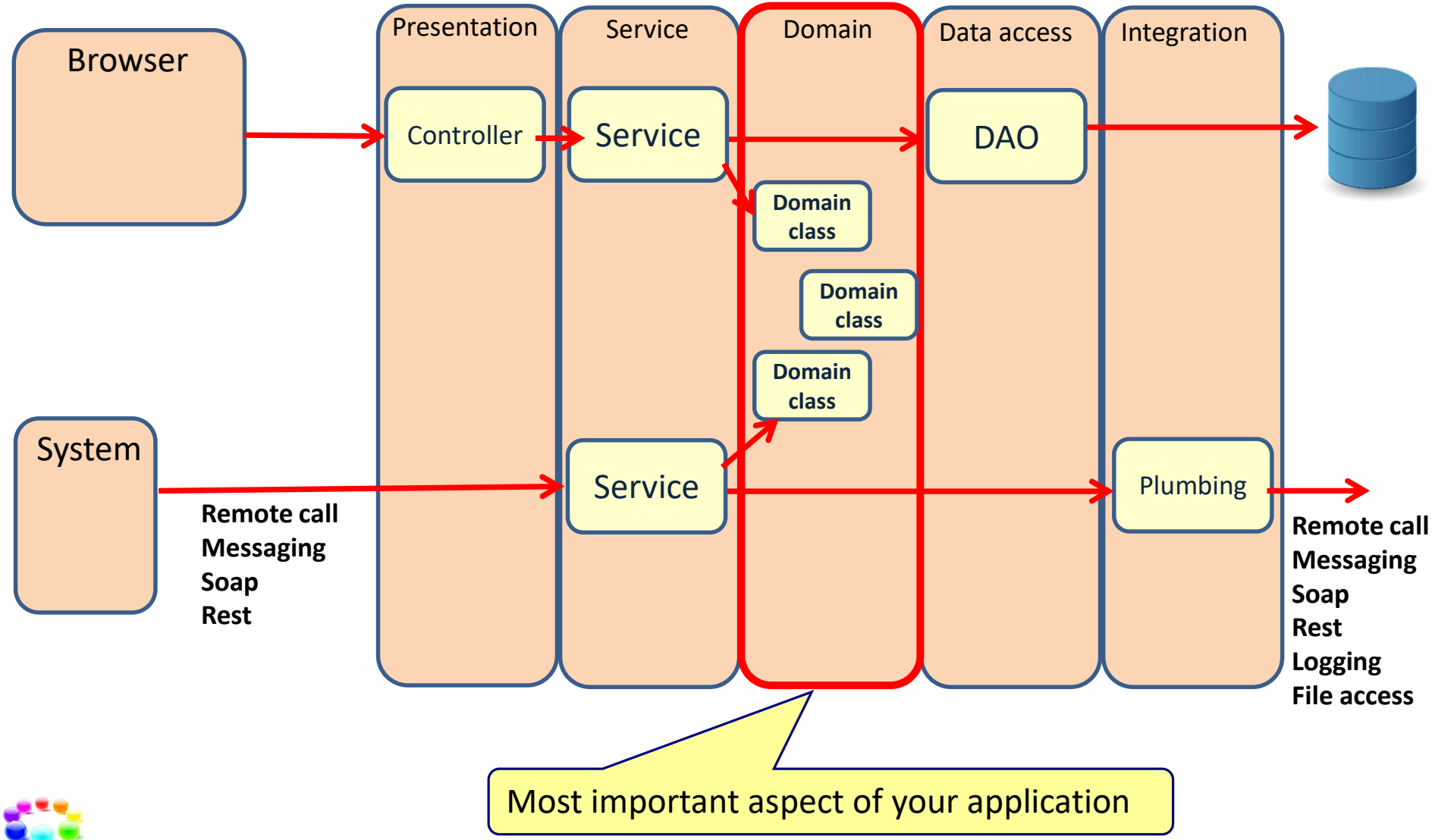
Lesson 3

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 Wiegers

People use their own language

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

domain expert

- Objects
- Databases
- HTML
- SQL
- XML messages

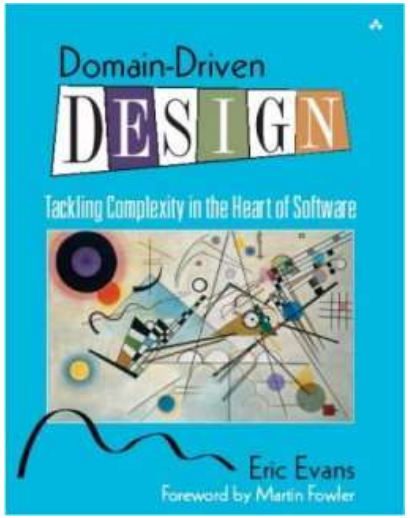
developer

- Applications
- Components
- Protocols
- Platforms
- Tooling

architect

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
  - Everything we do (discussions, design, coding, testing, documenting, etc.) is based on the domain model.

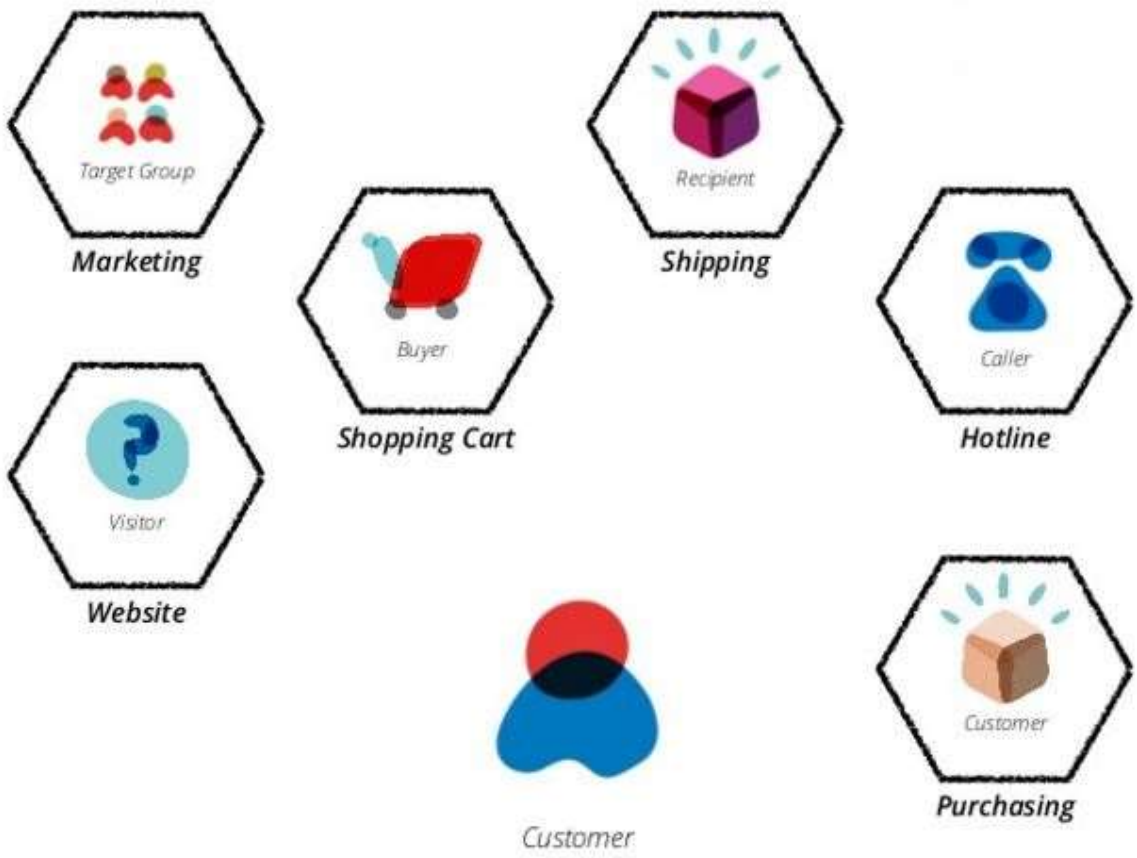


Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
  - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
  - Rich domain model
- Let the software be a reflection of the real world domain

Common language

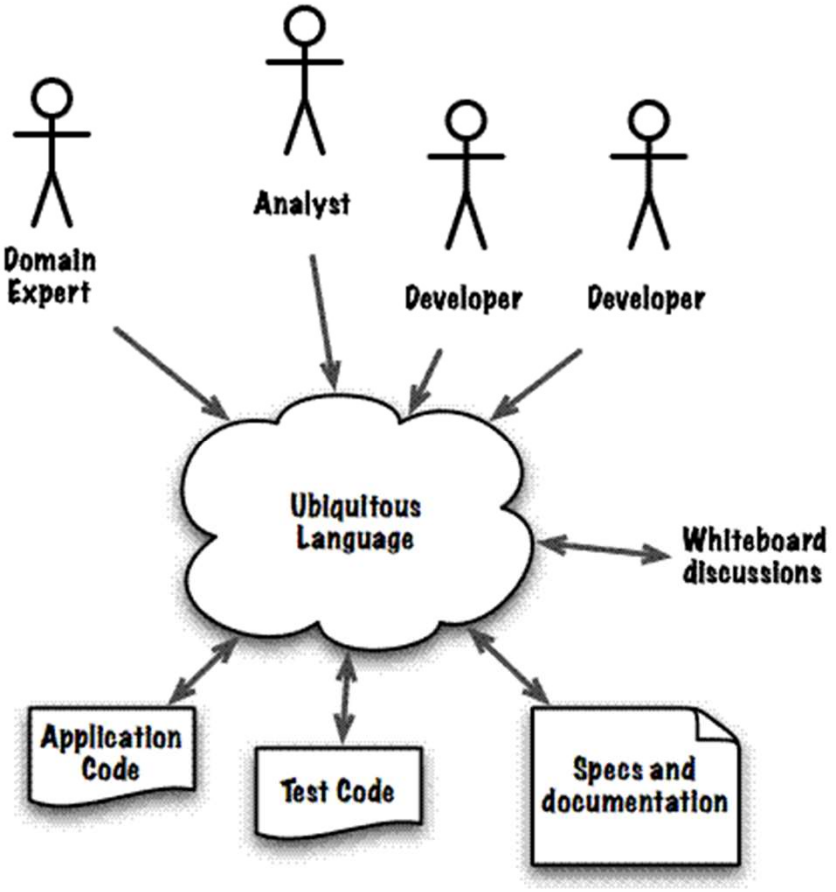
- Different people from the business use different names for the same thing.



We need a common language

Ubiquitous Language

- Language used by the team to capture the concepts and terms of a specific core business domain.
  - Used by the people
  - Used in the code
  - Used everywhere



Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
  - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
  - Rich domain model
- Let the software be a reflection of the real world domain

Model

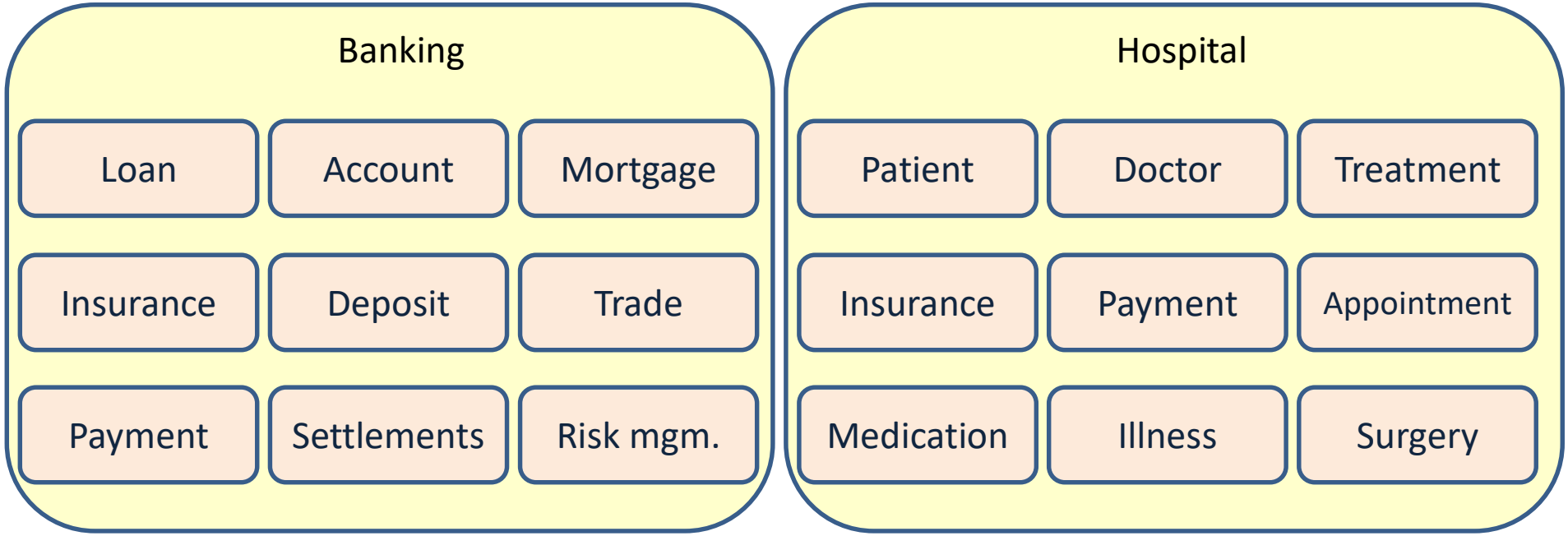
- More complexity -> More modeling
  - Higher level of abstraction
  - Allows for visualization
  - Vehicle of communication

Domain model

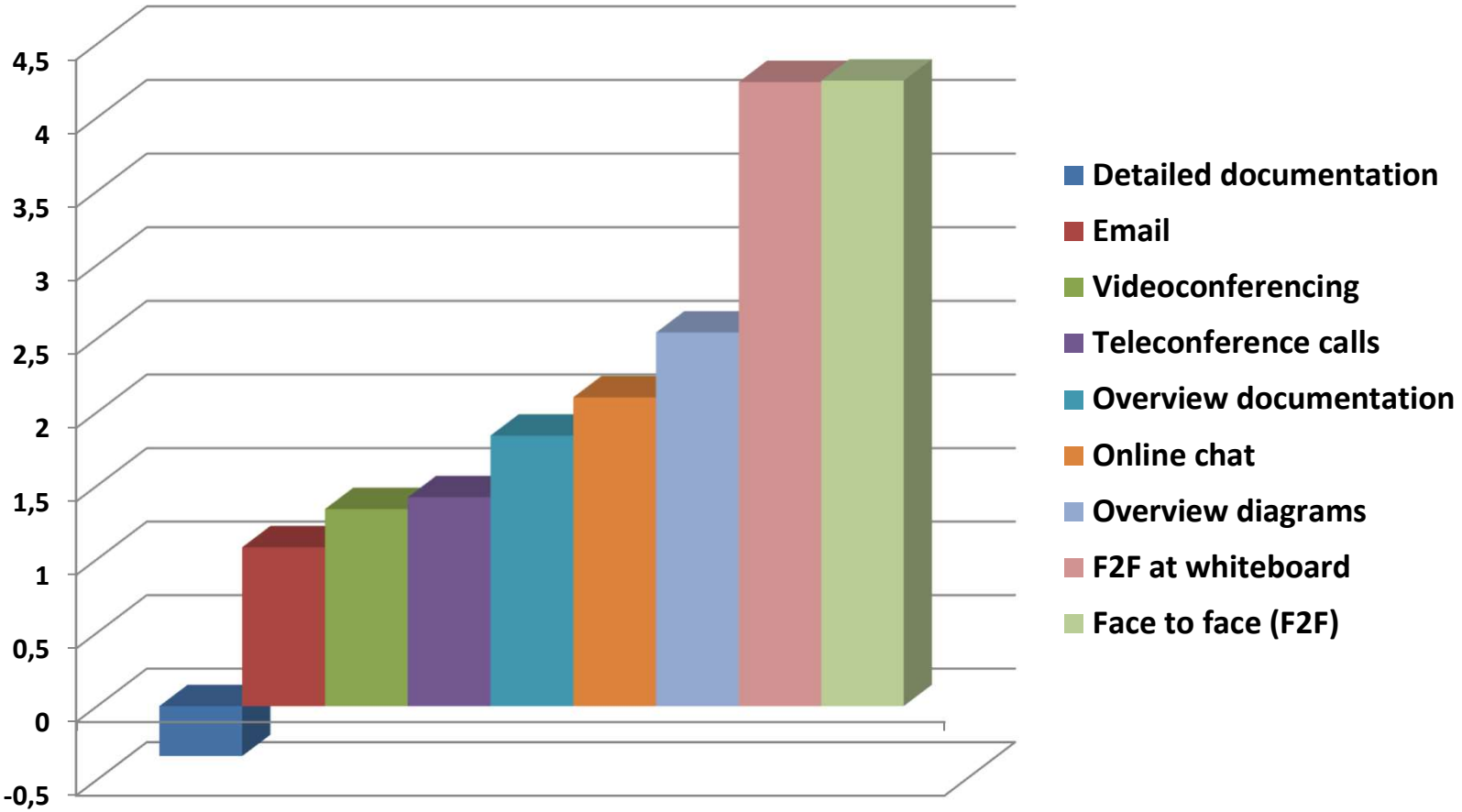
- Simplification of reality
- Area of interest

Domain model

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



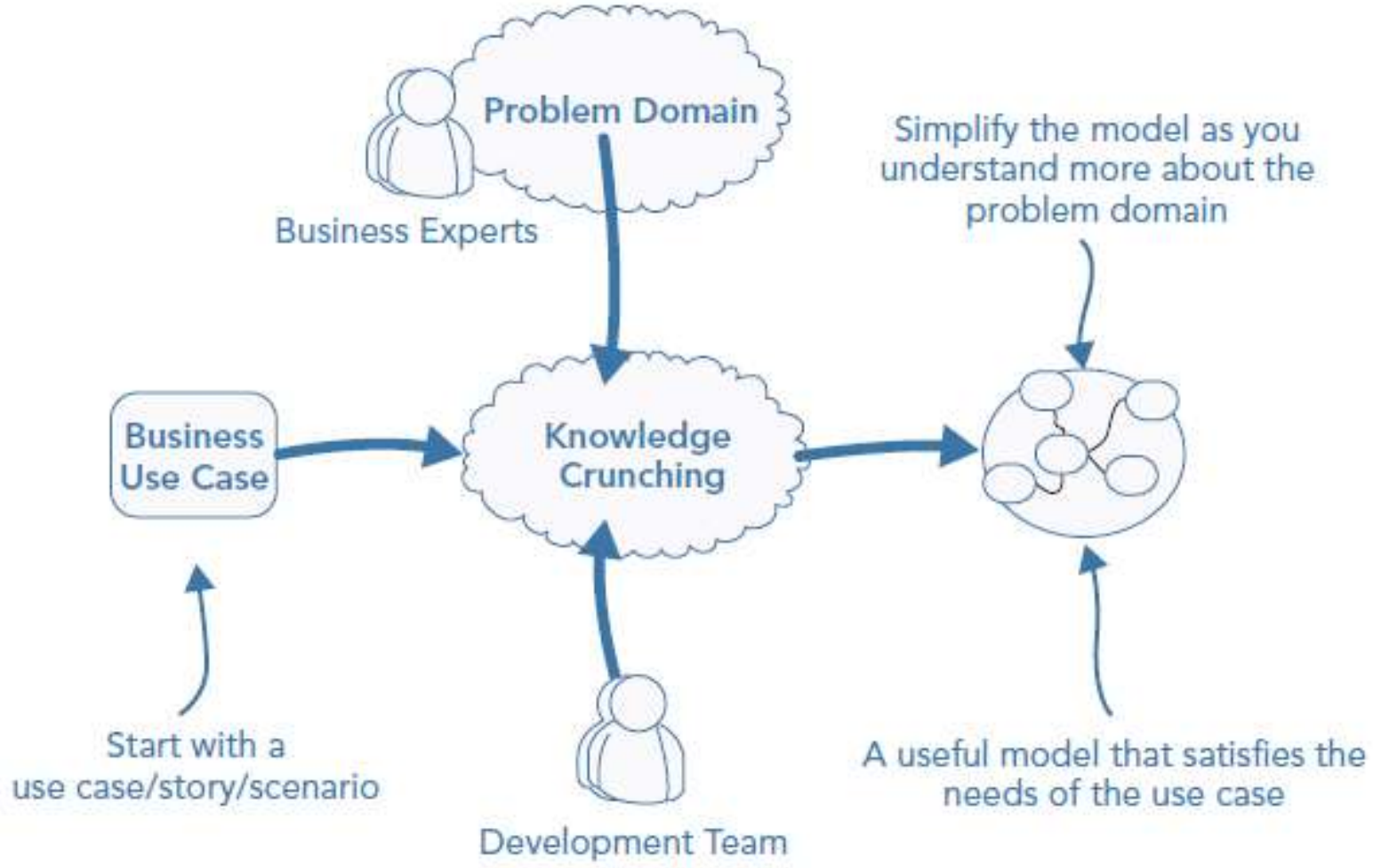
Effectiveness of communication



Advantages of a domain model

- Improves understanding
- Validates understanding
- Improves communication
- Shared glossary
- Improves discovery

Knowledge crunching



Principles of Domain Driven Design

- Use one common language to describe the concepts of a domain
  - Ubiquitous language
- Create a domain model that shows the important concepts of the domain
  - Rich domain model
- Let the software be a reflection of the real world domain

The software is a reflection of the real world

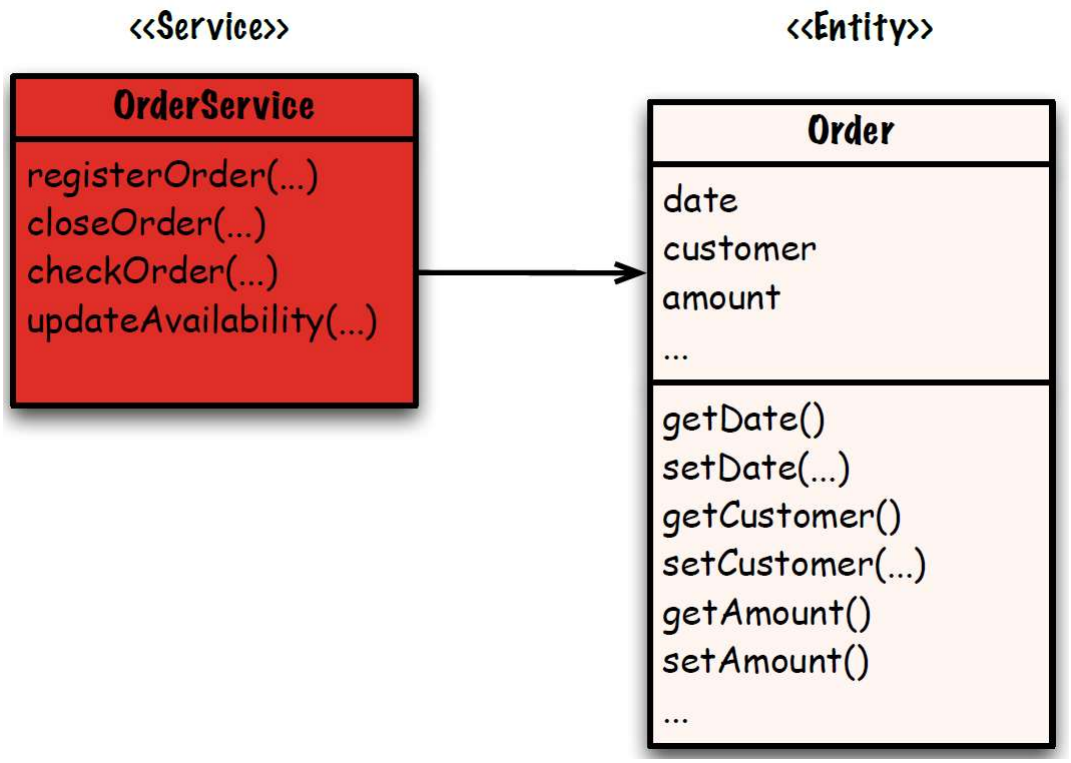
- It is easier to spot inconsistencies, errors, misconceptions.
- The software is easier to understand for
  - Existing developers
  - Testers
  - Business people (with guidance)
  - New developers and testers
- By looking at the code you can learn a lot of domain knowledge
- No translation necessary
- It is easier to write tests
- Easier to maintain the code

ANEMIC AND RICH DOMAIN MODEL

Anemic domain model

- Classes in the model have no business logic

NOT OK



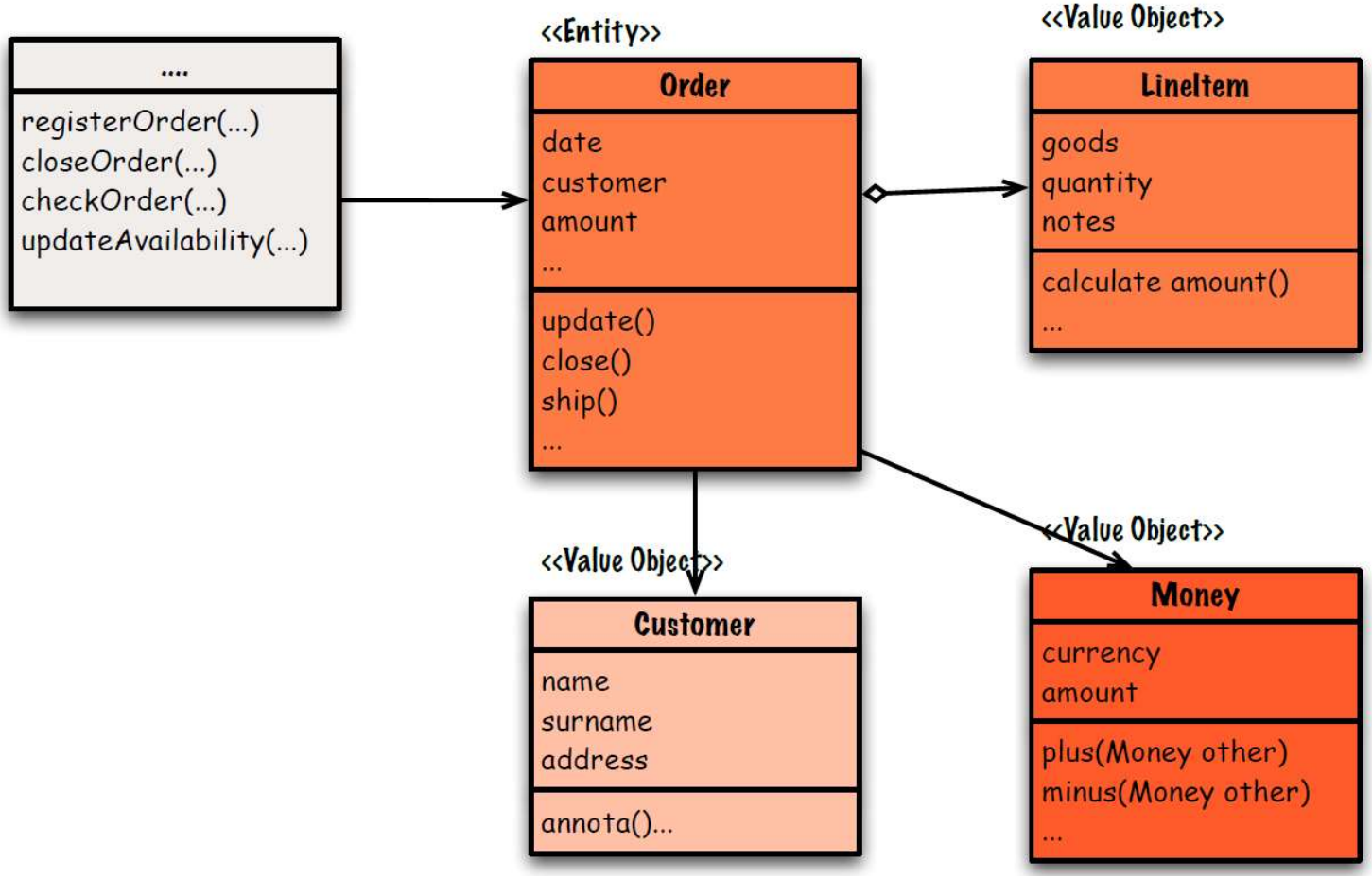
Disadvantages anemic domain model

- 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

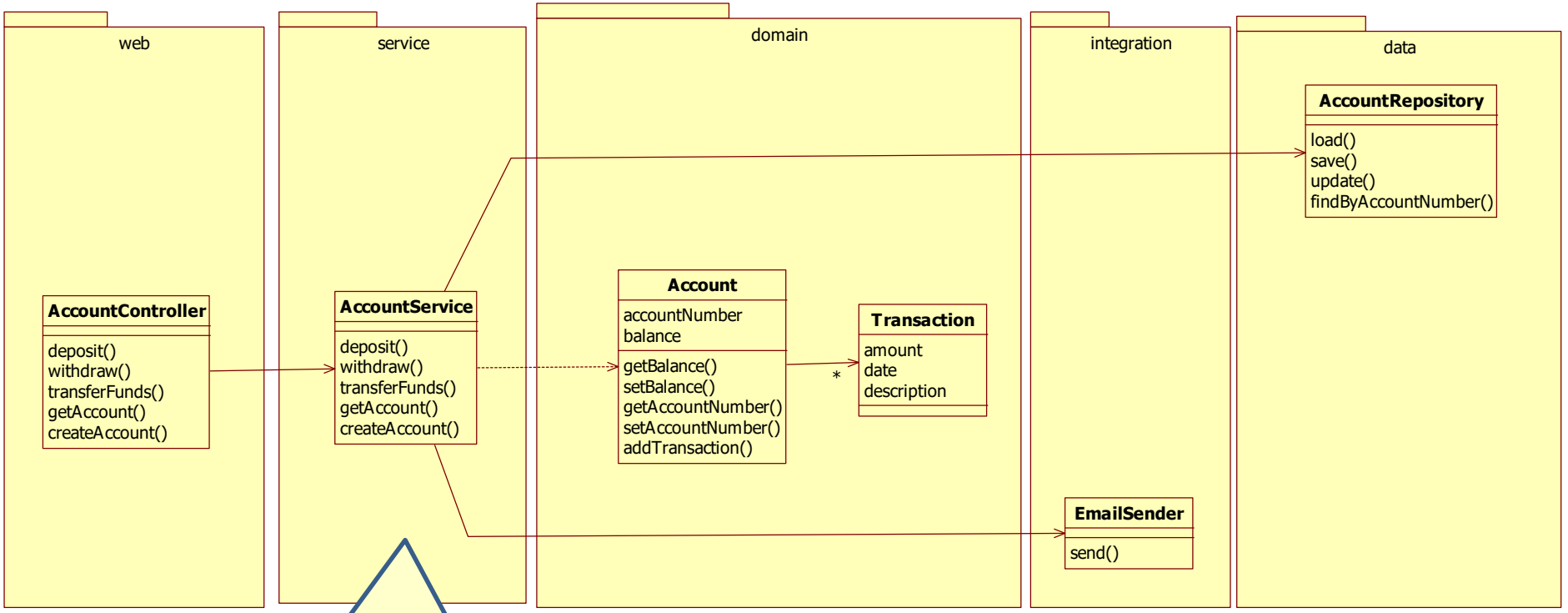
Rich domain model

- Classes with business logic

OK

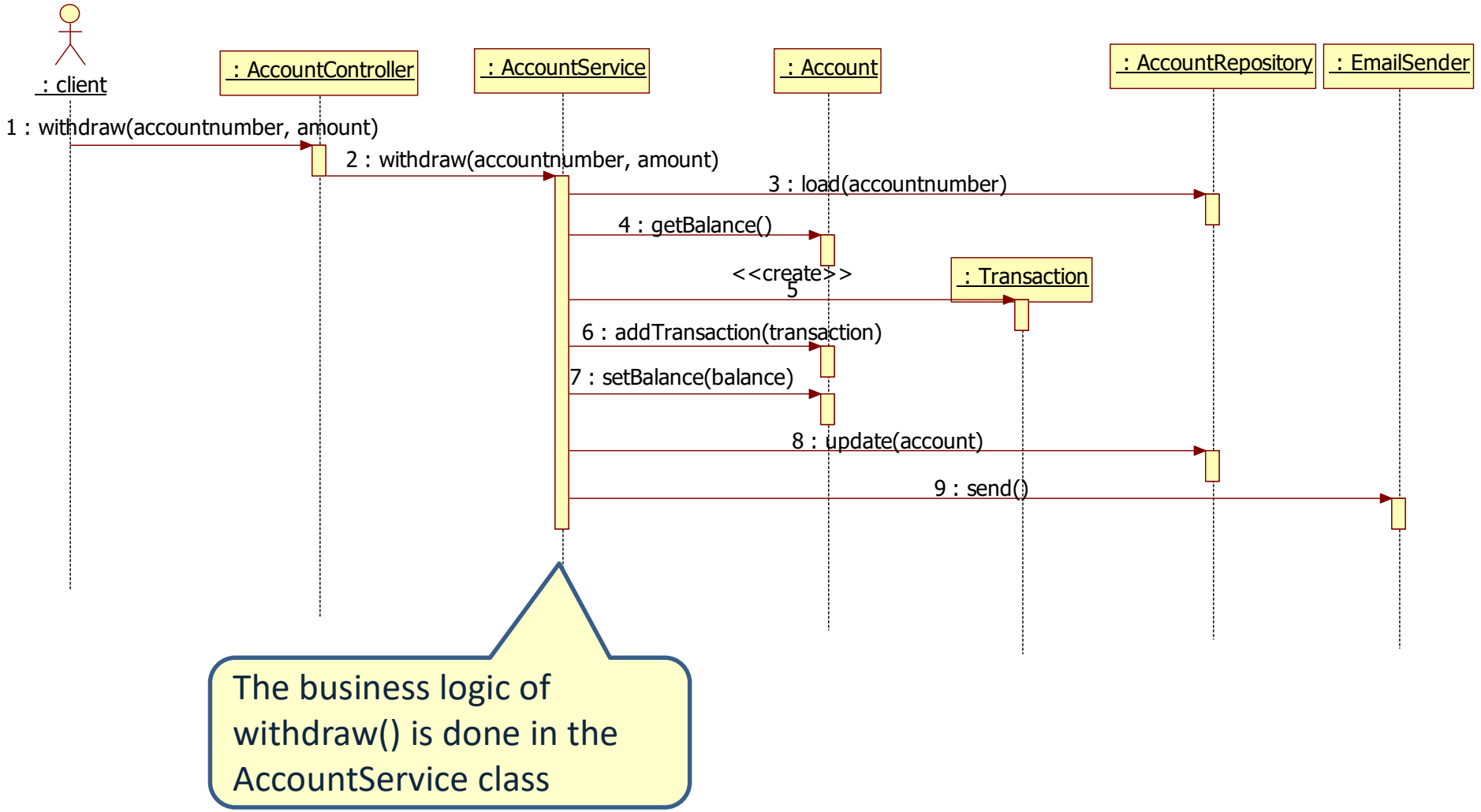


Anemic domain model example

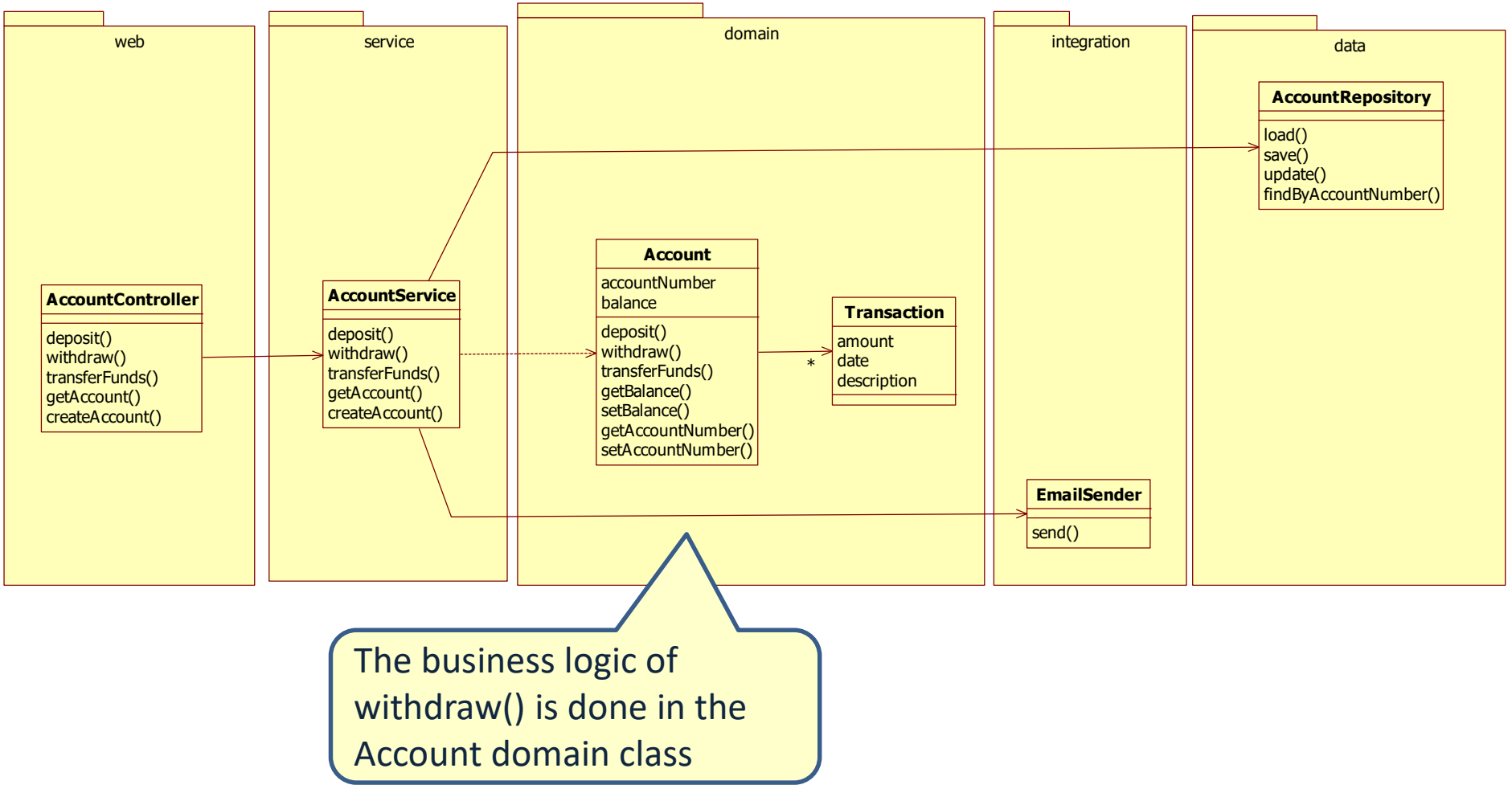


The business logic of withdraw() is done in the AccountService class

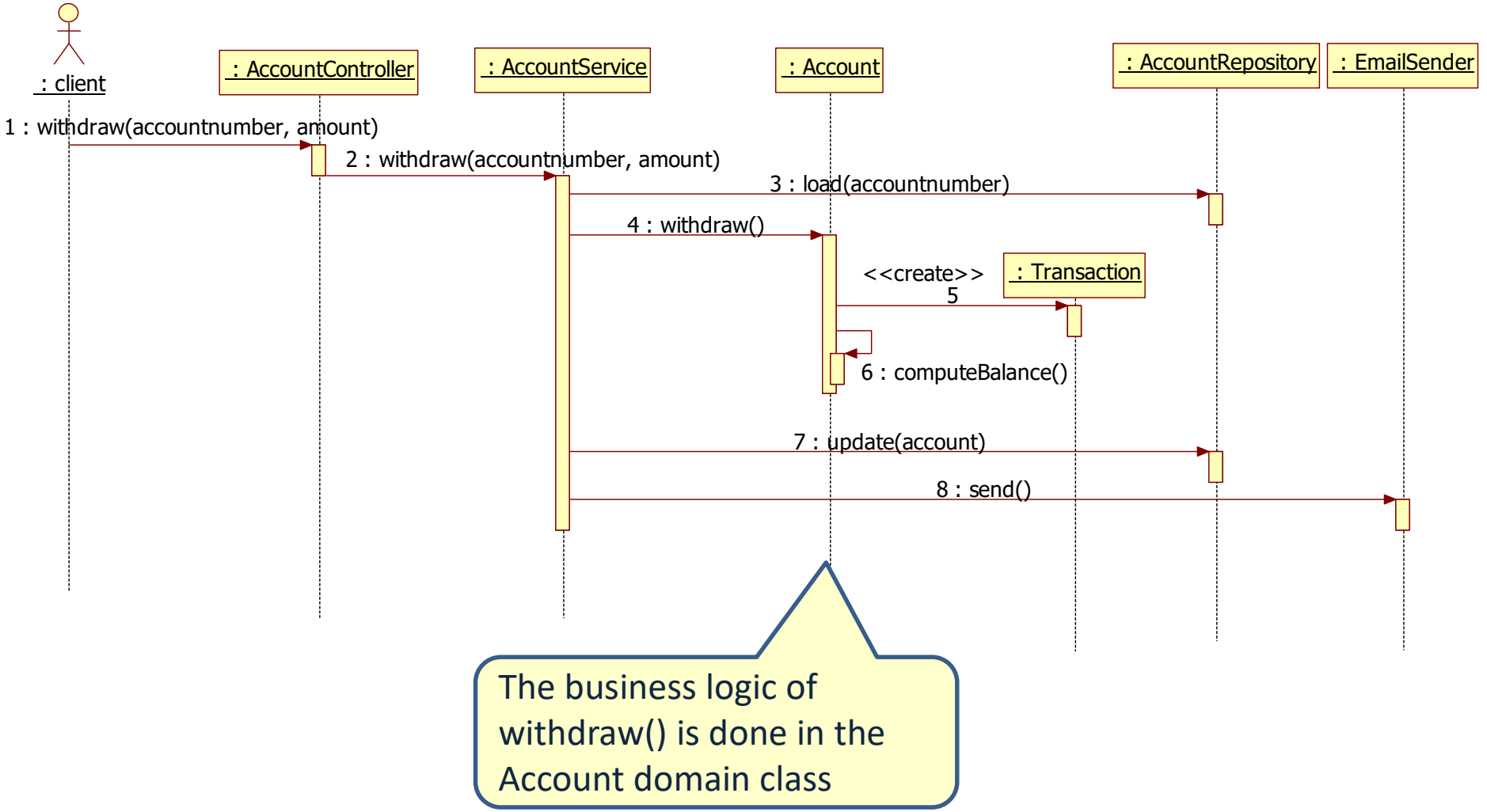
Anemic domain model example



Rich domain model example



Rich domain model example



Main point

- A rich domain model helps in communicating complex domain knowledge
- The daily experience of pure consciousness helps in a more happy and successful life.

ORCHESTRATION & CHOREOGRAPHY

Orchestration vs. choreography

- Orchestration
  - One central brain

Easy to follow the process

Does not work well in large and or complex applications

- Choreography
  - No central brain

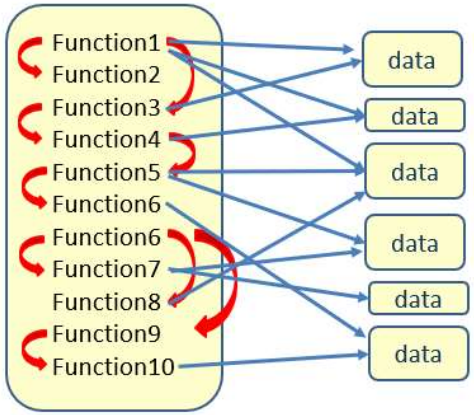
Hard to follow the process

Does work well in large and or complex applications

Orchestration vs. choreography

- Orchestration
  - One central brain

Procedural programming
(C, Pascal, Algol, Cobol)

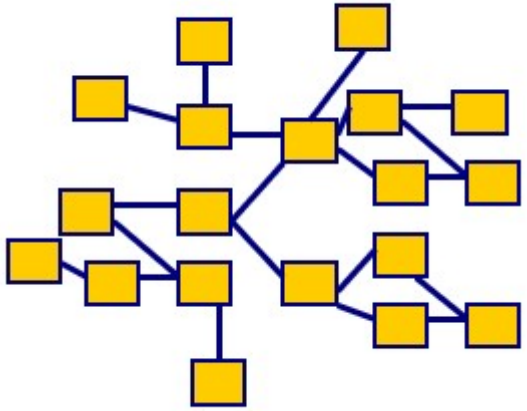


Easy to follow the process

Does not work well in large and or complex applications

- Choreography
  - No central brain

Object-Oriented programming
(Java, C#, Python, C++)



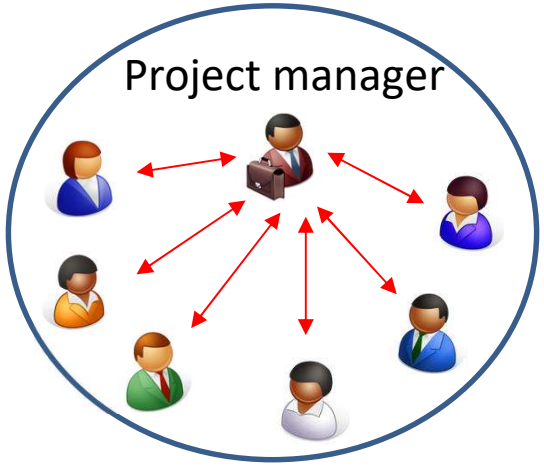
Hard to follow the process

Does work well in large and or complex applications

Orchestration vs. choreography

- Orchestration
  - One central brain

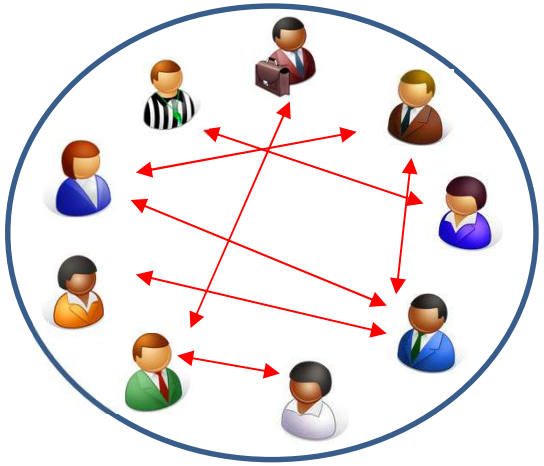
Waterfall



Does not work well complex projects

- Choreography
  - No central brain

Agile

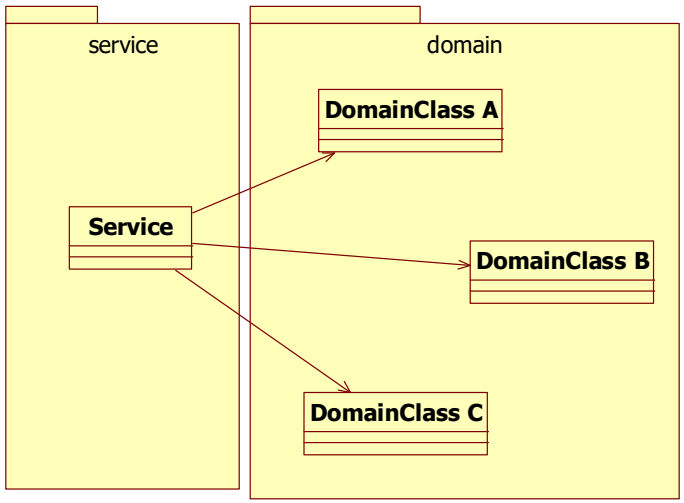


Does work well in complex projects

Orchestration vs. choreography

- Orchestration
  - One central brain

Anemic domain model

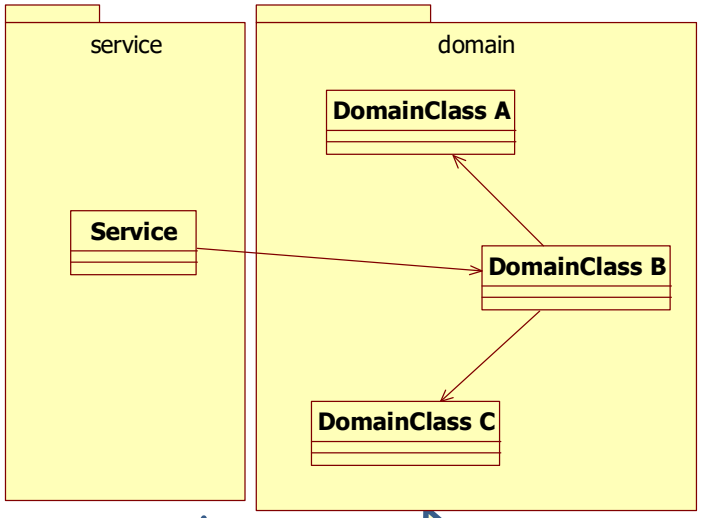


Easy to follow the process

Does not work well in large and or complex applications

- Choreography
  - No central brain

Rich domain model



Hard to follow the process

Does work well in large and or complex applications

DOMAIN MODEL PATTERNS

Domain Model Patterns

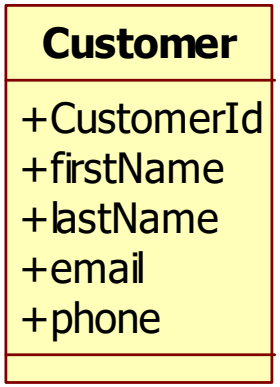
- Entities
- Value objects
- Domain services
- Domain events

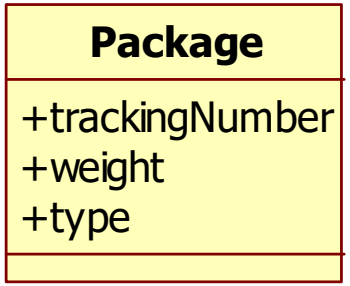
ENTITIES

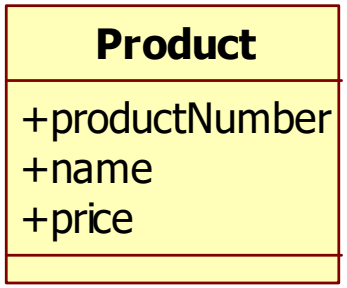
Entities

- 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

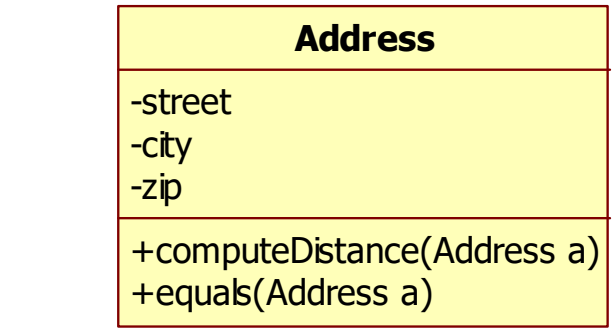
- Changing attributes doesn't change which one we're talking about
  - Identity remains constant throughout its lifetime

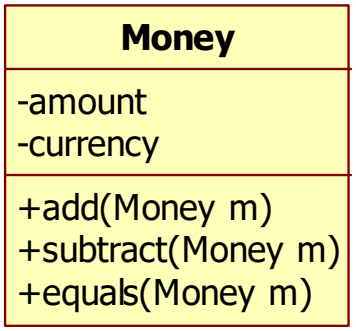
VALUE OBJECTS

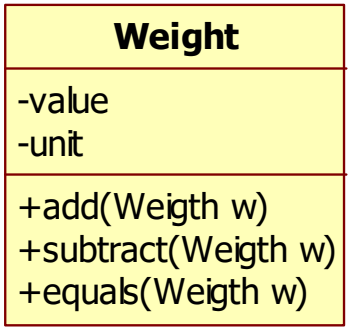
Value objects

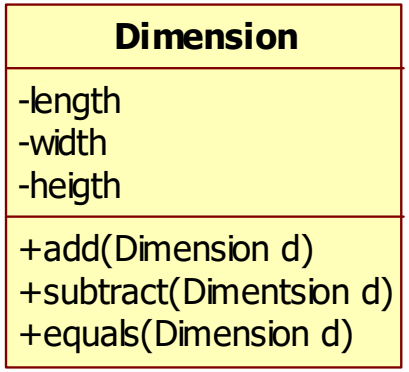
- Has no identity
  - Identity is based on composition of its values
- Immutable
  - State cannot be changed after instantiation

Example value object classes







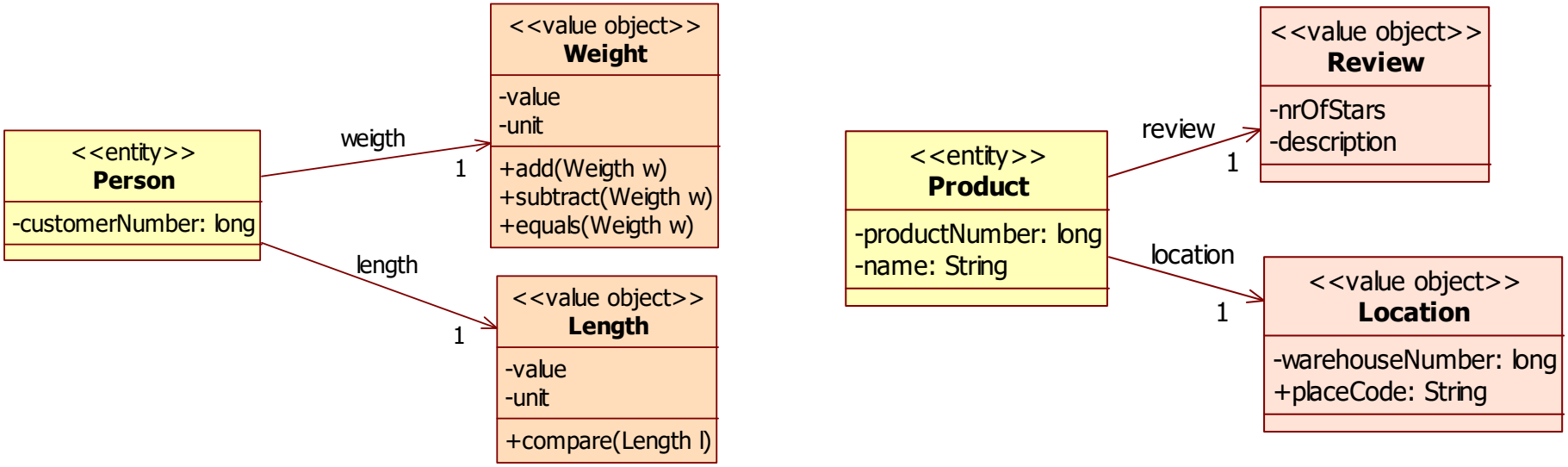


Value object characteristics

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

No identity

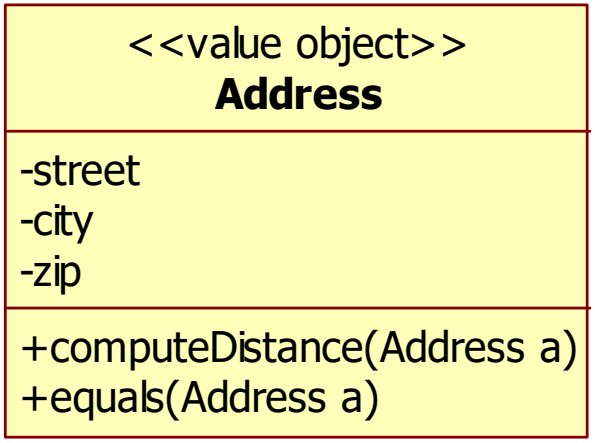
- Value objects tell something about another object

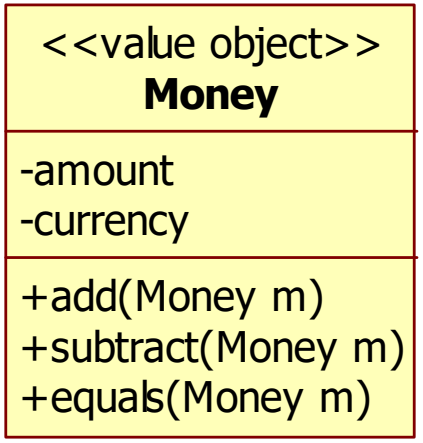


- Technically, value objects may have IDs using some database persistence strategies.
  - But they have no identity in the domain.

Attribute-based equality

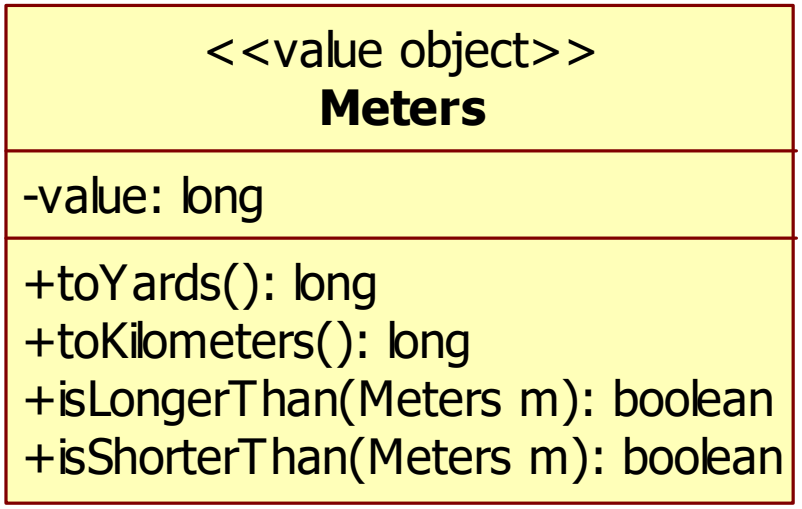
- 2 value objects are equal if they have the same attribute values





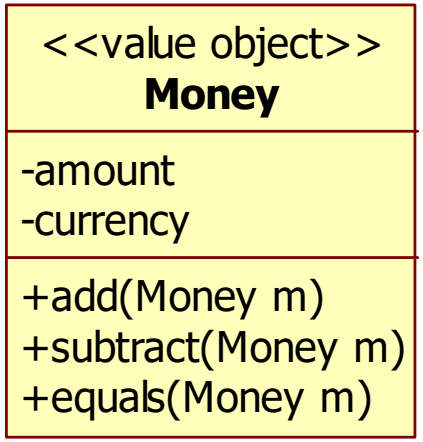
Behavior rich

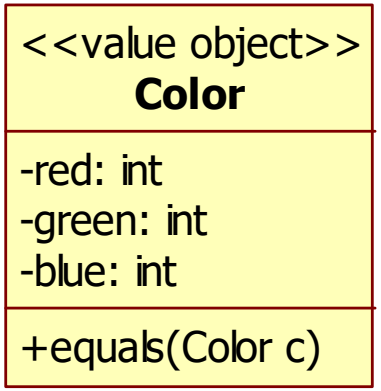
- Value objects should expose expressive domain-oriented behavior



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

Static factory methods

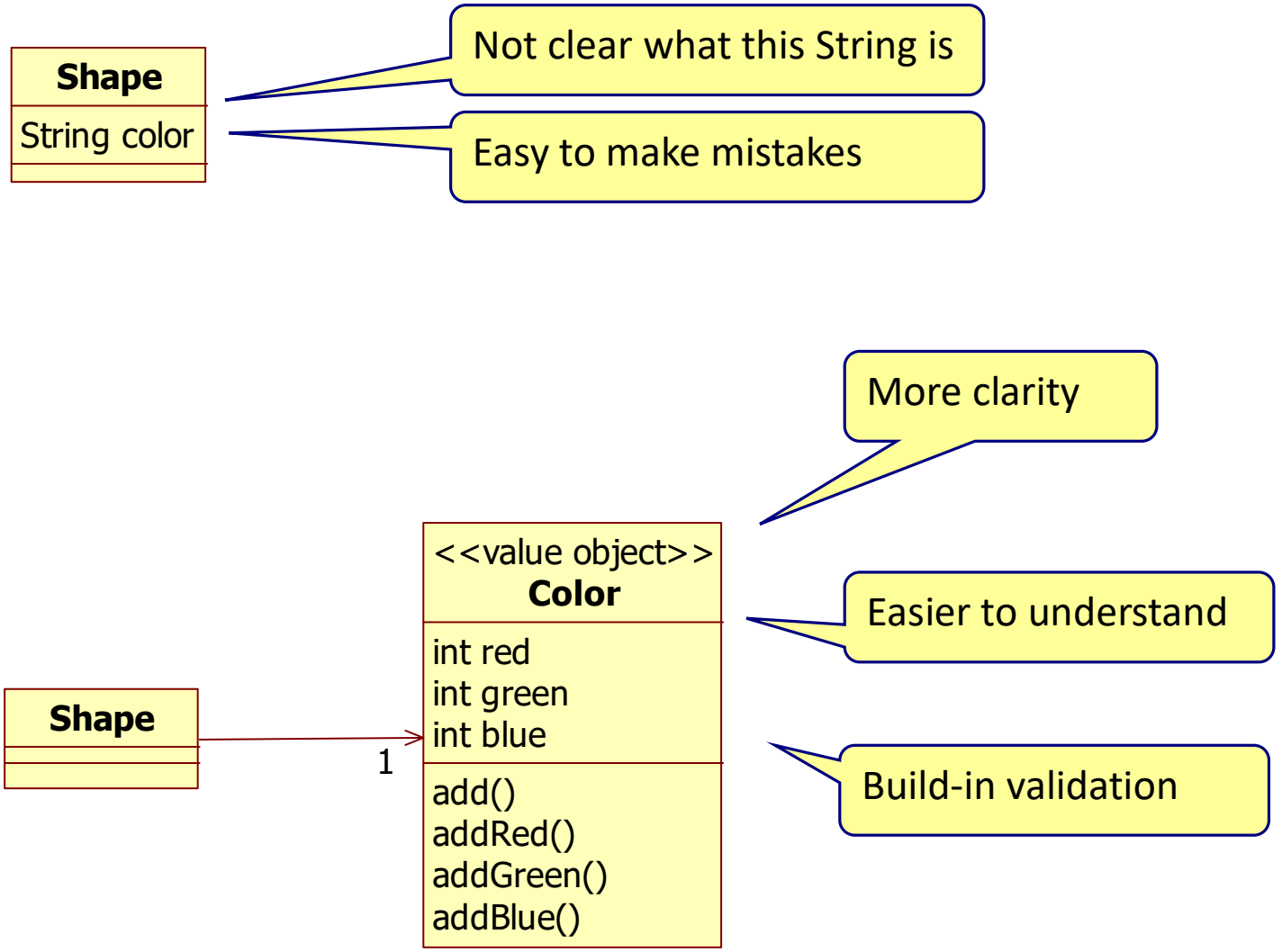
```
public class Heigth {
  private enum MeasureUnit {
    METER,
    FEET,
    YARD;
  }

  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

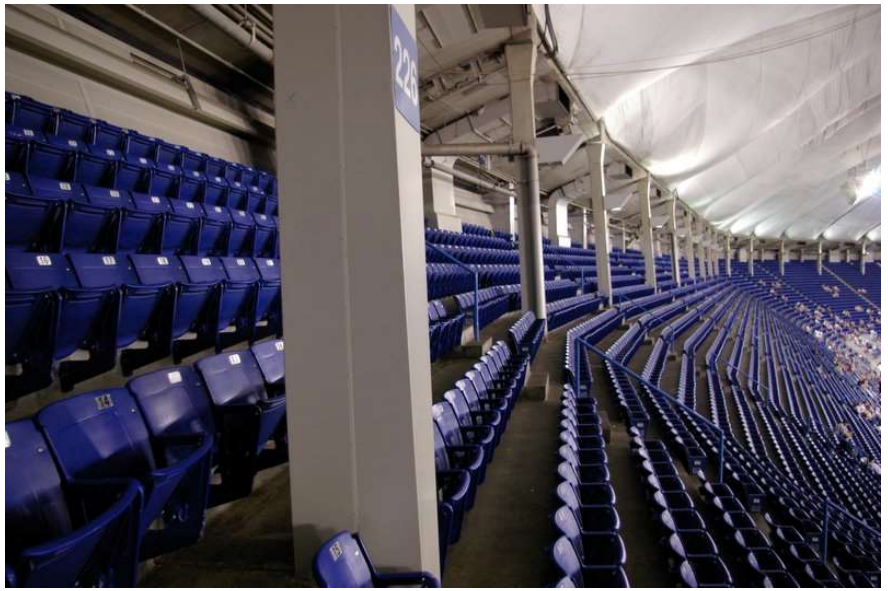
Decouple clients from MeasureUnit

Enhancing explicitness



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

<<entity>>
Customer

+customerNumber
-firstName
-lastName
-email
-phone
-street
-city
-zip
-state
-country
-userName
-password

+validateEmail()
+validatePhone()
+validateAddress()
+validateLogin()
+checkUserNameAndPassword()
+computeDistance()

<<value object>>
Contact

-email
-phone

+validate()

1

<<entity>>
Customer

+customerNumber
-firstName
-lastName

1

<<value object>>
Address

-street
-city
-zip
-state
-country

+validate()
+computeDistance(Address a)

1

<<value object>>
Login

+userName
+password

+validate()
+check()

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
- Always prefer value objects over entities in your domain model.

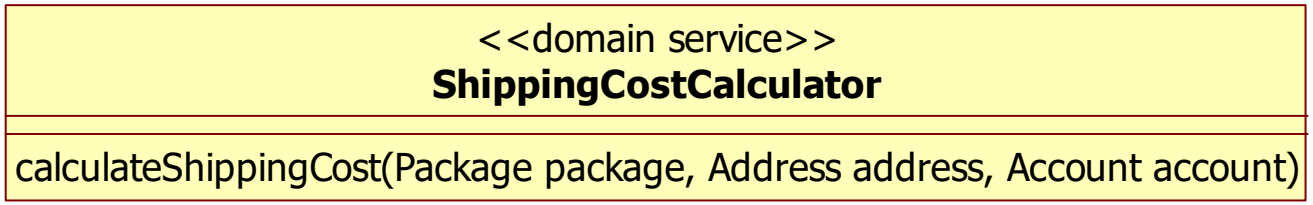
Main point

- 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

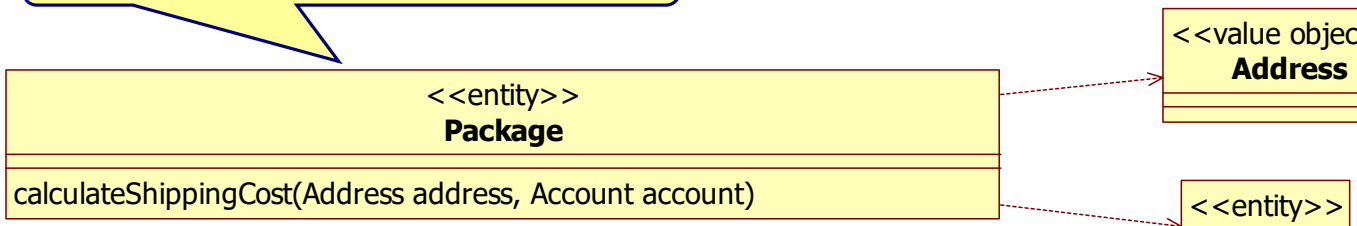
Domain service

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

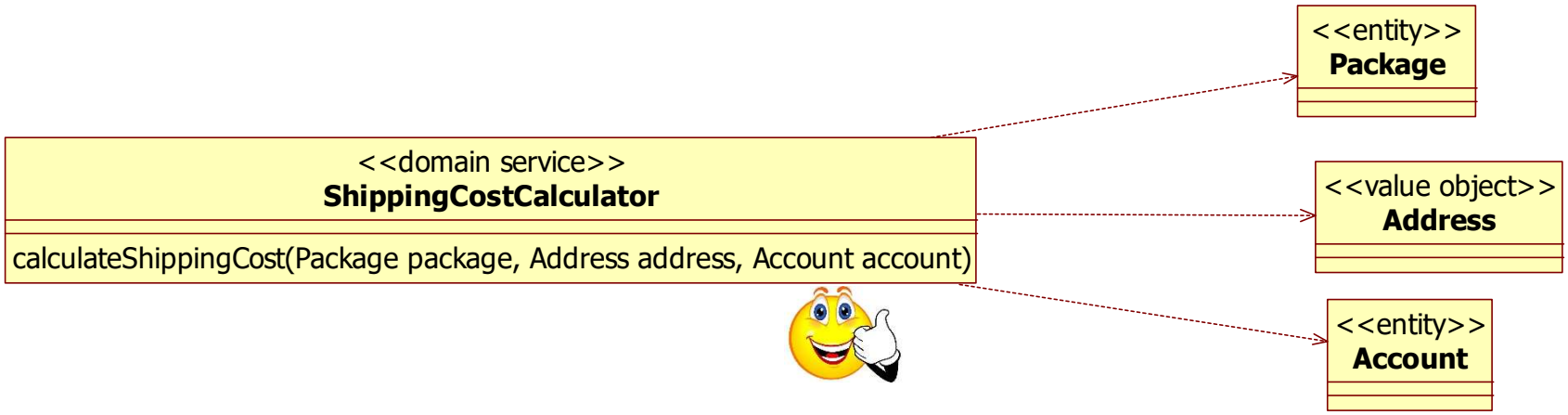


Domain service

OK if the calculation is simple

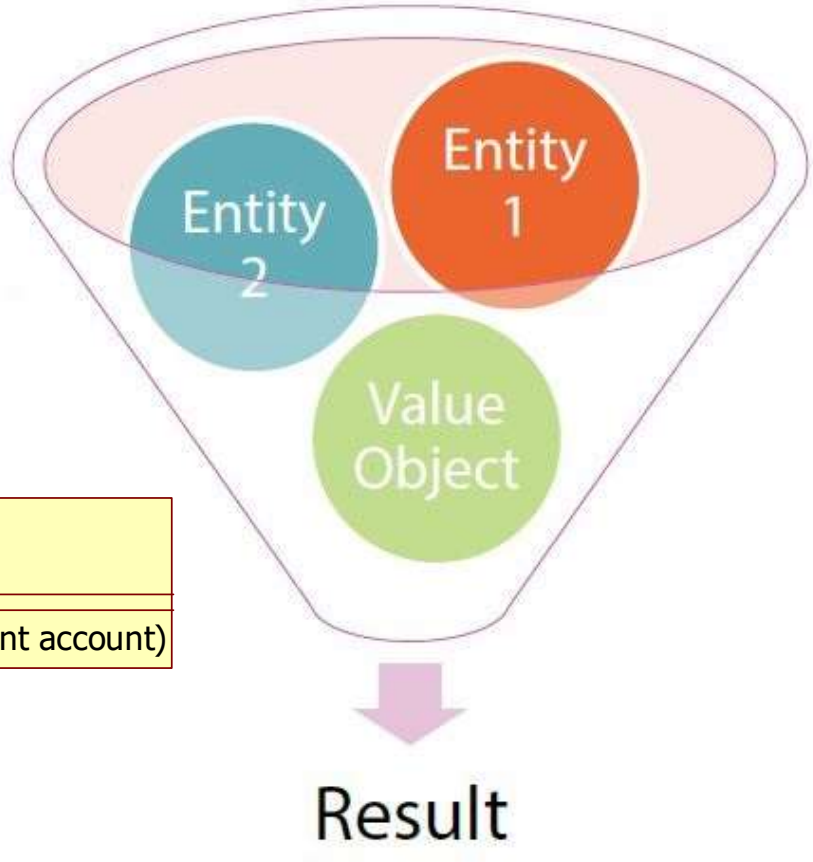


What if there are many business rules related to Address and Account to compute the shipping cost



Domain service

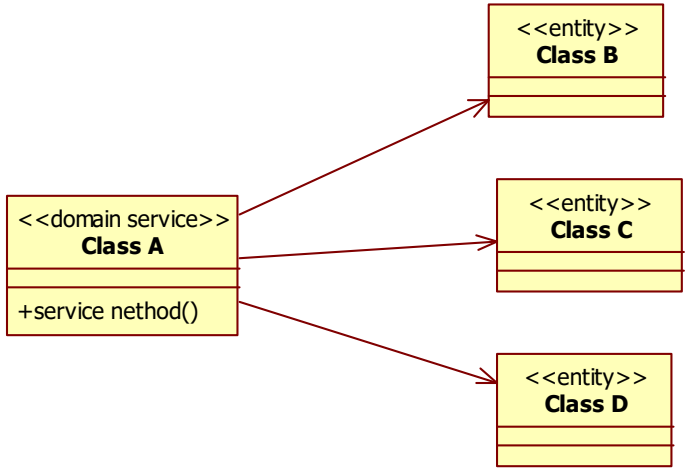
- Interface is defined in terms of other domain objects



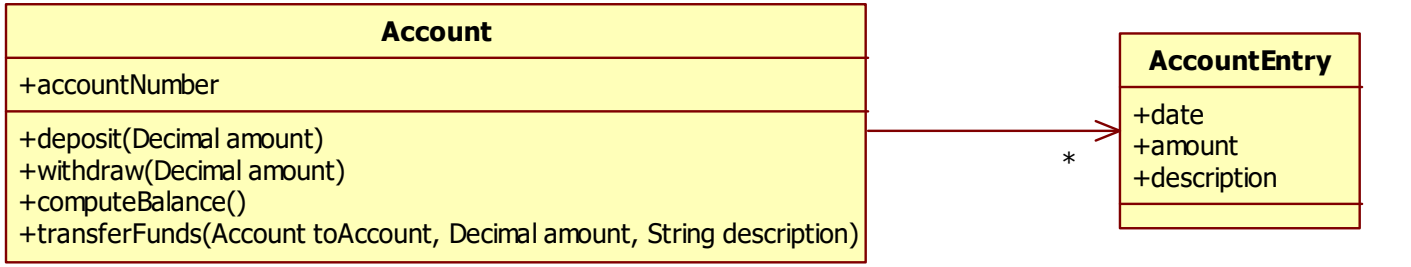
<<domain service>> **ShippingCostCalculator**
calculateShippingCost(Package package, Address address, Account account)

Domain service characteristics

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



Service example

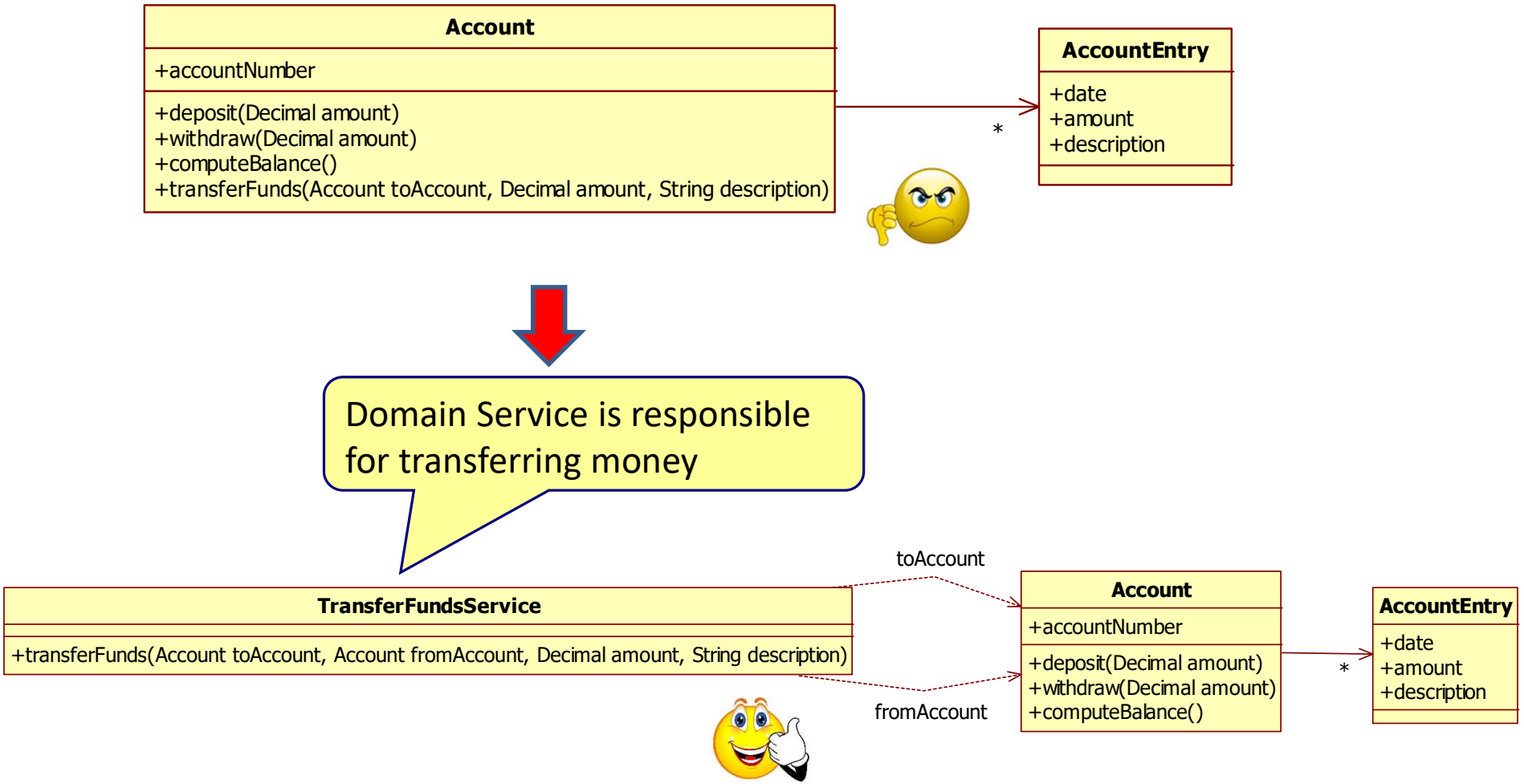


The Account is responsible for transferring money

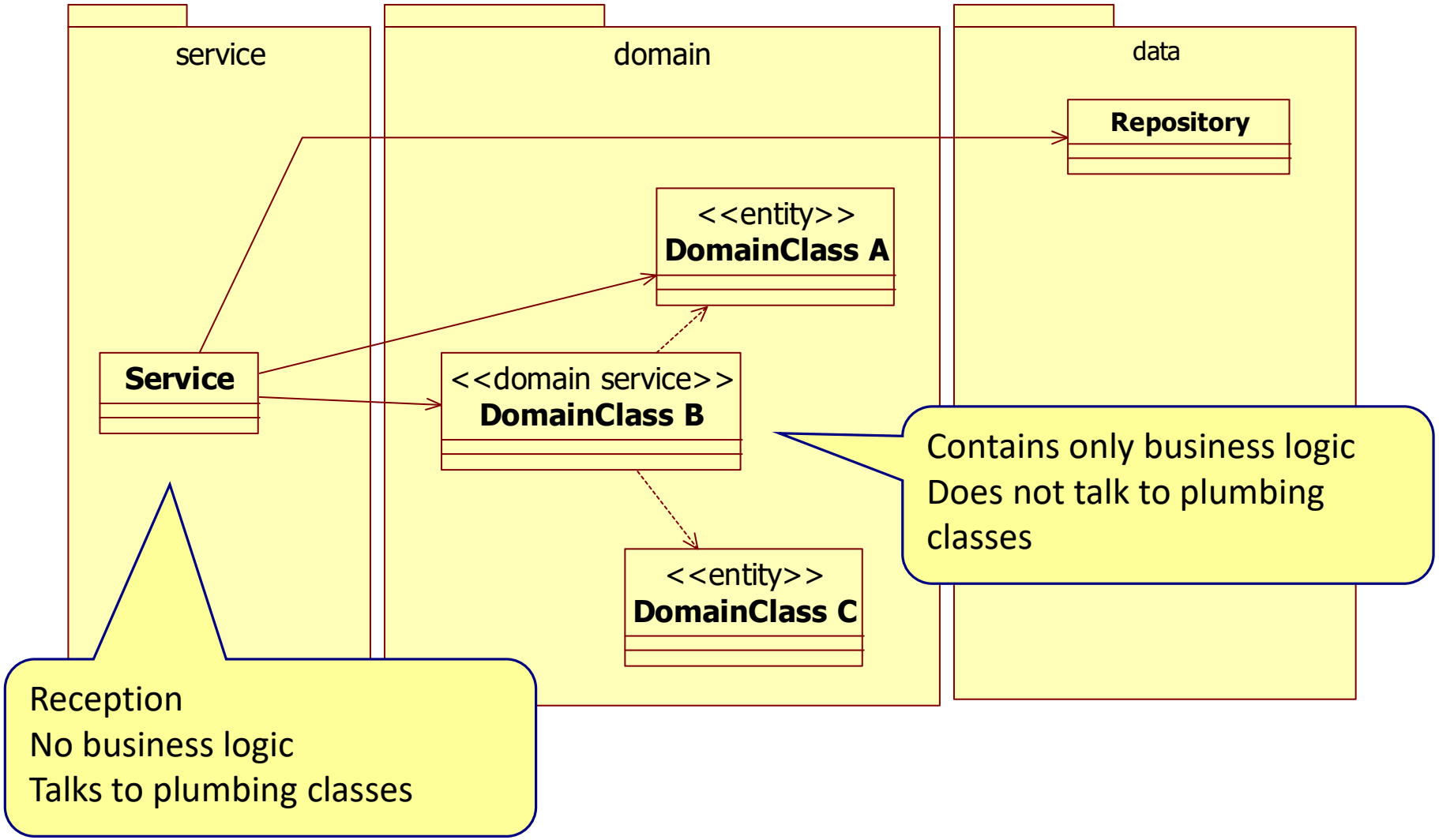
What if there are many business rules for transferFunds() related to different Accounts

- You cannot transferFunds() between these 2 types of accounts
- If the toAccount is this type you forst have to check this
- If the fromAccount is this type, you compute this first
- …

Service example



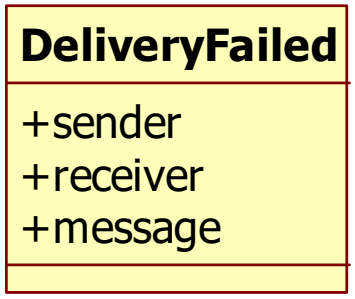
Service class and domain service class

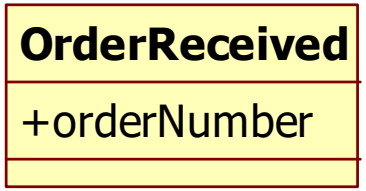


DOMAIN EVENTS

Domain event

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

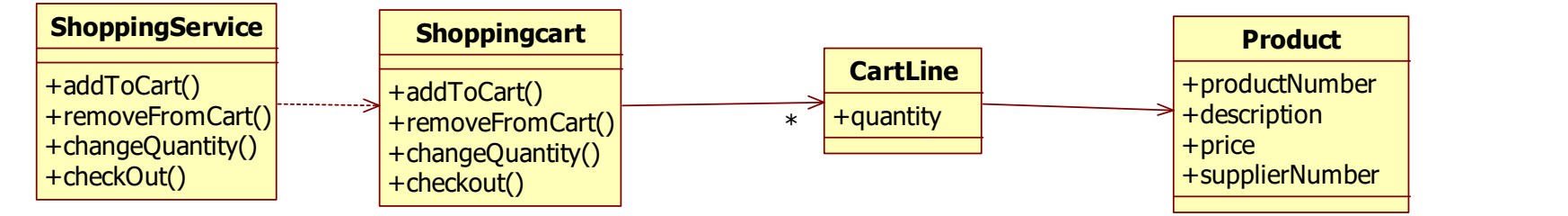


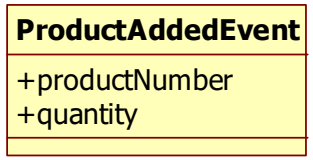


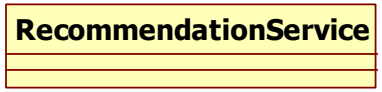
Domain event

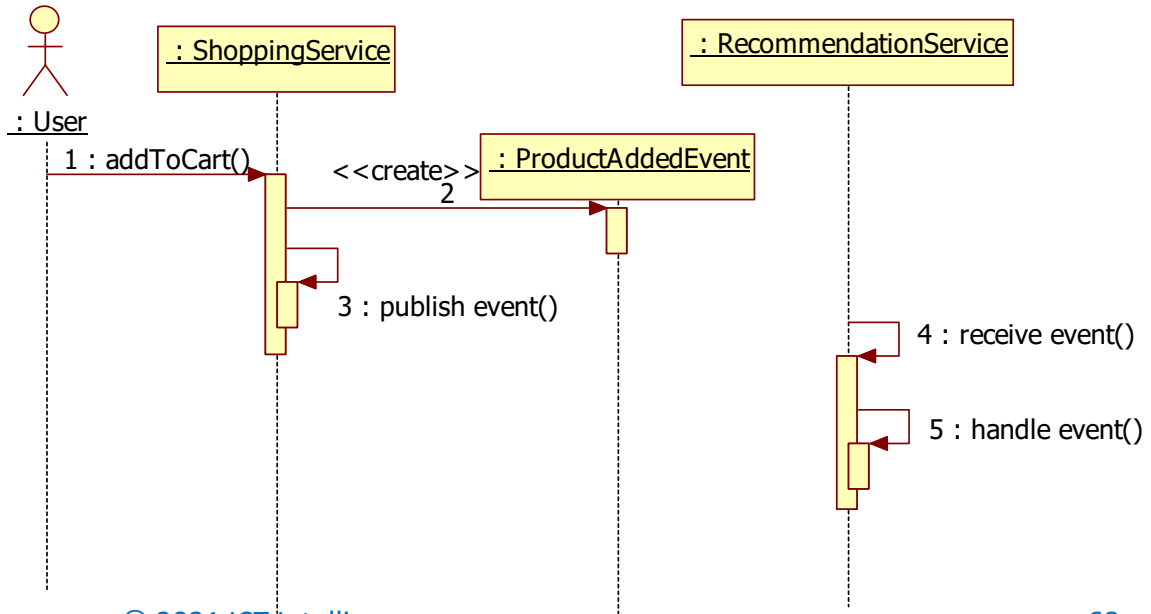
- Events are raised and event handlers handle them.
- Some handlers live in the domain, and some live in the service layer.

Domain event example









SUMMARY

Domain Model Patterns

- Entities
- Value objects
- Domain services
- Domain events

Key principle 4

- The hardest and most important aspect of software development is the domain
  - Create a domain model
  - Knowledge crunching between business and IT
  - Place the domain logic in a separate layer
  - Let the domain logic be a reflection of the real world

Key principle 5

- Orchestration works well if the application is simple and/or the scope is small
- Choreography works well if the application is complex and/or the scope is large

- Orchestration
  - One central brain

- Choreography
  - No central brain

Connecting the parts of knowledge with the wholeness of knowledge

1. The hardest and most important aspect of software development is the domain
2. A rich domain model contains all domain knowledge.

3. **Transcendental consciousness** is the source of all knowledge.
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.