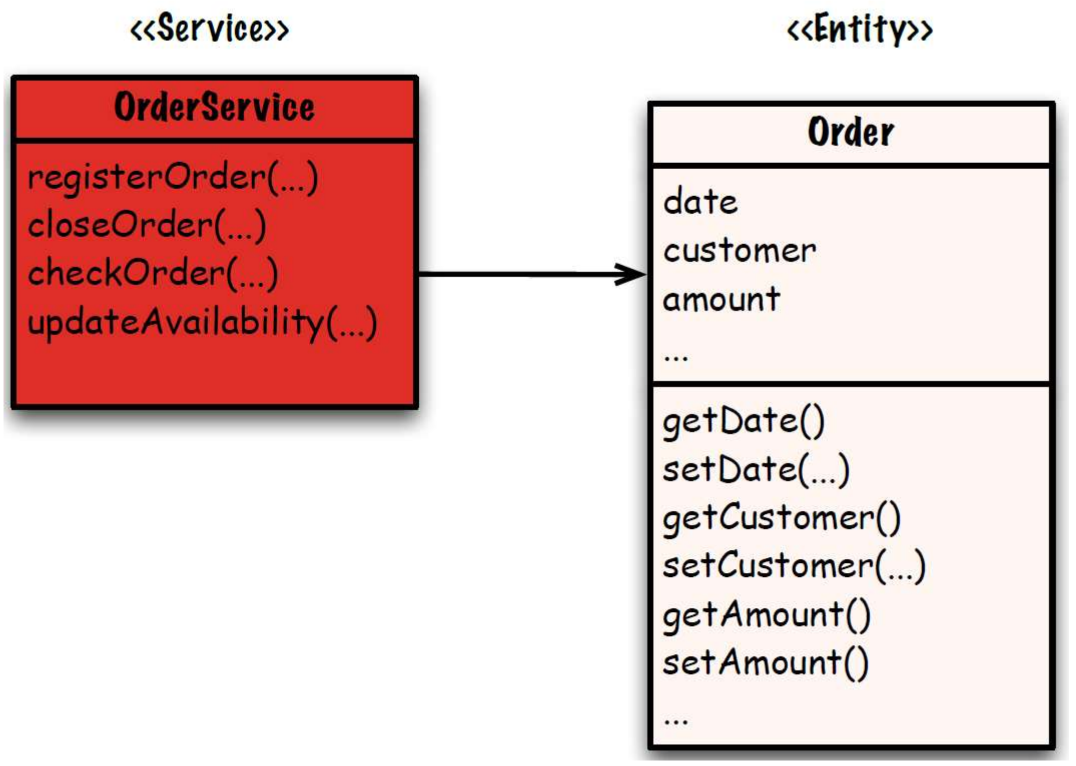
RICH DOMAIN MODELS

Anemic domain model

- Classes in the model have no business logic

NOT OK



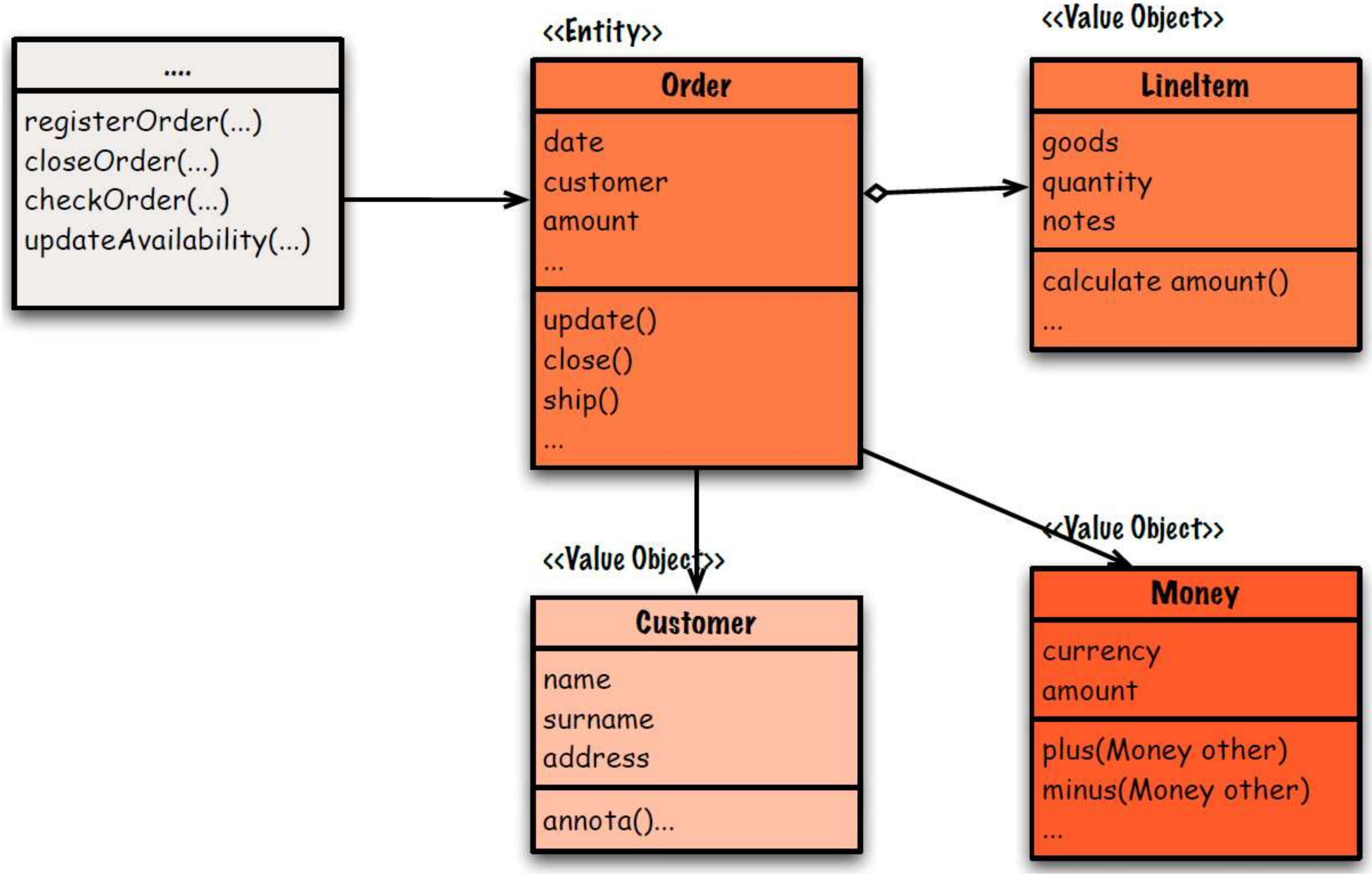
Disadvantages anemic domain model

- 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

OK



Domain Model Patterns

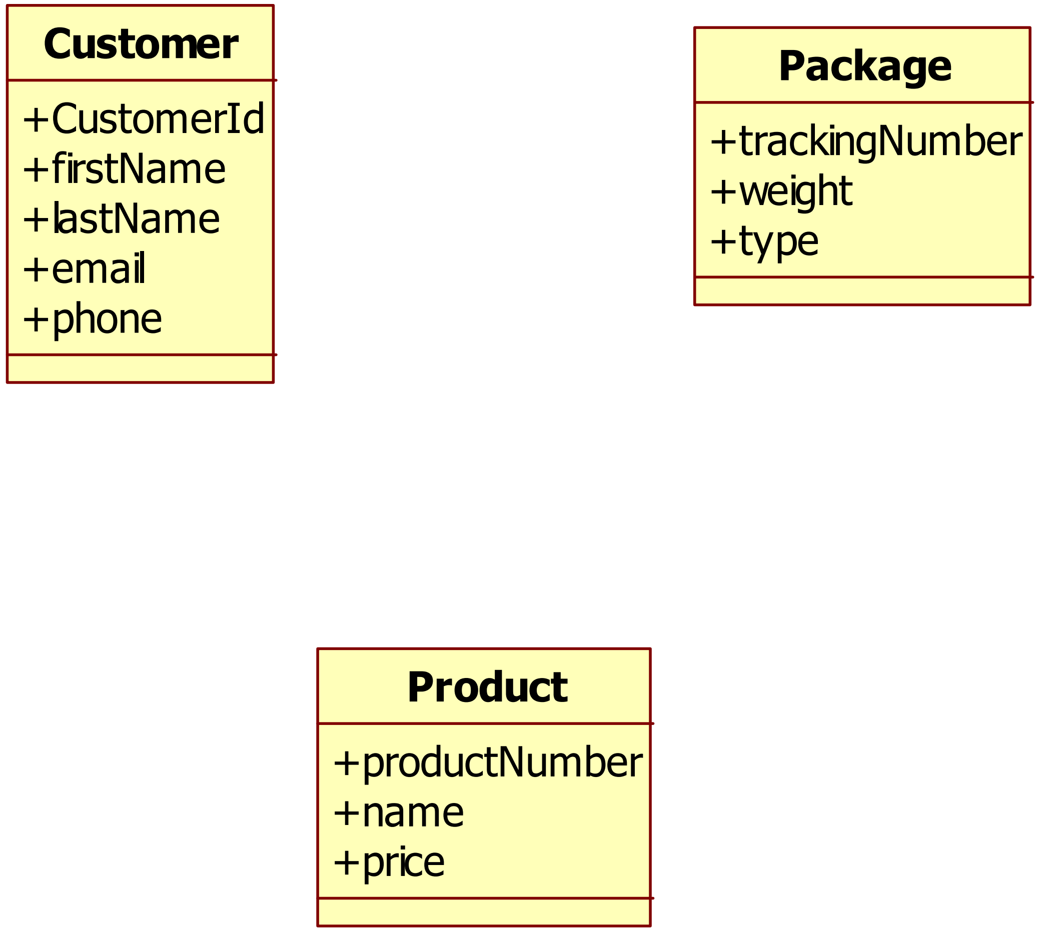
- Entities
- Value objects
- Domain services
- Domain events

ENTITIES

Entities

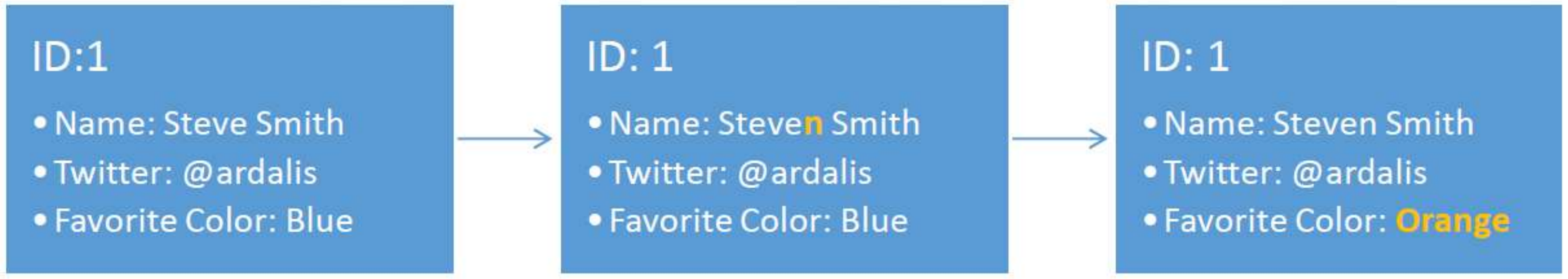
- A class with identity
- Mutable
  - State may change after instantiation
  - The entity has an lifecycle
    - The order is placed
    - The order is paid
    - The order is fulfilled

Example entity classes



Entities

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

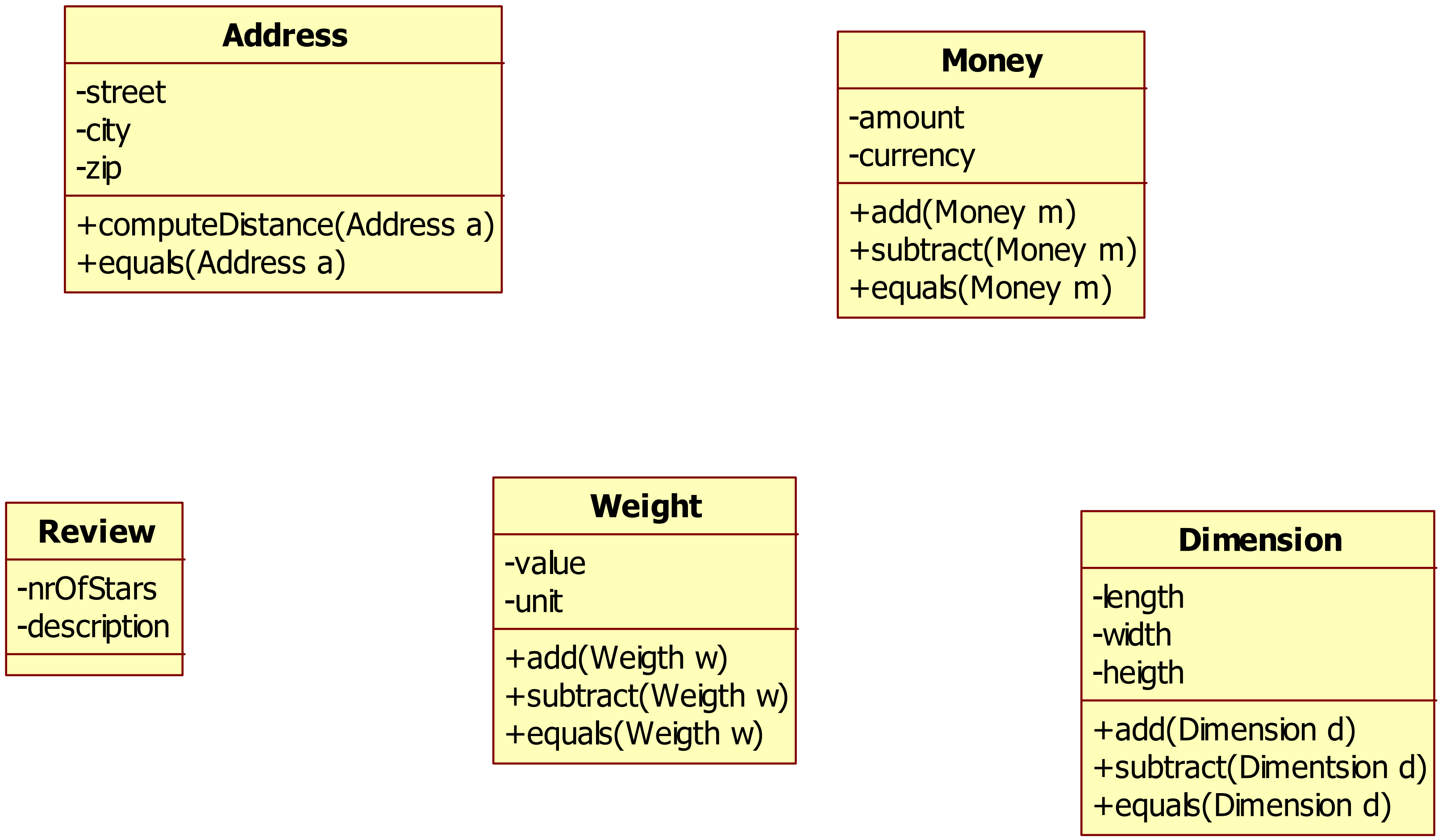


VALUE OBJECTS

Value objects

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

Example value object classes

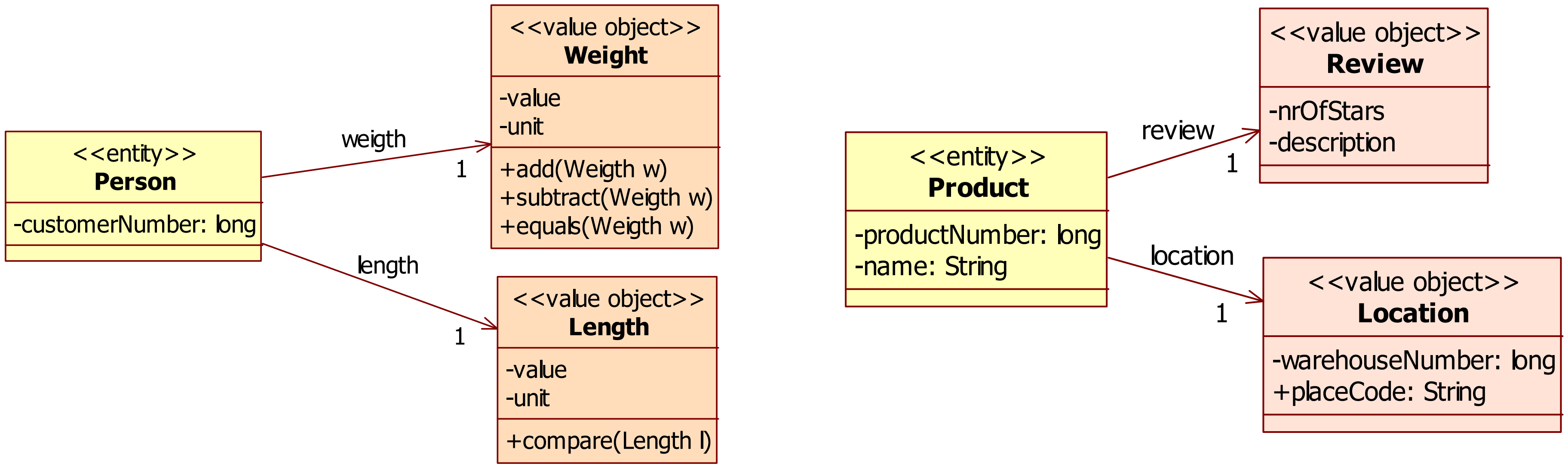


Value object characteristics

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

No identity

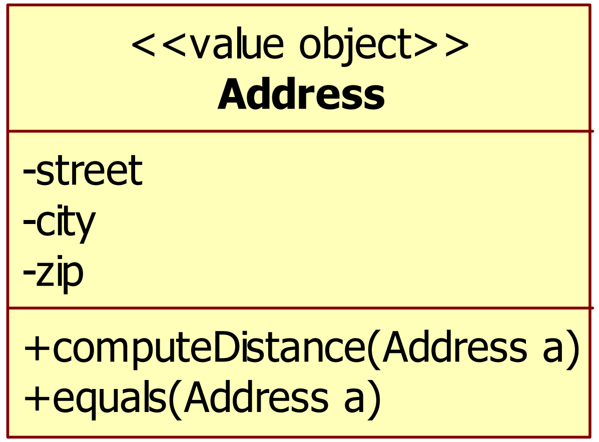
- Value objects tell something about another object

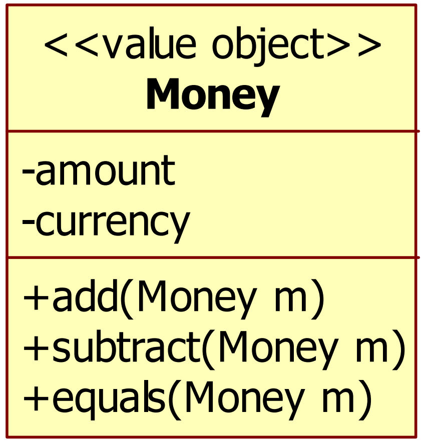


- Technically, value objects may have IDs using some database persistence strategies.
  - But they have no identity in the domain.

Attribute-based equality

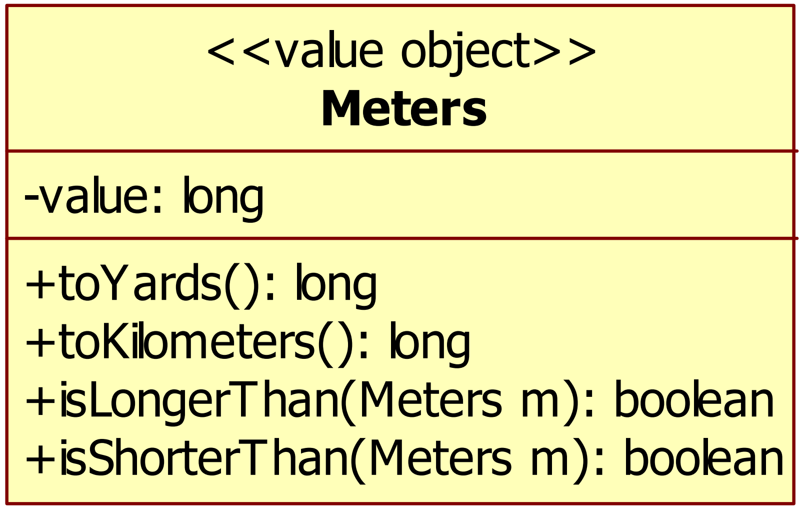
- 2 value objects are equal if they have the same attribute values





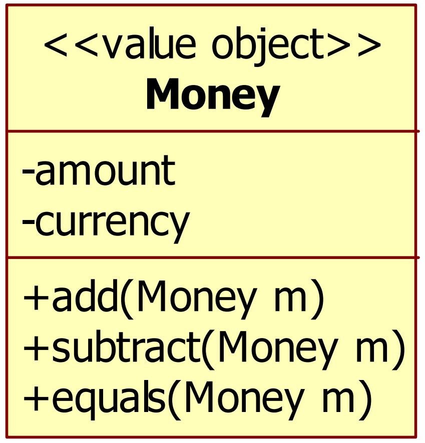
Behavior rich

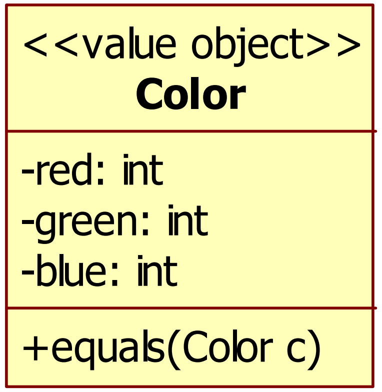
- Value objects should expose expressive domain-oriented behavior



Cohesive

- Encapsulate cohesive attributes





Immutable

- Once created, a value object can never be changed

```
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;
  }
}
```

No setter methods

Mutation leads to the creation of new instances

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;
   }
}
```

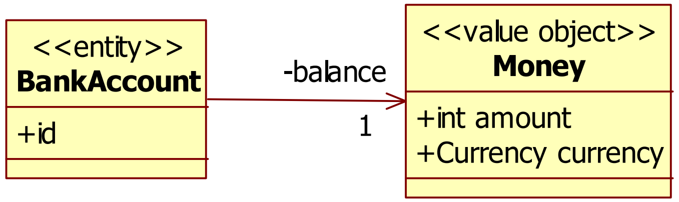
Self-validation

Testable

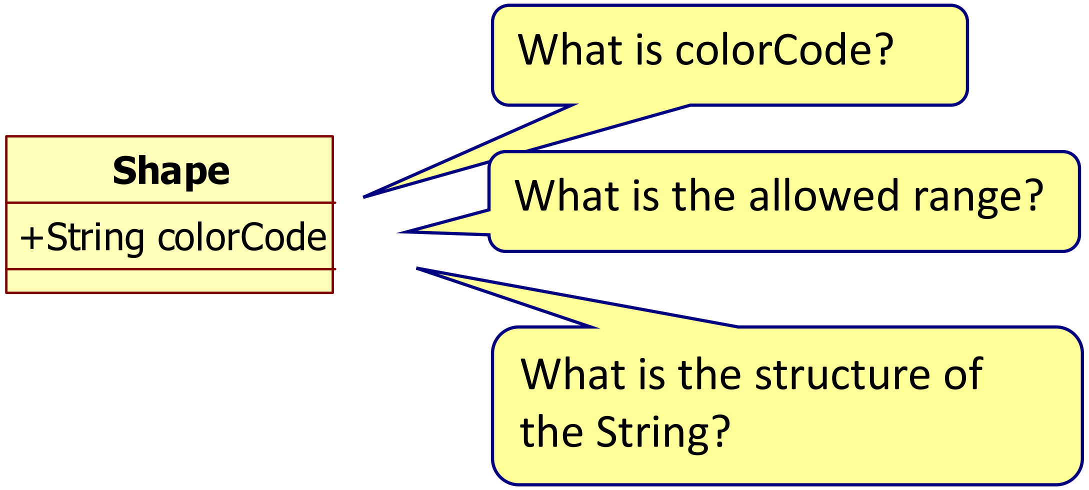
- Value objects are easy to test because of these qualities
  - Immutable
    - We don’t need mocks to verify side effects
  - Cohesion
    - We can test the concept in isolation
  - Combinability
    - Allows to express the relationship between 2 value objects

When to use value objects?

1. Representing a descriptive identity-less concept

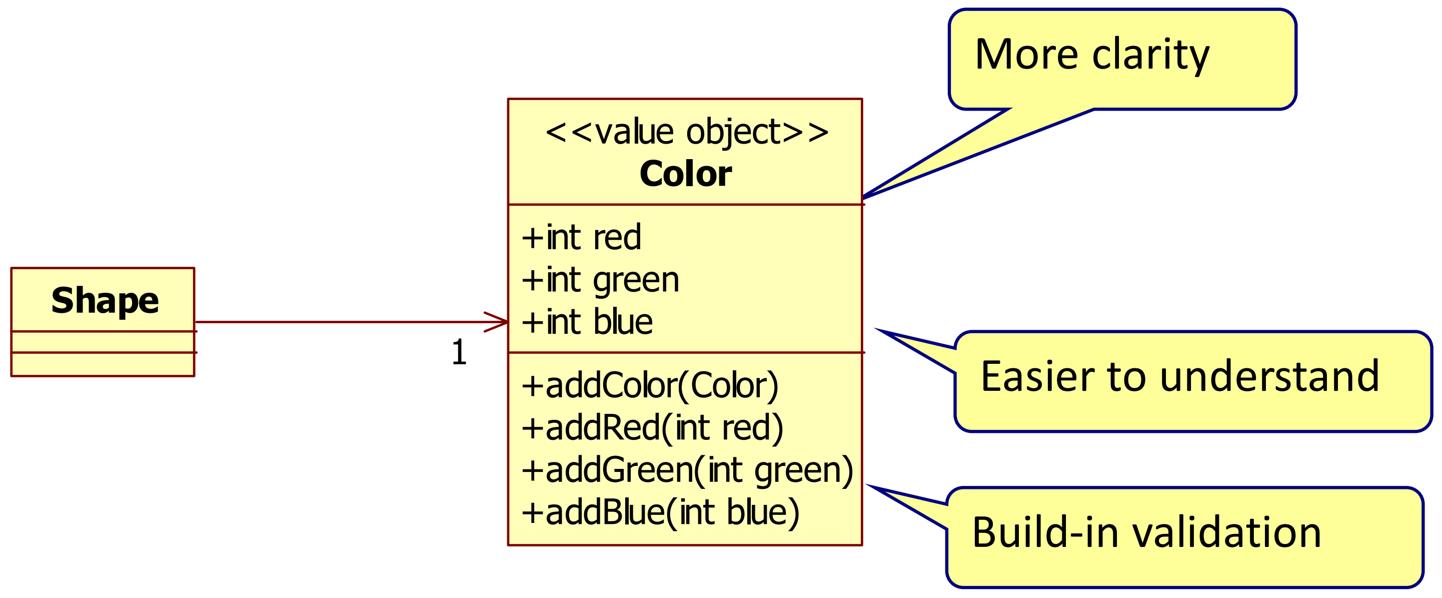


2. Enhancing explicitness



Shape
+String colorCode
What is colorCode?
What is the allowed range?
What is the structure of the String?

Enhancing explicitness



Static factory methods

```
public class Heigth {
  private enum MeasureUnit {
    METER,
    FEET,
    YARD;
  }

  private int value;
  private MeasureUnit unit;

  public Heigth(int value, MeasureUnit unit) {
    this.value = value;
    this.unit = unit;
  }

  public static Heigth fromFeet(int value) {
    return new Heigth(value, MeasureUnit.FEET);
  }

  public static Heigth fromMeters(int value) {
    return new Heigth(value, MeasureUnit.METER);
  }
}
```

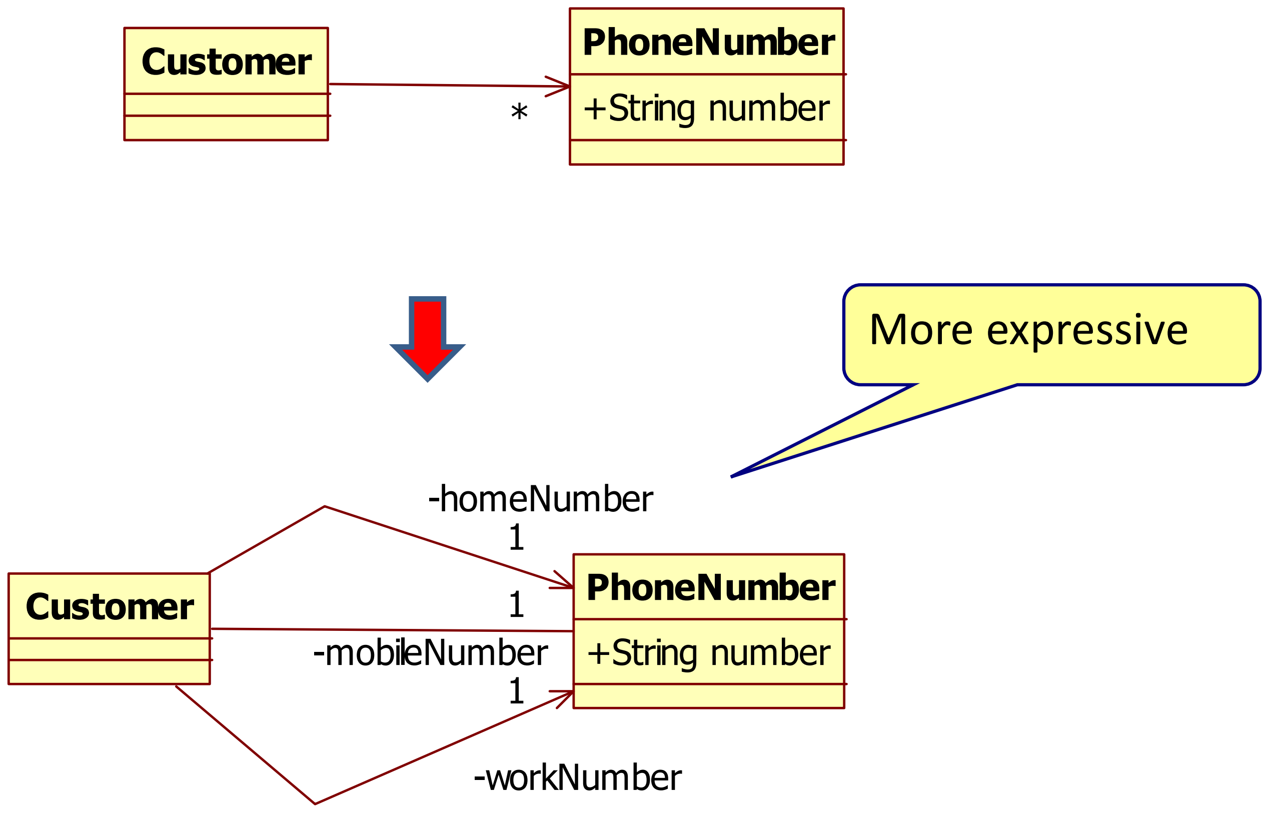
More expressive

Easier for clients to call

Decouple clients from MeasureUnit

Collection avoidance

- Be careful with collections of value objects

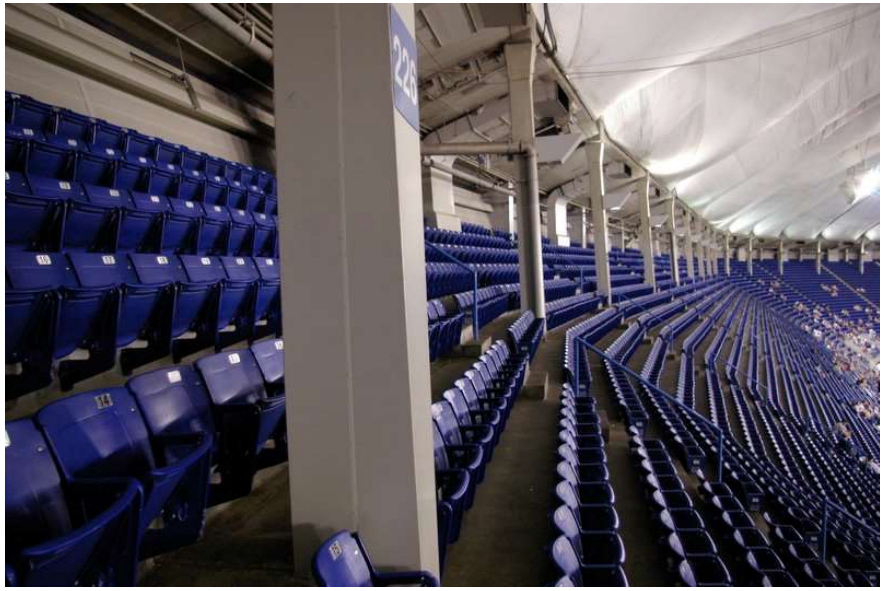


Persisting value objects

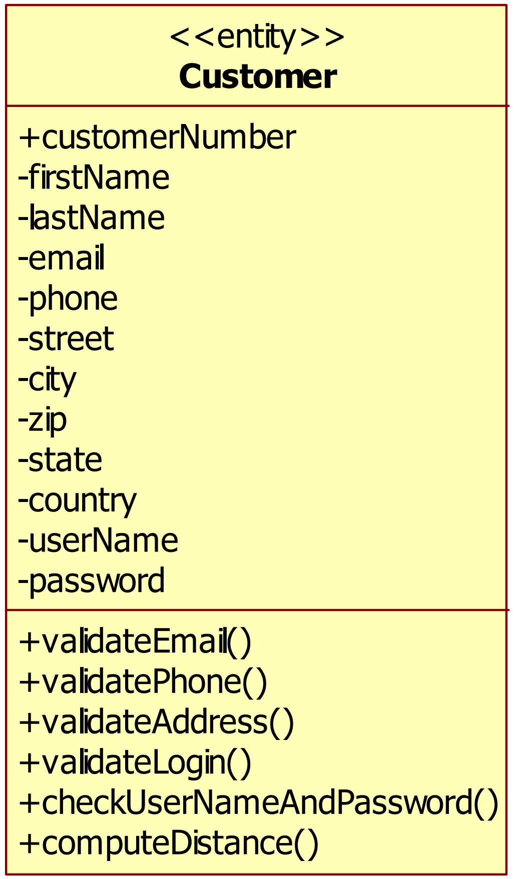
- Persist them into a denormalized form
  - Relational
    - Save them as String (using toString())
  - NoSQL
    - Embed them into the entity document
- Persist them into a separate relational table
  - Give the value object an id.

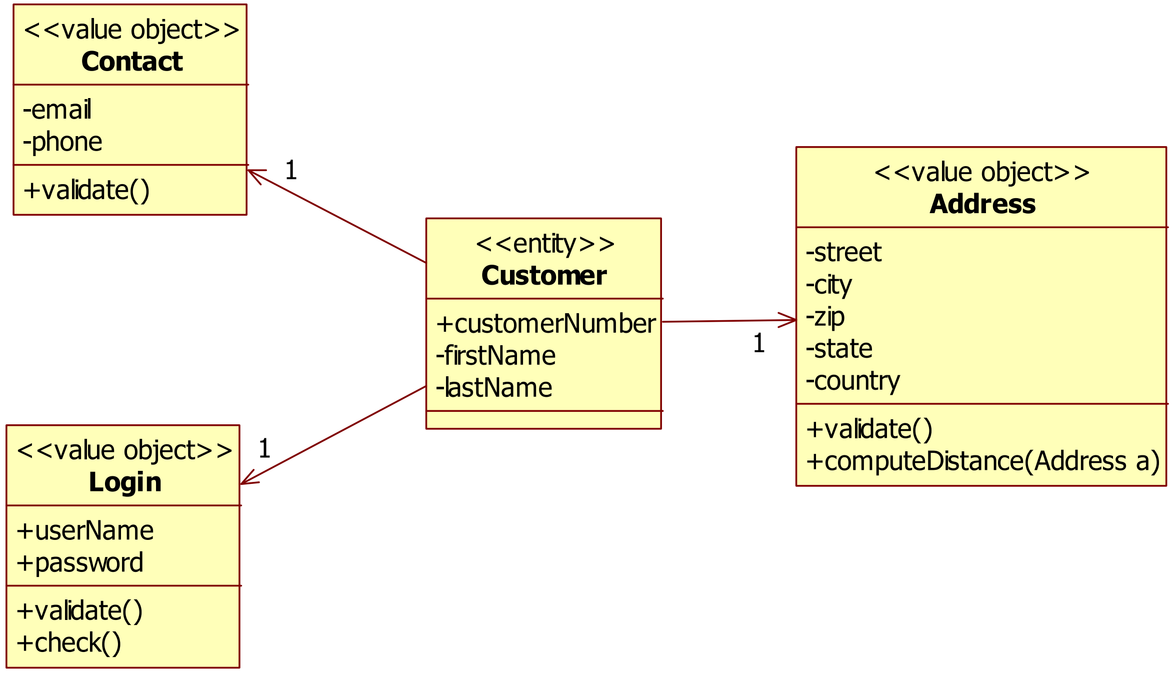
Entity versus value objects

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



Pushing behavior into value objects





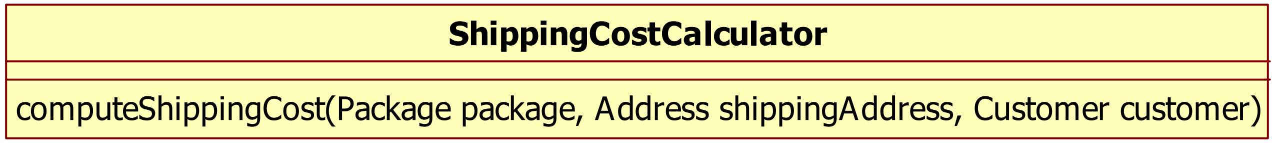
Entities versus Value objects

- 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
  - 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.
- 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.

DOMAIN SERVICES

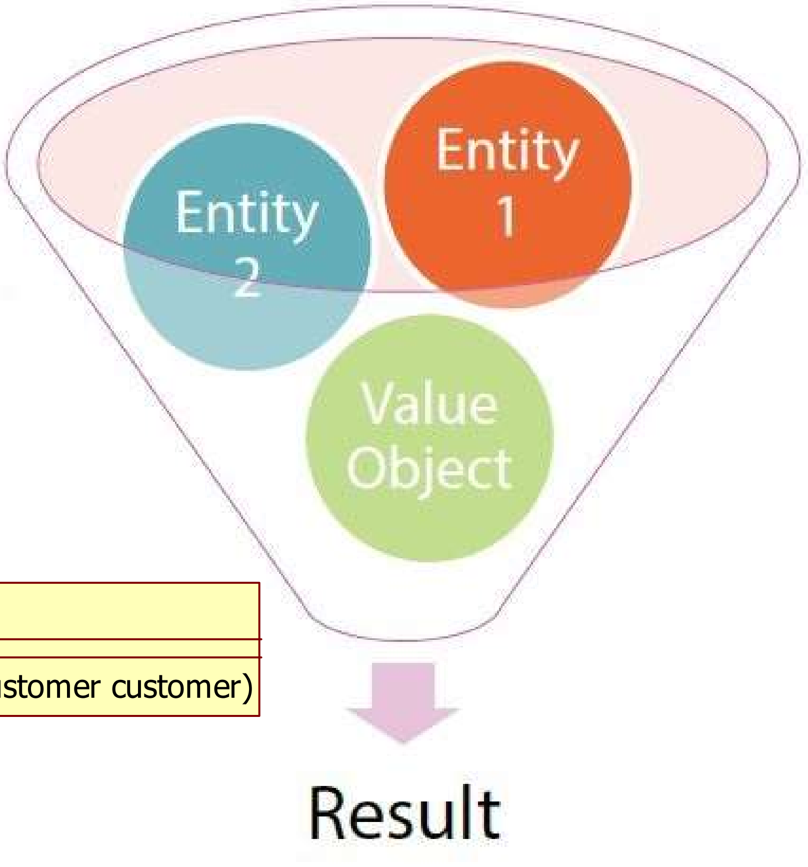
Domain service

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


ShippingCostCalculator
computeShippingCost(Package package, Address shippingAddress, Customer customer)

Domain service

- Interface is defined in terms of other domain objects

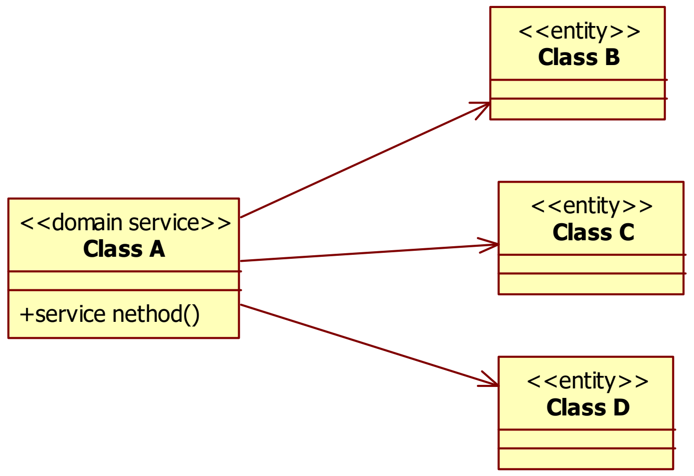


ShippingCostCalculator

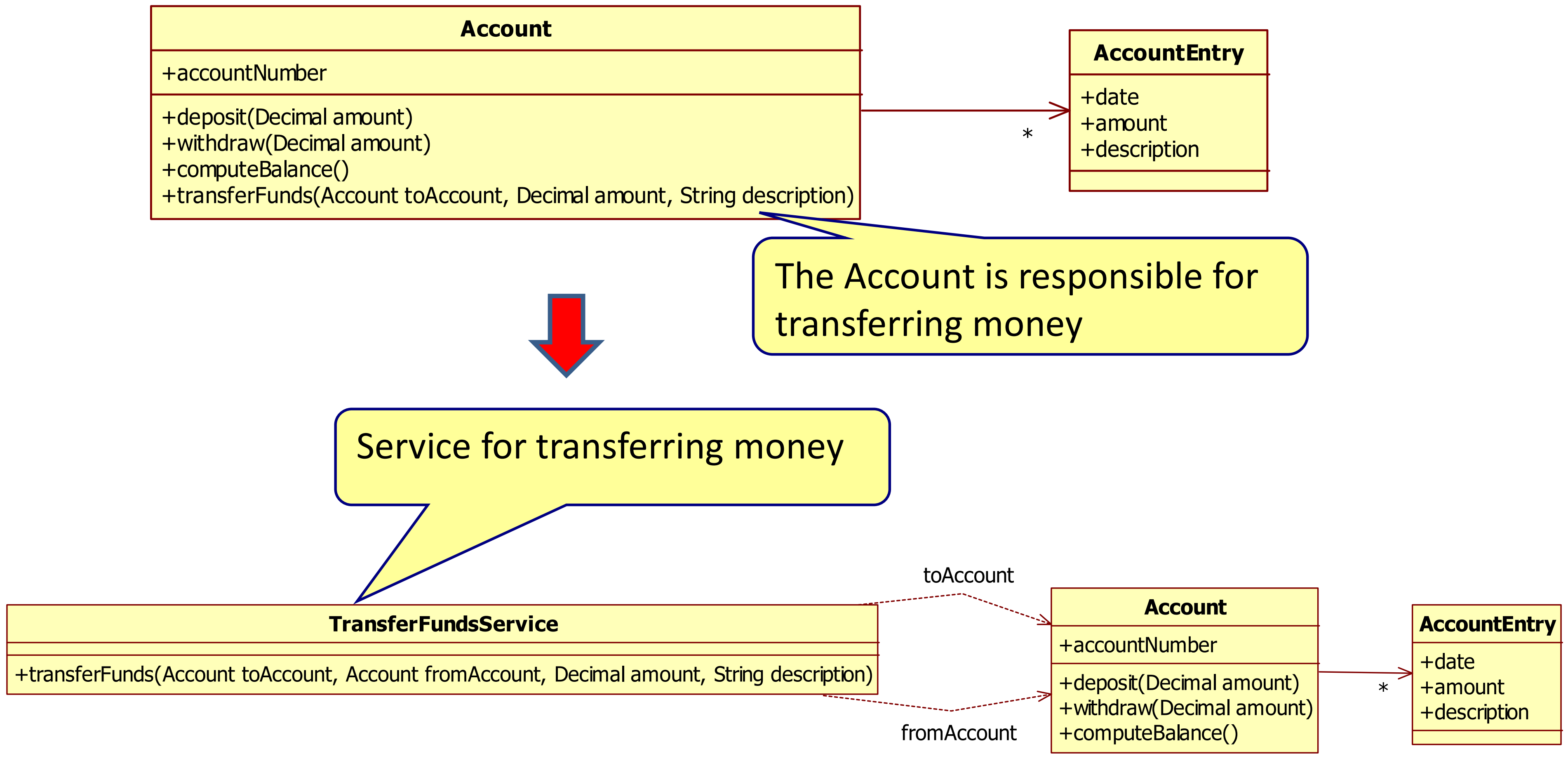
computeShippingCost(Package package, Address shippingAddress, Customer customer)

Domain service characteristics

- Stateless
  - Have no attributes
- Represent behavior
  - No identity
- Often orchestrate multiple domain objects



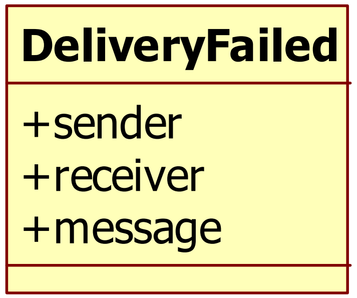
Domain service example

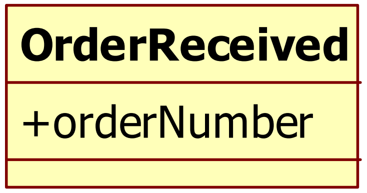

Account
+accountNumber
+deposit(Decimal amount)
+withdraw(Decimal amount)
+computeBalance()
+transferFunds(Account toAccount, Decimal amount, String description)
*
AccountEntry
+date
+amount
+description
The Account is responsible for transferring money
Service for transferring money
TransferFundsService
+transferFunds(Account toAccount, Account fromAccount, Decimal amount, String description)
toAccount
fromAccount
Account
+accountNumber
+deposit(Decimal amount)
+withdraw(Decimal amount)
+computeBalance()
*
AccountEntry
+date
+amount
+description

DOMAIN EVENTS

Domain event

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

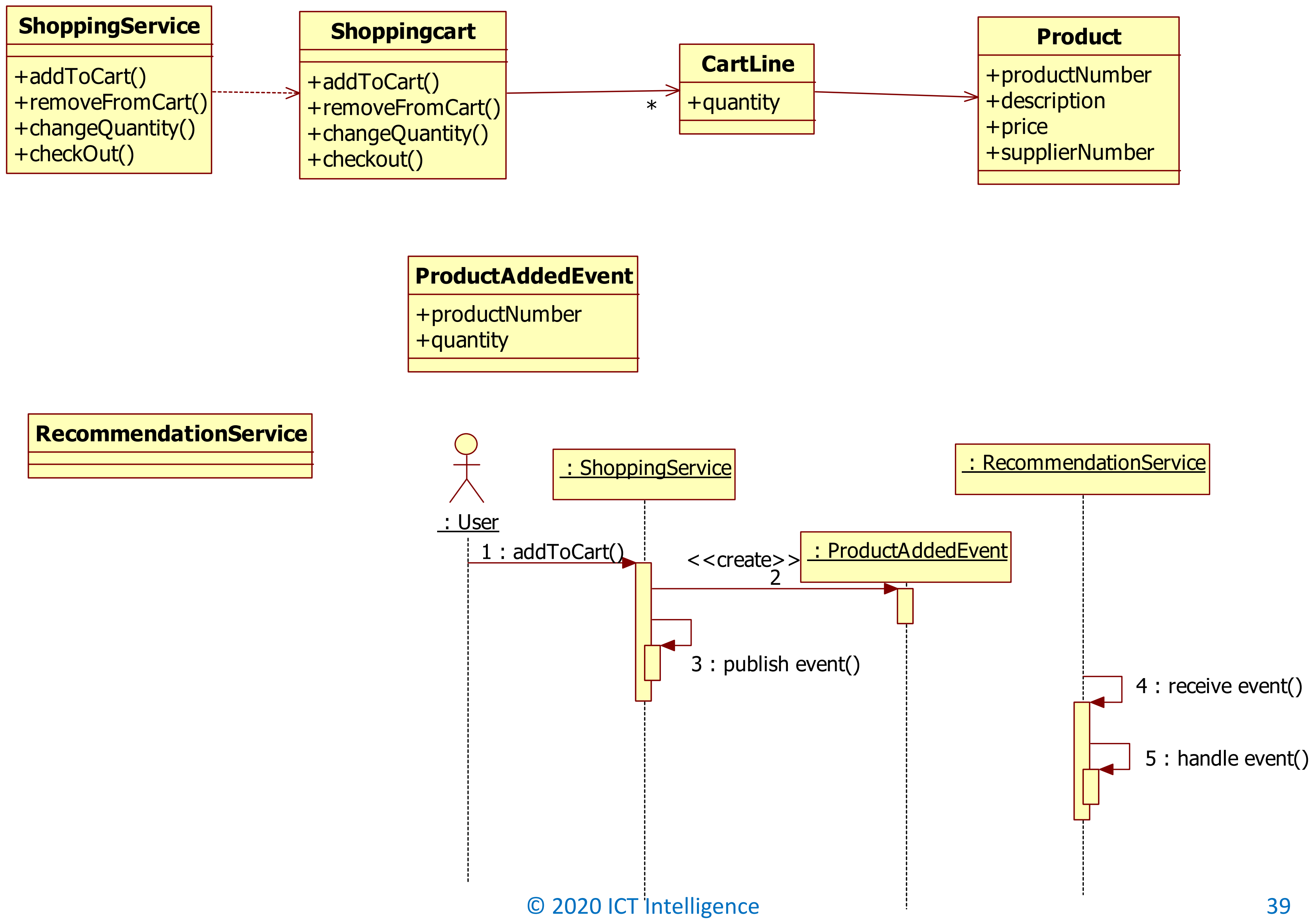




Domain event

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

Domain event example



ShoppingService
+addToCart()
+removeFromCart()
+changeQuantity()
+checkOut()
Shoppingcart
+addToCart()
+removeFromCart()
+changeQuantity()
+checkout()
*
CartLine
+quantity
Product
+productNumber
+description
+price
+supplierNumber
ProductAddedEvent
+productNumber
+quantity
RecommendationService
: User
: ShoppingService
: RecommendationService
1 : addToCart()
<<create>>
2
: ProductAddedEvent
3 : publish event()
4 : receive event()
5 : handle event()