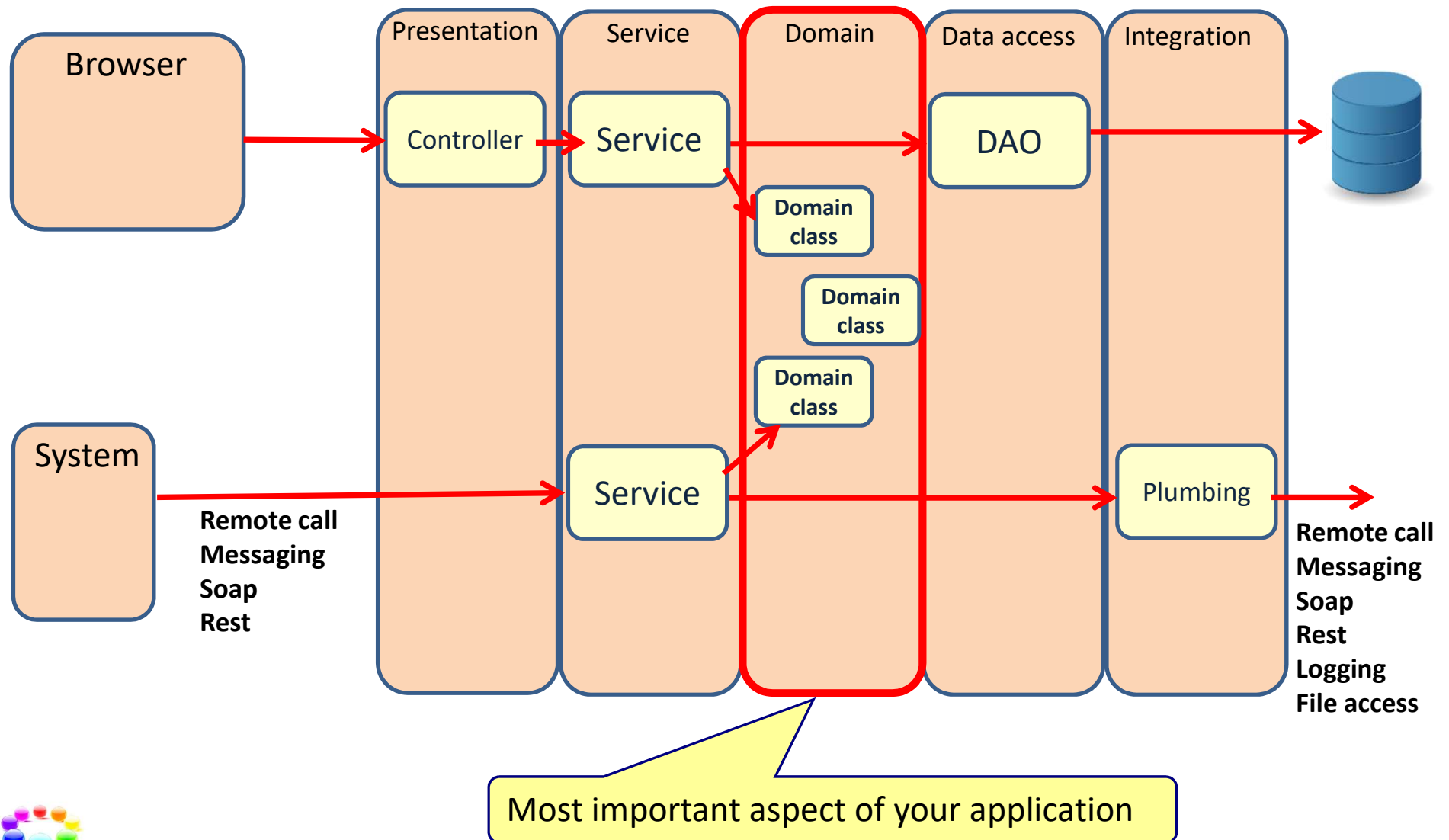


Lesson 3

DOMAIN DRIVEN DESIGN



Domain Driven Design



Building software

- Before you can start writing code you have to understand the domain first

The hardest single part of building a software system is deciding precisely what to build.

Fred Brooks - "No Silver Bullet" 1987

If you don't get the requirements right, it does not matter how well you do anything else.

Karl Wieggers



People use their own language

- Business process
- Business events
- Business rules
- Business structure



domain expert

- Objects
- Databases
- HTML
- SQL
- XML messages



developer

- Applications
- Components
- Protocols
- Platforms
- Tooling



architect



Business and IT working close together

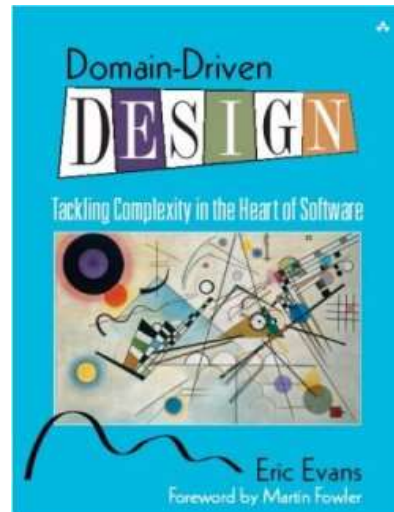
- Both use a different language
- The code/design often does not reflect the business
 - But reflects the developers view
- The business does not understand the developers view
 - A lot of translation need to be done



What is Domain Driven Design?

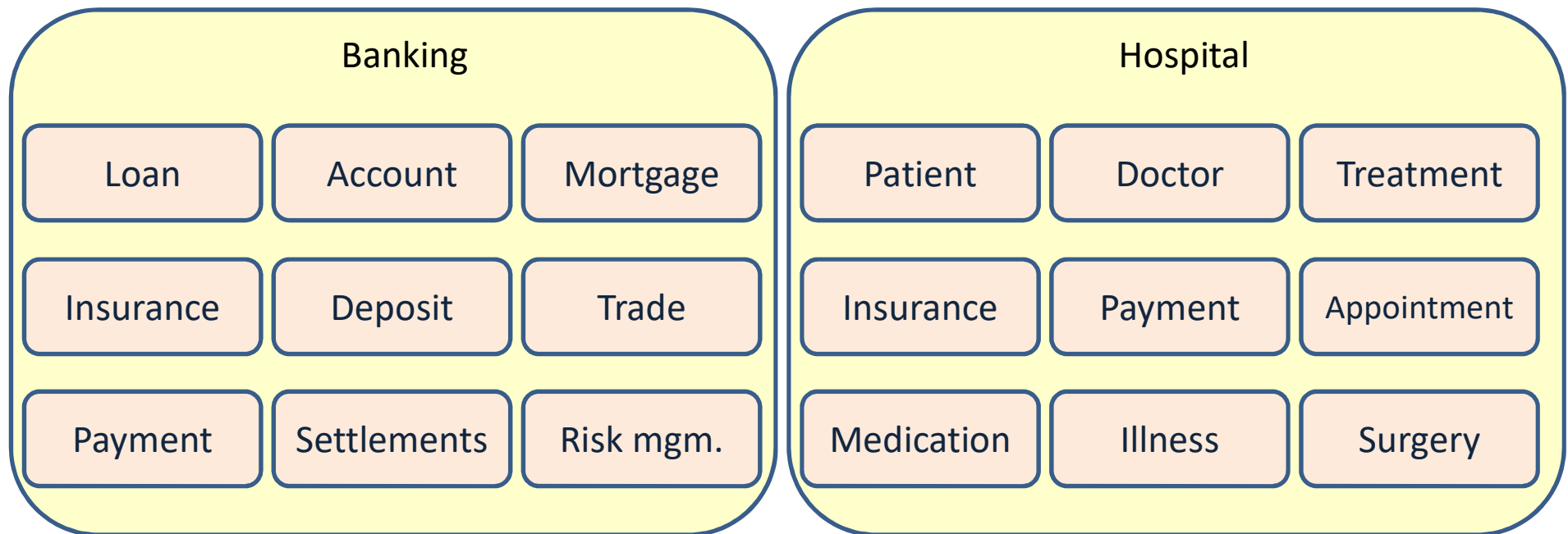
- An approach to software development where the focus is on the core **Domain**.
 - We create a **domain model** to communicate the domain

the idea of DDD is when you build something complex you have to put some effort whatever you put in the code should be in the language of business
 - Everything we do (discussions, design, coding, testing, documenting, etc.) is based on the domain model.



Domain

What a business does and the world it does it in.

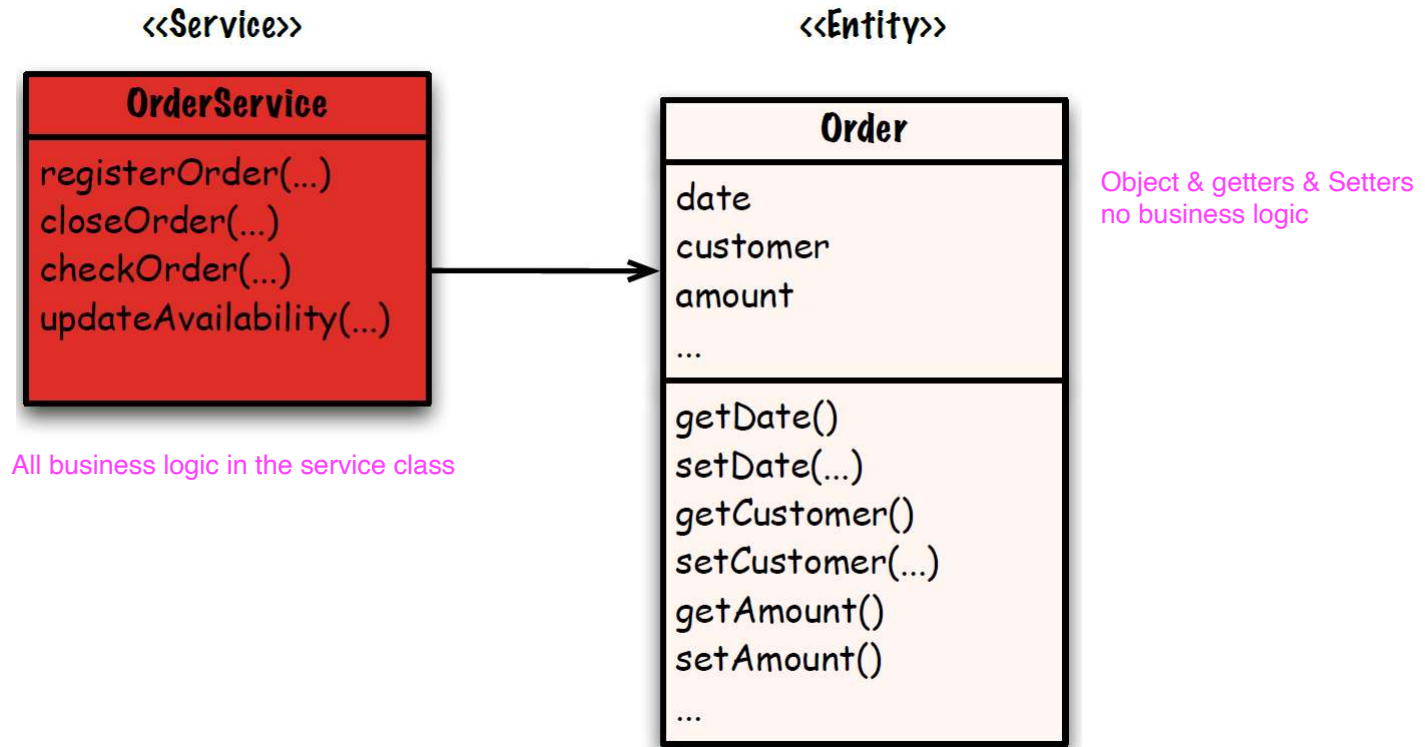


Anemic domain model

- Classes in the model have no business logic



NOT OK



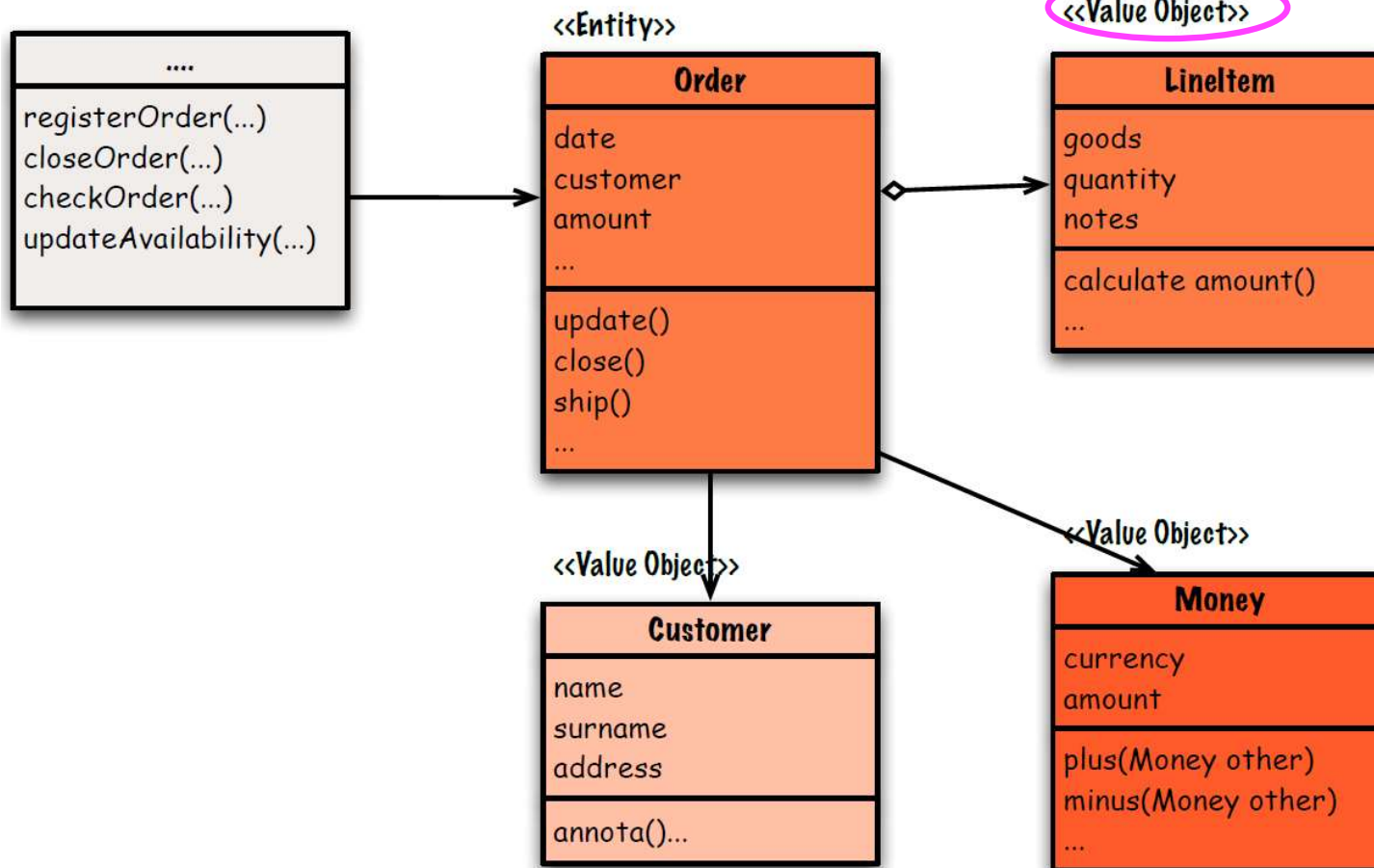
Disadvantages anemic domain model

- You do not use the powerful OO techniques to organize complex logic.
- Business logic (rules) is hard to find, understand, reuse, modify.
- The software reflects the data structure of the business, but not the behavioral organization
- The service classes become too complex
 - No single responsibility
 - No separation of concern



Rich domain model

- Classes with business logic





Domain Model Patterns

- **Entities** Classes that stores in DB <<Entity>>
- **Value objects** A Value Object is an immutable type that is distinguishable only by the state of its properties. That is, unlike an Entity, which has a unique identifier and remains distinct even if its properties are otherwise identical
- **Domain services** is used to perform domain operations and business rules.
- **Domain events** Classes that represent important events in the problem domain that have already happened
- **Aggregates** group of obj that belongs together
aggregates are basically small application by its own



ENTITIES



Entities

- A class with identity
- Mutable
 - State may change after instantiation
 - The entity has an lifecycle
 - The order is placed
 - The order is paid
 - The order is fulfilled



Example entity classes

Customer
+CustomerId
+firstName
+lastName
+email
+phone

Package
+trackingNumber
+weight
+type

Product
+productNumber
+name
+price



Entities

- Changing attributes doesn't change which one we're talking about
 - Identity remains constant throughout its lifetime



VALUE OBJECTS



Value objects

- Has no identity
 - Identity is based on composition of its values
- Immutable
 - State cannot be changed after instantiation



Example value object classes

Address
-street -city -zip
+computeDistance(Address a) +equals(Address a)

Money
-amount -currency
+add(Money m) +subtract(Money m) +equals(Money m)

Review
-nrOfStars -description

Weight
-value -unit
+add(Weight w) +subtract(Weight w) +equals(Weight w)

Dimension
-length -width -height
+add(Dimension d) +subtract(Dimension d) +equals(Dimension d)



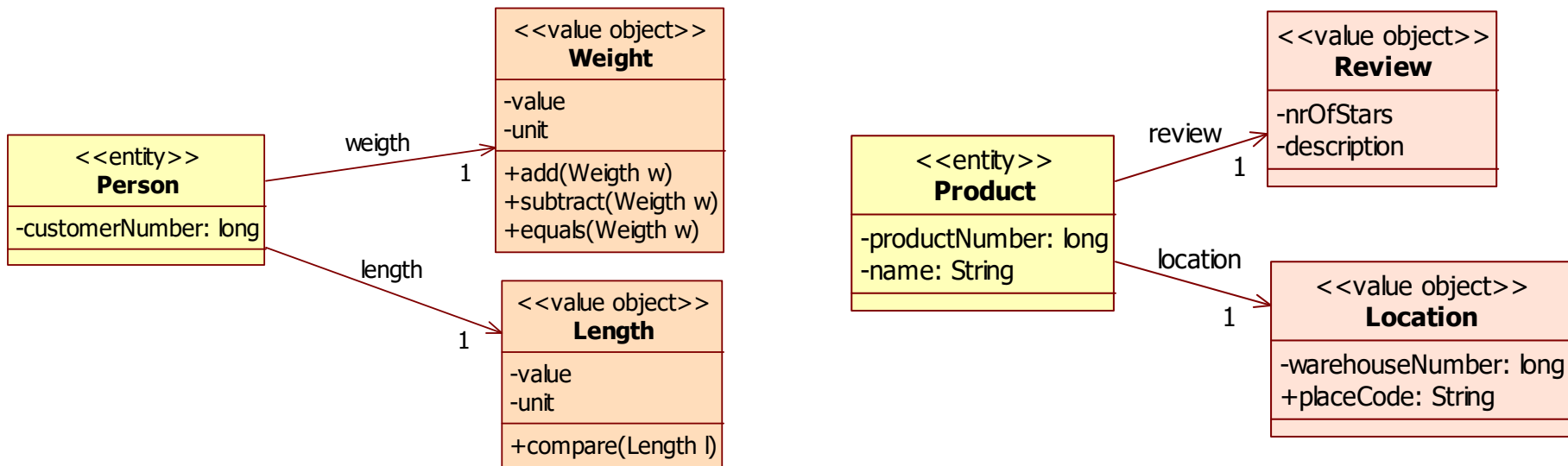
Value object characteristics

- No identity
- Attribute-based equality
- Behavior rich they are functionality
- Cohesive
- Immutable
- Combinable
- Self-validating
- Testable



No identity

- Value objects tell something about another object



- Technically, value objects may have IDs using some database persistence strategies.
- But they have no identity in the domain.



Attribute-based equality

- 2 value objects are equal if they have the same attribute values

<<value object>> Address
-street -city -zip
+computeDistance(Address a) +equals(Address a)

<<value object>> Money
-amount -currency
+add(Money m) +subtract(Money m) +equals(Money m)



Behavior rich

- Value objects should expose expressive domain-oriented behavior

<code><<value object>></code> Meters
-value: long
+toYards(): long +toKilometers(): long +isLongerThan(Meters m): boolean +isShorterThan(Meters m): boolean



Cohesive

Related attribute

- Encapsulate cohesive attributes

<<value object>> Money
-amount -currency
+add(Money m) +subtract(Money m) +equals(Money m)

<<value object>> Color
-red: int -green: int -blue: int
+equals(Color c)



Immutable

- Once created, a value object can never be changed

```
public class Money {  
    private BigDecimal value;
```

No setter methods

```
    public Money(BigDecimal value) {  
        this.value = value;  
    }
```

Mutation leads to the creation of new instances

```
    public Money add(Money money){  
        return new Money(value.add(money.getValue()));  
    }
```

```
    public Money subtract(Money money){  
        return new Money(value.subtract(money.getValue()));  
    }
```

```
    public BigDecimal getValue() {  
        return value;  
    }
```

```
}
```



Minimize Mutability

- Reasons to make a class immutable:
 - Less prone to errors
 - Easier to share
 - Thread safe
 - Combinable
 - Self-validating
 - Testable



Combinable

- Can often be combined to create new values

```
public class Money {  
    private BigDecimal value;  
  
    public Money(BigDecimal value) {  
        this.value = value;  
    }  
  
    public Money add(Money money){  
        return new Money(value.add(money.getValue()));  
    }  
  
    public Money subtract(Money money){  
        return new Money(value.subtract(money.getValue()));  
    }  
  
    public BigDecimal getValue() {  
        return value;  
    }  
}
```

Combine 2 Money instances



Self-validating

- Value objects should never be in an invalid state

```
public class Money {  
    private BigDecimal value;  
  
    public Money(BigDecimal value) {  
        validate(value);  
        this.value = value;  
    }  
  
    private void validate(BigDecimal value){  
        if (value.doubleValue() < 0)  
            throw new MoneyCannotBeANegativeValueException();  
    }  
  
    public Money add(Money money){  
        return new Money(value.add(money.getValue()));  
    }  
  
    public BigDecimal getValue() {  
        return value;  
    }  
}
```

Self-validation



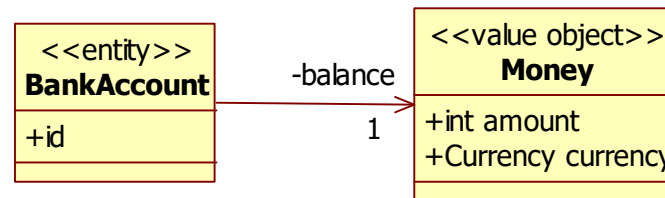
Testable

- Value objects are easy to test because of these qualities
 - Immutable
 - We don't need mocks to verify side effects
 - Cohesion
 - We can test the concept in isolation
 - Combinability
 - Allows to express the relationship between 2 value objects

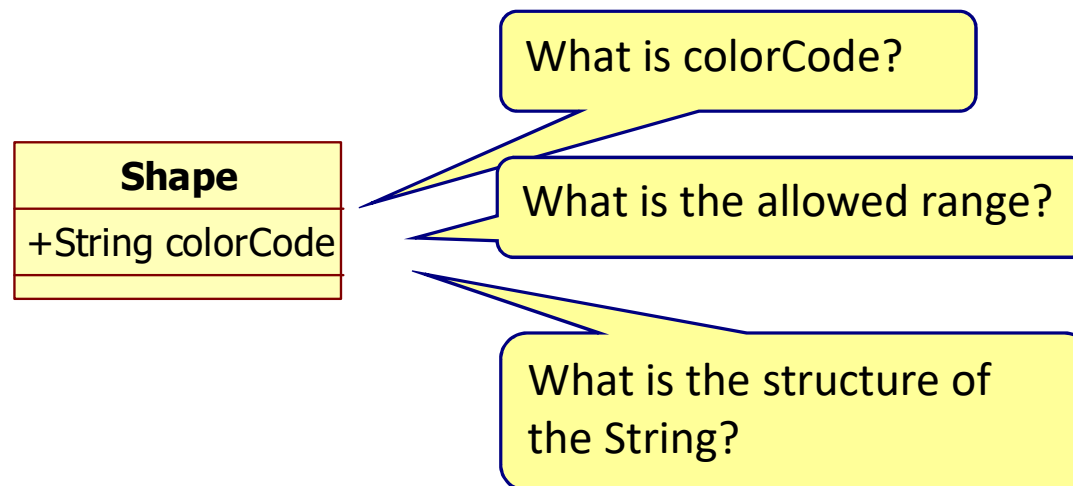


When to use value objects?

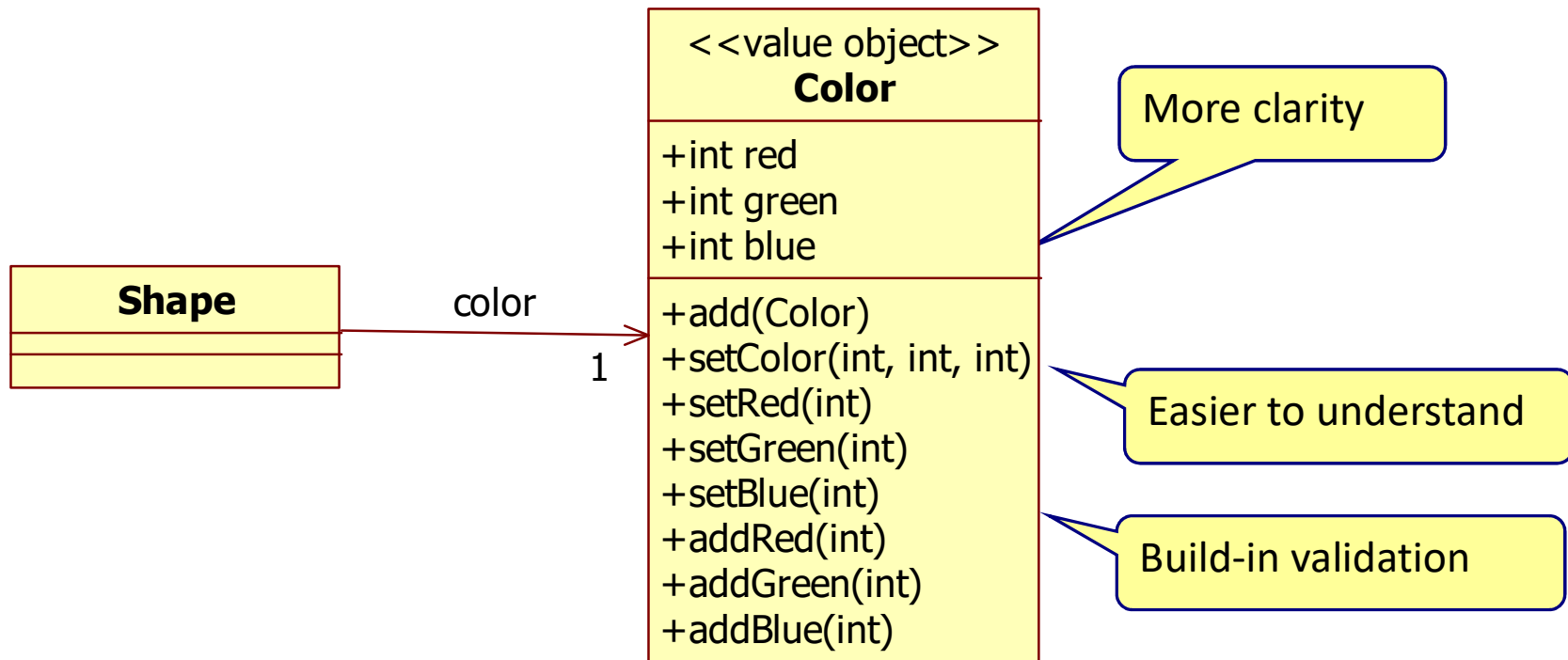
1. Representing a descriptive identity-less concept



2. Enhancing explicitness



Enhancing explicitness



Static factory methods

```
public class Height {  
    private enum MeasureUnit {  
        METER,  
        FEET,  
        YARD;  
    }  
  
    private int value;  
    private MeasureUnit unit;  
  
    public Height(int value, MeasureUnit unit) {  
        this.value = value;  
        this.unit = unit;  
    }  
  
    public static Height fromFeet(int value) {  
        return new Height(value, MeasureUnit.FEET);  
    }  
  
    public static Height fromMeters(int value) {  
        return new Height(value, MeasureUnit.METER);  
    }  
}
```

More expressive

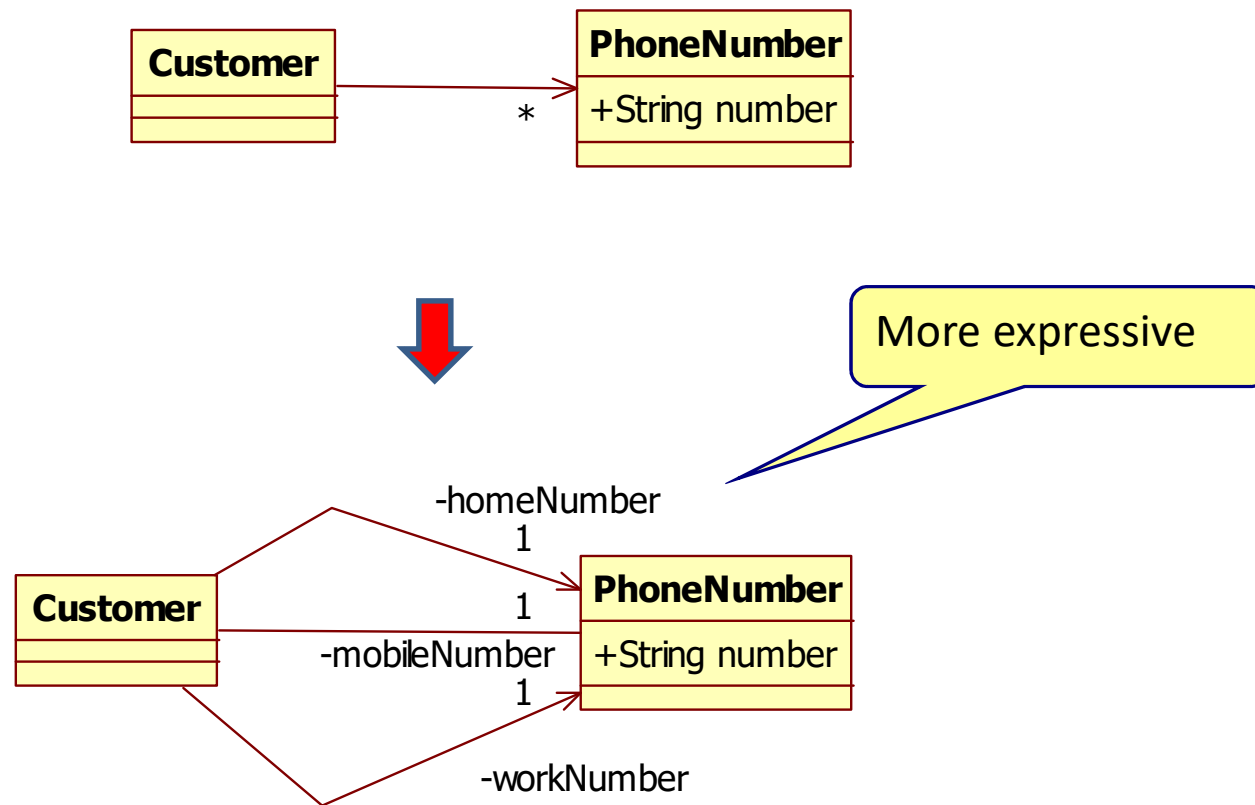
Easier for clients to call

Decouple clients
from MeasureUnit



Collection avoidance

- Be careful with collections of value objects



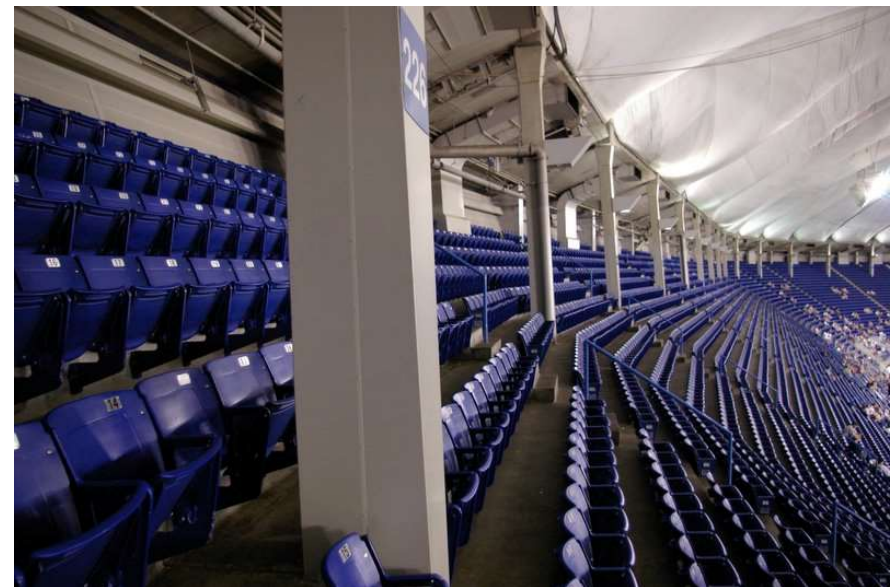
Persisting value objects

- Persist them into a denormalized form
 - Relational
 - Save them as **String** (using toString())
 - NoSQL
 - **Embed** them into the entity document
- Persist them into a separate relational table
 - Give the value object an id.

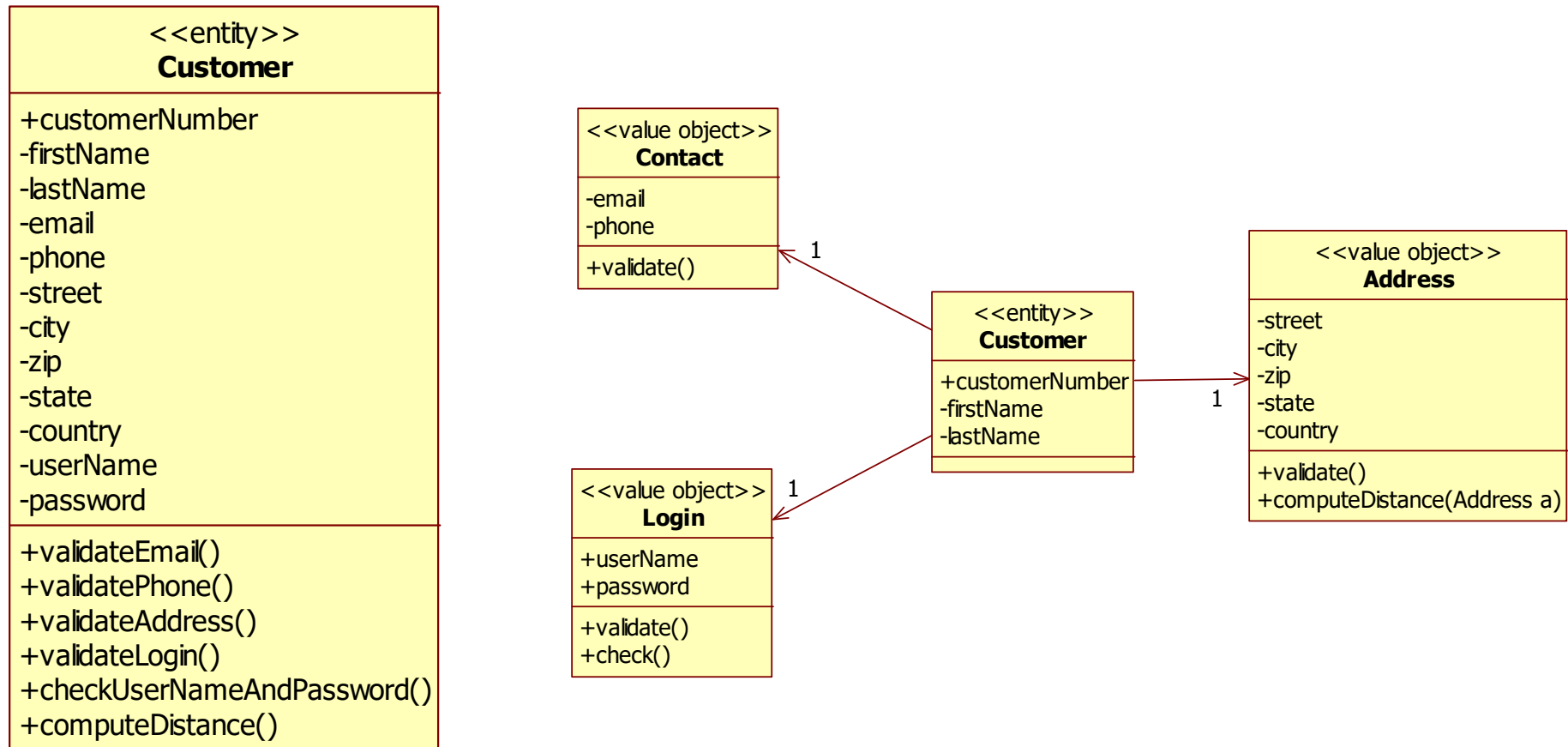


Entity versus value objects

- If visitors can sit wherever they find an empty seat then seat is a...^{vo}
- If visitors buy a ticket with a seat number on it, then seat is a...^{En}



Pushing behavior into value objects



Entities versus Value objects

- Entities have their own intrinsic **identity**, value objects don't.
- The notion of identity **equality** refers to entities
 - Two entities are the same if their id's are the same
- The notion of structural equality refers to value objects
 - Two value objects are the same if their data is the same
- Entities have a **history**; value objects have a zero lifespan.
- A value object should always **belong to one or several entities**.
 - It can't live by its own.
- Value objects should be **immutable**; entities are almost always **mutable**.
 - If you change the data in a value object, create a new object.
- If you can safely **replace** an instance of a class with another one which has the same set of attributes, that's a good sign this concept is a value object
- Value objects don't need their own tables in the database.
 - The data can be embedded into the entity table
- Always prefer value objects over entities in your domain model.



Main point

- Instead of a large entity class, we strive for a small and simple entity class with many value classes
- The Unified Field contains all knowledge in its simplest and most abstract form.



DOMAIN SERVICES



For complicated business rules
business for more than 1 entity

Domain service

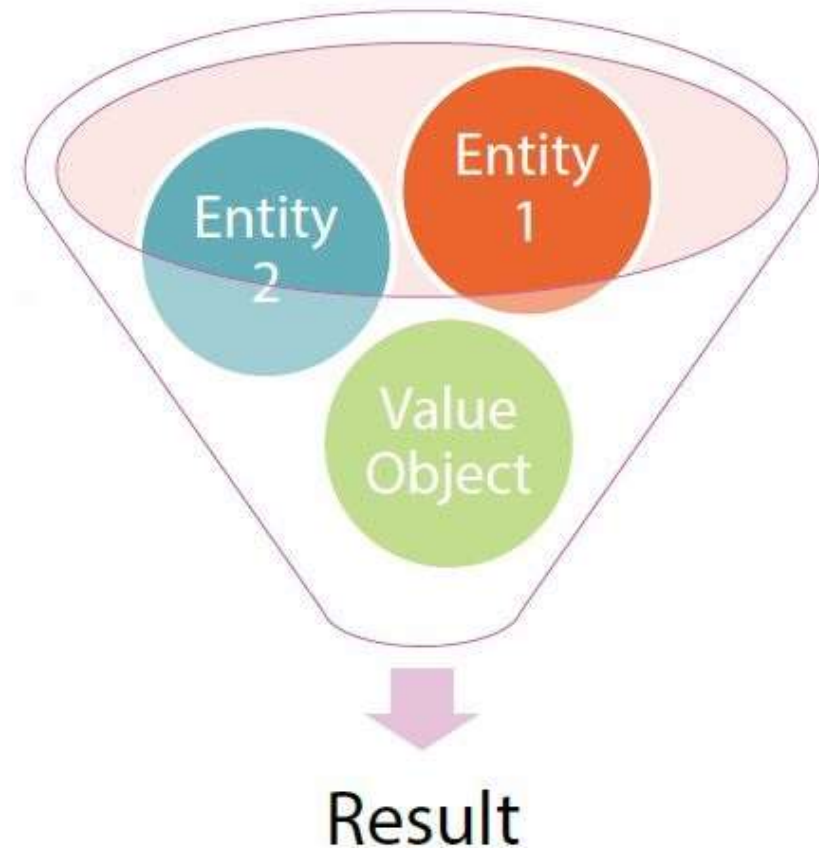
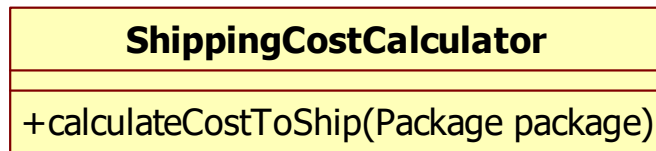
- Sometimes behavior does not belong to an entity or value object
 - But it is still an important domain concept
- Use a domain service.

ShippingCostCalculator
+calculateCostToShip(Package package)



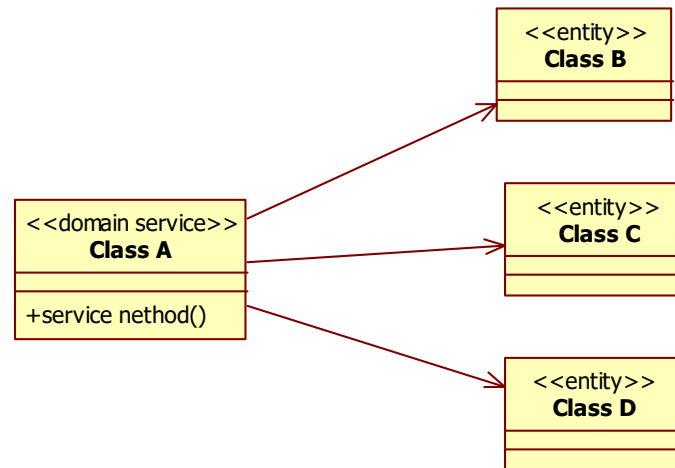
Domain service

- Interface is defined in terms of other domain objects

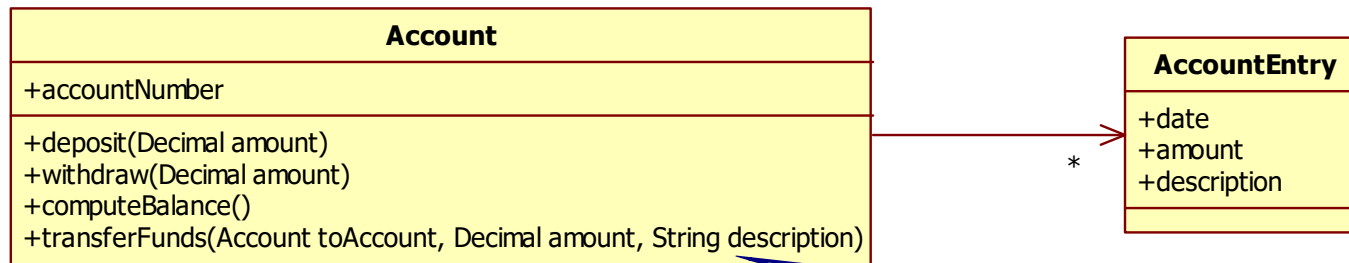


Domain service characteristics

- Stateless
 - Have no **attributes** just has logic
- Represent behavior
 - No **identity** will not be saved in db
- Often orchestrate multiple domain objects



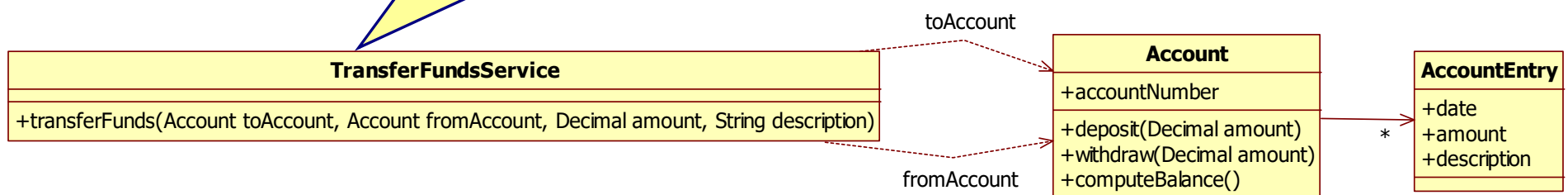
Service example



The Account is responsible for transferring money



Service for transferring money

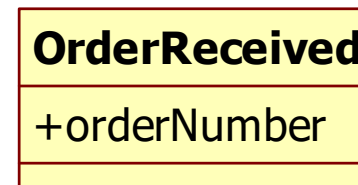
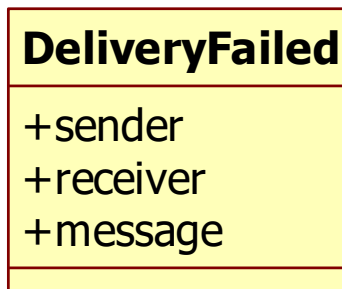


DOMAIN EVENTS



Domain event

- **Classes** that represent important events in the problem domain that have already happened
 - **Immutable**

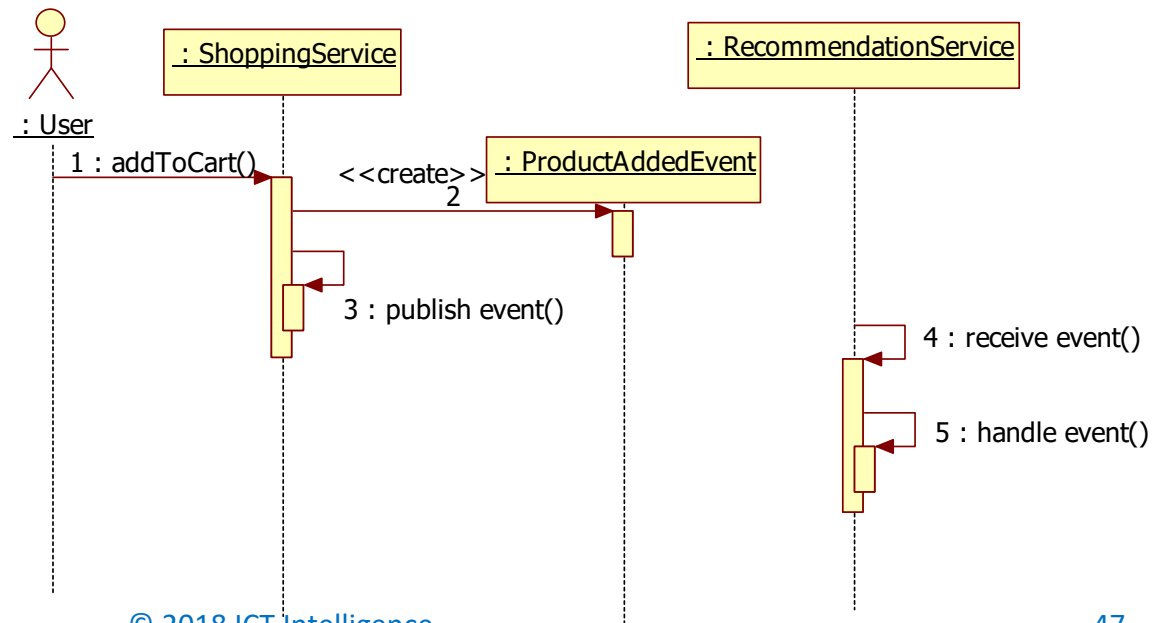
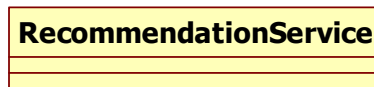
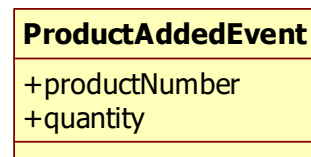
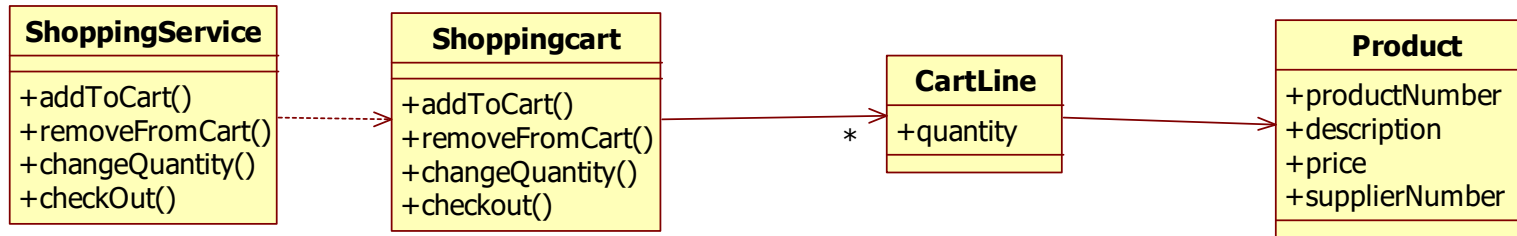


Domain event

- Events are raised and event handlers handle them.
- Some handlers live in the domain, and some live in the service layer.



Domain event example



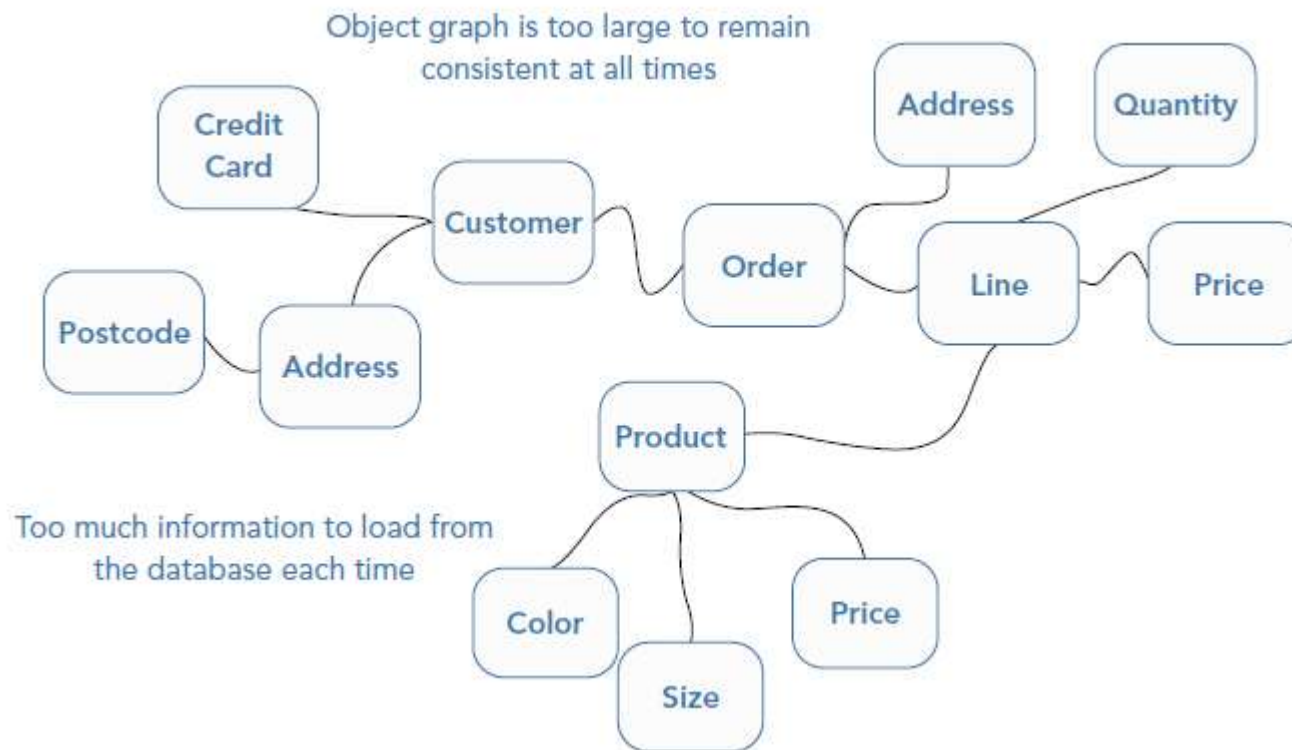
AGGREGATES



Aggregates

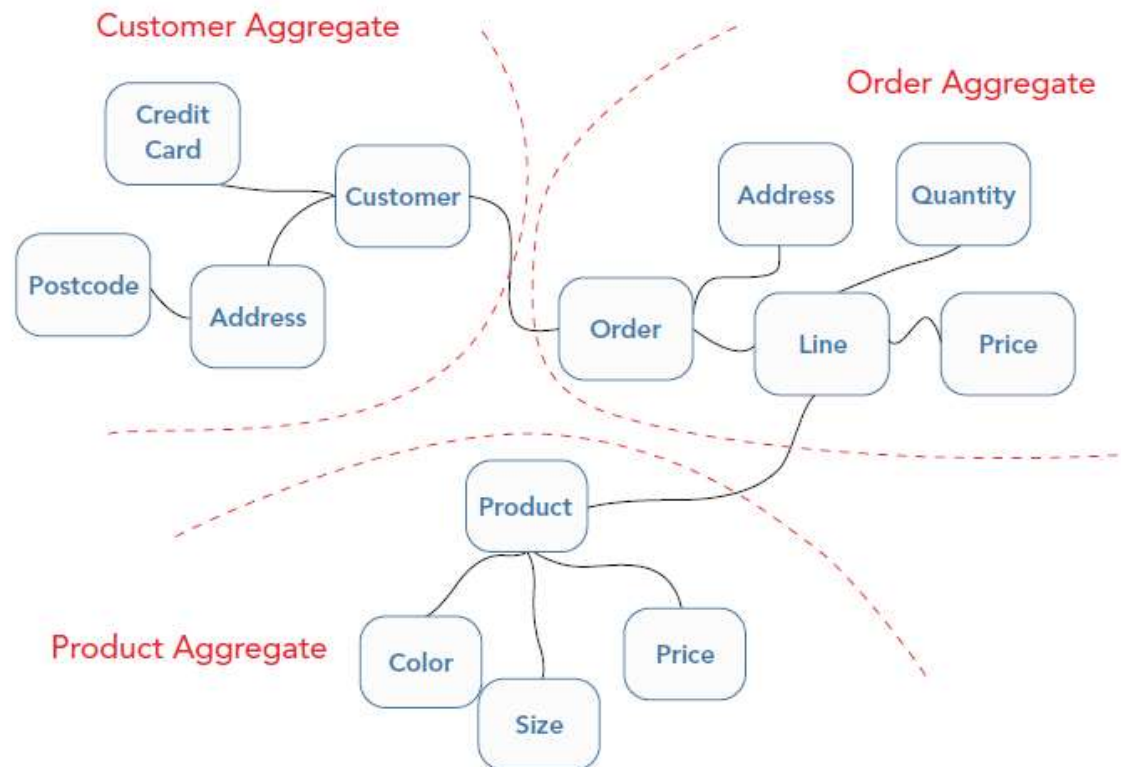
group of obj that belongs together
aggregates are basically small application by its own

- Large object graphs are difficult to keep consistent



Aggregates

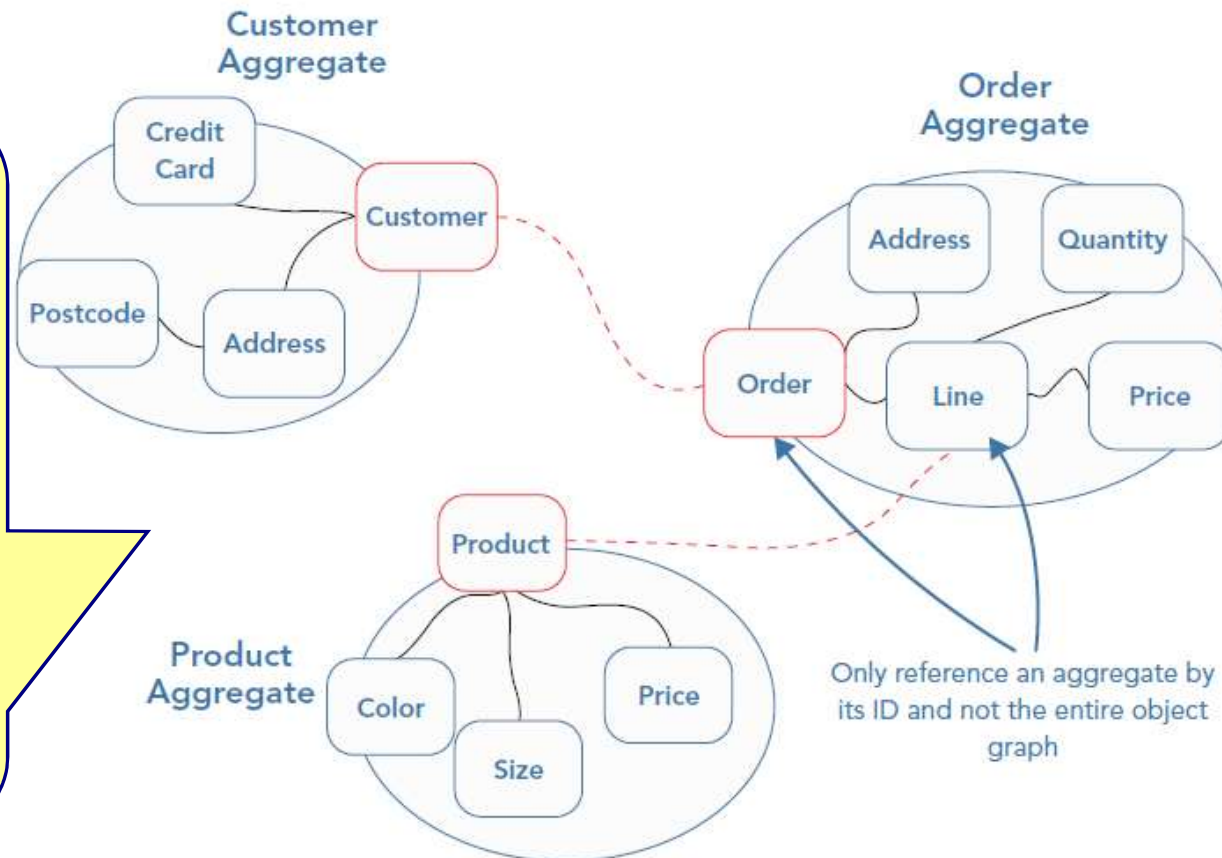
- Large models are **split** and grouped into **aggregates of entities** and value objects that are treated as a conceptual whole.



Aggregate root

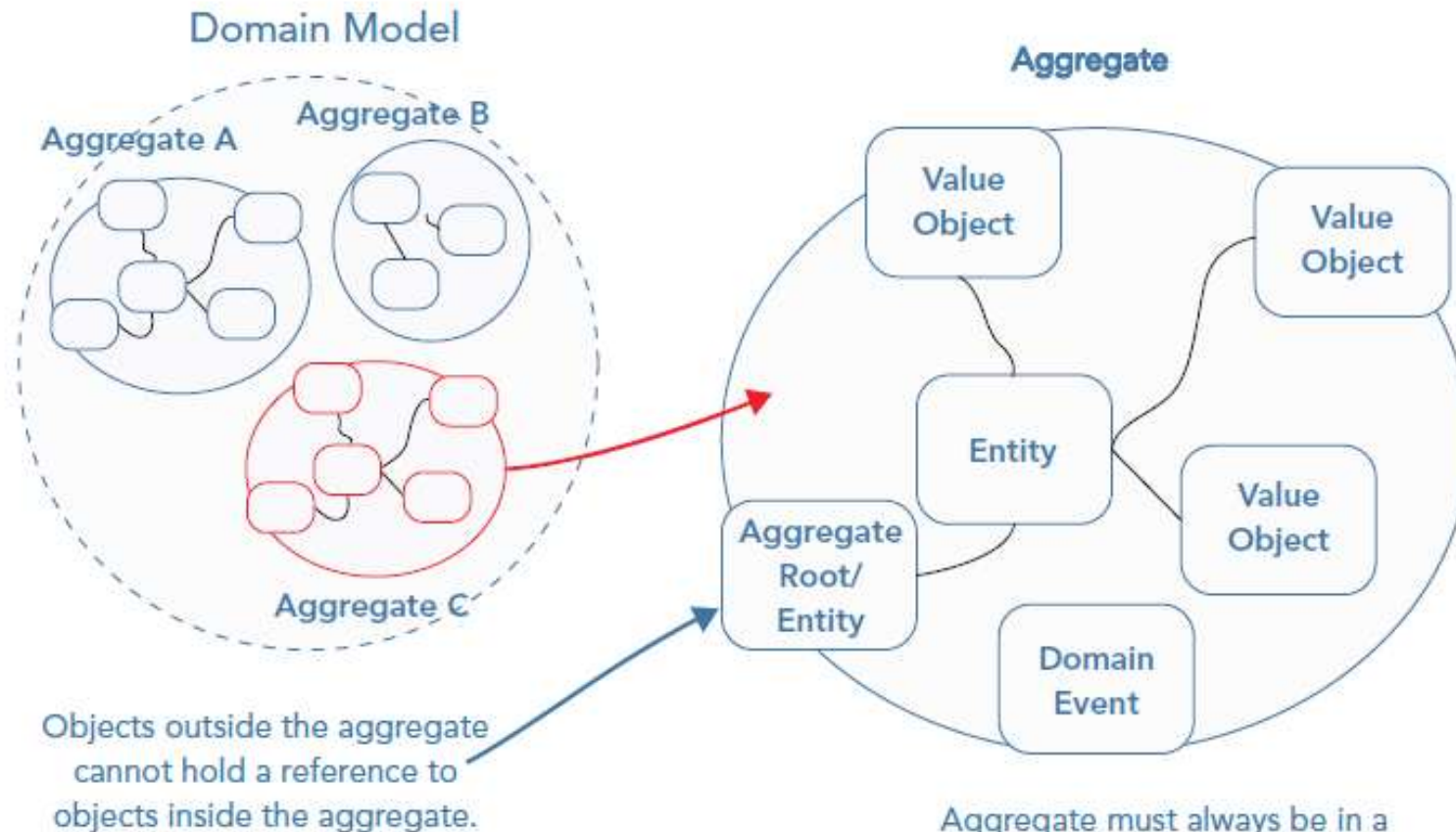
- An aggregate root acts as the entry point to the aggregate.

Relationships between aggregate roots should be implemented by keeping the ID of another aggregate root and not a reference to the object itself



Aggregate root is entry point into the aggregate

Aggregates do not have association with each other, they can have dependency

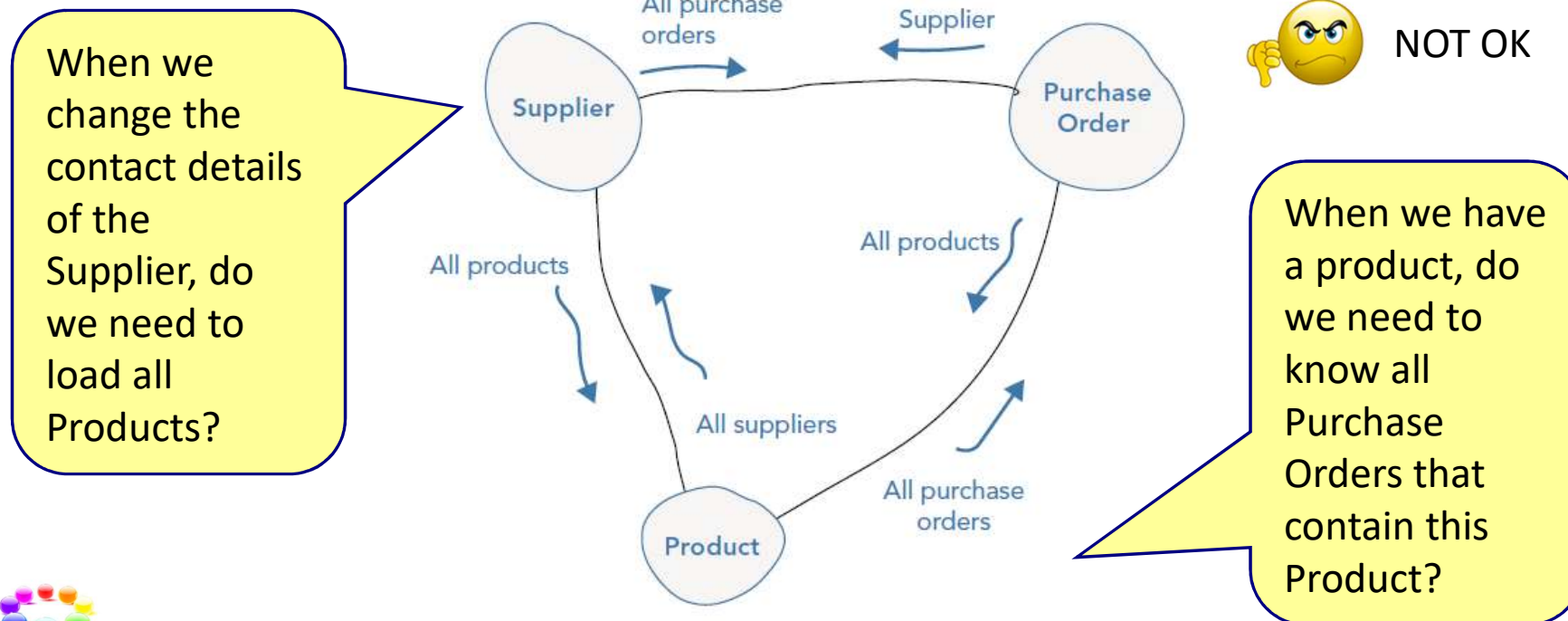


association in UML is a line
type A always has attr of type B
dependency is temporal
relationship, obj A will talk to B but
only for this one method



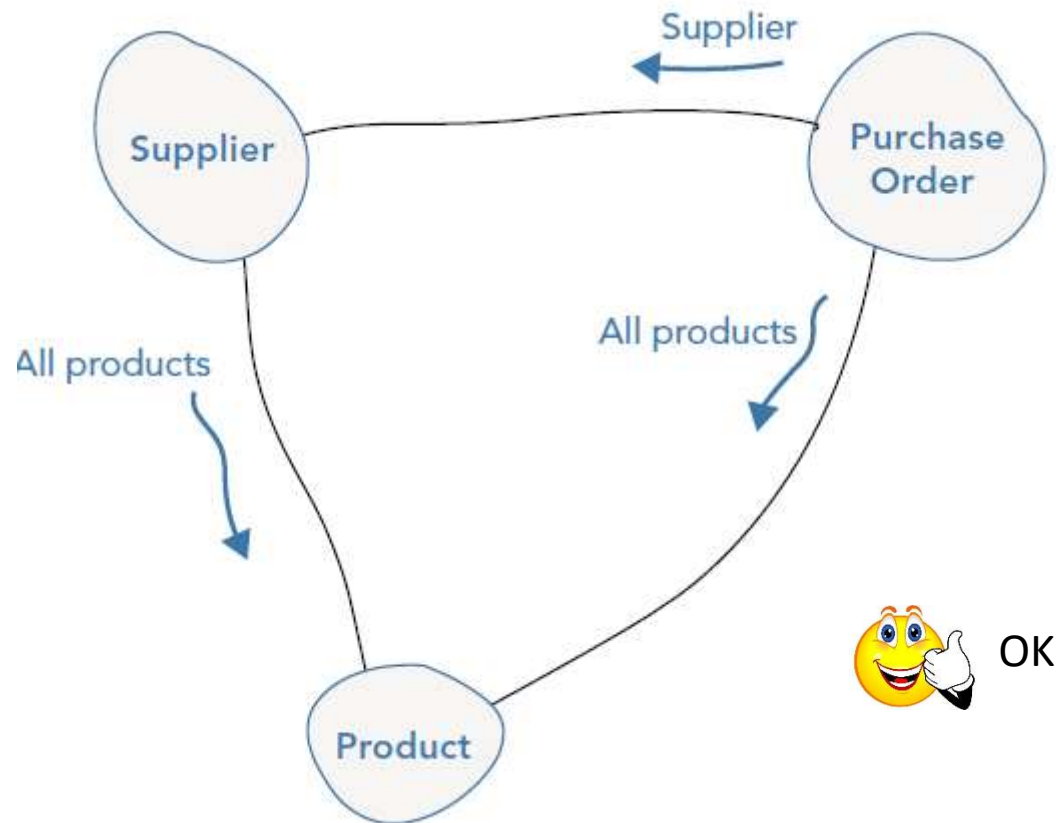
Bi-directional relationship

- Many-to-many relationships can become overwhelmingly complex
- The model does not need to reflect the real world

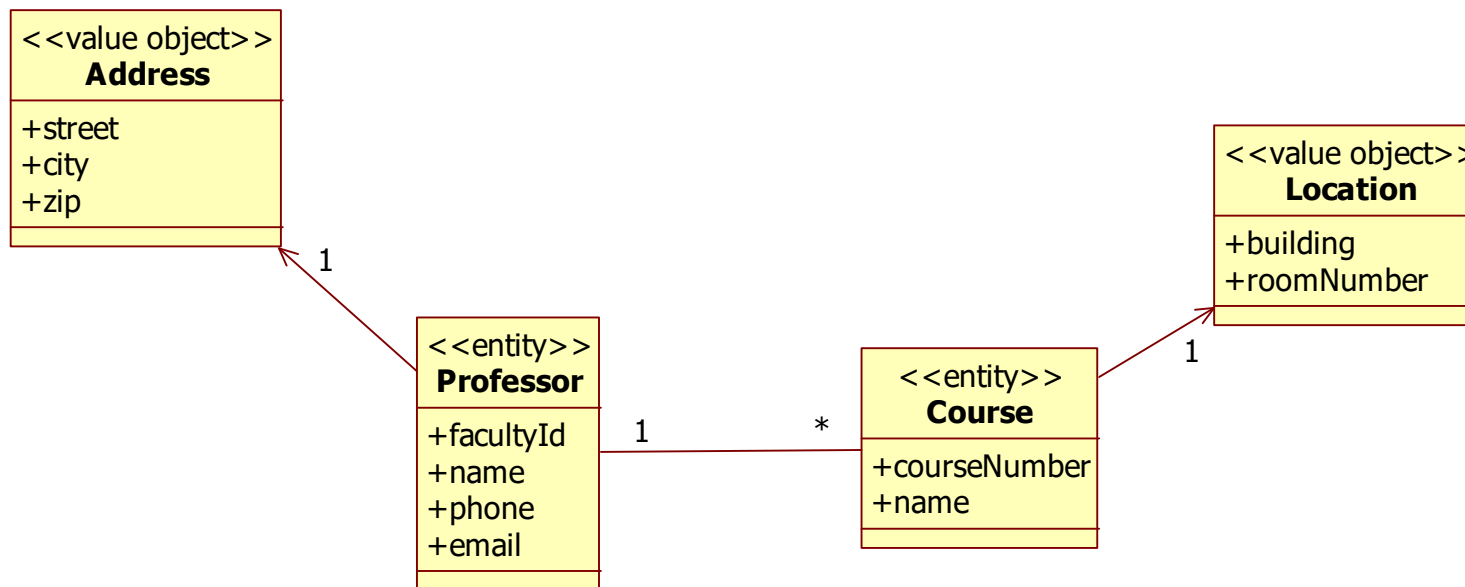


Favor single traversal directions

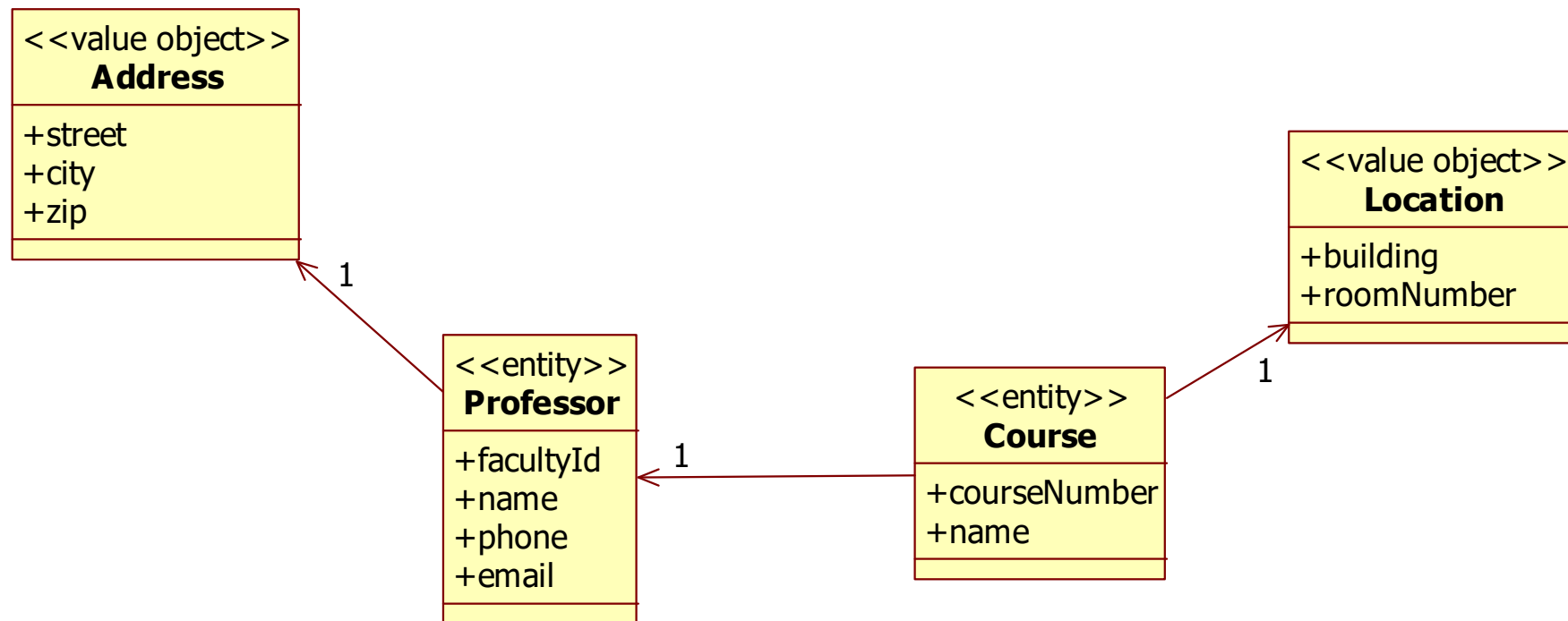
- Simplicity
- Shows who is the owner of a relation



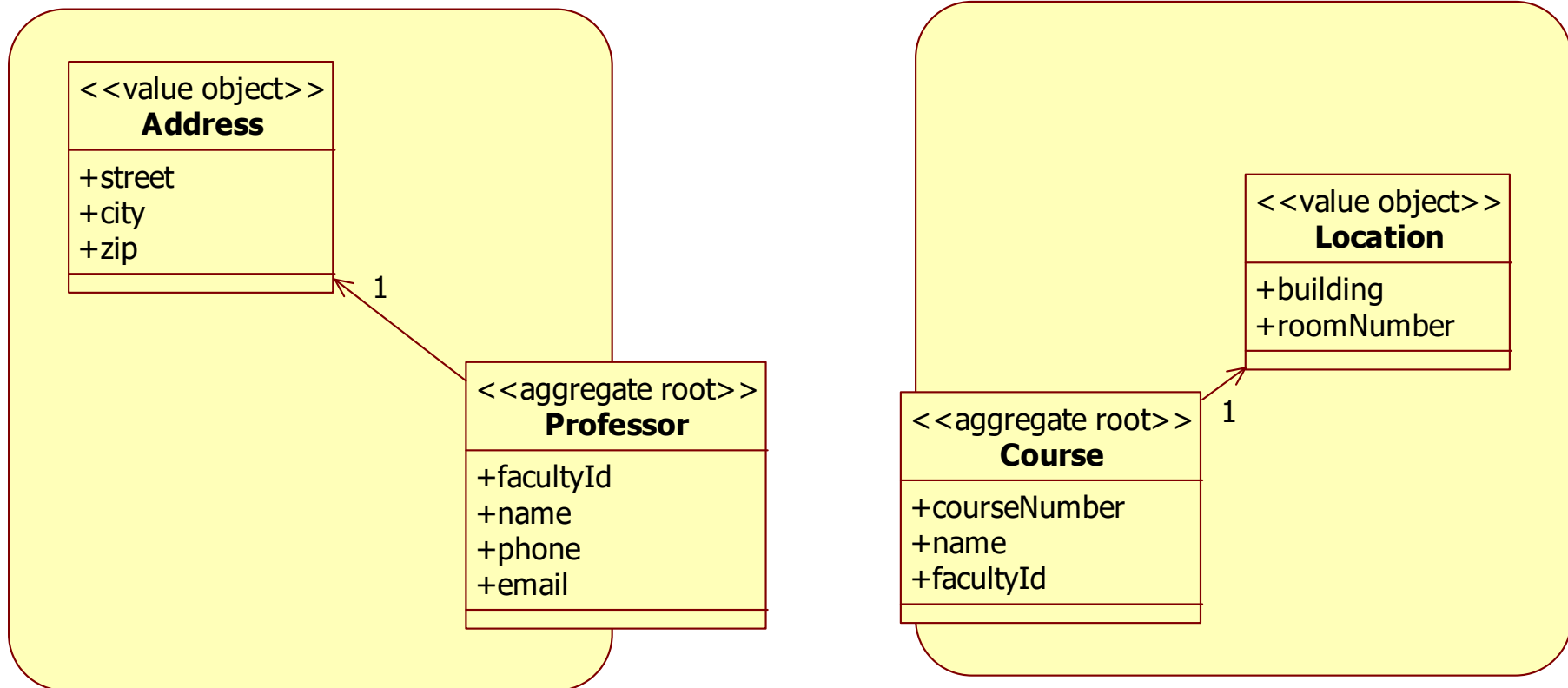
Bi-directional relation



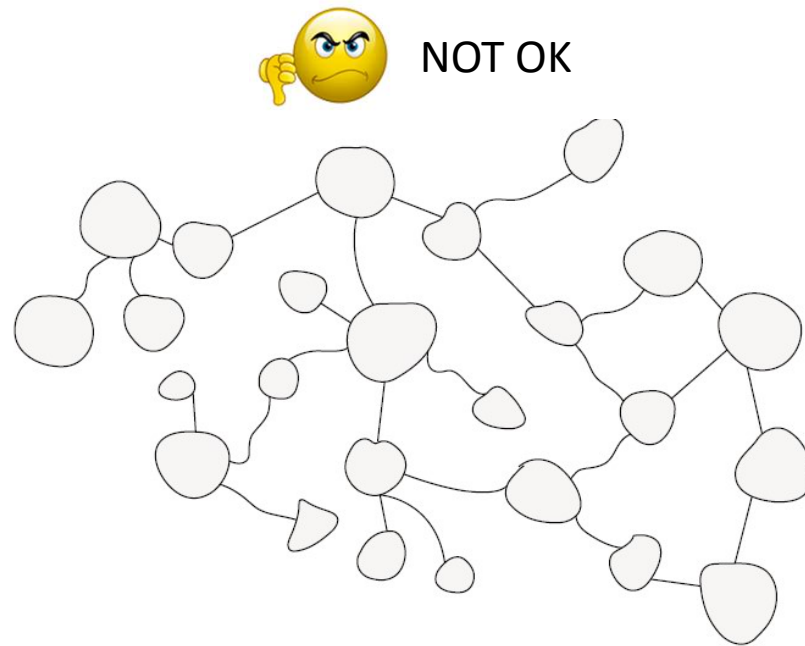
Uni-directional relation



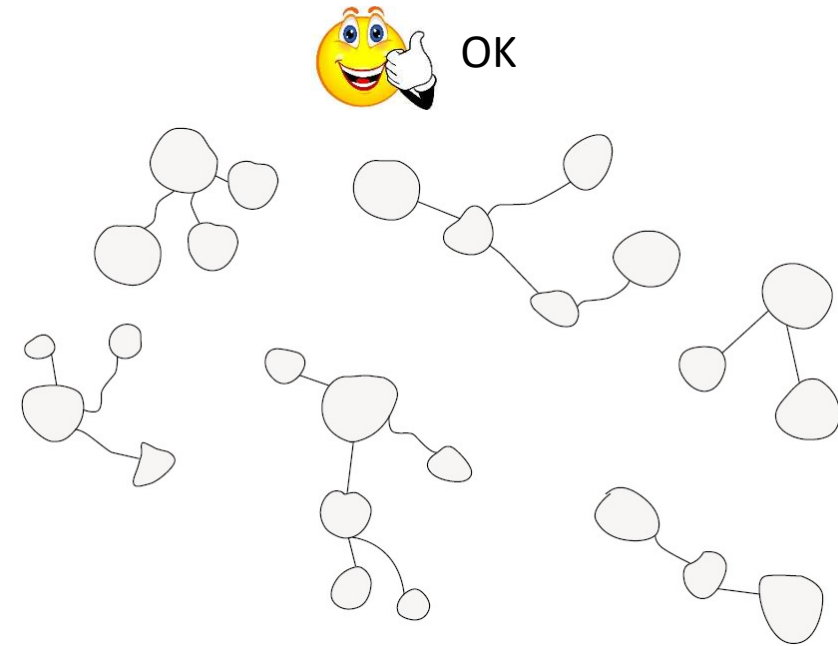
Using aggregates



Simplicity of the domain model



Complex domain model with many unnecessary associations

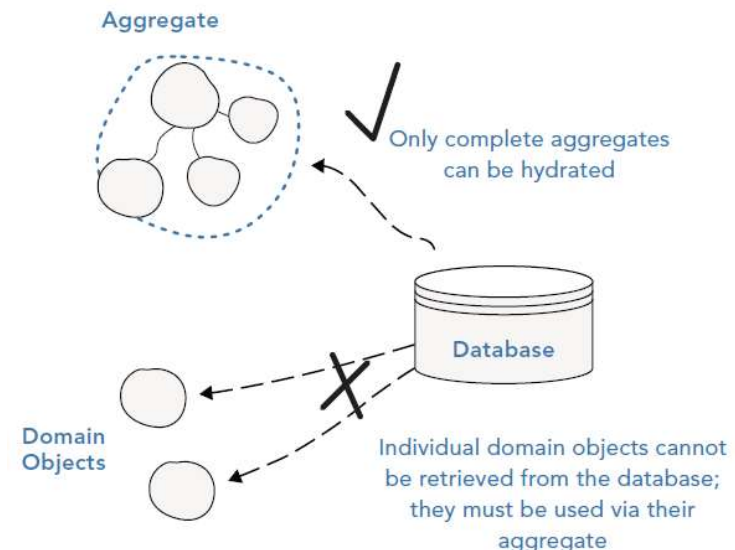


Clearer domain model with only the essential associations



Aggregates and database actions

- Aggregates are saved, updated and deleted as a whole
- Aggregates are loaded from the database as a whole
- One Data Access Object (DAO) per aggregate
- The whole aggregate is within one transactional boundary



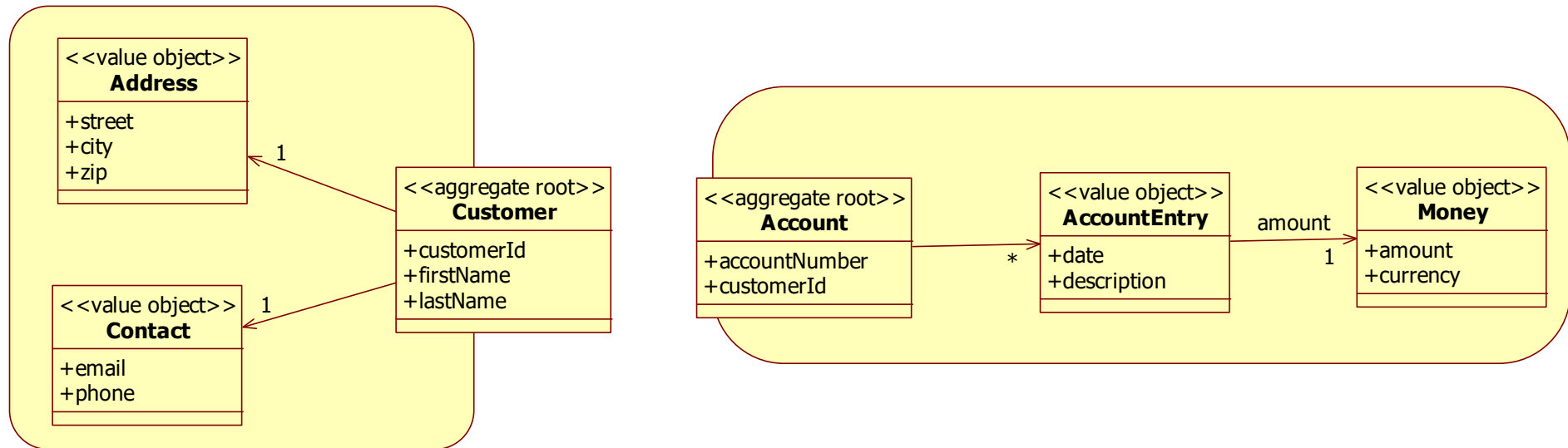
Referencing other aggregates

- Aggregate roots never holds a reference to another aggregate root
 - It should keep the ID of another aggregate root
 - Or we add a new class that it references



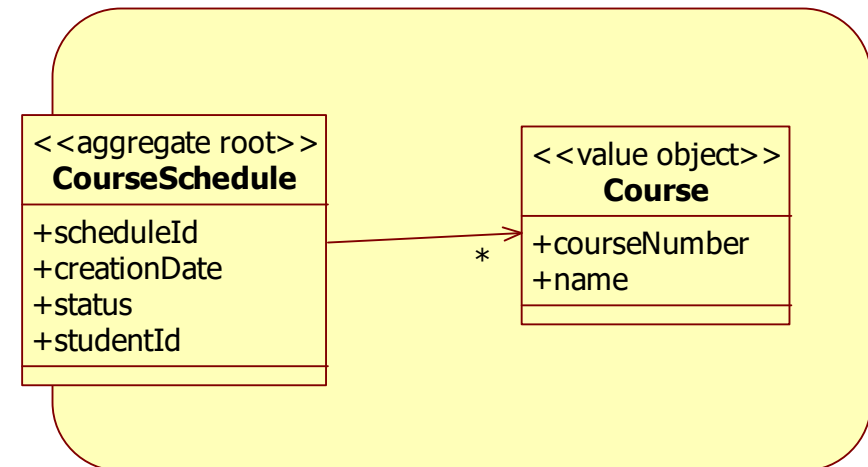
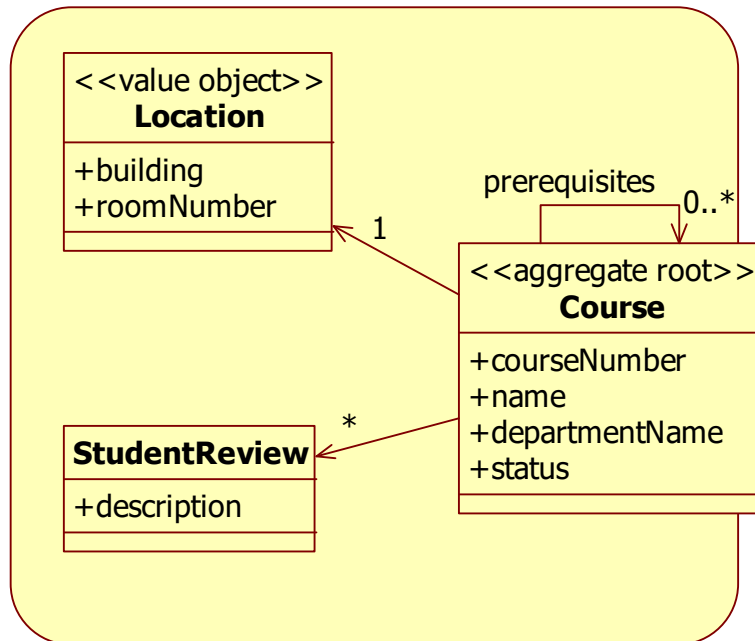
Referencing other aggregates

- Using an Id



Referencing other aggregates

- Add a new class



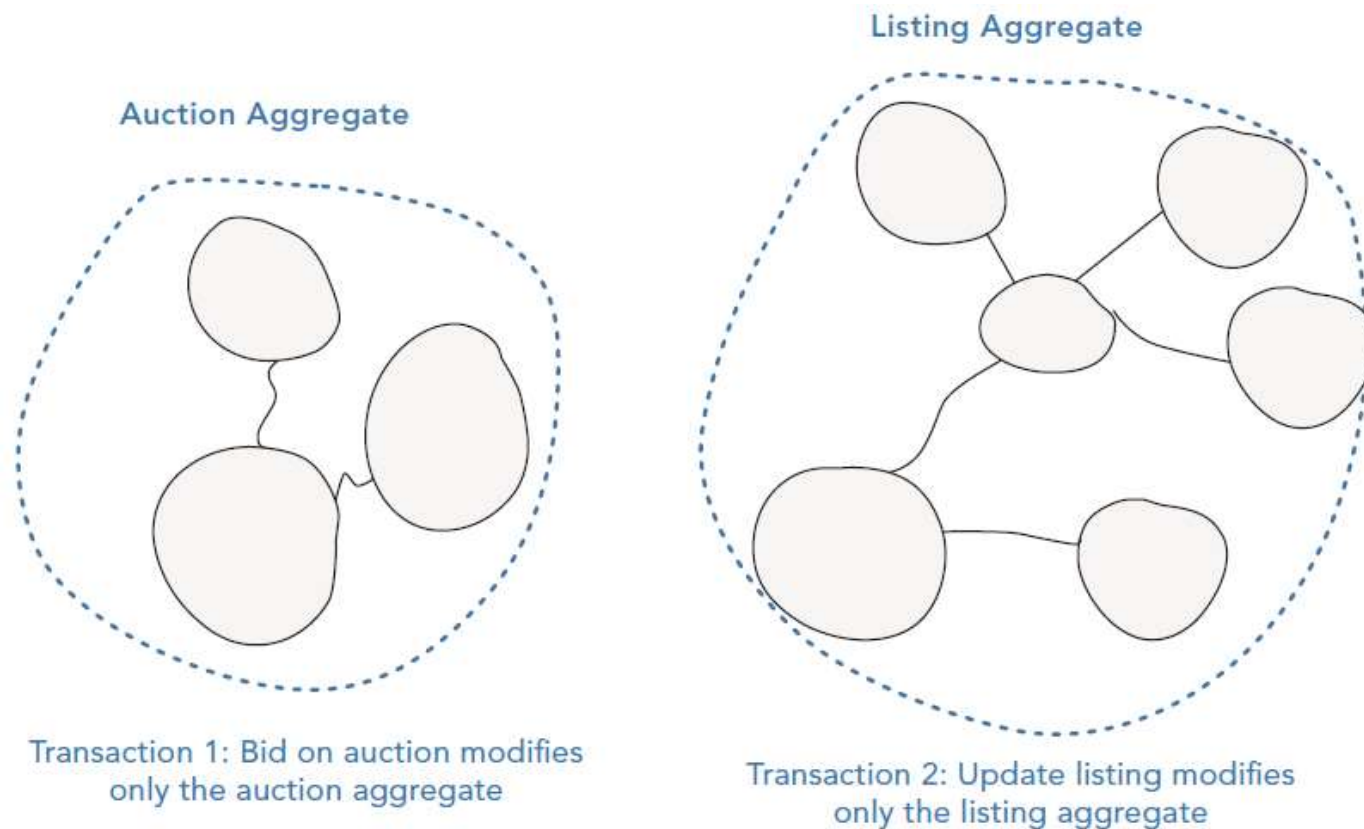
Size of the aggregate

- Favor **smaller** aggregates
 - Large aggregates can decrease performance
 - Many database calls with additional join queries
 - Large aggregates are often involved in multiple use cases
 - More concurrency conflicts
 - Large aggregates may not scale well
 - The whole aggregate needs to be placed in one database





Transactions and consistency

- Strive to modify a single aggregate per use case



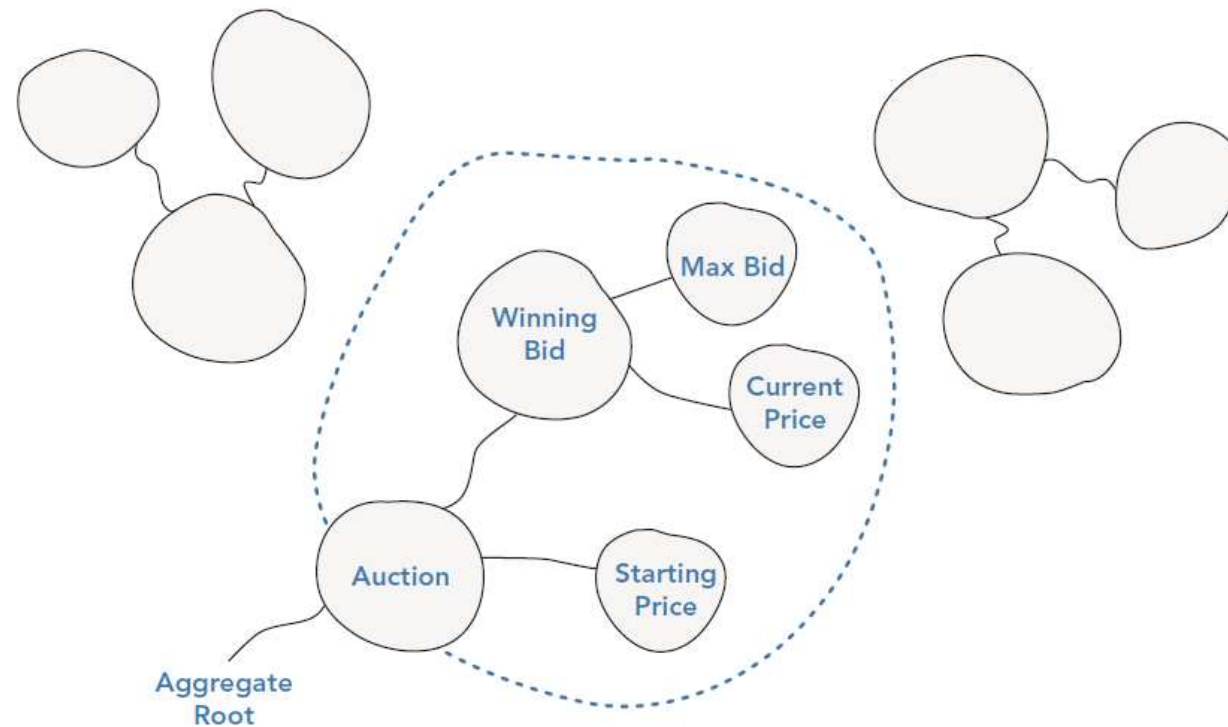
Finding aggregates

- Which object are logically connected
 - High cohesion, low coupling  OK
- Design aggregates around invariants/business rules
- Strive to modify a single aggregate per use case
- Do not design aggregates around UIs  NOT OK
- Do not design aggregates around the data model
- Avoid dumb collections and containers
 - An aggregate is not a just a container for other objects



Aggregate Root

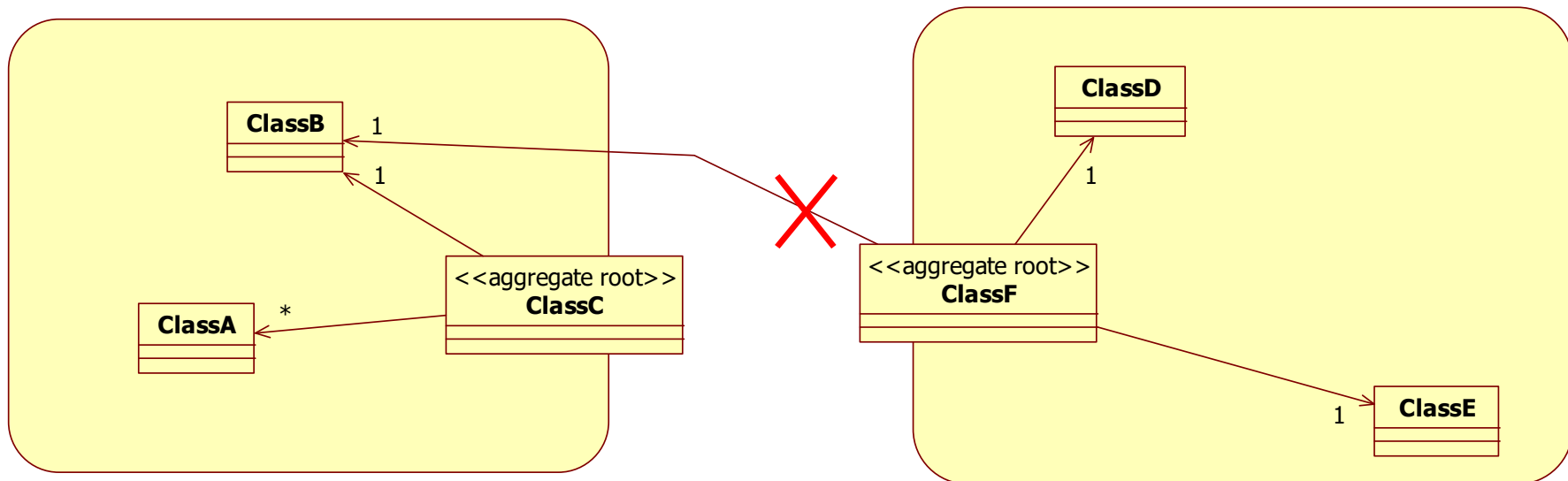
- All communication with an aggregate should go via its root



Encapsulation

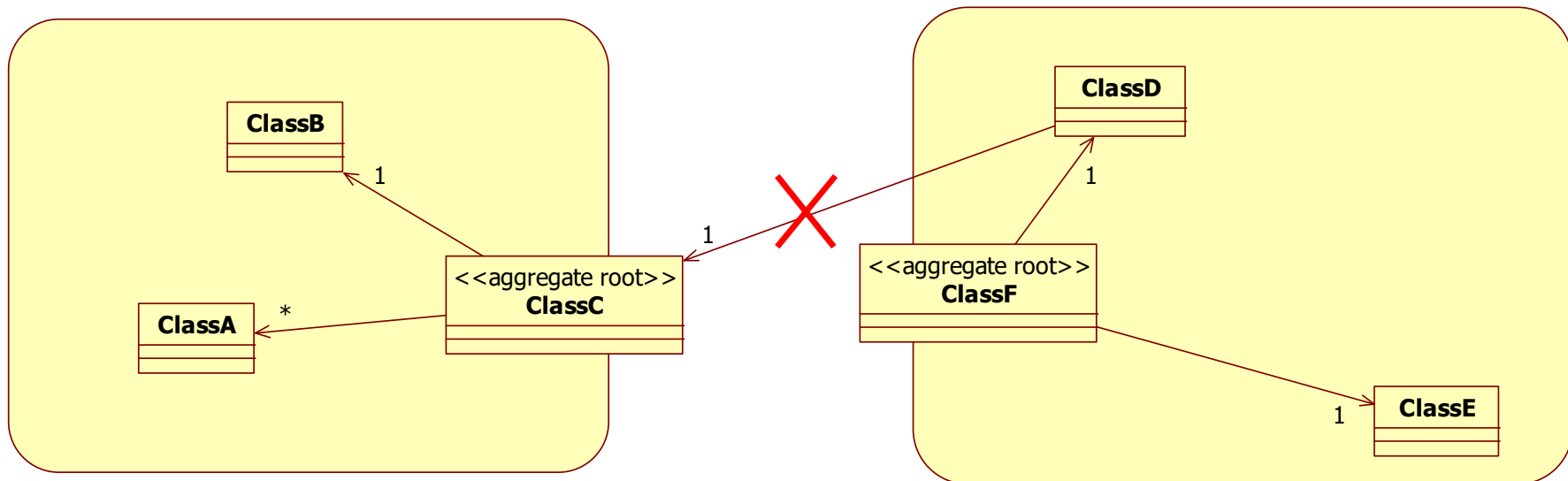
- Nothing outside an aggregate should hold a reference to its inner members

aggregates are encapsulated

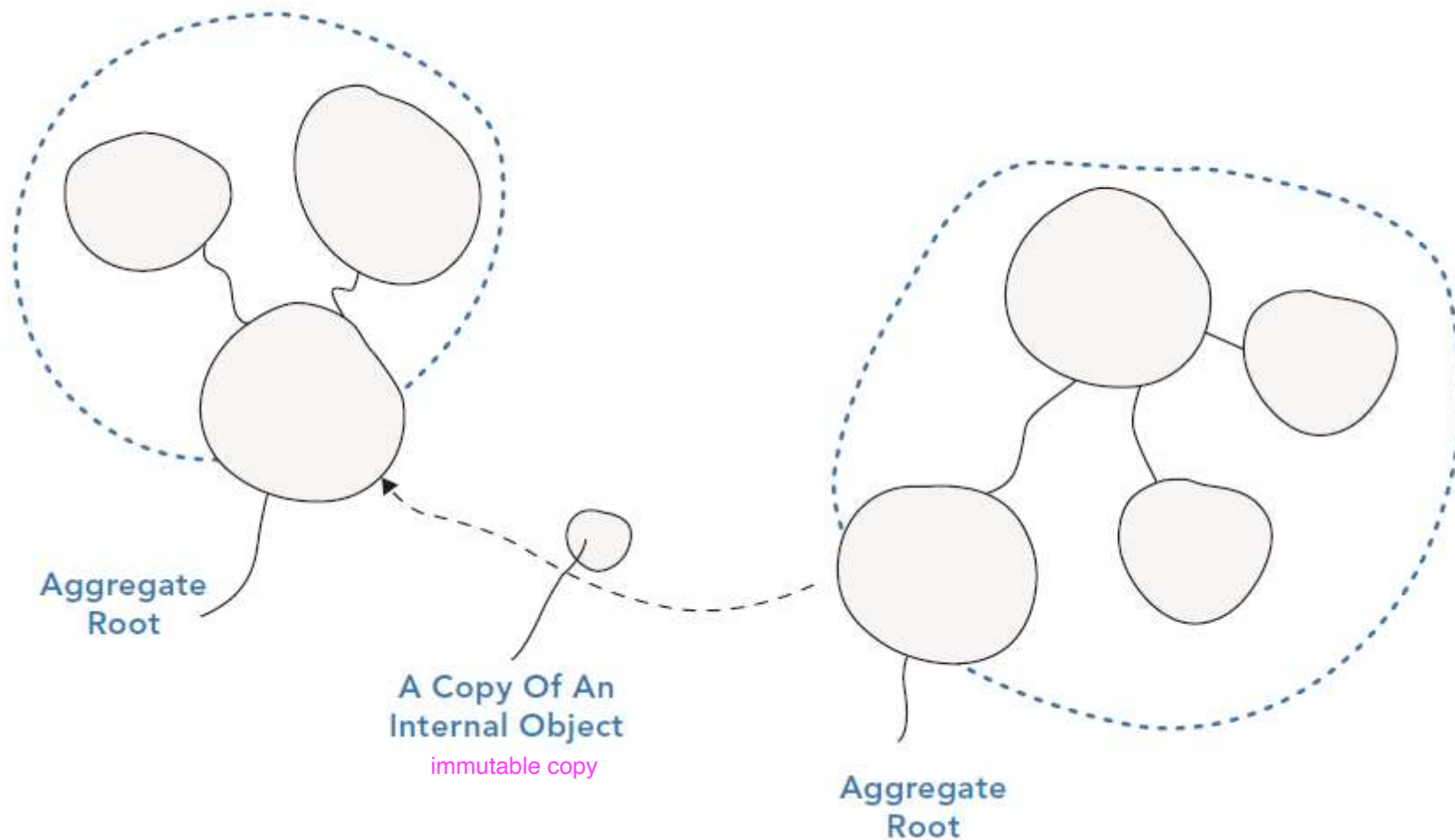


Internal reference to aggregate root

- Non aggregate roots can hold a reference to other aggregate roots



Sharing copies

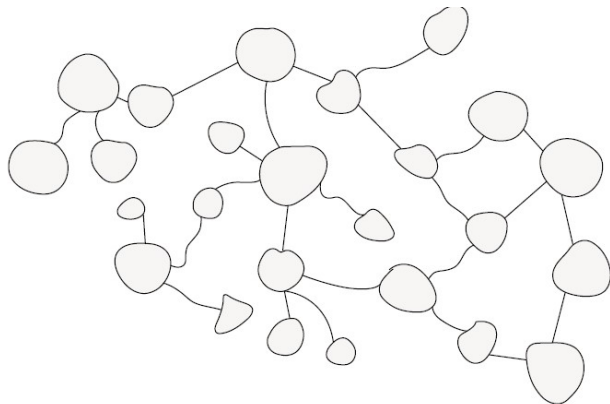


Main point

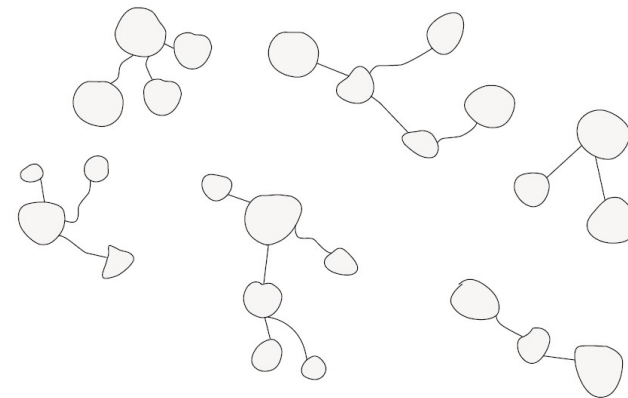
- A large complex class structures can be split up is smaller and simpler class structures using aggregates.
- By gaining full support of nature by tapping into pure consciousness, life gets simpler and more enjoyable.



NOT OK



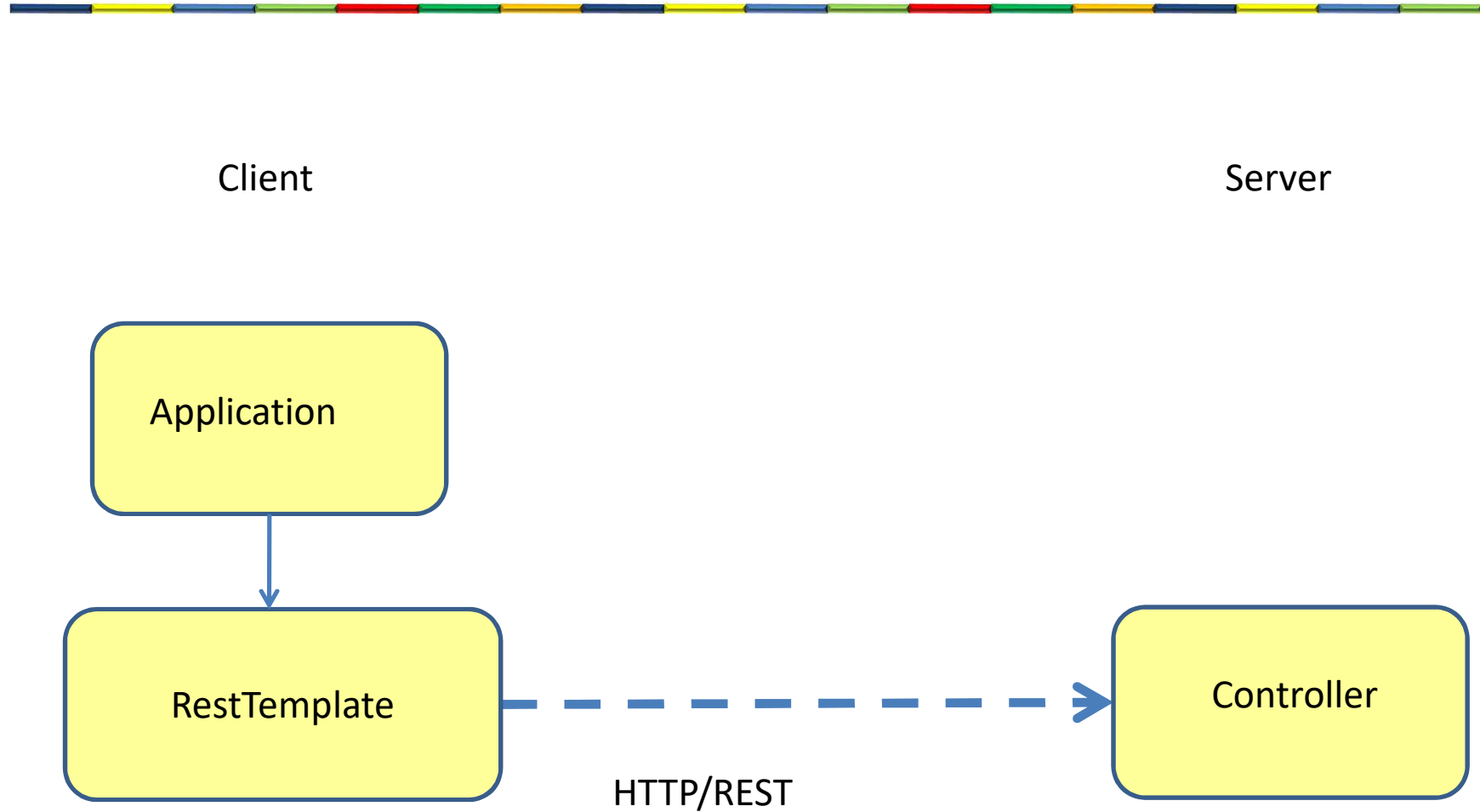
OK



SPRING BOOT REST CLIENT



Creating a REST client



RestServer

```
@RestController
public class GreetingController {

    @RequestMapping("/greeting")
    public Greeting greeting() {
        return new Greeting("Hello World");
    }
}
```

```
public class Greeting {
    private String content="";

    public Greeting() {}

    public Greeting(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
}
```



RestClient

server.port=8081

```
@SpringBootApplication
public class RestClientApplication implements CommandLineRunner {
    @Autowired
    private RestOperations restTemplate;

    public static void main(String[] args) {
        SpringApplication.run(RestClientApplication.class, args);
    }

    @Bean      Spring object, created when app run
    RestTemplate restTemplate() {
        RestTemplate restTemplate = new RestTemplate();
        restTemplate.getMessageConverters().add(new MappingJackson2HttpMessageConverter());
        restTemplate.getMessageConverters().add(new StringHttpMessageConverter());
        return restTemplate;
    }

    @Override
    public void run(String... args) throws Exception {
        Greeting greeting = restTemplate.getForObject("http://localhost:8080/greeting",
                                                    Greeting.class);
        System.out.println("Receiving message:" + greeting.getContent());
    }
}
```

Connecting the parts of knowledge with the wholeness of knowledge

1. A rich domain model contains all domain knowledge.
 2. An aggregate is a small group of classes that belong together
-

3. **Transcendental consciousness** is the source of all activity.
4. **Wholeness moving within itself:** In Unity Consciousness, one realizes that all activity in the universe are expressions from and within one's own silent pure consciousness.

