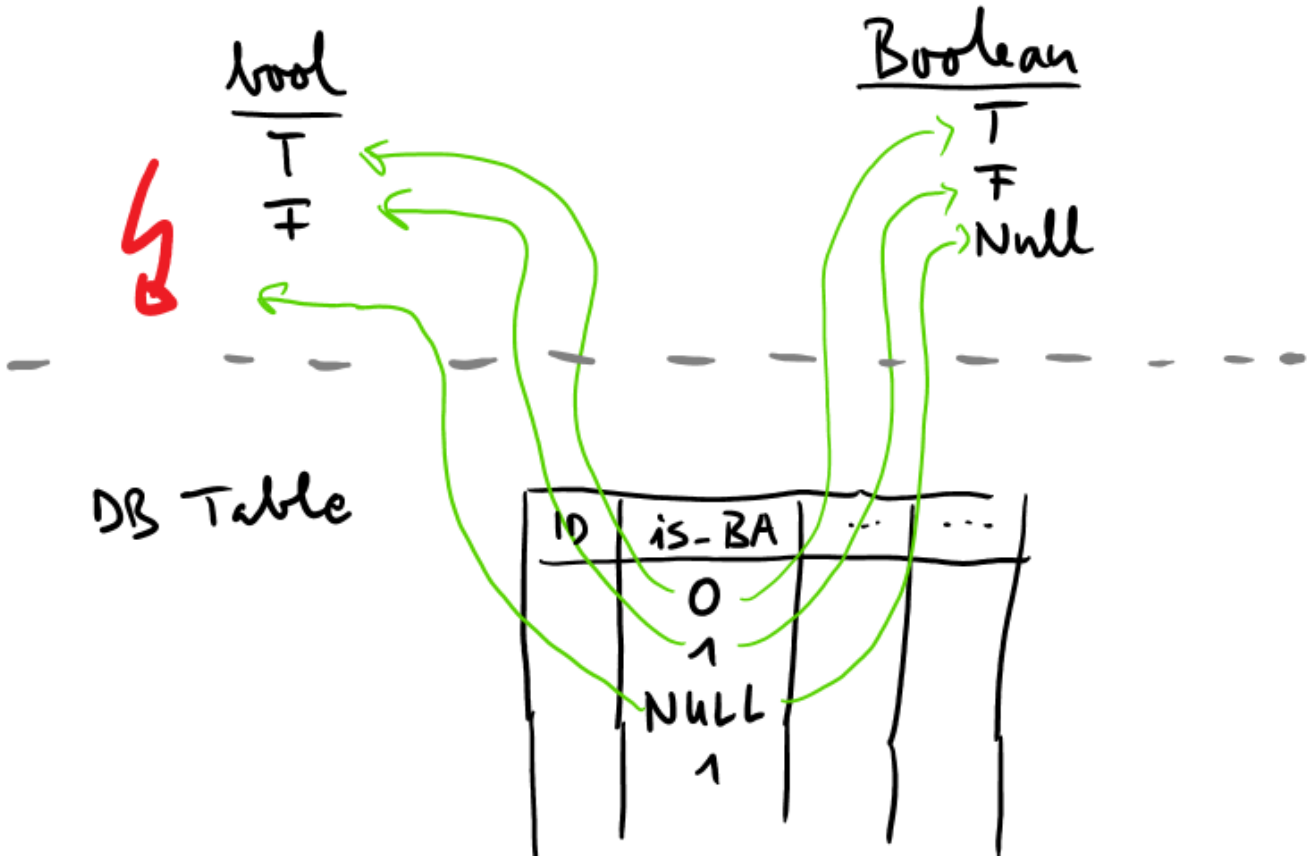
3. Attribute als Wrapperklassen mit domänen-spezifischer Validierung

4. Attribute als „Domain Primitive“ Value Objects
   - Erlaubt Builder / Konverter, spezifische Validations / Exceptions, domänenspezische Funktionalität

5. ~~Attribute als native *Domain Primitives*~~
   - Voller IDE-Support, so wie native Datentypen, aber domänen-spezifisches Verhalten

zumindest in Java leider nicht ver-fügbar, ggfs. in moderneren Sprachen

Option 1) Warum keine „primitive data types“ in Entities?



- Bei einer NULLABLE DB Column kann eine NULL nicht auf einen primitiven Datentypen abgebildet werden, auf einen Wrapper-Typen aber schon (siehe Beispiel).
- Generell sind Frameworks wie Spring eher auf den Umgang mit Objekten ausgelegt; die Wrapperklassen sind damit konsistenter zu benutzen.

Wichtigste 2 Vorteile

1. Viel Basisfunktionalität kann zentral getestet werden
   - ganz oft sind die Ursache von komplexen Bugs ganz simple Fehler
   - Beispiel:
     - Zuordnung eines Warenlagers über die erste stelle der Postleitzahl
     - wenn das über den Code verstreut ist, findet man einen Fehler schwer
     - => kann in „Postleitzahl“-DP verkapselt werden!

2. Ein DP ist immer valide, wenn es instanziiert wurde
   - Java: muss nur auf **`null`** getestet werden
     - (andere Sprachen: noch nicht mal das …)
   - man spart sich Quantillionen von if-Abfragen im Code

Do‘s & Don‘ts mit Domain Primitives

(bezogen auf unser Praktikum)

Don‘ts (macht man manchmal unbedacht falsch …)

```
public class ShoppingCart {
    @EmbeddedId
    @Setter(AccessLevel.PRIVATE)     // only for JPA
    private ShoppingCartId id;

    @Embedded
    private CustomerId customerId;

    @Embedded
    private MoneyType totalPrice;
```

```
org.springframework.orm.jpa.JpaSystemException: Cannot
instantiate abstract class or interface
'thkoeln.archilab.ecommerce.usecases.domainprimitivetypes.
MoneyType'
```

Do‘s (bitte machen, rentiert sich)

- **Adapter-Pattern**

```java
@Service
public class ShoppingCartUseCasesAdapterService implements ShoppingCartUseCases {
    private final ShoppingCartService shoppingCartService;

    @Autowired
    public ShoppingCartUseCasesAdapterService( ShoppingCartService shoppingCartService ) {
        this.shoppingCartService = shoppingCartService;
    }

    @Override
    public void addProductToShoppingCart( MailAddressType customerMailAddress, UUID productId, int quantityFor ) {
        shoppingCartService.addProductToShoppingCart(
                (MailAddress) customerMailAddress, new ProductId( productId ), quantityFor );
    }
```

- damit hält man „fremde“ Typen (hier: das Interface) aus dem eigenen Code raus und spart sich einen Haufen von Casting im eigenen Code
- (PS: warum kann ich hier einfach casten? Und wieso funktioniert das in anderen Codebasen nicht?)

Do‘s (bitte machen, rentiert sich)

- **Ausgiebiges Unit-Testen von Domain Primitives**
  - damit kann man einfach sicher sein, dass keine Basis-Operationen schief gehen
  - schafft ein hohes Maß an „Grundvertrauen“ in den eigenen Code

- Siehe Aufgabe E2

Generelles zum Code

(unsortiert)

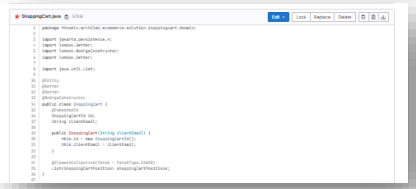
Was passiert hier?

```
return basket.getParts().stream()  Stream<ShoppingBasketPart>
        .mapToInt( ShoppingBasketPart part -> part.getQuantity())  IntStream
        .sum();


    stockLevels.stream()  Stream<StockLevel>
            .filter( StockLevel sl -> sl.getMerchandiseId().equals(merchandiseId))
            .findFirst()  Optional<StockLevel>
            .ifPresentOrElse(
                    StockLevel sl -> sl.setQuantity(sl.getQuantity() + quantity),
                    () -> stockLevels.add(new StockLevel(merchandiseId, quantity))
            );
}
```

- ChatGPT und CoPilot lieben Lambda-Expressions
  - … aber die sind schwer zu verstehen
  - … und noch schwerer zu debuggen

Domain-Klasse und Service



- Man tut sich keinen Gefallen, wenn man zu viel Business-Logik in den Service packt
  - Domain-Klassen erlauben Reuse!