



Technische Hochschule
Ingolstadt

Fakultät für Elektrotechnik
und Informatik

*Zukunft in
Bewegung*

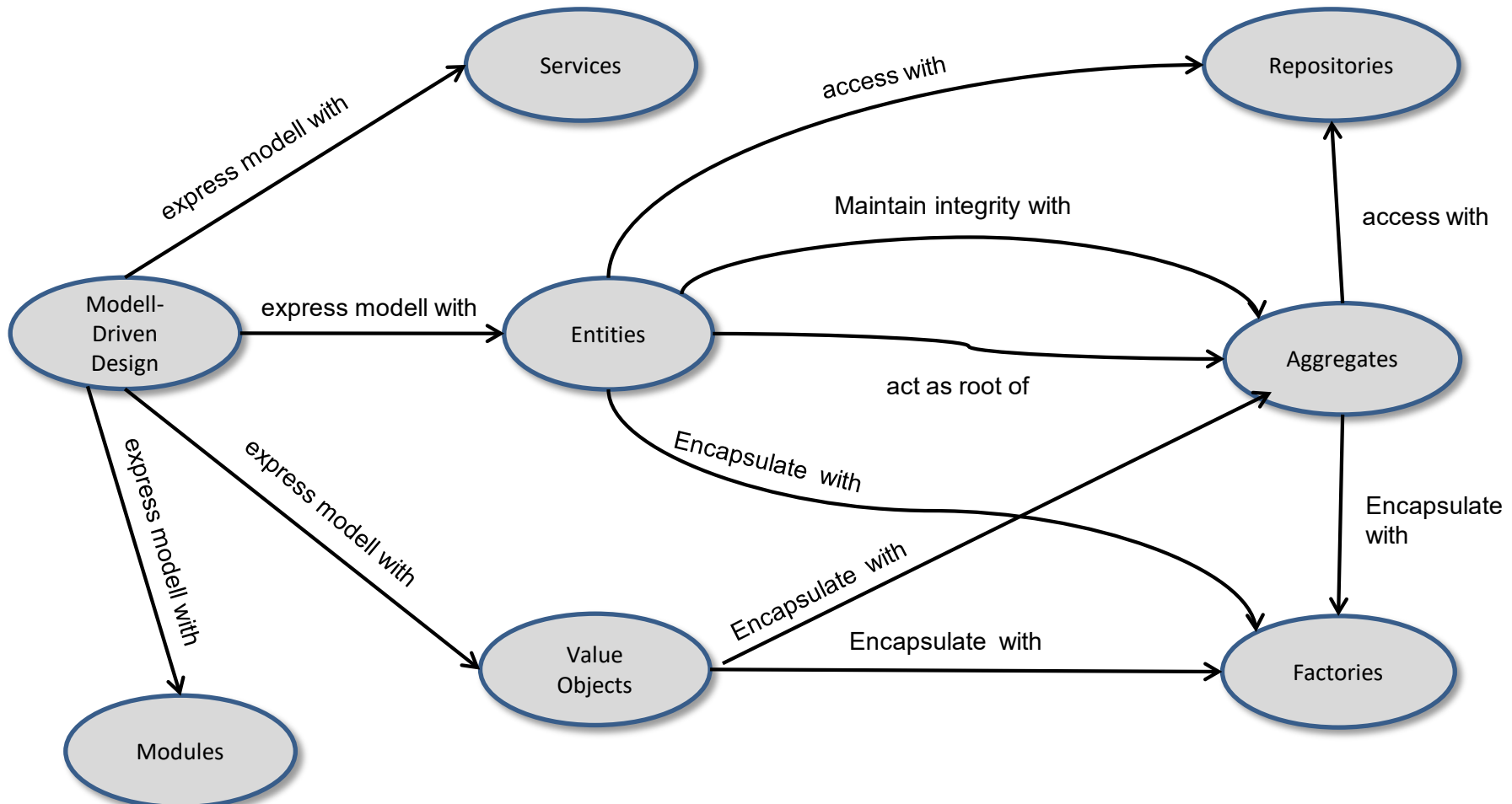
Domain-Driven Design

*Architektur- und Entwurfsmuster
der Softwaretechnik*

Prof. Dr. Bernd Hafenrichter 06.03.2015

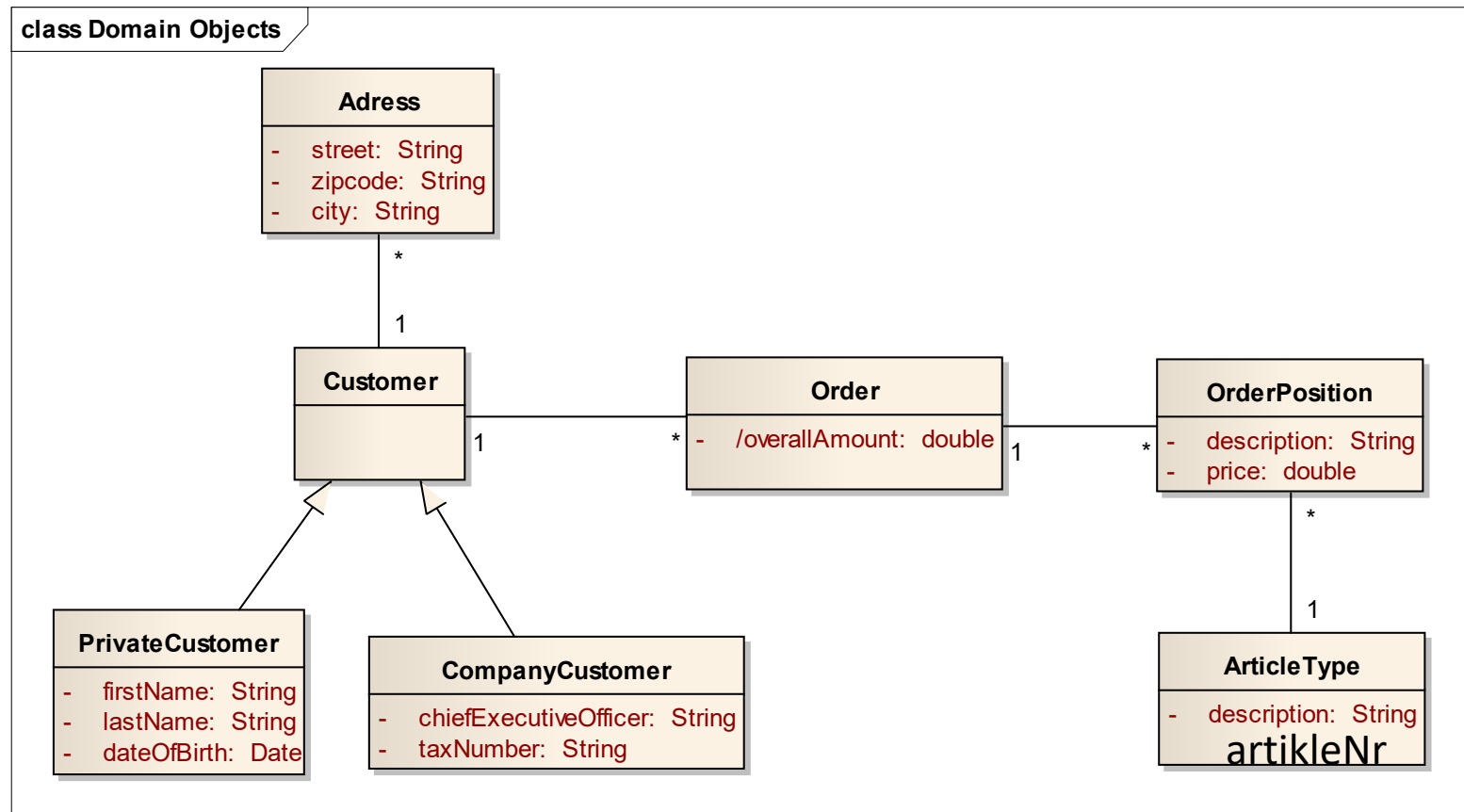


Die Grundelemente des Modell Driven Deisgn

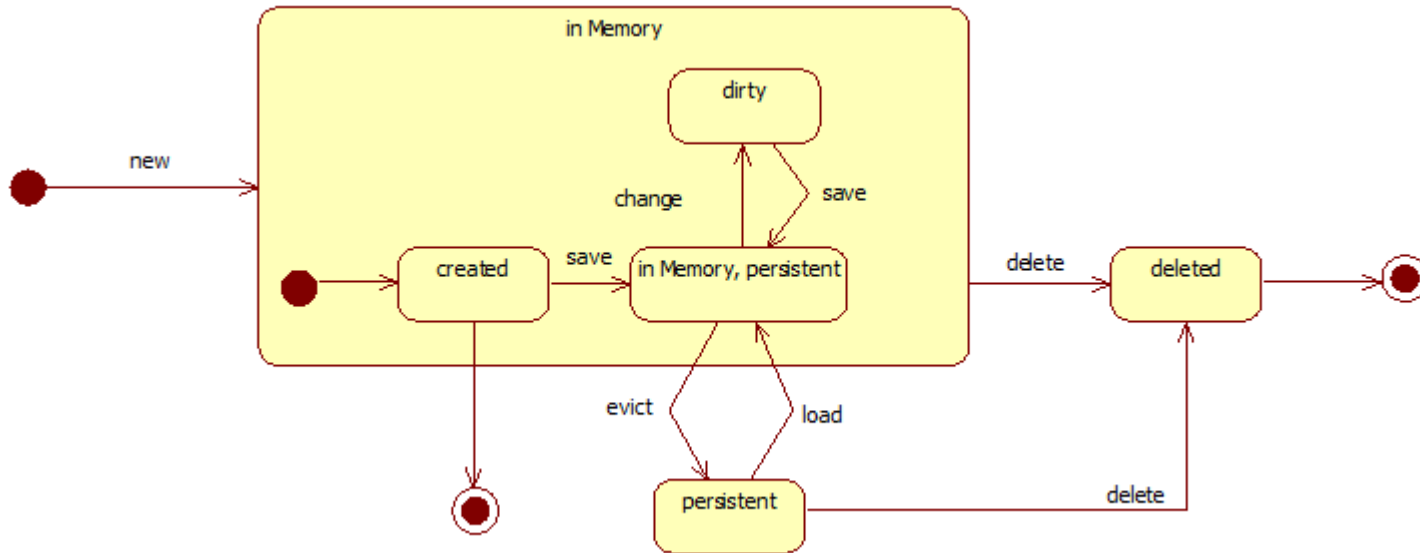


Ausgangspunkt: Das Domänenmodell

Wie können Klassen des Domänenmodells persistiert werden?



Lebenszyklus eines Domänenobjekts



Motivation:

- Domänenobjekte existieren nicht nur im Hauptspeicher sondern auch in der Datenbank. Sie haben unterschiedliche Repräsentationen
- Der Lebenszyklus, wie ihn eine Programmiersprache definiert, ist für Domänenobjekte zu klein gefasst
- Eine Persistenzschicht muss die unterschiedlichen Lebenszyklen zwischen den Domänenobjekten (Assoziationen) berücksichtigen

Persistenz von Domänenobjekten

Separates Speichern
jedes einzelnen
Objektes



Speichern des
kompletten
Objektgraphen

Navigation über Persistence-Layer(-)
Viele Lade/Speicher-Vorgänge (-)
Effizienter im Speicherverbrauch (+)

Navigation über Assoziationen(+)
Ein Lade/Speicher-Vorgang (+)
Hohe Speicherverbrauch (-)

Idee:

- Finde Gruppen von Objekten, die als eine Einheit für Lade und Speichervorgänge verwendet werden können (= gleicher Lebenszyklus)
- Mache Gruppen voneinander unabhängig, wenn der Lebenszyklus unterschiedlich ist
- Pro Gruppe existiert eine explizite RootEntity, über welche die Gruppe extern identifiziert werden kann

Schritte zur Implementierung einer Persistenzschicht

Schritt 1: Welche Elemente des Domänenmodells müssen persistiert werden?

- Definition von Entitäten und Value-Objects

Schritt 2: Finden von zusammenhängenden Einheit welche als ganzes gespeichert und geladen werden

- Definition von Aggregaten

Schritt 3: Implementierung der Zugriffsoperationen pro Aggregat

Entities



Ein Objekt welches nicht ausschließlich durch seine Attribute beschrieben wird sondern primär durch seine Identität wird Entity genannt

Design of Entities

- Über den Lebenszyklus hinweg kann sich der Inhalt/Form stark ändern.
- Über die Identität können die Entities jederzeit nachvollzogen und zugeordnet werden
- Die Identität eines Domänen-Objektes ist nicht mit der Identität innerhalb der Programmiersprache zu verwechseln
- Die Identität ist explizit zu machen
- Es ist eine Funktion zu definieren welche für jedes Objekt eine eindeutige Identität erzeugt

Value Objects



Value
Object

Ein Objekt welches einen beschreibenden Aspekt der Domäne enthält wird „Value Object“ genannt, wenn es keine eigene konzeptuelle Identität besitzt.

Design of Value-Objects

- Unterschiedliche Objekte können sich das gleiche Value-Object (Instanz) teilen.
- Value-Object's sind als immutable zu definieren. Dadurch werden negative Seiteneffekte aufgrund von gemeinsamer Verwendung ausgeschlossen



Value Objects

Spezialfall: Mutability of Value-Objects

- Folgende Gründe können dafür sprechen eine Value-Object nicht als immutable zu implementieren
 - Der Wert ändert sich sehr häufig
 - Objekt Erzeugung/Löschung ist sehr teuer
 - Probleme bei verteilten Applikationen
 - Value-Objects werden selten gemeinsam (shared) benutzt




Aggregates

Motivation:

„Es ist schwierig die Konsistenz innerhalb eines Modells zu garantieren insbesondere dann wenn komplexe Assoziationen existieren. Es müssen Invarianten für zusammenhängende Objekte sichergestellt werden. Darüberhinaus besteht die Notwendigkeit Zusammenhängende Objekte vor gemeinsamen Zugriff zu schützen“

Aggregates



Aggregat

- Eine Gruppe von zusammenhängenden Objekten die als eine Einheit für den Datenaustausch verwendet werden.
- Das Aggregat kapselt alle Referenzen der beteiligten Objekte
- Jedes Aggregat hat einen ausgezeichnete Root-Entity.
- Außenstehende Objekte dürfen nur auf die Root-Entity verweisen.
- Alle Entities (ausser der Root) haben eine lokale Identität welche nur innerhalb des Aggregats eindeutig sein muss.
- Das Aggregat sorgt für die Einhaltung von Invarianten wenn immer sich ein Objekt innerhalb des Aggregats ändert.



Aggregates (continued)

Design of aggregates:

- Die Root-Entity hat eine globale Identität und ist für die Prüfung von Invarianten zuständig
- Entities innerhalb des Aggregats haben eine lokale Identität, welche nur innerhalb des Aggregates eindeutig sein muss
- Ausschließlich die Root-Entity darf von außen referenziert werden.
- Die Root-Entity darf Referenzen auf interne Objekte herausgeben. Diese sind aber nur temporärer Natur (transient)
- → Nur die Root-Entität kann über eine Query ermittelt werden. Alle anderen Objekte über eine Traversierung von Assoziationen

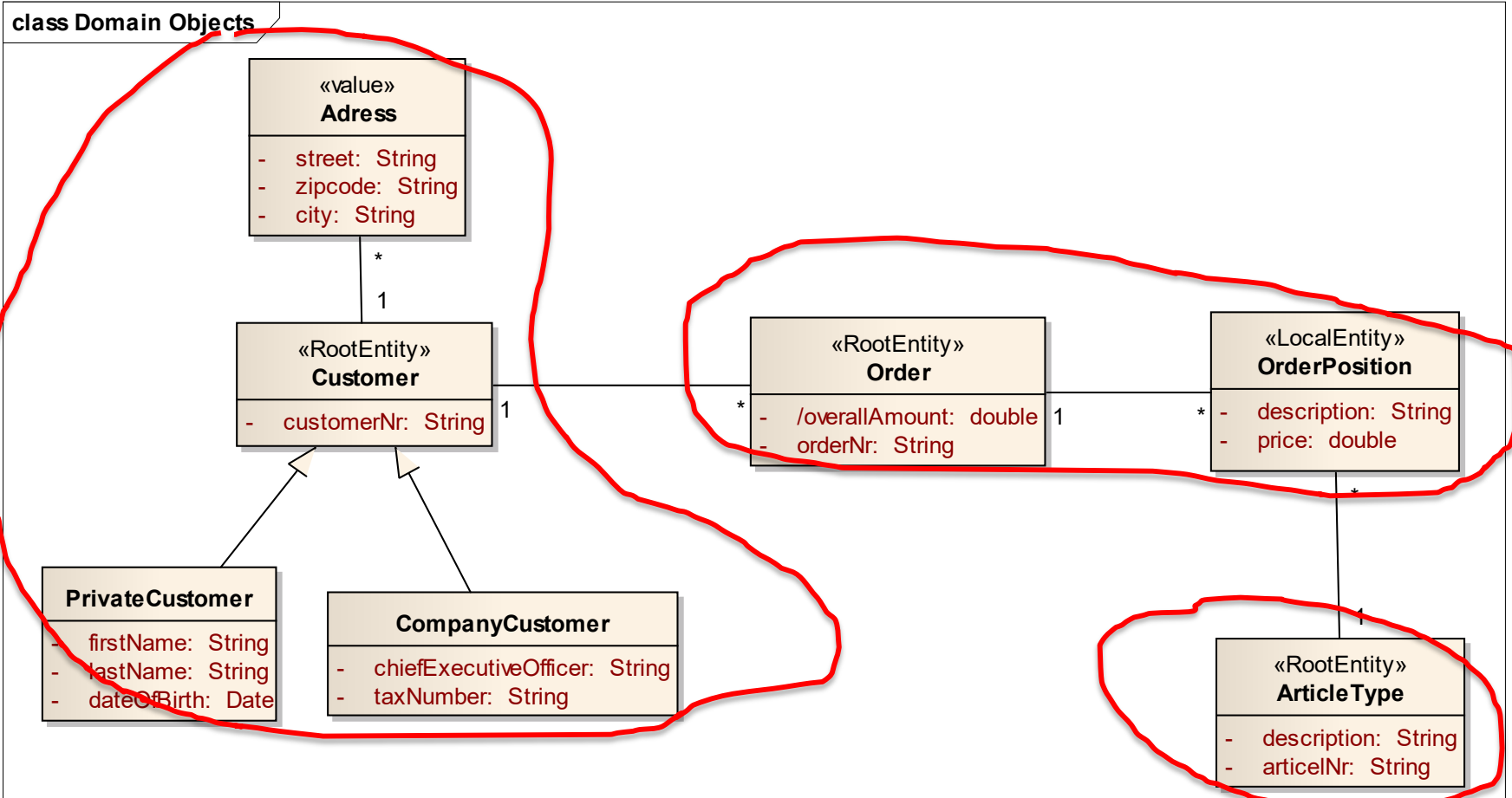


Aggregates (continued)

Design of aggregates:

- Objekte innerhalb des Aggregats können andere Aggregate referenzieren
- Ein Löschoperation muss alles innerhalb eines Aggregates auf einmal löschen
- Wenn ein Objekt innerhalb des Aggregates verändert wird (committed) müssen alle Invarianten des Aggregates erfüllt sein.

Zerlegen des Objekt-Modells in Aggregate



Achtung: Eine Root-Entity muss eine extern wahrnehmbare Identität haben



Assoziationen

Die Abbildung von Assoziationen zwischen Objekten ist schwierig.
Grundsätzlich gilt:

„For every traversable association in the model, there is a mechanism in the software with the same properties“

Implementierung von Assoziationen

- Direkt mit Hilfe von Objektreferenzen
- Indirekt mit Hilfe von Fremdschlüsselreferenzen
- Indirekt mit Hilfe von Lookup-Methoden



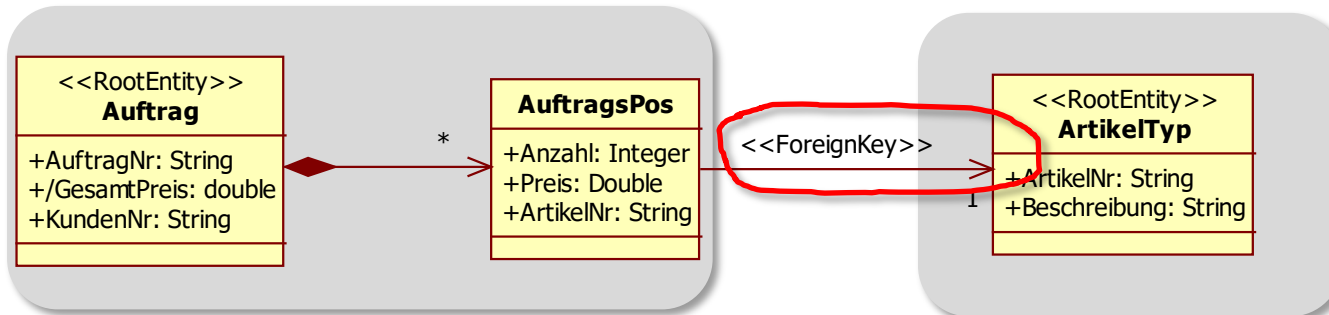
Assoziationen (continued)

„Es ist wichtig Assoziationen so einfach wie möglich zu gestalten. Nur dadurch kann die Komplexität des Modells reduziert werden“

Design of associations

- Definiere für jede Assoziation die Navigationsrichtung
- Reduktion der Komplexität mit Hilfe von Qualifieren (Zugriff über Schlüssel)
- Eliminieren von Überflüssigen Assoziationen

Assoziationen zwischen Aggregaten



```
public class AuftragPos {  
    private ArtikelTyp artZyp;  
}
```

- (-) Lebenszyklus von AuftragPos und ArtikelTyp sind voneinander abhängig.
- (+) Einfach zu programmieren, Objektnavigation möglich
- (+) Geeignet innerhalb eines Aggregats

```
public class AuftragPos {  
    private String artZyp;  
}
```

- (+) Lebenszyklus von AuftragPos und ArtikelTyp sind unabhängig.
- (-) Aufwändiger zu programmieren, Navigation über Zugriff auf Persistence-Layer
- (+) Geeignet zwischen Aggregaten



Factories

Factories

- Die Erstellung eines Aggregates/Objektes kann sehr komplex sein bzw. viel internes Wissen benötigen (Kopplung).
- Factories sind ein Mittel der Kapselung um die interne Objekterzeugung von einem Client zu verstecken.
- Dadurch ist eine bessere Trennung von Verantwortlichkeiten gewährleistet.



Factories

Design of Factories

- Jede Factormethode ist atomar und stellt die Einhaltung aller Invarianten des erzeugten Objektes bzw. Aggregates sicher.
 - Für eine Entity: Das gesamte Aggregat (ausser optionalen Elementen) wird erzeugt
 - Für ein Value-Object: Alle Attribute werden korrekt initialisiert
- Die Factory soll die erzeugten Typen von den tatsächlichen (Implementierungs-)Klassen extrahieren




Factories

Fälle in denen keine Factory verwendet werden sollte

- Die erzeugte Klasse selbst ist bereits der Typ. Keine relevante Vererbungshierarchie bzw. keine Polymorphie
- Der Client ist an der implementierenden Klasse interessiert
- Die Konstruktion ist nicht kompoliziert
- Achtung: Ein Konstruktor sollte ebenfalls darauf achten das alle Invarianten eingehalten sind.

Repositories

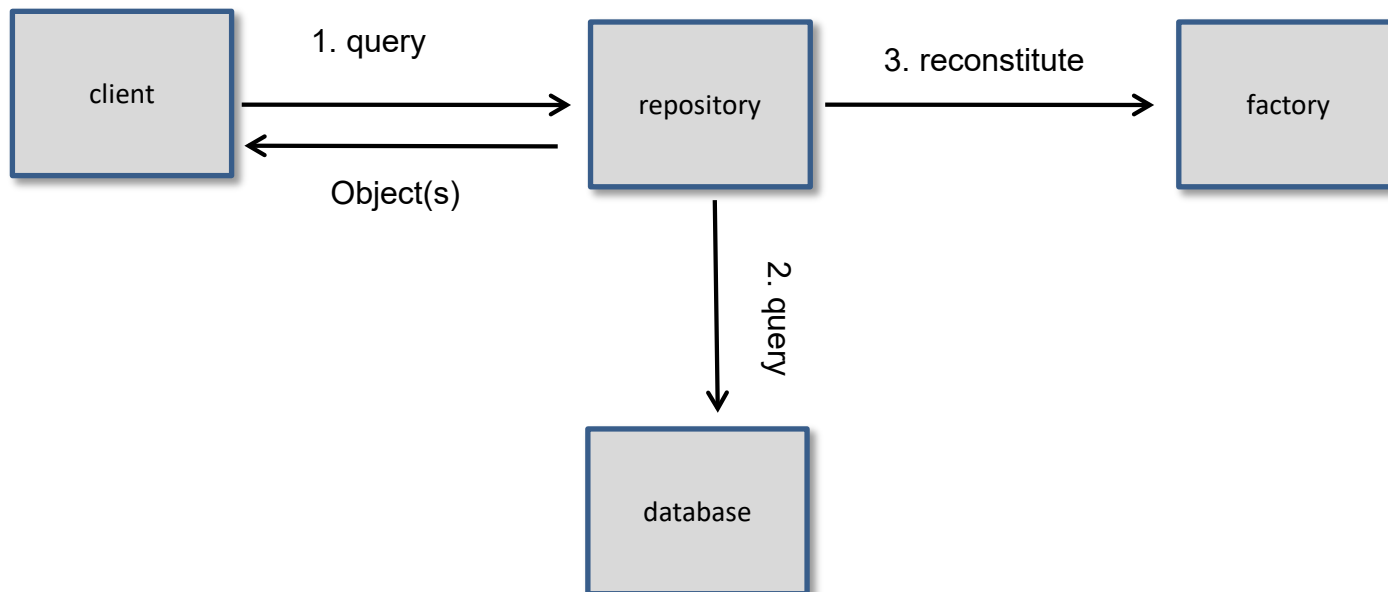


Repository

- Ein Repository ist ein Objekt welche eine Menge von Objekten (eines Typs) als konzeptuelles Set repräsentiert.
- Es realisiert eine virtuelle Collection welche jedoch auf einen Persistenzmechanismus abgebildet ist.
- Ein Repository stellt folgende Methoden zur Verfügung
 - Hinzufügen, Ändern und löschen
 - Suchen von Objekten
- Für den Benutzer eines Repositories soll die Illusion eines „In-Memory“-Speichers existieren
- Repositories sollten nur für Aggregat-Roots existieren welche einen direkten Zugriff benötigen
- Alle Art von Objekt-Speicher und Zugriff sollten an das Repository delegiert werden

Repositories

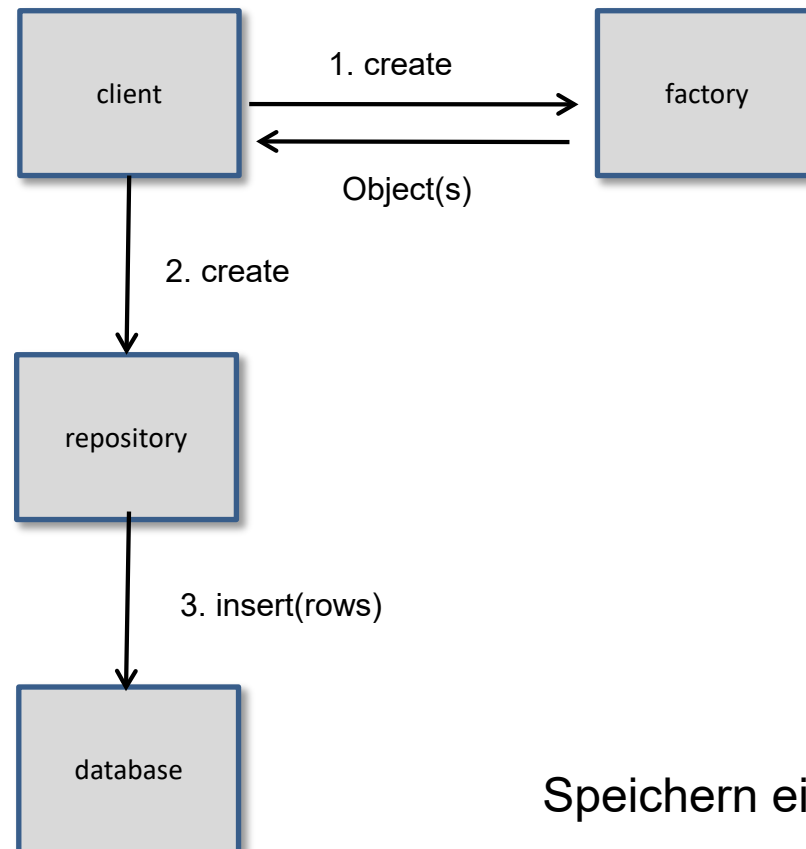
Zusammenspiel zwischen Factories & Repositories



Laden eines existierenden Objektes über die Datenbank

Repositories

Zusammenspiel zwischen Factories & Repositories



Speichern eines neuen Objektes



Repositories

Vorteile beim Einsatz eines Repositories

- Ein Repository stellt ein einfaches Modell für den Zugriff auf Persistente Objekte und deren Lebenszyklus dar
- Ein Repository stellt eine Trennung zwischen Applikations- und Domainlogik von der Persistenztechnologie
- Ein Repository stellt die Design-Entscheidungen über den Objekt-Zugriff explizit dar
- Durch ein Repository kann eine einfache Ersetzung durch eine Dummy-Implementierung bzw. andere Strategien vorgenommen werden.
- Ein Repositories ist selbst Teil der Domänenschicht und greift auf die Objekte der Persistenzschicht, welche meist mittels der Entwurfsmuster Data Access Objects, Query Objects oder Metadata Mapping Layers umgesetzt werden, zu.

Service

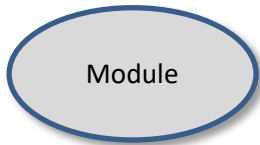


- Eine Operation welche für sich alleine innerhalb des Domänenmodells steht
- D.h. eine notwendige Operation welche aber nicht direkt einem Objekt (Entität/Value) zugeordnet werden kann

Design of Services

- Die Operation bezieht sich auf ein Konzept der Domäne welches nicht natürlicher Bestandteil einer Entität oder Value-Objekt ist
- Das Interface ist auf Basis des Domänenmodells definiert
- Die Operation ist stateless

Module



Problem

- Domänen Modelle können sehr groß werden
- Dadurch wird die Komplexität erhöht
- Dies führt zu Schwierigkeiten im Verständnis

Lösung

- Teilen Sie das Domänenmodell in fachliche Bestandteile auf
- Ein Modul soll eine hohe (fachliche) Kohäsion aufweisen. D.h. es sollen zusammengehörige Konzepte enthalten sein
- Geben Sie den Modulen Namen welche Bestandteil der allgemeinen Projektsprache werden. (z.B. Das Kundenmodul)
- Der Name eines Modules soll die Innensicht der Domäne reflektieren

Domain-Driven Design

Architektur- und Entwurfsmuster der Softwaretechnik

