The POS case

Programming 2.2

KdG
University of Applied Sciences and Arts

Agenda

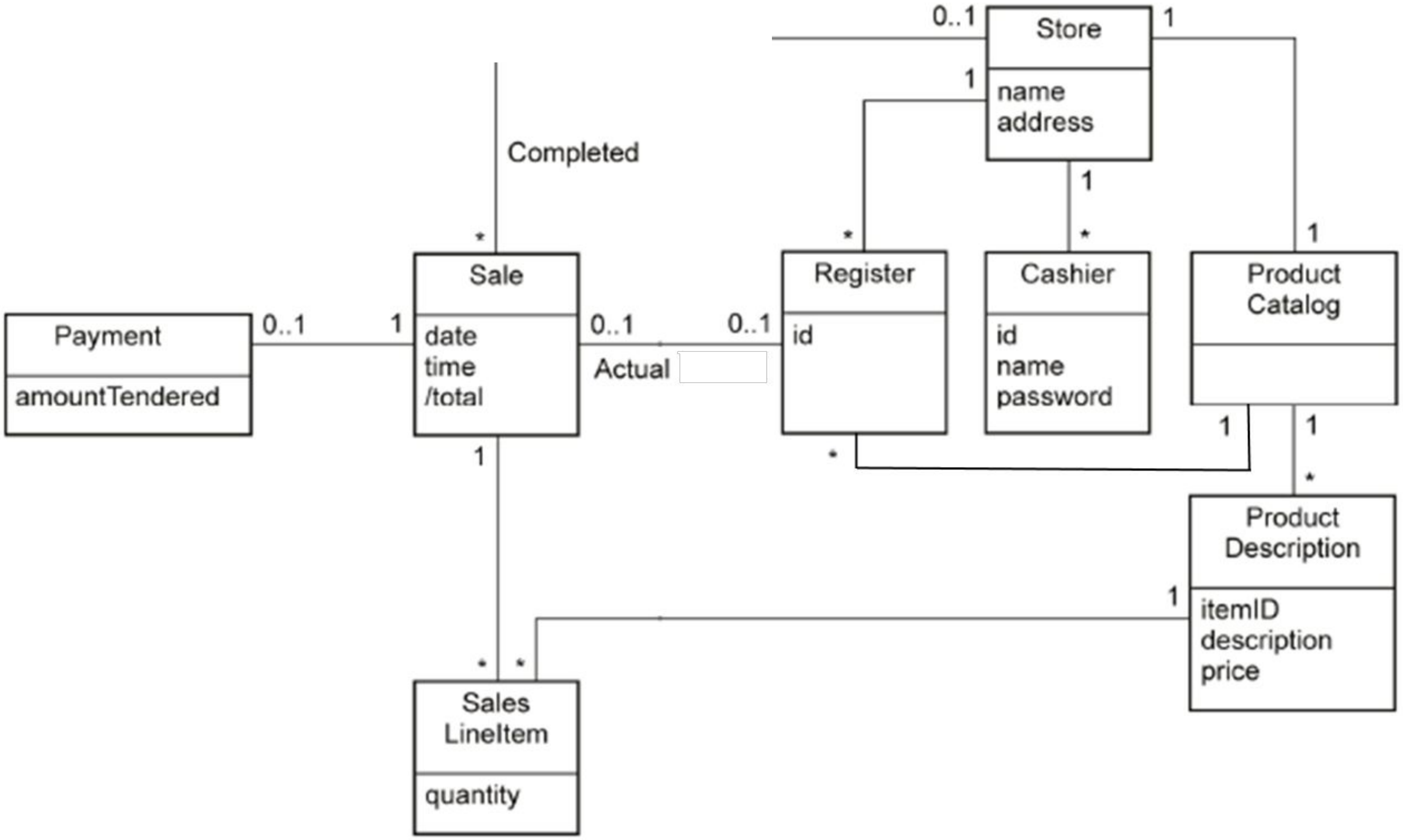
1. Input
2. Design
3. Initialisation
4. DDD: service pattern

Karel de Grote Hogeschool

Input

University of Applied Sciences and Arts

input: Domain model



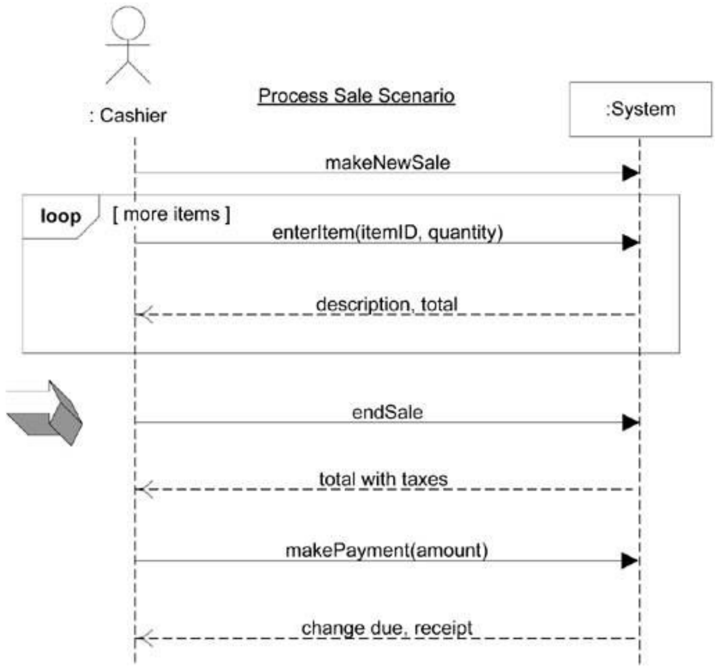
input: user stories / acceptance criteria / SSD

As a cashier

I want to enter products

So that the amount due can be correctly calculated

```
1.    Scenario: new sale
   Given there is no active sale for register 5
   When 2 items of product "Mars" are entered for register 5
   Then there is an active sale with 1 salesline(s) for cashier 5
 2. Scenario: active sale
   Given there is an active sale 1235 containing 2 salesline(s)
      for register 4
   When 5 items of product "Twix" are entered for register 4
   Then sale 1235 contains 3 salesline(s)
   And salesline 3 of sale 1235 contains 5 items
 3. Scenario: close sale
   Given there is an active sale 1235 for register 4
   When the cashier of register 4 closes the sale
   Then sale 1235 is "closed"
   And there is no active sale for register 4
```



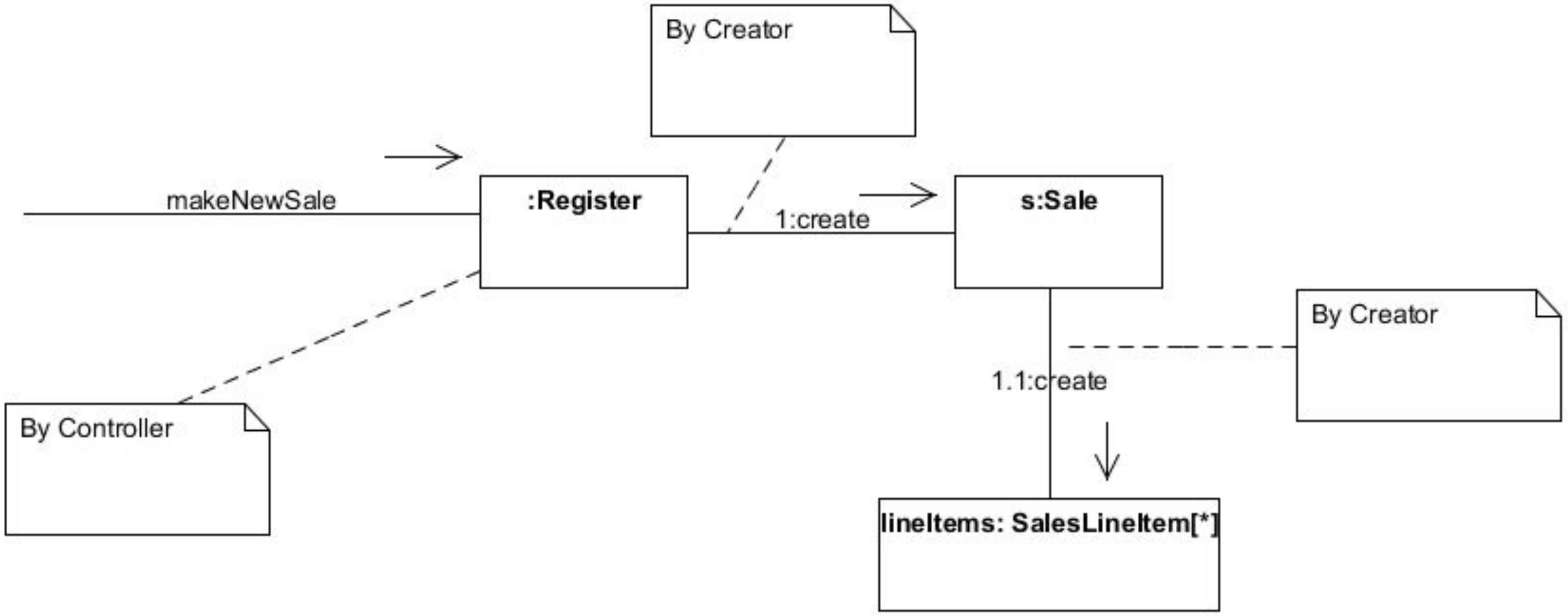
2

Design

University of Applied Sciences and Arts

Operation	**makeNewSale():Sale**
Cross References	**Use Cases: Process Sale**
Preconditions	none
Postconditions	- A Sale instance s was created (instance creation). - s was associated with the Register (association formed). - Attributes of s were initialized.

Operation	**makeNewSale():Sale**
Cross References	**Use Cases: Process Sale**
Preconditions	none
Postconditions	- A Sale instance s was created (instance creation). - s was associated with the Register (association formed). - Attributes of s were initialized.



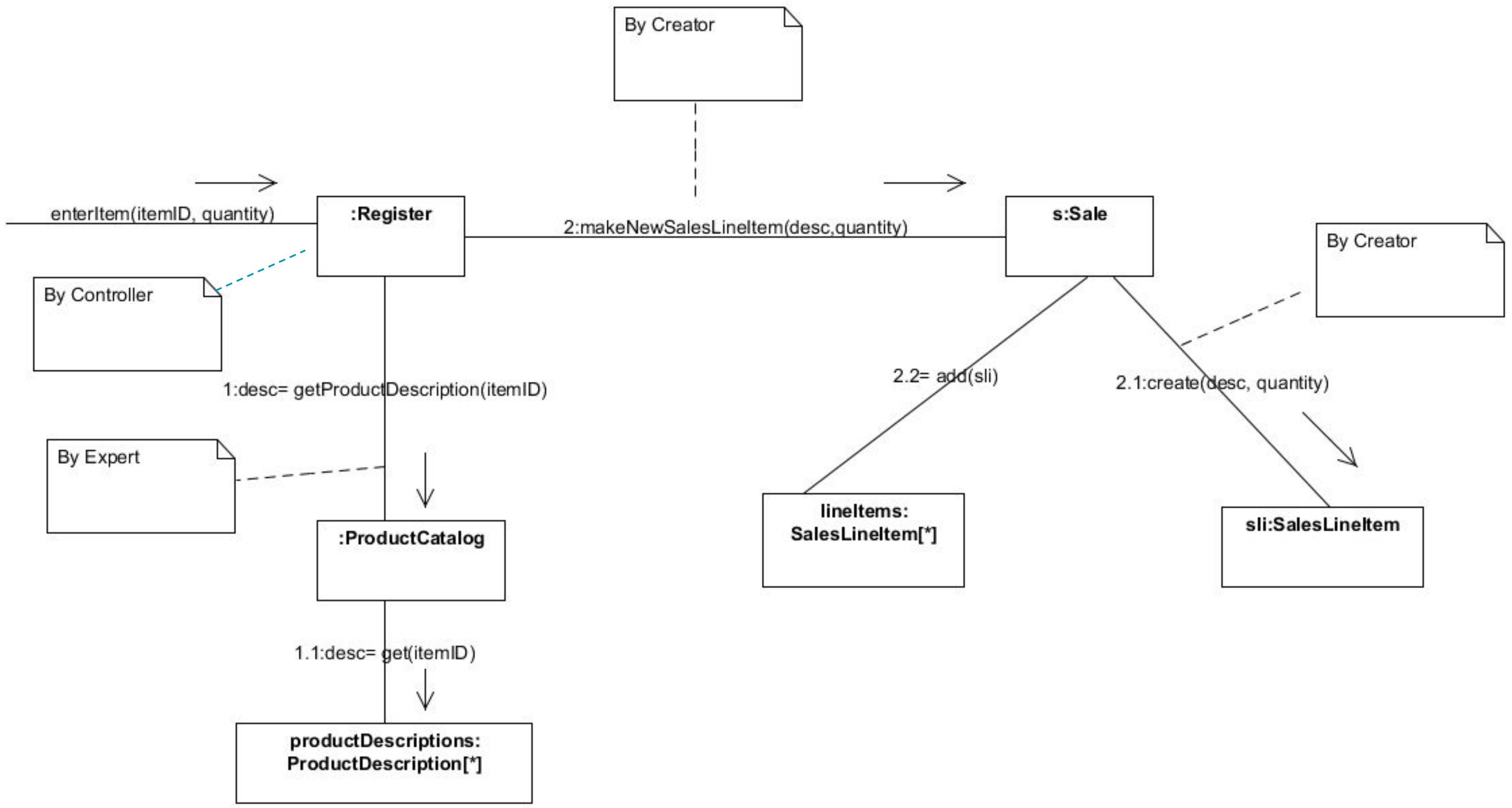
KdG University of Applied Sciences and Arts

Operation	**enterItem(itemID : long, quantity : integer): ProductDescription**
Cross References	**Use Cases: Process Sale**
Preconditions	There is an underway sale.
Postconditions	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). - sli.quantity became quantity (attribute modification). - sli was associated with a ProductDescription, based on itemID match (association formed).

Design questions

- We only have to productId, where do we find the description?
- Who creates a new SalesLine?
- How to associate salesline with ProductDescription

Operation enterItem(itemID : ItemID, quantity : integer)


By Creator
enterItem(itemID, quantity)
:Register
2:makeNewSalesLineItem(desc,quantity)
s:Sale
By Creator
By Controller
1:desc= getProductDescription(itemID)
2.2= add(sli)
2.1:create(desc, quantity)
By Expert
:ProductCatalog
lineItems: SalesLineItem[*]
sli:SalesLineItem
1.1:desc= get(itemID)
productDescriptions: ProductDescription[*]

KdG University of Applied Sciences and Arts

Operation	**endSale():float**
Cross References	**Use Cases: Process Sale**
Preconditions	There is an underway sale.
Postconditions	- Sale.isComplete became true

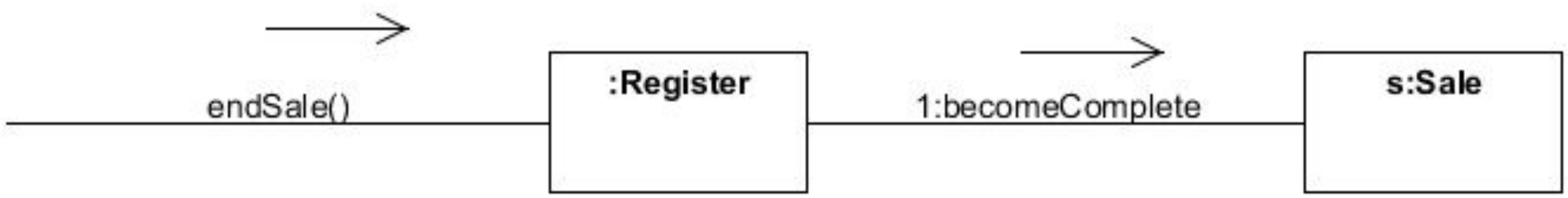
We're making two diagrams

set sale completed

Get and return total

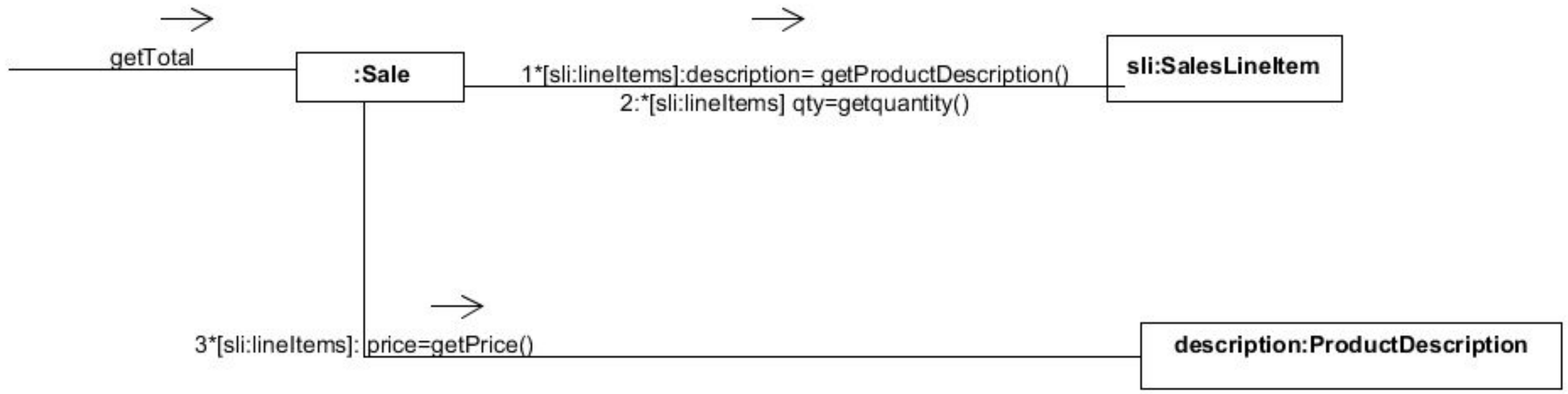
Operation	**endSale():float**
Cross References	**Use Cases: Process Sale**
Preconditions	There is an underway sale.
Postconditions	- Sale.isComplete became true

Set sale completed



KdG University of Applied Sciences and Arts

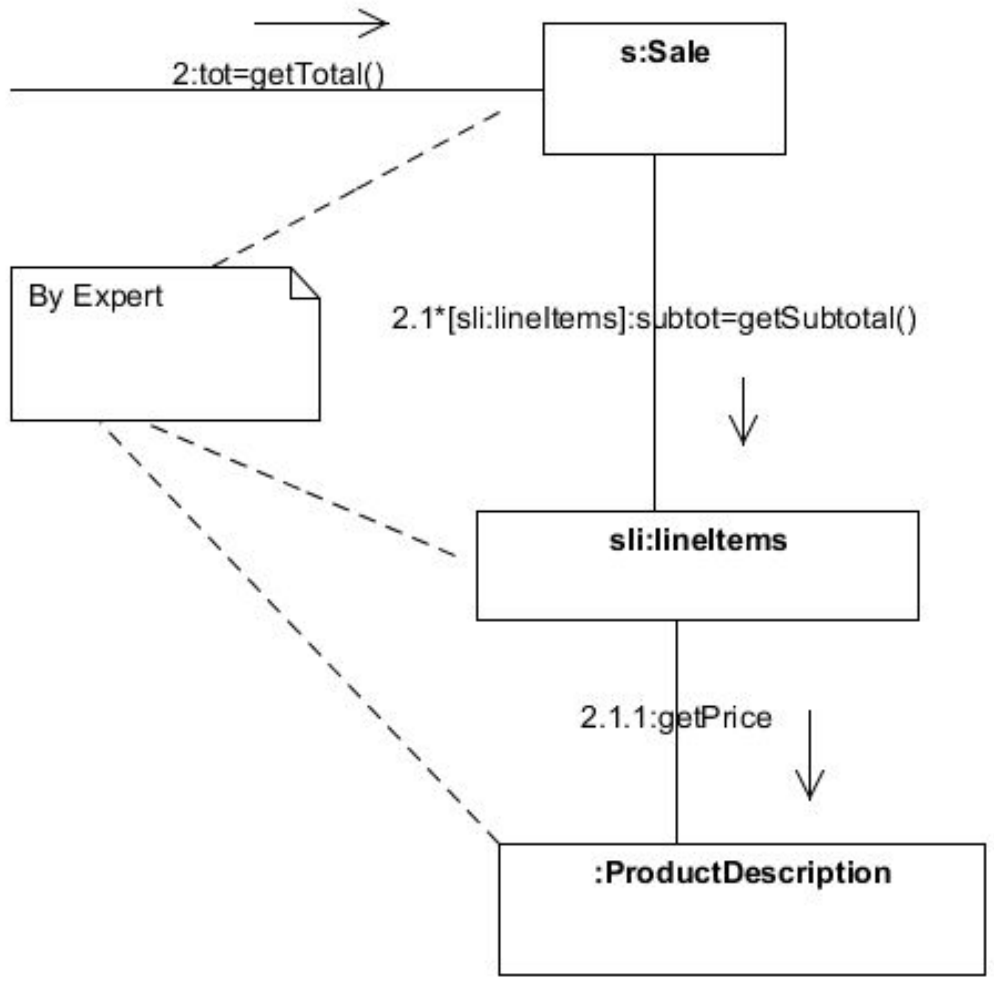
endSale: getTotal


getTotal
:Sale
1*[sli:lineItems]:description= getProductDescription()
2:*[sli:lineItems] qty=getquantity()
sli:SalesLineItem
3*[sli:lineItems]: price=getPrice()
description:ProductDescription

Pattern: Information Expert

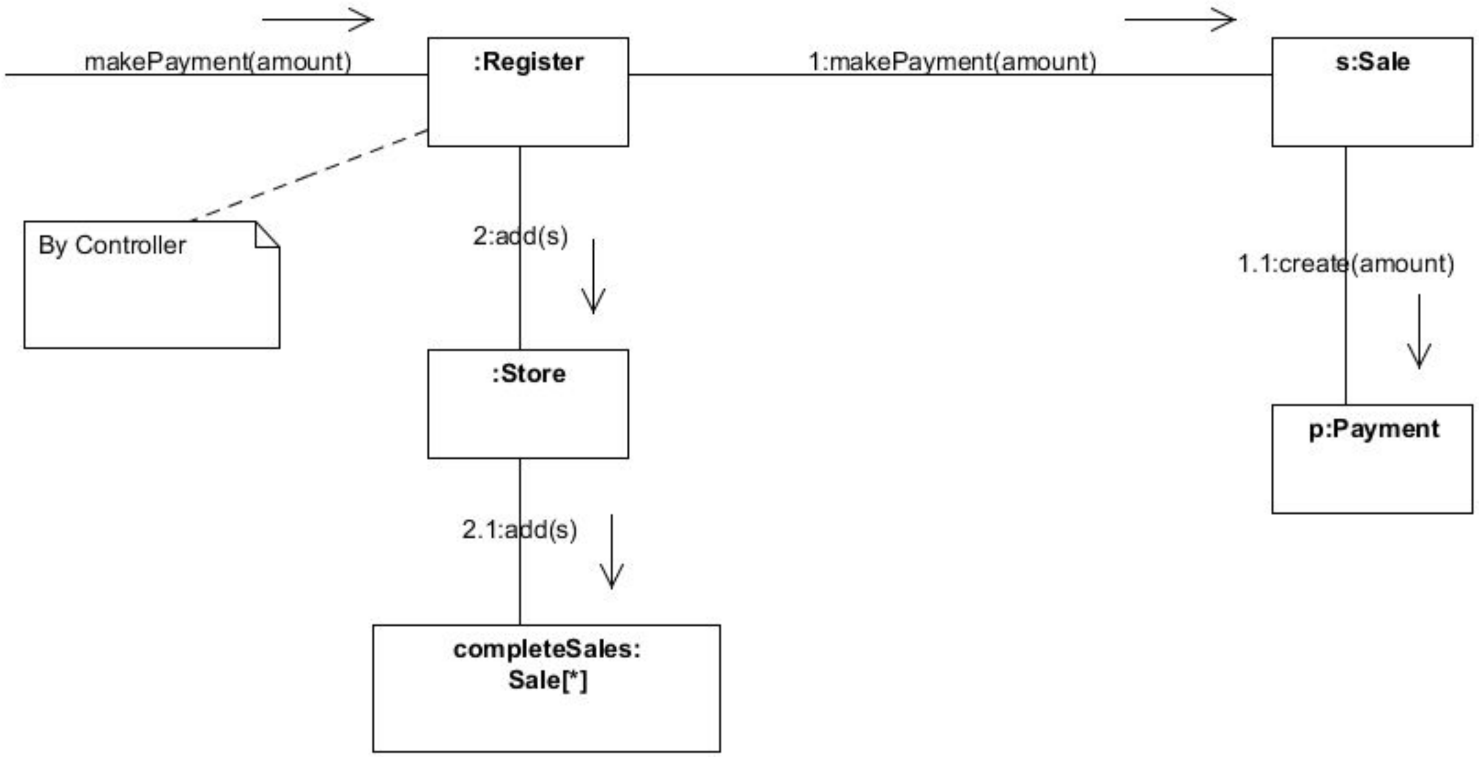
- Problem
  - What is the principle for assigning responsibilities?
- Solution
  - Assign the responsibilities to the class that has the information to fulfil the responsibility
- Related patterns
  - Don't talk to strangers: if a class has no reference to the information expert, the information expert is a stranger. Do not retrieve the information expert, but ask the class with the reference to do the job. It will delegate the job to the information expert
  - Delegation

Get and return total



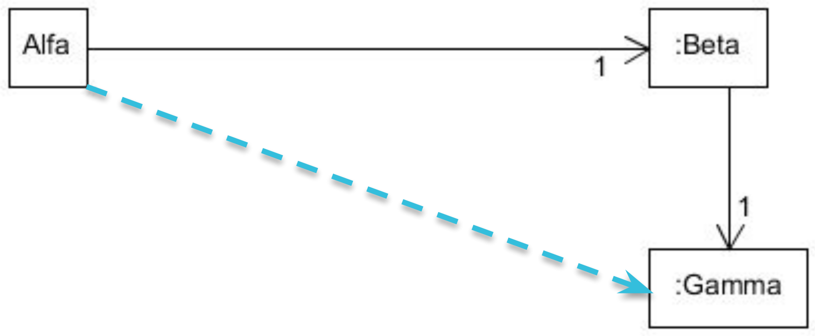
Operation	**makePayment(amount: Money)**
Cross References	**Use Cases: Process Sale**
Preconditions	There is a completed, unpaid sale.
Postconditions	- A Payment instance p was created (instance creation). - p was associated with the current Sale (association formed). - The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales).

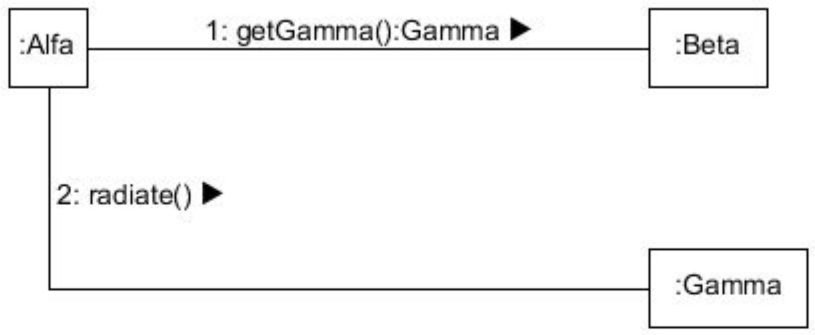
Operation	**makePayment(amount: Money)**
Cross References	**Use Cases: Process Sale**
Preconditions	There is a completed, unpaid sale.
Postconditions	- A Payment instance p was created (instance creation). - p was associated with the current Sale (association formed). - The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales).



Low Coupling

- A relation in a class diagram implies a dependency
- If you have a link in a communication diagram and there is no corresponding link in the class diagram, you can indicate this in the class diagram using a dependency arrow (-->)
  - This example violates *don't talk to strangers*



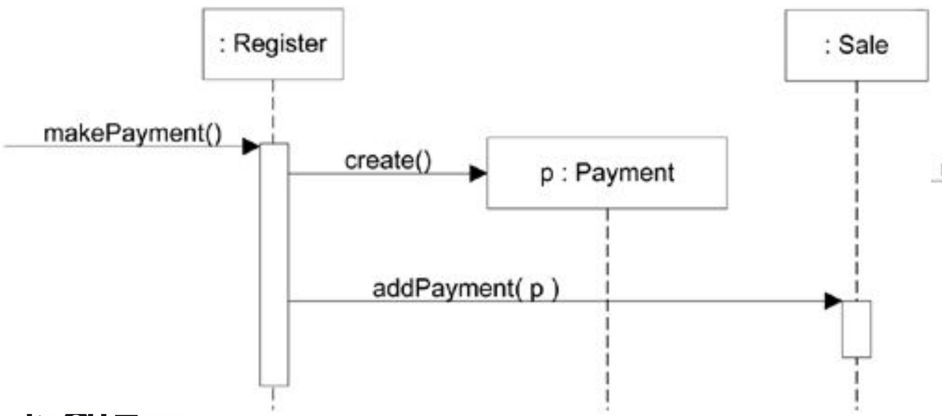


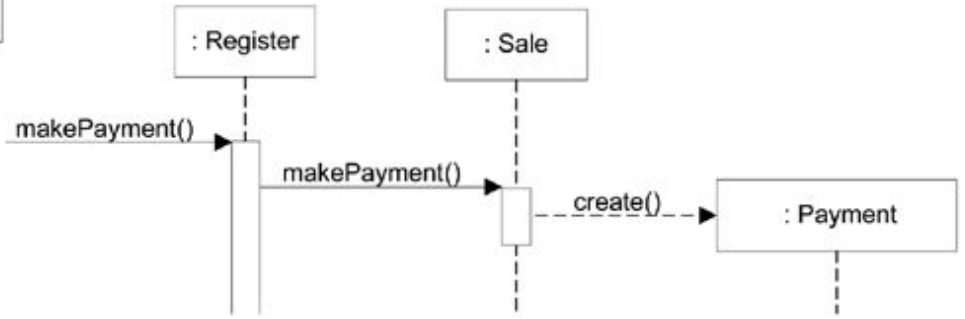
Low Coupling

- Problem: How to keep impact of changes local (and reusability high)
- Solution: Assign responsibilities so that coupling is low.

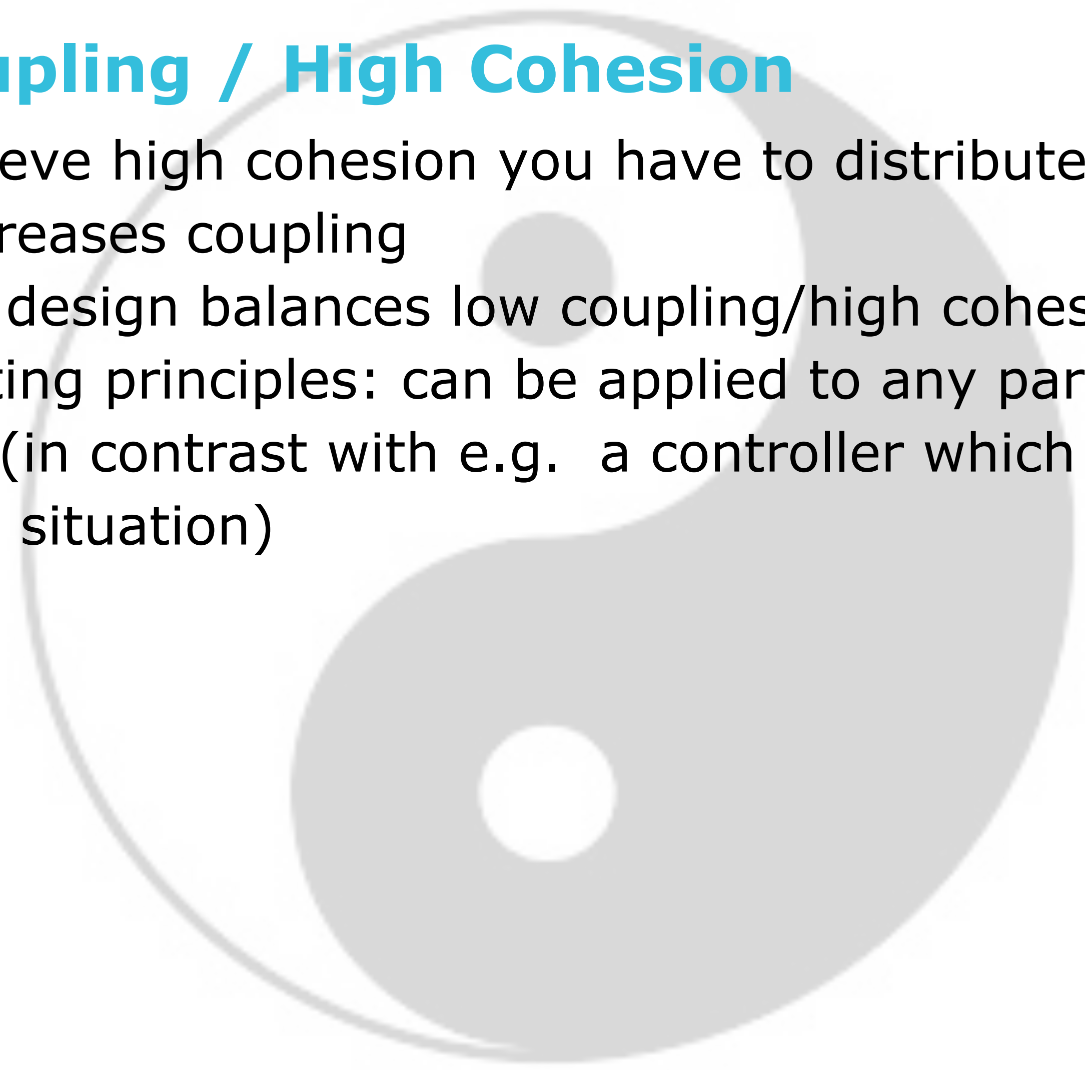
High Cohesion

- Problem: How to keep classes focused, clear and maintainable?
- Solution: Determine a clear and limited responsibility for the class. All methods in the class should collaborate towards this responsibility
- Related: Single Responsibility Principle
- Antipattern (opposite): God class, DDD: Big ball of mud





Low Coupling / High Cohesion



- To achieve high cohesion you have to distribute tasks => can increases coupling
- a good design balances low coupling/high cohesion
- Evaluating principles: can be applied to any part of the design (in contrast with e.g. a controller which handles a specific situation)

KdG
University of Applied Sciences and Arts

Initialisation

University of Applied Sciences and Arts

System initialisation

- *At the end of design*
  When designing interaction diagrams carefully note which object and relations you use. Create them in initialisation

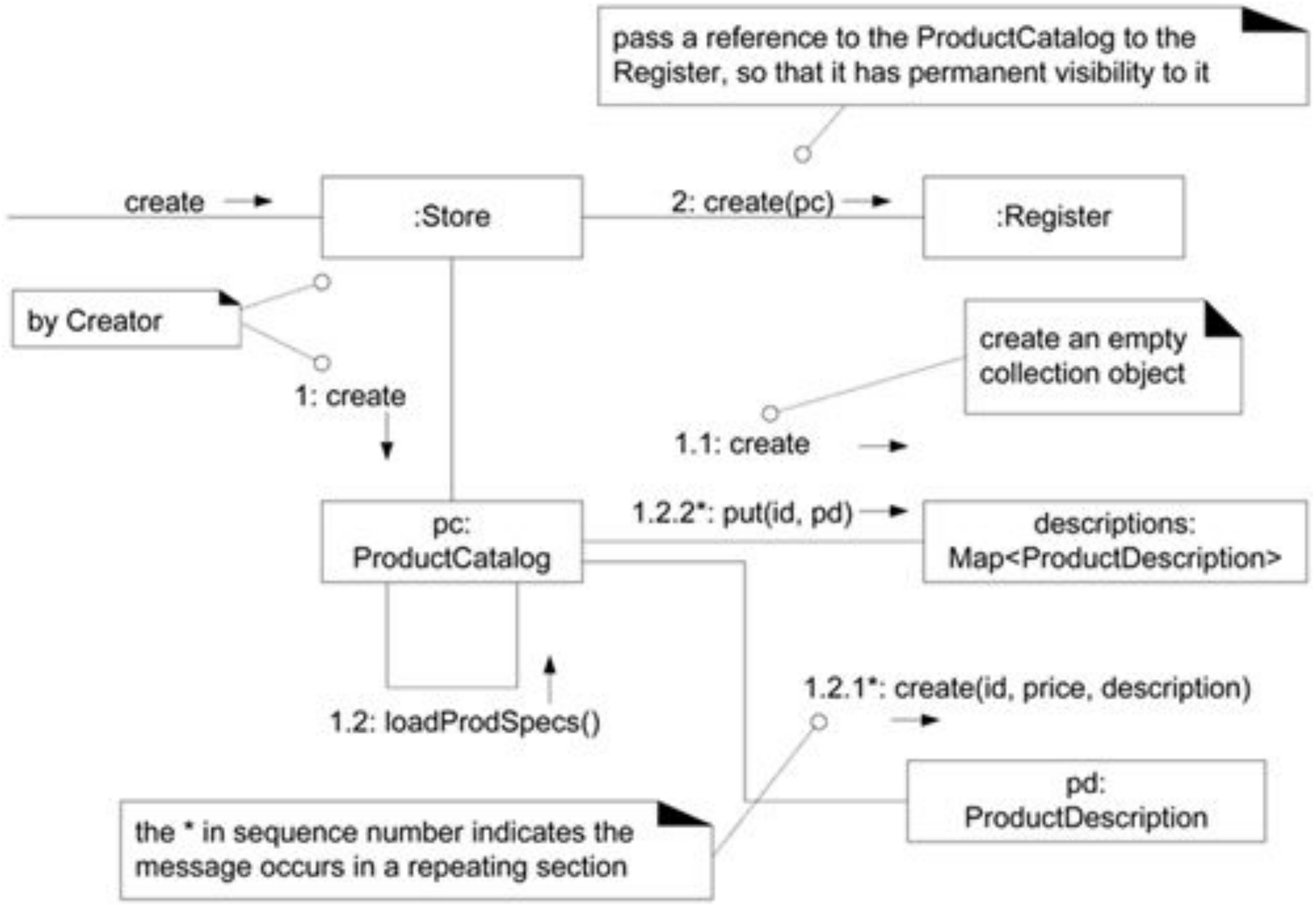
- Initialise coordinating objects:
  - controller
  - Repositories (custom classes that manage collections e.g. ProductCatalog)

System initialisation

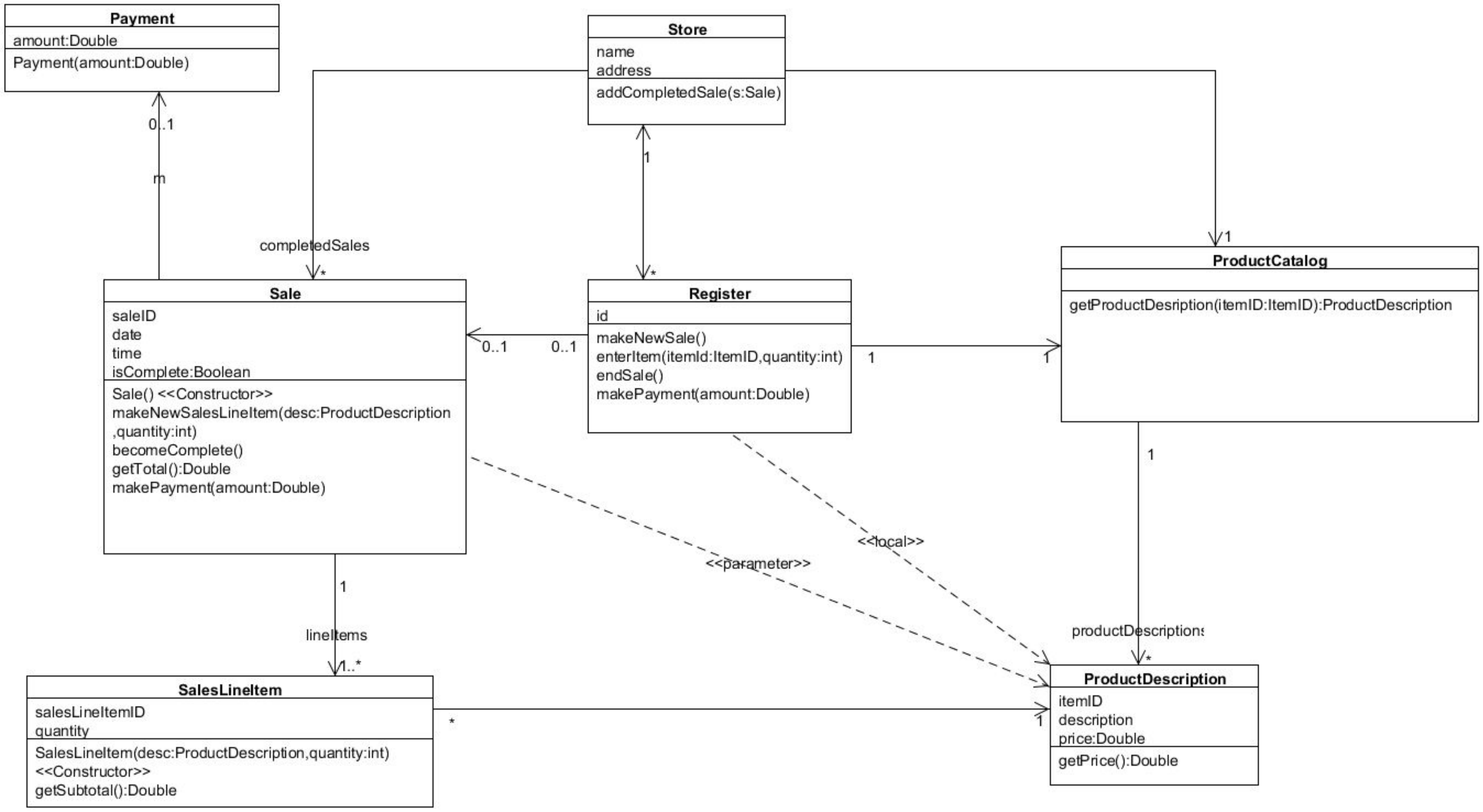
```
public class Main{
    public static void main( String[] args ){
    // Store is the initial domain object.
    // The Store creates some other domain objects.
   Store store = new Store();
  SaleJFrame frame = new SaleJFrame(store.getRegister());
    ...
    }
}
```

University of Applied Sciences and Arts

System initialisation


pass a reference to the ProductCatalog to the Register, so that it has permanent visibility to it
create →
:Store
2: create(pc) →
:Register
by Creator
1: create ↓
create an empty collection object
1.1: create →
pc: ProductCatalog
1.2.2*: put(id, pd) →
descriptions: Map<ProductDescription>
1.2: loadProdSpecs() ↑
1.2.1*: create(id, price, description) →
pd: ProductDescription
the * in sequence number indicates the message occurs in a repeating section

Partial class diagram after elaboration use case Process Sale



DDD: Service pattern

4

University of Applied Sciences and Arts

DDD: Service Pattern

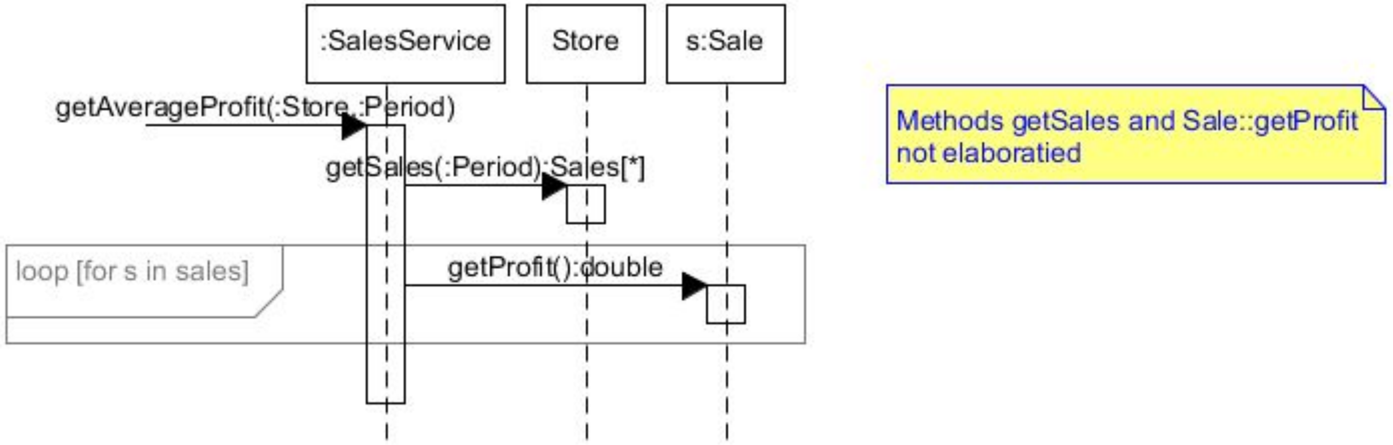
What if there is no good candidate for a function?

Examples:

- Multiple entities/aggregates are involved in a function, but none of the classes can take responsibility for changes in the other.
- There is a transaction on a collection of objects (possibly of the same class) but its logic is beyond common repository features (sorting, searching…)
- You do not want to put business logic in the controller (keep the responsibility limited to interaction with the outside world): delegate to a service

DDD: Service Pattern

Service Example:


:SalesService
Store
s:Sale
getAverageProfit(:Store,:Period)
getSales(:Period):Sales[*]
loop [for s in sales]
getProfit():double
Methods getSales and Sale::getProfit not elaboratied

DDD: Service Pattern

- AKA: Manager
- Service: a new class containing the business logic for orchestrating the collaboration between the objects involved.
  - Reduces coupling between collaborating classes
  - Is a behavioural class (contains actions) not a structural class (thing)
    - The interface (methods) are important
  - Is part of the domain and the ubiquitous language.
  - The service is rather an **action** (dynamic) than a thing (structure). The service is part of the domain and the ubiquitous language.
  - Stateless: does not keep intermediate data between service method calls in attributes (but can have associations established at initialisation time)
  - Naming guidelines
    - Imaginary actor (e.g. Authenticator)
    - Main *task* + Manager/Service (e.g. *Authentication*Service)
    - Main *concept* on which actions take place + Manager/Service (e.g. *SaleService*)

DDD: Service Pattern

- Tension with information expert. If you find a class that has most of the information, do not add a service, but apply information expert.
- Combining behaviour and data is an important OO characteristic
  - Pattern: rich domain model
  - Anti-pattern: anaemic domain model

Overview

1. input
2. Design
3. Initialisation
4. DDD: service pattern

Karel de Grote Hogeschool