Rich Domain Model mit JPA 2.0

Adrian Hummel
Consultant
adrian.hummel@trivadis.com

Mischa Kölliker
Principal Consultant
mischa.koelliker@trivadis.com

Java Forum Stuttgart, 1. Juli 2010

trivadis
makes IT easier.

Ein Vorzeige-Domänenmodell

… und wie es dann oft in der Praxis aussieht

Rich Domain Model Konzepte

Knackpunkte bei der Umsetzung

Lösung mit JPA 2.0

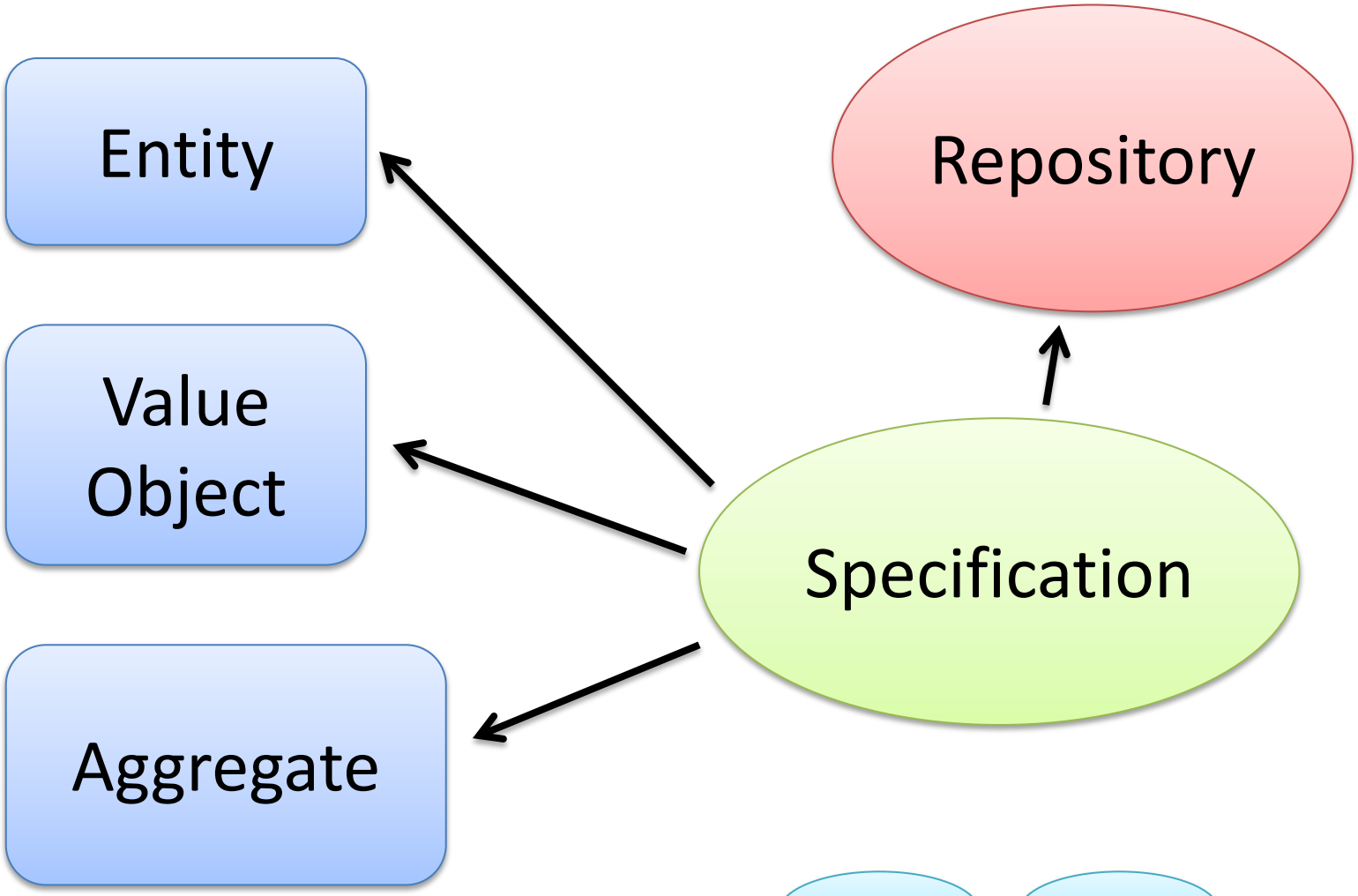
Wer wir sind

- Adrian Hummel
  - Java Consultant
  - JPA & Modelling-Enthusiast

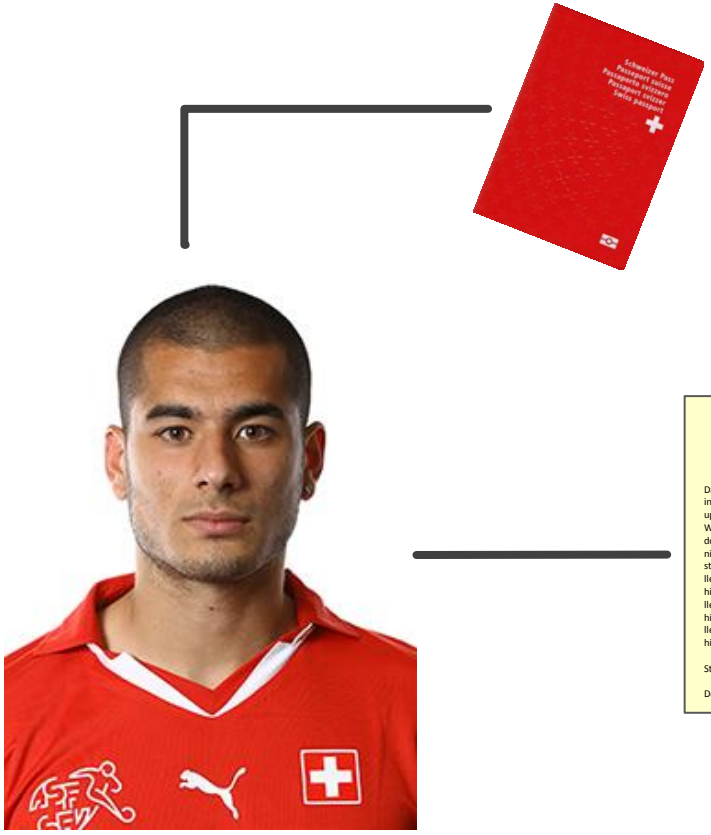
- Mischa Kölliker
  - Java & SOA Consultant
  - Buchautor
  - Architektur, Konzeption

- Trivadis AG
  - Enterprise-Lösungen mit Java-, Oracle- und Microsoft-Technologien
  - 550 Mitarbeiter an 11 Standorten in DACH

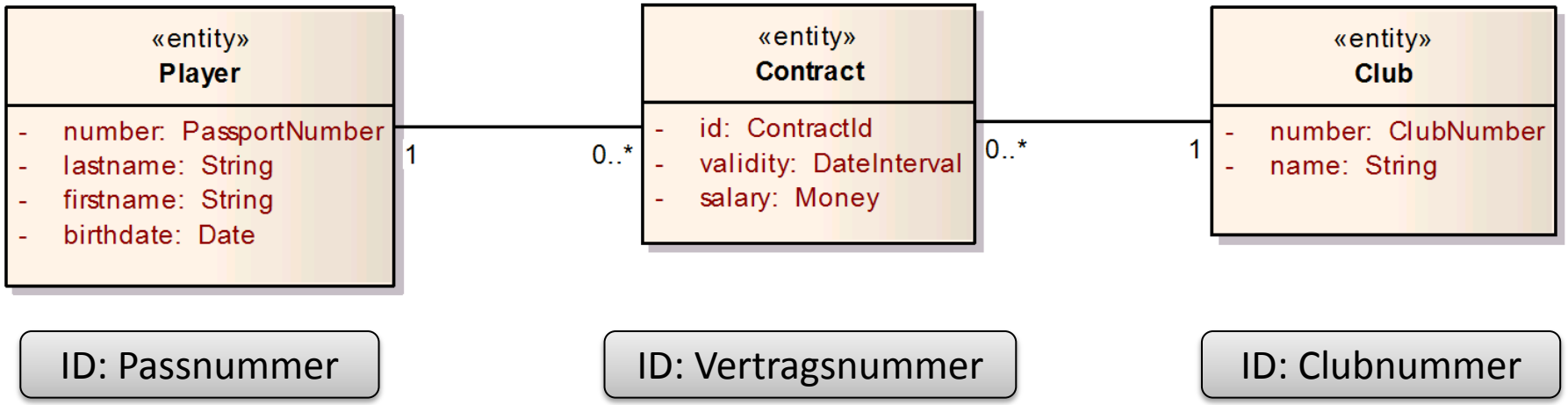
Rich Domain Model - Konzepte



Aus aktuellem Anlass: Die Beispieldomäne



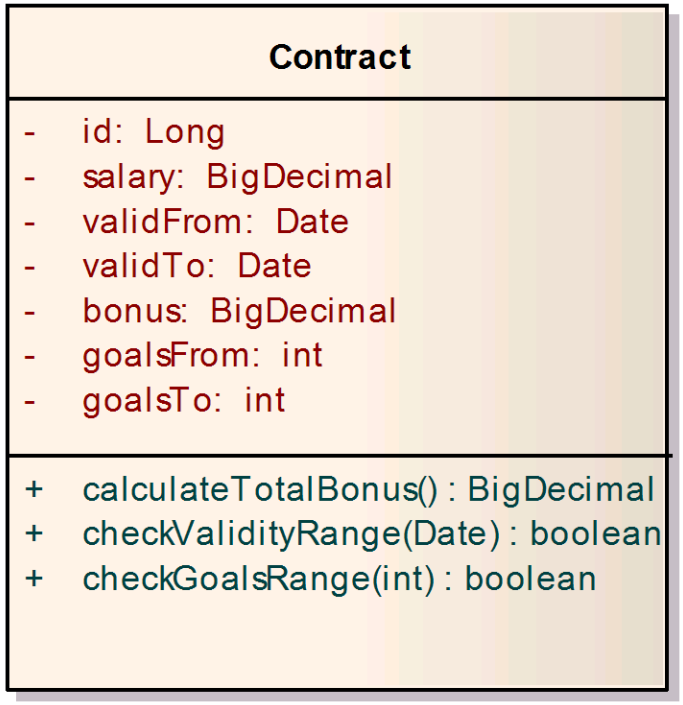
Entities – Haben eine fachliche Identität



Value Objects – Der simple Ansatz

Gültigkeitsbereich

betrifft Bonus

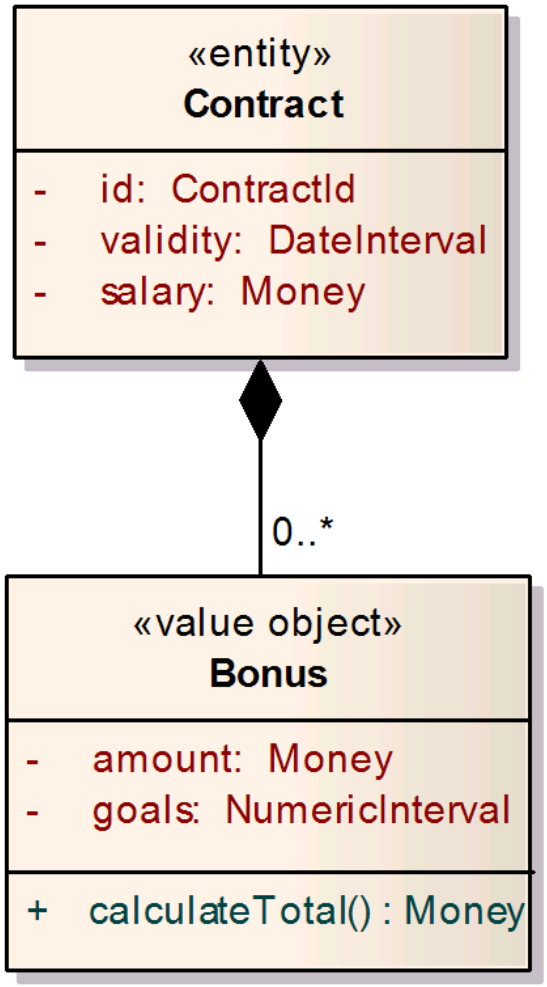


„Contract“ ist hier für alles mögliche zuständig

Value Objects – Attribute einer Entität

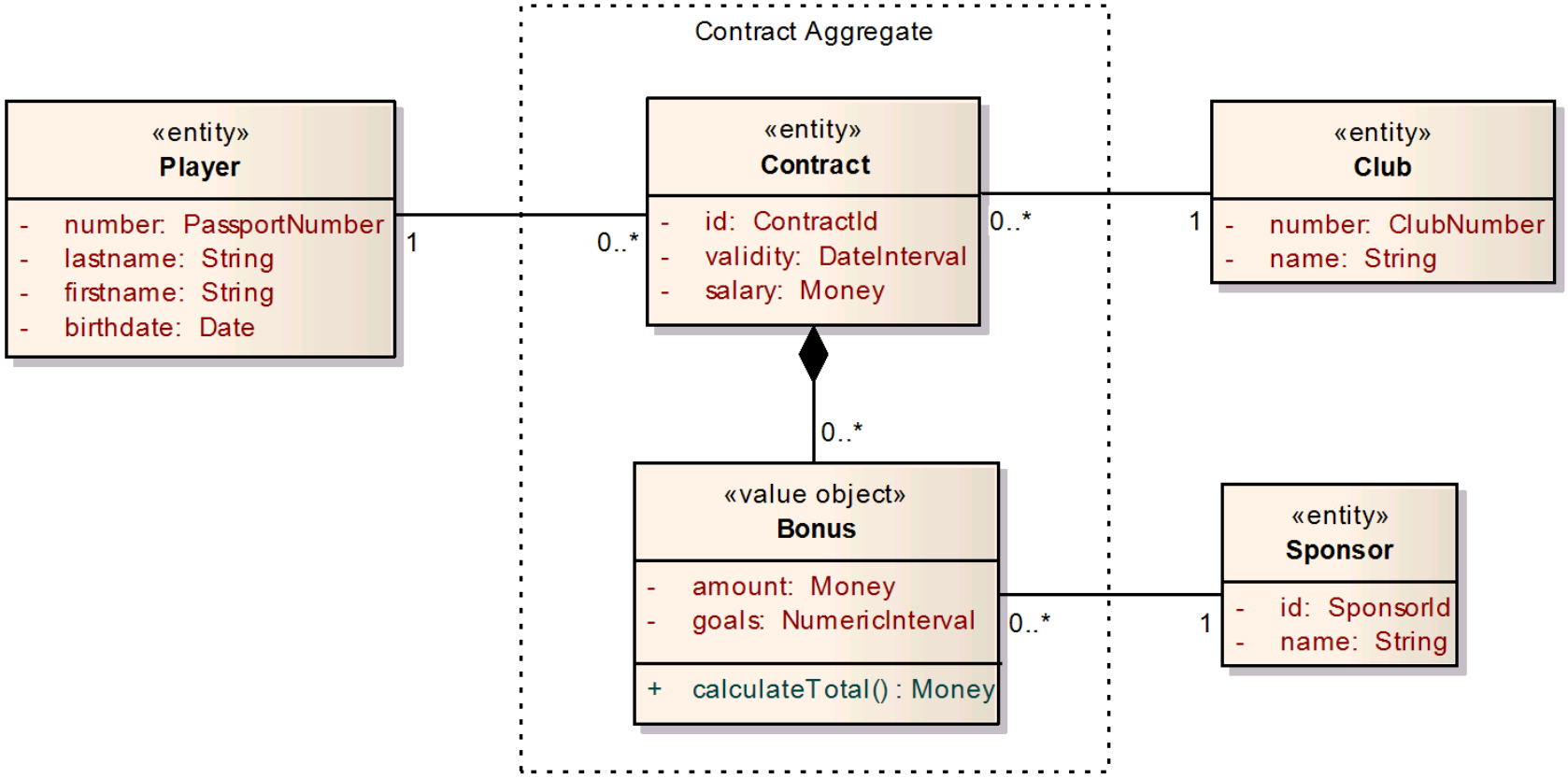
Keine eigene Identität

Haben typischerweise eigenes Verhalten



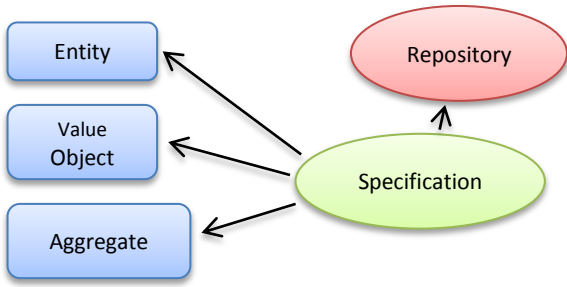
Aggregates – Verantwortlichkeiten regeln

Set von zusammengehörenden Objekten



Repository: Verwaltung der Entitäten

- Repräsentiert Collection von Domänenobjekten
  - Add, Find, Update, Remove
- Abstrahiert den Datenbank-Zugriff
- Repository ⇔ EntityManager
  - Repository Interface definiert klare Schnittstelle, vereinfacht Testing
  - Wiederverwendbare Funktionalität (z.B. komplexe Finder Methoden)



Repository: Finder-Methoden sind unflexibel

Beispiel:

`findCheapPlayer()`

`findAvailablePlayer()`

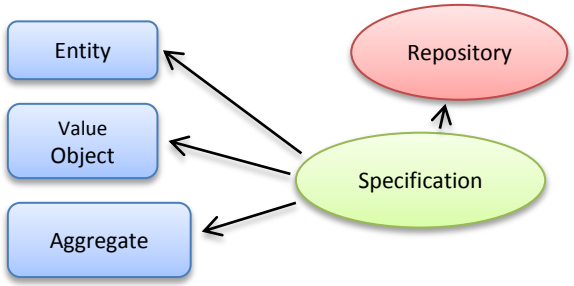
→ Neue Anforderung:

`findCheapAvailablePlayer()`

Specification – Implizite Konzepte explizit

Aufgaben

- Selektiere Daten
- Validiere Domänenobjekte
- Erstelle Objekte



Knackpunkte bei der Umsetzung

#1 Komplexe Value Objects

#2 Aggregat-Konsistenz

#3 Specifications in Repositories

Knackpunkte bei der Umsetzung

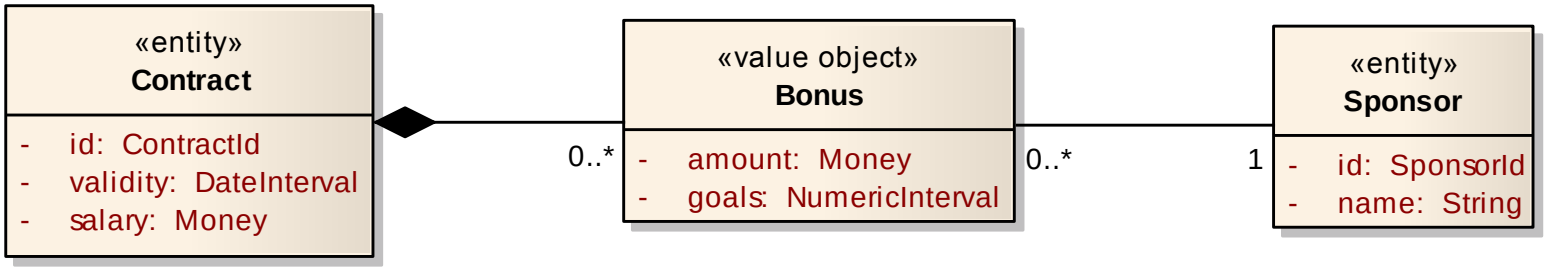
#1 Komplexe Value Objects

#2 Aggregat-Konsistenz

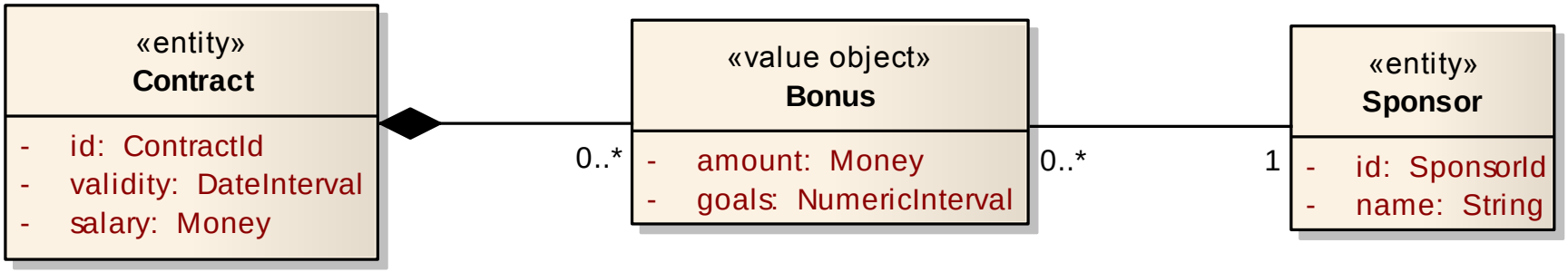
#3 Specifications in Repositories

Komplexe Value Objects abbilden

- Value Objects nennt man in JPA Embeddables
- Einschränkungen mit JPA 1.0
  - Keine verschachtelten Embeddables
  - Keine Collections
  - Keine Referenzen auf Entitäten
- Wie bilden wir dieses Modell ab?



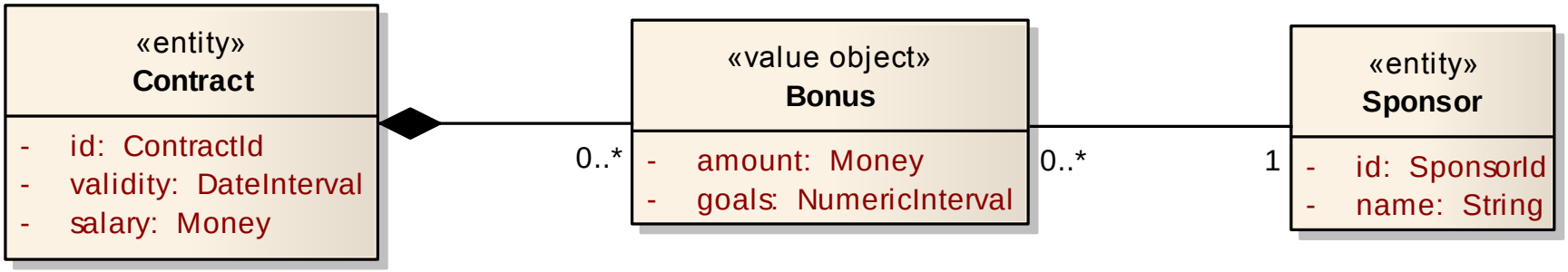
JPA 2.0 liefert Antworten



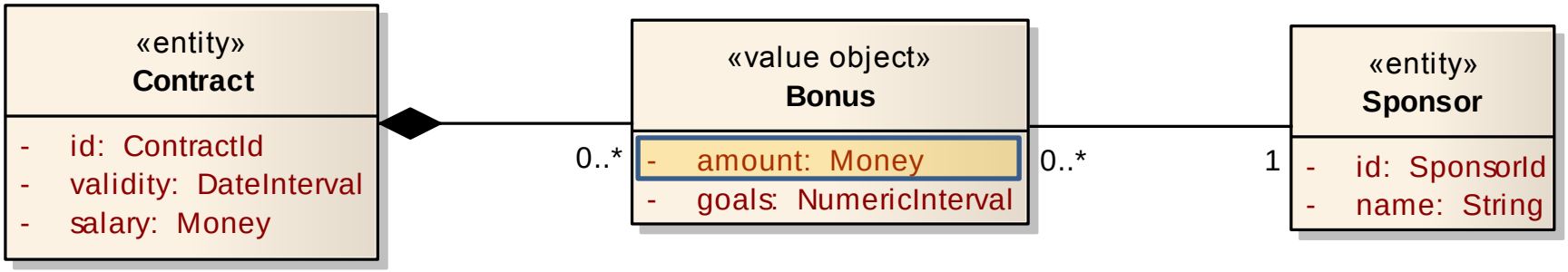

```java
@Entity
@Table(name="t_contract")
public class Contract {
  @ElementCollection
  @CollectionTable(name="t_bonus")
  private Set<Bonus> bonuses;
  // ...
}
```

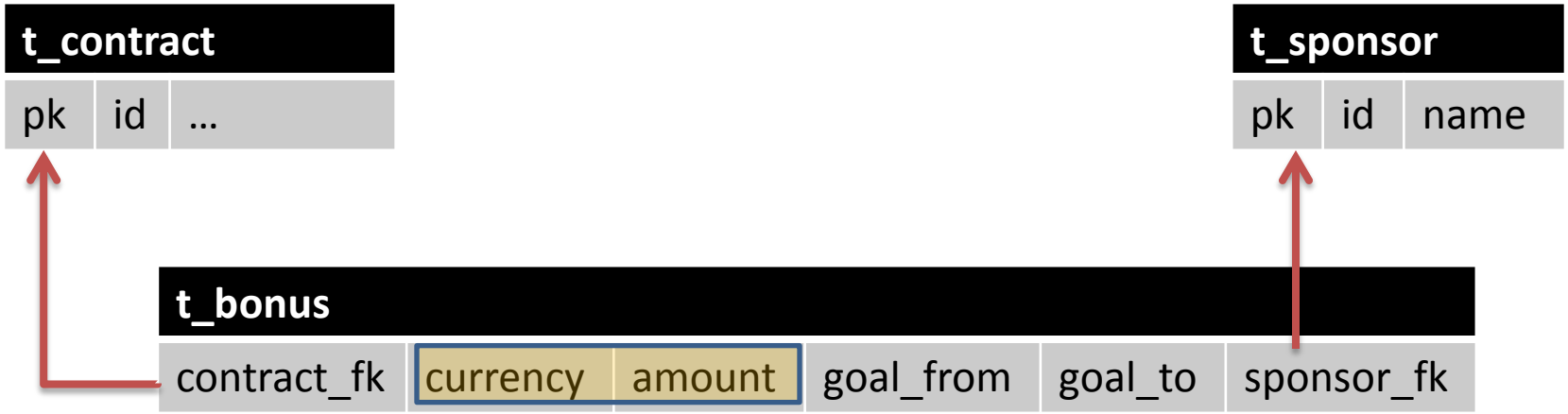
```java
@Embeddable
public class Bonus {
  @Embedded
  private Money amount;
  @Embedded
  private NumericInterval goals;
  @ManyToOne
  private Sponsor sponsor;
  // ...
}
```

Embeddables aus DB-Sicht



Embeddables aus DB-Sicht





Knackpunkte bei der Umsetzung

#1 Komplexe Value Objects 

#2 Aggregat-Konsistenz

#3 Specifications in Repositories

Aggregat–Konsistenz bewahren

- Aggregat (Root) verantwortlich für Konsistenz
  - Kapselung der Daten → API (fachlich motiviert)
  - Validierungen

- JPA 1.0
  - Programmatorische Validierungen
  - Fehleranfällige Templates/Callbacks
  - Keine Integration mit anderen Schichten

Validierung mit JPA 2.0

- Bean Validation 1.0 (Teil von Java EE 6)
- Integration mit JPA 2.0
  - Validierung bei persist, merge, remove

```
@Entity
public class Contract {
  @Valid
  @BonusContinuity
  private Set<Bonus> bonuses;
}
```

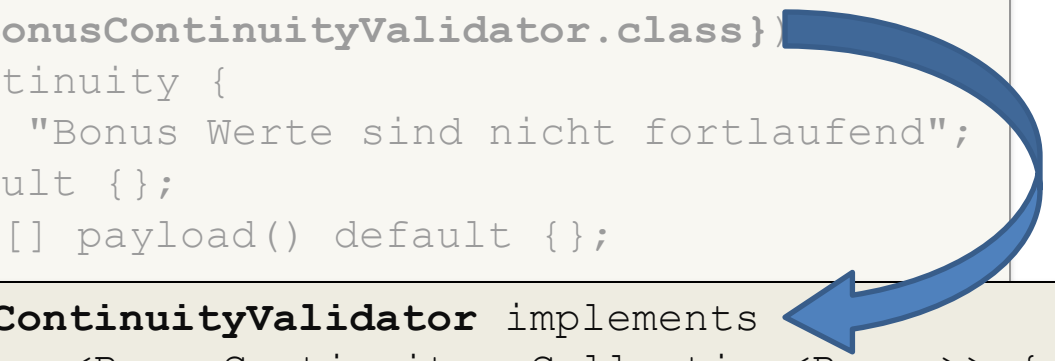
- Integration mit JSF 2.0

Eigene Constraints und Validatoren

```
@Target(FIELD)
@Retention(RUNTIME)
@Constraint(validatedBy={BonusContinuityValidator.class})
public @interface BonusContinuity {
  String message() default "Bonus Werte sind nicht fortlaufend";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Eigene Constraints und Validatoren

```
@Target(FIELD)
@Retention(RUNTIME)
@Constraint(validatedBy={BonusContinuityValidator.class})
public @interface BonusContinuity {
  String message() default "Bonus Werte sind nicht fortlaufend";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```



```
public class BonusContinuityValidator implements
  ConstraintValidator<BonusContinuity, Collection<Bonus>> {

  @Override
  public boolean isValid(Collection<Bonus> bonuses,
    ConstraintValidatorContext context) {

    return isValidBonusContinuity(bonuses);
  }

  @Override
  public void initialize(BonusContinuity annotation) {}
}
```

Knackpunkte bei der Umsetzung

#1 Komplexe Value Objects 

#2 Aggregat-Konsistenz 

#3 Specifications in Repositories

Specifications als *findByXyz* Ersatz

- Teilaufgabe von Specifications
  - Selektiere Objekte aus Collection (= Repository)
- Limitationen in JPA 1.0
  - Keine Unterstützung für dynamische Queries
  - Keine typisierten Queries

Criteria API in JPA 2.0

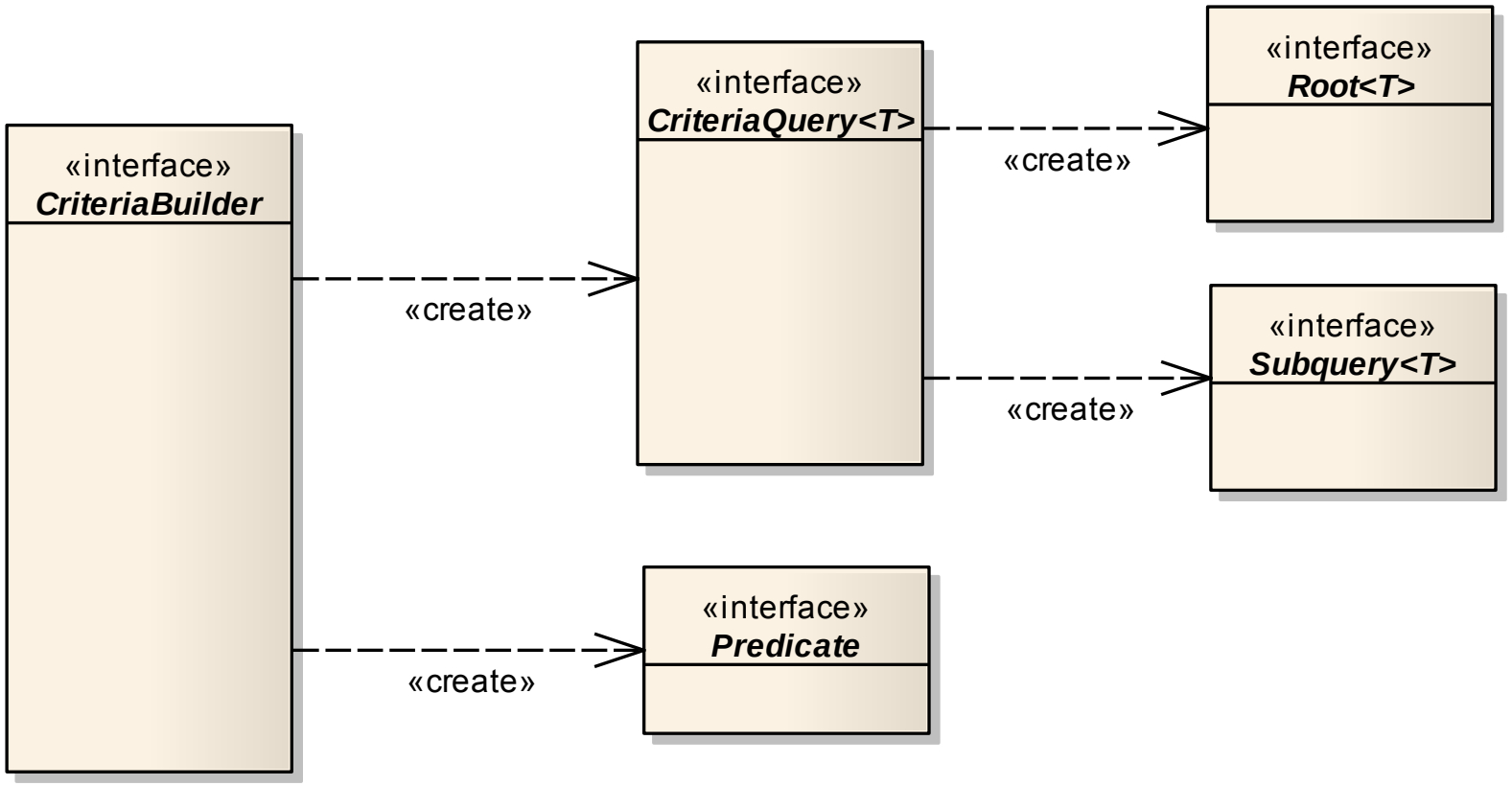
- Objektorientiertes JPQL
- Stark typisiert – basierend auf Metamodell

```
List<Player> list = em.createQuery(
   q.where(cb.equal(p.get(Player_.name), "Derdiyok")))
    .getResultList();
```

- String-basierte Navigation möglich

```
List<Player> list = em.createQuery(
   q.where(cb.equal(p.get("name"), "Derdiyok")))
    .getResultList();
```

Wichtige Criteria API Konzepte



Wichtige Criteria API Konzepte

- CriteriaBuilder
  - Factory für CriteriaQuery, Selektionen, Prädikate und Sortierungen
- CriteriaQuery
  - Query Manipulationen
    - from, groupBy, having, select, subquery, where
- Predicate
  - Boolean Ausdrücke (and, or, not Verknüpfungen)
  - Specifications repräsentieren ebenfalls Prädikate

Criteria API Beispiel – Finde faire Spieler

- JPQL

```
SELECT p FROM Player p
 WHERE (SELECT count(c) FROM Card c
         WHERE c.player = p AND c.season = 2009) < 5
```

- Criteria API

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Player> q = cb.createQuery(Player.class);
Root<Player> player = q.from(Player.class);

Subquery<Long> sq = q.subquery(Long.class);
Root<Card> card = sq.from(Card.class);
sq.where(cb.and(cb.equal(card.get(Card_.player), player),
                cb.equal(card.get(Card_.season), 2009)));

q.where(cb.lt(sq.select(cb.count(card)), 5));
List<Player> players = em.createQuery(q).getResultList();
```

Repository – Finde faire Spieler

```
public List<Player> findFairPlayers() {

  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Player> q = cb.createQuery(Player.class);
  Root<Player> player = q.from(Player.class);

  Subquery<Long> sq = q.subquery(Long.class);
  Root<Card> card = sq.from(Card.class);
  sq.where(cb.and(cb.equal(card.get(Card_.player), player),
                  cb.equal(card.get(Card_.season), 2009)));

  q.where(cb.lt(sq.select(cb.count(card)), 5));
  return em.createQuery(q).getResultList();
}
```

Repository – Identifiziere Specification

```
public List<Player> findFairPlayers() {

  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Player> q = cb.createQuery(Player.class);
  Root<Player> player = q.from(Player.class);

  Subquery<Long> sq = q.subquery(Long.class);
  Root<Card> card = sq.from(Card.class);
  sq.where(cb.and(cb.equal(card.get(Card_.player), player),
                  cb.equal(card.get(Card_.season), 2009)));

  q.where(cb.lt(sq.select(cb.count(card)), 5));
  return em.createQuery(q).getResultList();
}
```

FairPlayerSpecification

Repository – Delegiere an Specification

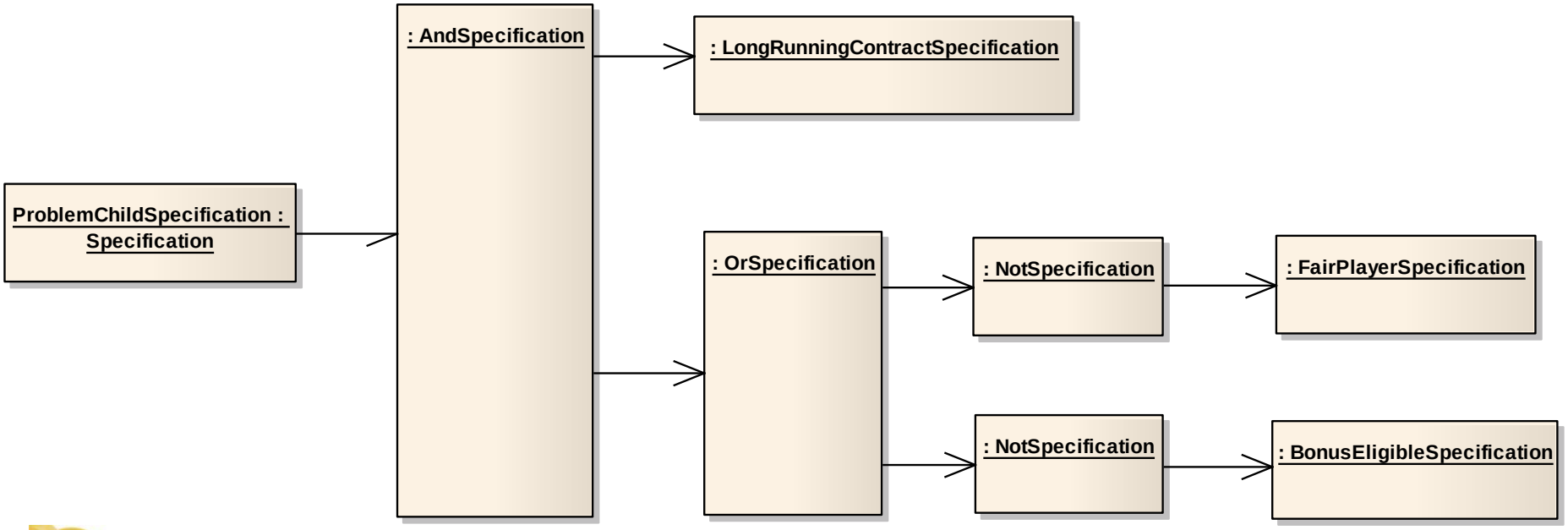
```
public List<Player> findAll(Specification<Player> specification) {

  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Player> q = cb.createQuery(Player.class);
  Root<Player> player = q.from(Player.class);

  q.where(specification.toPredicate(cb, q, player));

  return em.createQuery(q).getResultList();
}
```

Implementiere die Specification

```
public class FairPlayerSpecification implements Specification<Player> {

  @Override
  public Predicate toPredicate(CriteriaBuilder cb,
    CriteriaQuery<Player> q, Root<Player> player) {

    Subquery<Long> sq = q.subquery(Long.class);
    Root<Card> card = sq.from(Card.class);

    sq.where(cb.and(cb.equal(card.get(Card_.player), player),
                    cb.equal(card.get(Card_.season), 2009)));

    return cb.lt(sq.select(cb.count(card)), 5);
  }

  //...
}
```

Kompositionen – Die wahre Stärke

- Finde alle „Problem-Spieler“
  - Unfair ODER Nicht bonus-relevant UND
  - Lang-laufender Vertrag



Kompositionen – Repository bleibt stabil

```
public List<Player> findAll(Specification<Player> specification) {

  CriteriaBuilder cb = em.getCriteriaBuilder();
  CriteriaQuery<Player> q = cb.createQuery(Player.class);
  Root<Player> player = q.from(Player.class);

  q.where(specification.toPredicate(cb, q, player));

  return em.createQuery(q).getResultList();
}
```

Knackpunkte bei der Umsetzung

#1 Komplexe Value Objects 

#2 Aggregat-Konsistenz 

#3 Specifications in Repositories 

Fazit: Vorsprung durch JPA 2.0

- Komplexe Embeddables
  - Collections
  - Verschachtelung
  - Referenzen auf Entitäten
- Integration mit Bean Validation 1.0
  - Constraints 1x definieren, überall validieren
- Specifications mit Criteria API
  - Selektierung von Objekten in Repository
  - Composite Specifications möglich

Mit JPA 2.0 haben wir unser Ziel erreicht…

…und sind gewappnet für komplexe Modelle!