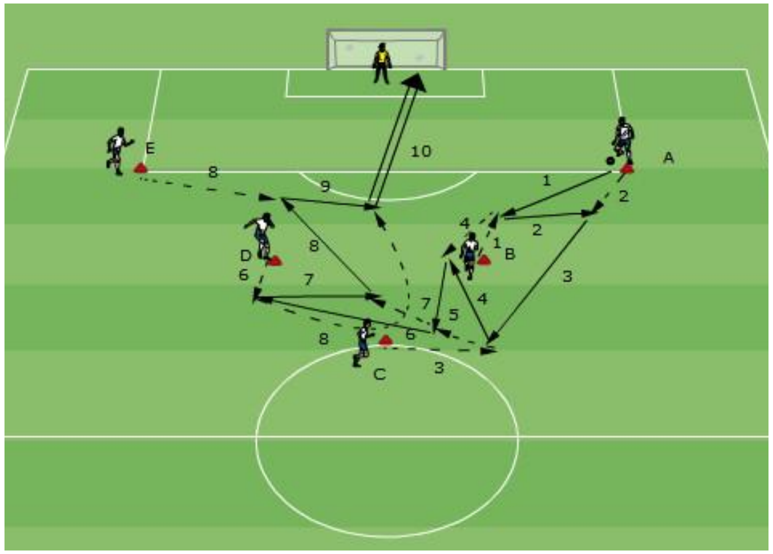
UML Interaction Diagrams

Programming 2.2

KdG
University of Applied Sciences and Arts

Agenda

Karel de Grote Hogeschool



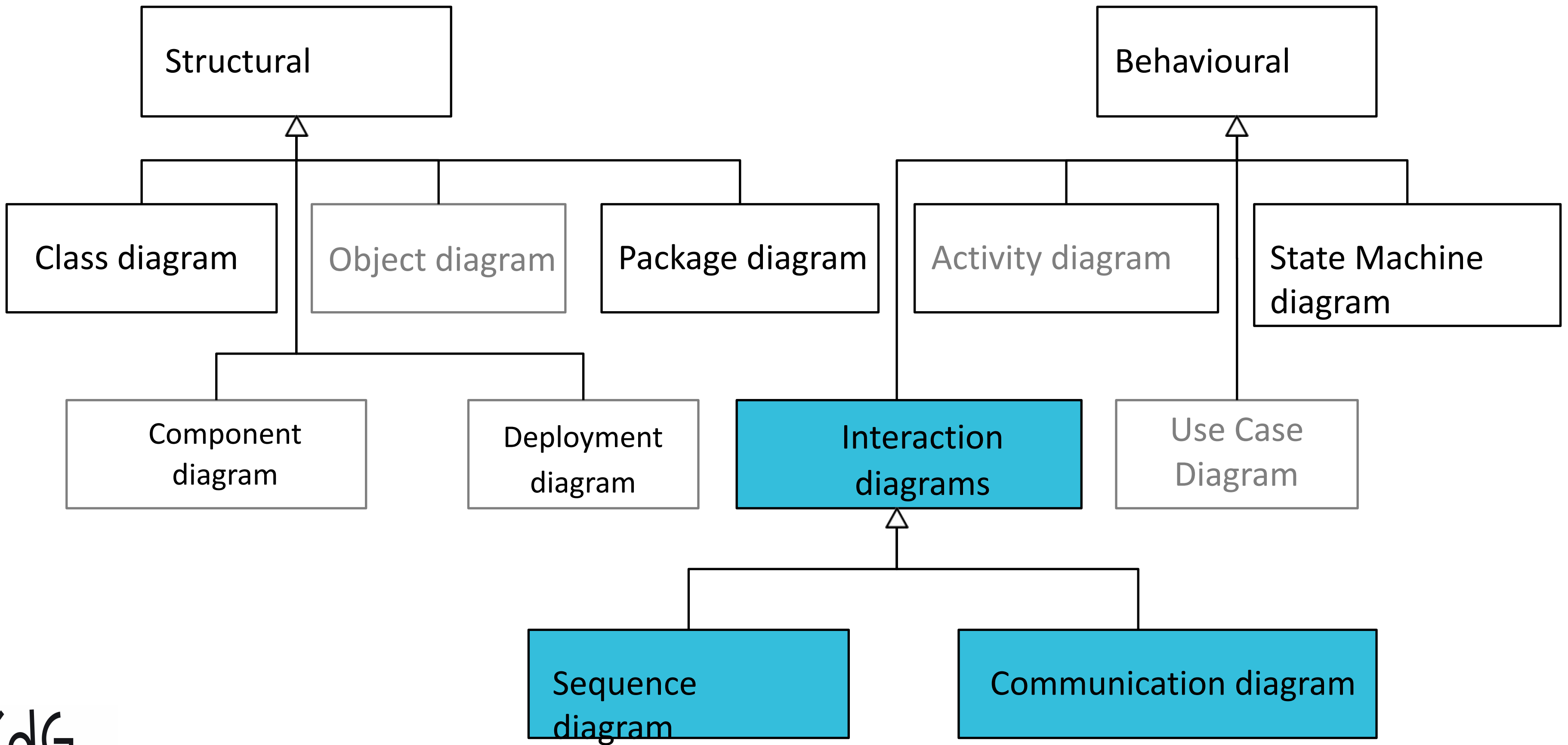
Introduction

Software design

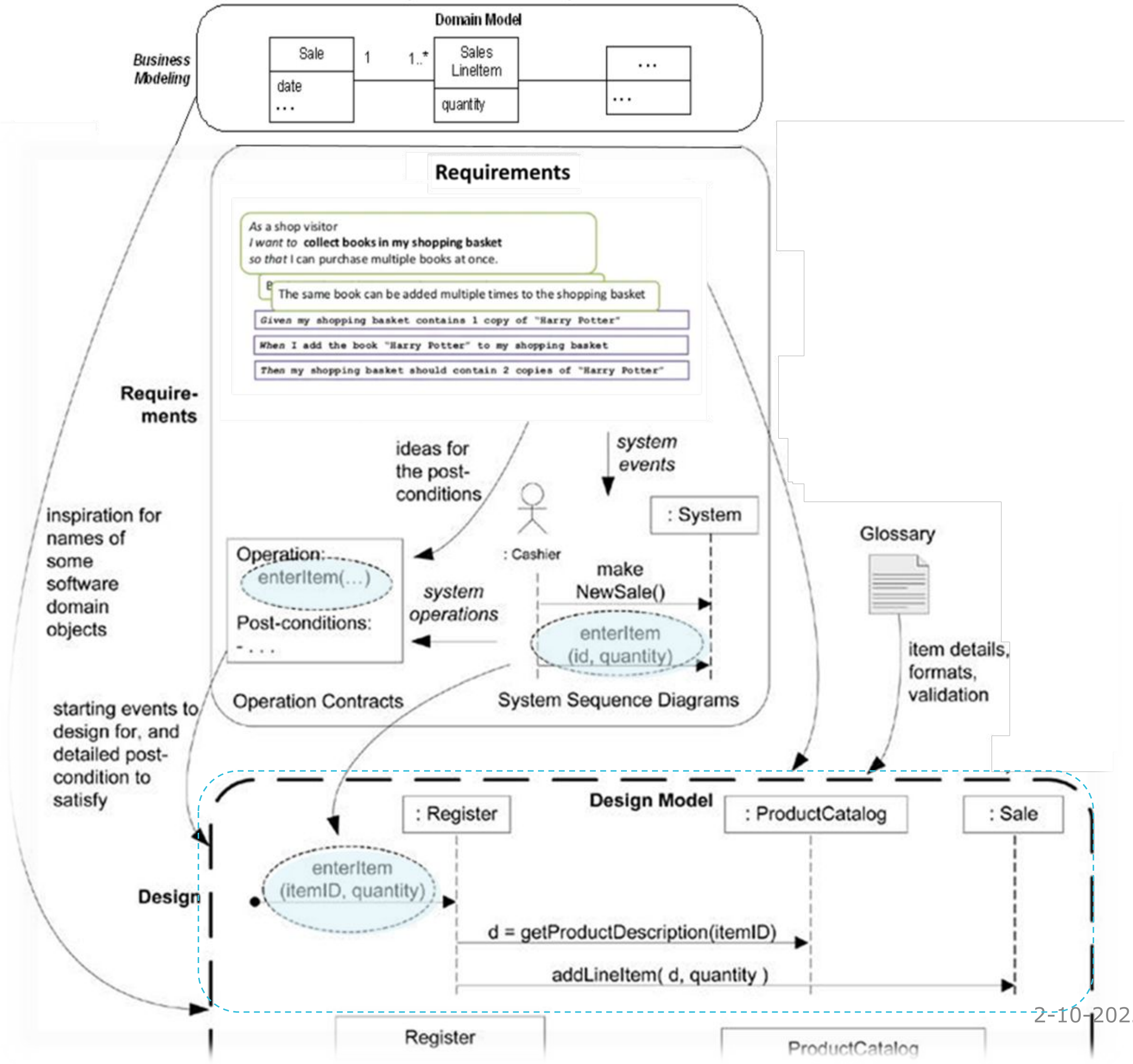
- In analysis we make a class diagram with a conceptual (real world) model of the business domain
  - what do we need to build
- In design we evolve the domain model to specify how we will build the system

KdG University of Applied Sciences and Arts

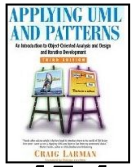
UML diagrams



University of Applied Sciences and Arts


Domain Model
Business Modeling
Sale
date
...
1
1..*
Sales LineItem
quantity
...
...
Requirements
As a shop visitor
I want to collect books in my shopping basket
so that I can purchase multiple books at once.
The same book can be added multiple times to the shopping basket
Given my shopping basket contains 1 copy of "Harry Potter"
When I add the book "Harry Potter" to my shopping basket
Then my shopping basket should contain 2 copies of "Harry Potter"
Require-ments
ideas for the post-conditions
system events
: System
: Cashier
make NewSale()
enterItem (id, quantity)
Operation: enterItem(...)
Post-conditions:
- . . .
system operations
Operation Contracts
System Sequence Diagrams
inspiration for names of some software domain objects
Glossary
item details, formats, validation
starting events to design for, and detailed post-condition to satisfy
Design Model
Design
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
d = getProductDescription(itemID)
addLineItem(d, quantity)
Register
ProductCatalog

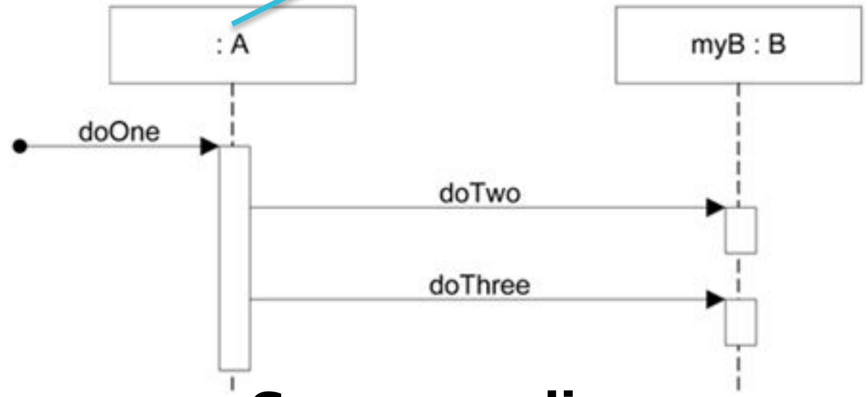
Interaction Diagrams



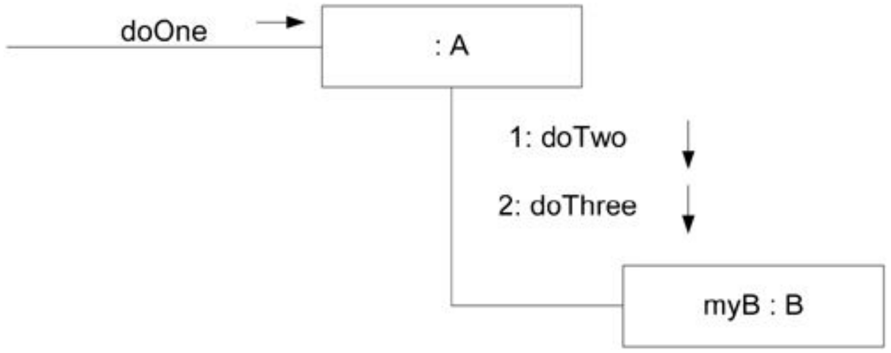
15.1

- Behavioural object diagrams
- Focus on interactions between objects
  - Show messages sent (=methods called) from one object to another
  - Show internals of operations
- 2 variants of interaction diagrams □ contain ~ the same information

:A indicates an instance (object) of **A**



Sequence diagrams



Communication diagram

Exercise Convert to java

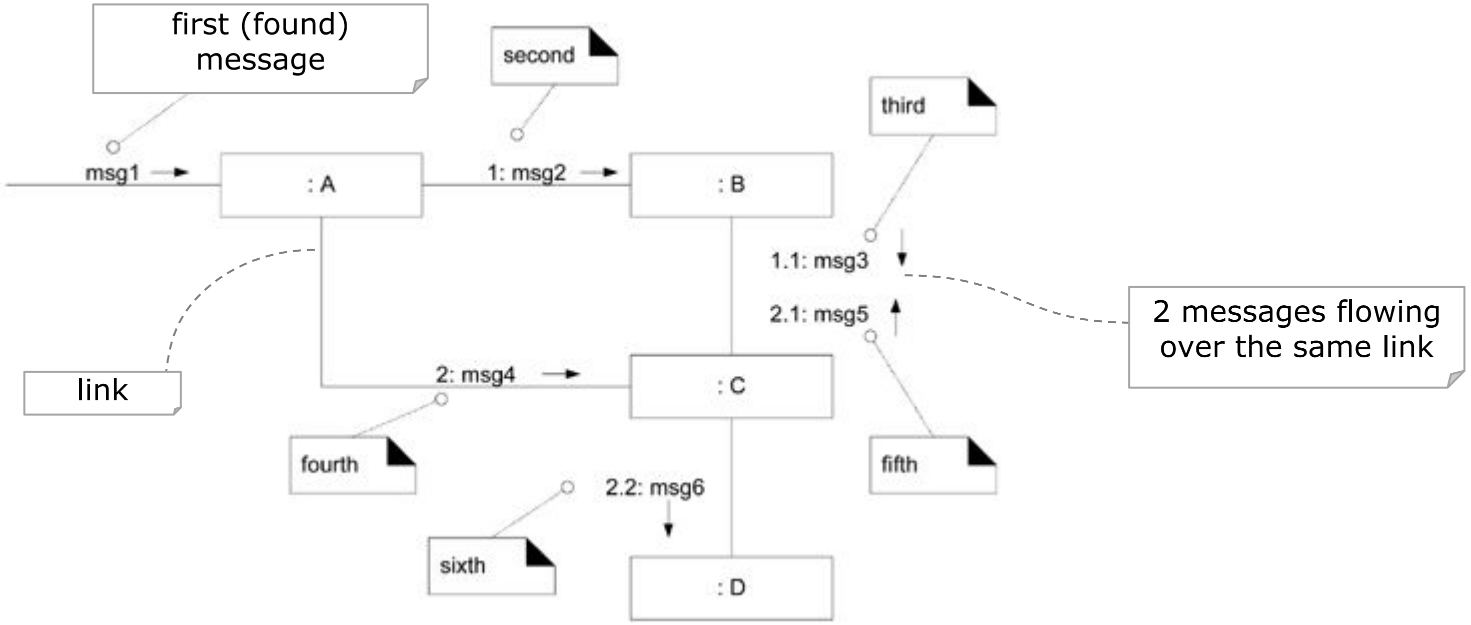
Exercise 1

Communication diagram

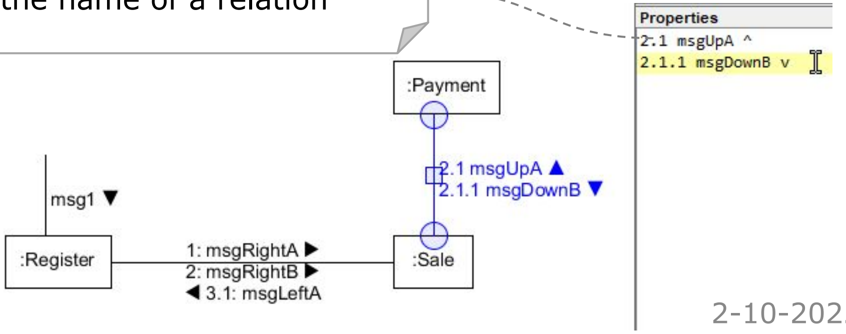
2

University of Applied Sciences and Arts

Messages



In UMLet use the arrow annotation (normally intended to indicate the reading direction of the name of a relation

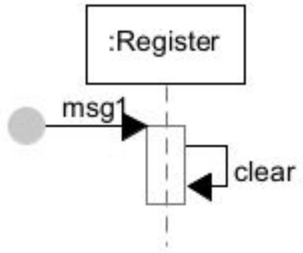


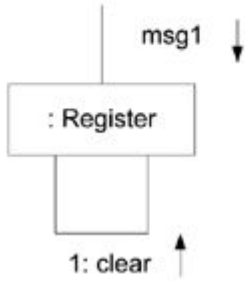
```
2.1 msgUpA ^
2.1.1 msgDownB v
```

Sequence

Communication

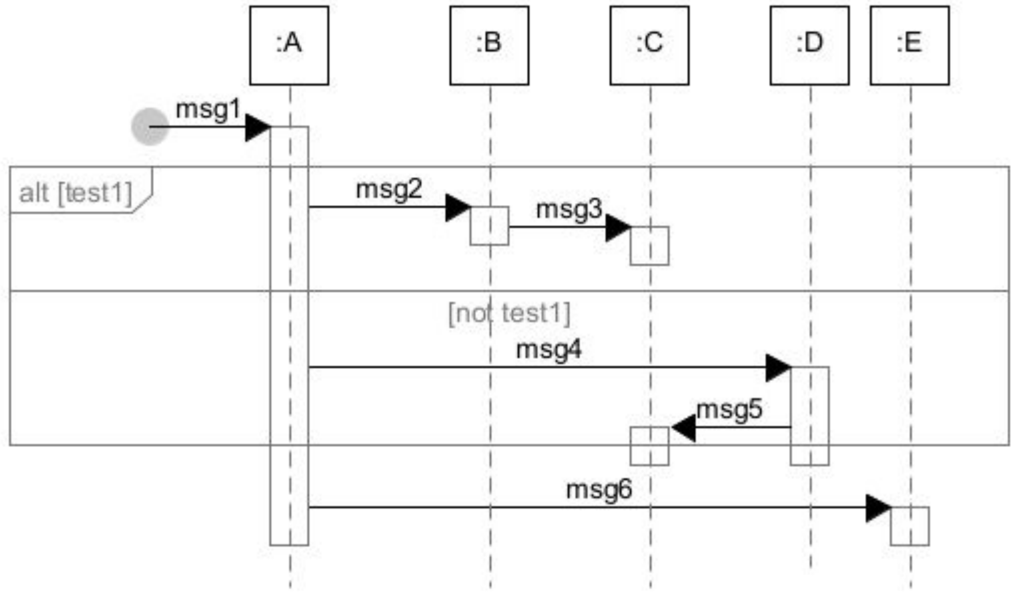
- Reflexive message
  - You only need to draw communication between objects
  - Only use reflexive messages to highlight an internal action



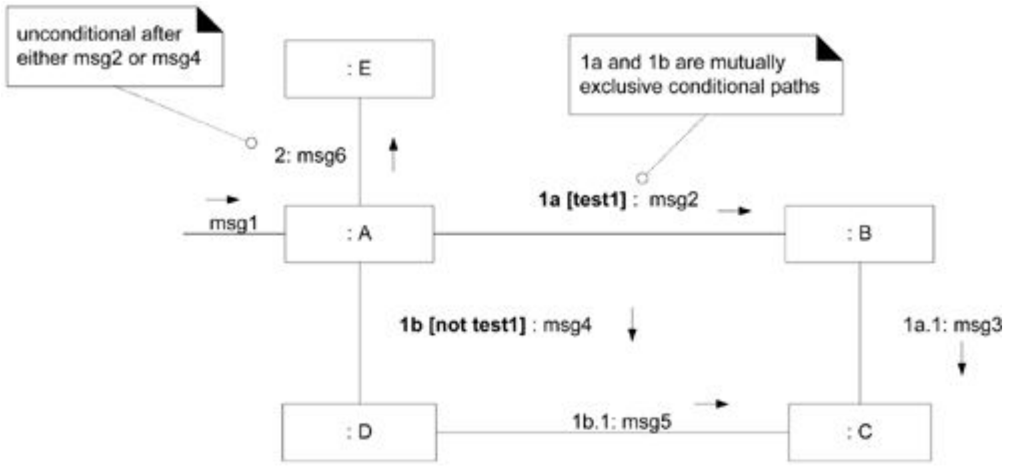


Mutually exclusive

- Sequence



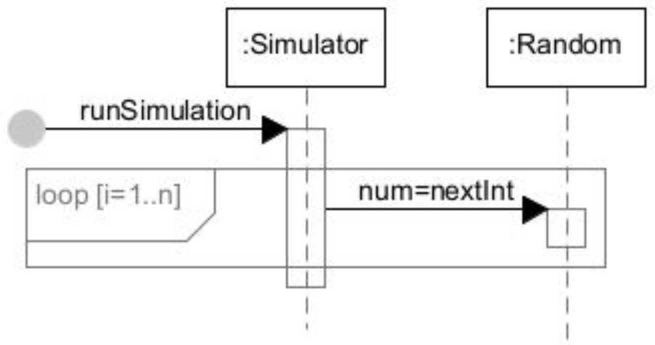
- Communication

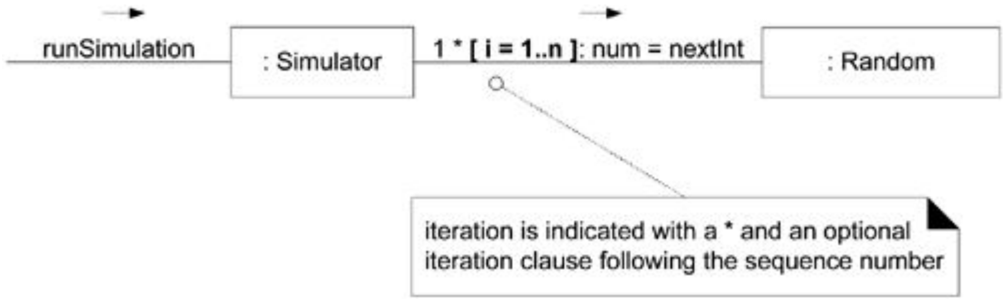

unconditional after either msg2 or msg4
: E
2: msg6
1a and 1b are mutually exclusive conditional paths
1a [test1] : msg2
msg1
: A
: B
1b [not test1] : msg4
1a.1: msg3
: D
1b.1: msg5
: C

Sequence

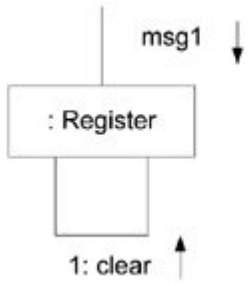
Communication

- Iteration





iteration is indicated with a * and an optional iteration clause following the sequence number



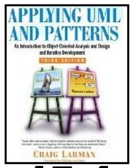
KdG University of Applied Sciences and Arts

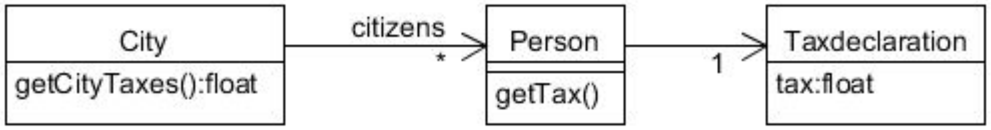
Exercise 2

Revisited

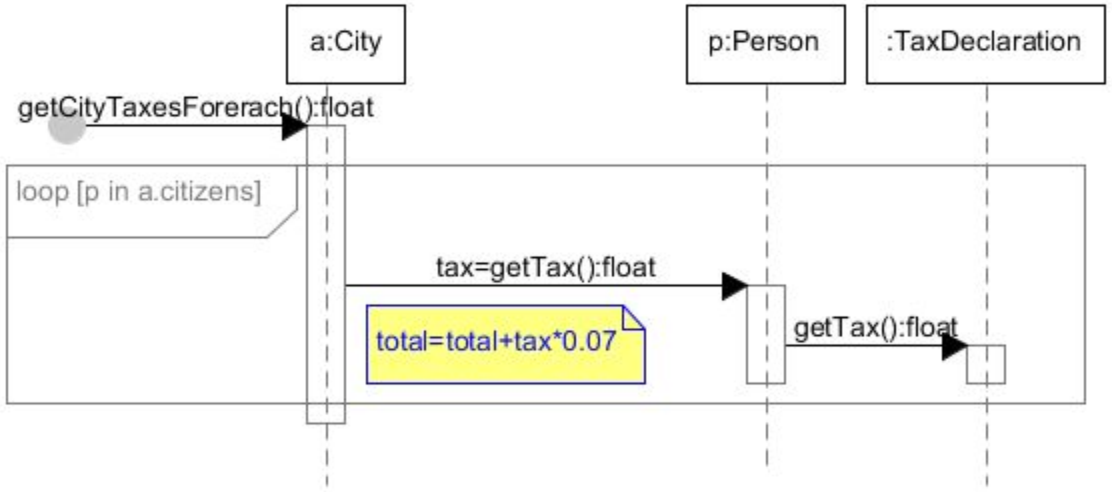
Sequence diagram

Flow control: loop frame foreach


15.4



Same logic, modeled as for each loop



Remark: Larman uses yet another shorthand for looping over a collection (which you may use if you like)

KdG University of Applied Sciences and Arts

Flow control: loop frame foreach code

```
be.kdg.prog22.cityTax
```

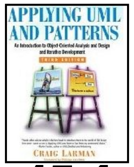
```
public class City {
   private static double TAX_SURCHARGE=0.07;
   private List<Person> citizens= new ArrayList<>();
   //…

  public double getCityTaxesForEach(){
   double total=0.0;
   for (Person p : citizens) {
     total+=p.getTax();
   }
   return total*TAX_SURCHARGE;
  }

}
```

University of Applied
Sciences and Arts

What do you need to model in the sd?

- Guideline: draw method calls to other objects in your code
  - `[p in a.citizens]:` called from `a.getCityTaxesForEach.` This is an internal access in a, no need to draw this
  - `citizens.get(i)`: called from `a.getCityTaxes.` Draw an arrow from a -> citizens.
    `get()` is not written by you (Collections class), no need to model what happens inside `get()`
  - `p.getTax()`: called from `a.getCityTaxes.` Draw an arrow from a -> p.
    `getTax()` is written by you (Person class): model what happens inside (call to TaxDeclaration)
  - `total += total+tax*0,07`: called from `a.getCityTaxes.` This is internal logic using local variables, no need to model this. You may add a note to how this works, but that is entirely optional

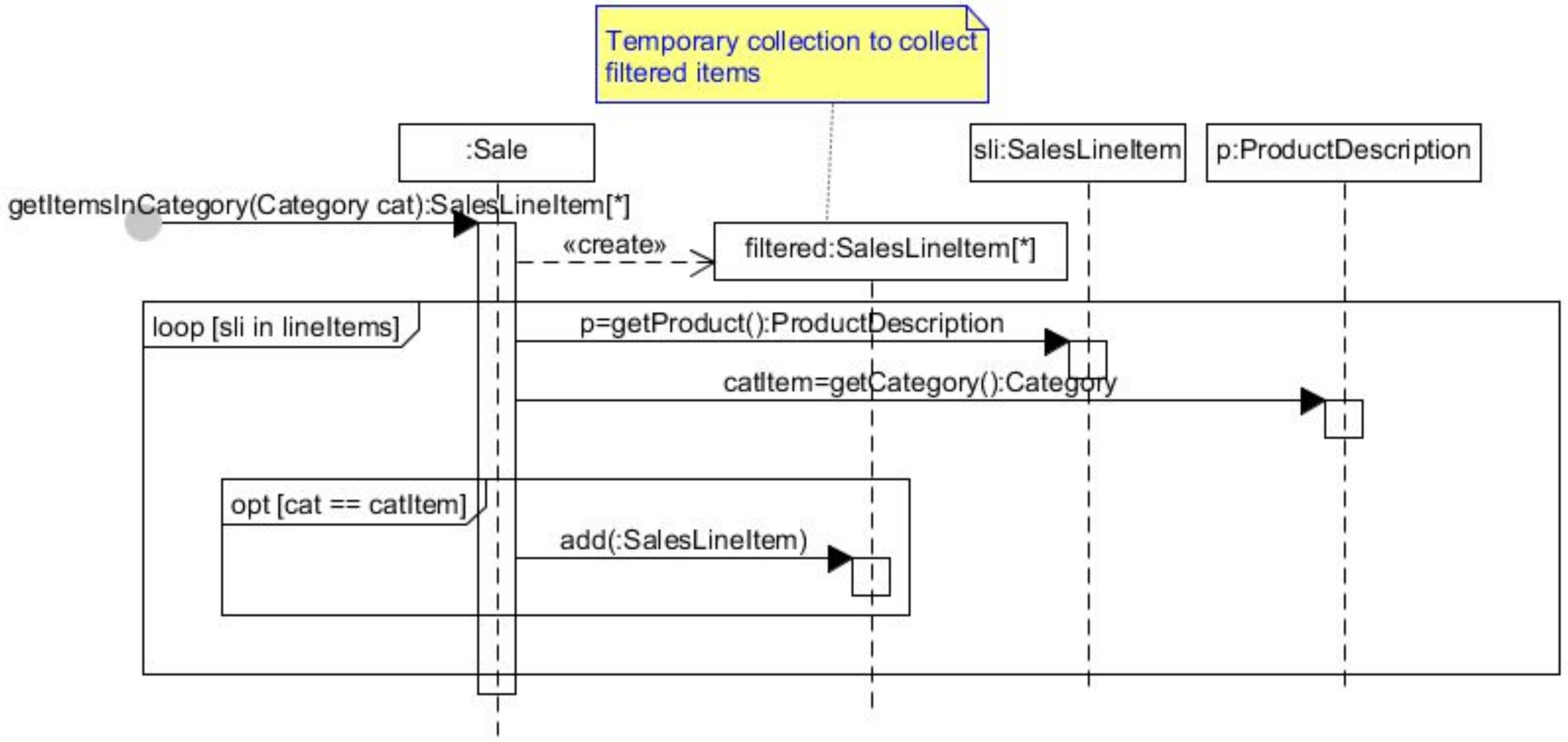
sd example v1: filter collection


15.4

```
public class Sale {
    private List<SalesLineItem> lineItems;
    …

  public List<SalesLineItem> GetItemsInCategory(Category cat){
    List<SalesLineItem> result = new ArrayList<>();
    for (SalesLineItem sli:lineItems){
      if (cat == sli.getProduct().getCategory()){
        result.add(sli);
      }
    }
   return result;
  }
…
}
```

University of Applied Sciences and Arts

sd example v1: filter collection


Temporary collection to collect filtered items
:Sale
sli:SalesLineItem
p:ProductDescription
getItemsInCategory(Category cat):SalesLineItem[*]
«create»
filtered:SalesLineItem[*]
loop [sli in lineItems]
p=getProduct():ProductDescription
catItem=getCategory():Category
opt [cat == catItem]
add(:SalesLineItem)

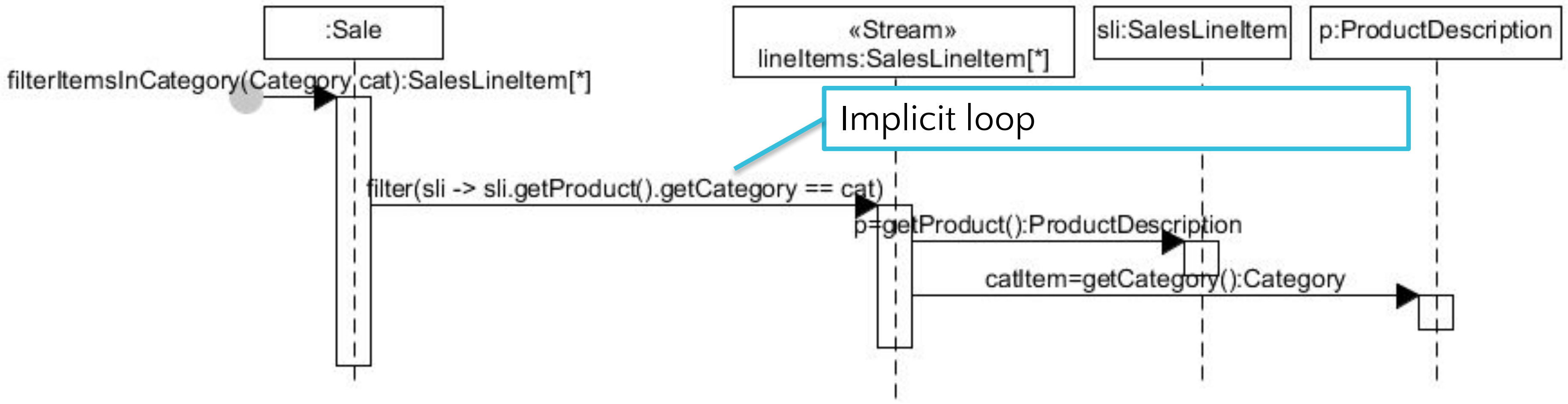
sd example v2: filter collection with stream and lambda

```
public class Sale {
    private List<SalesLineItem> lineItems;
    …

  public List<SalesLineItem> filterItemsInCategory(Category cat){
     return lineItems.stream()
       .filter(sli -> cat == sli.getProduct().getCategory())
       .collect(Collectors.toList());
  }
…
}
```

sd example v2: how to model a lambda

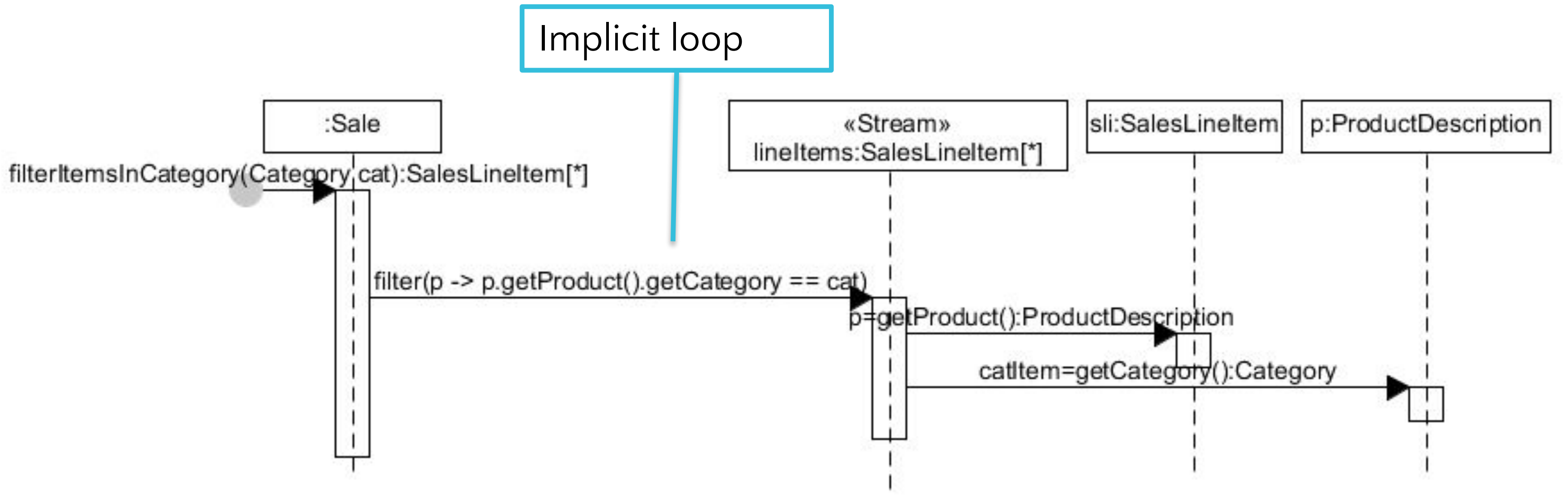
- De filter method is part of the Streams API: no need to model it
  - But the filter method uses a lambda parametr. The lambda is our code: we need to model it

⇒ When passing a lambda to an API: only model the lambda.
  - The reader should understand how the API uses it
  - The filter API method loops over the collection. We do not draw the loop fragment: but it IS present in the logic!

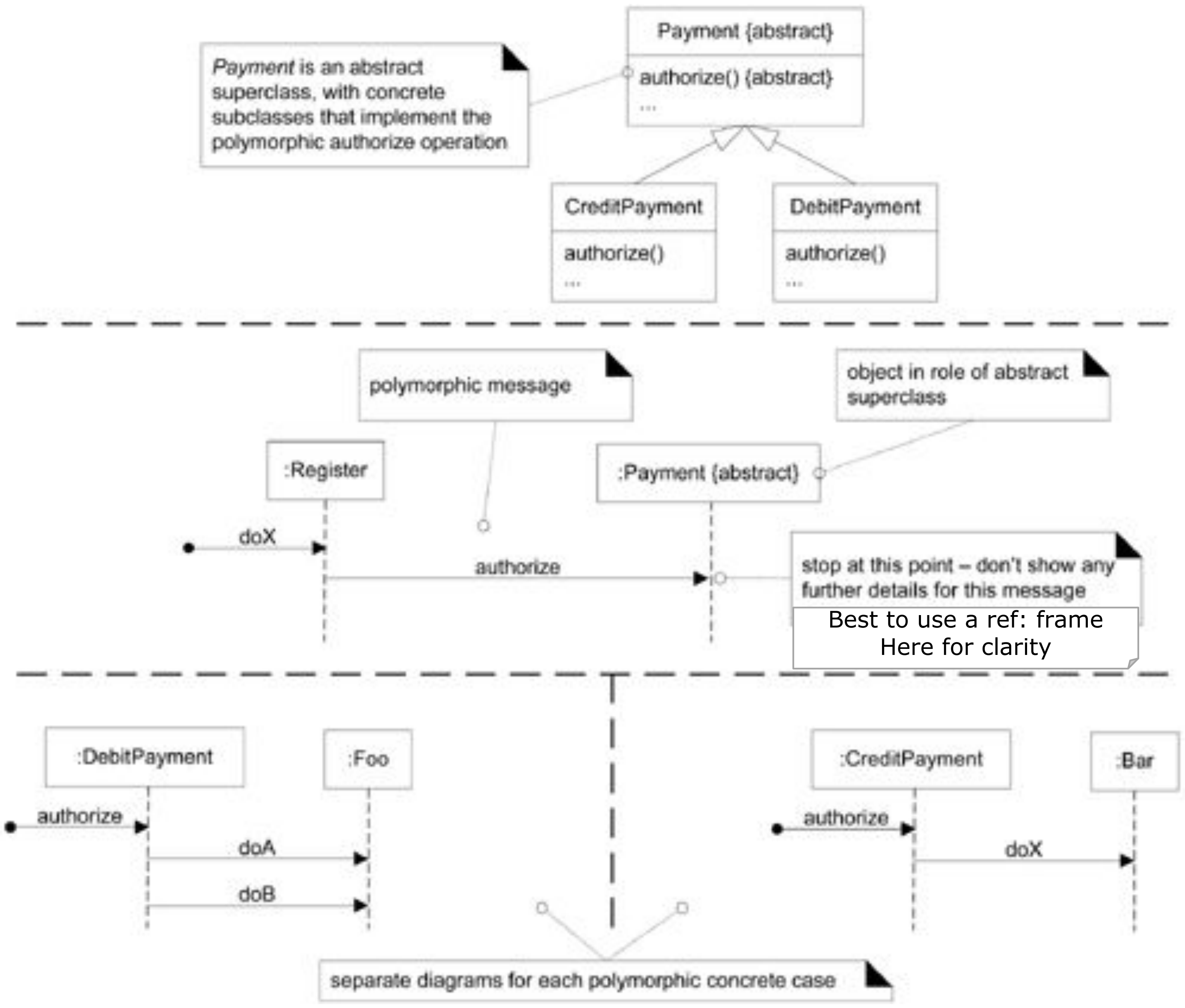

:Sale
«Stream»
lineItems:SalesLineItem[*]
sli:SalesLineItem
p:ProductDescription
filterItemsInCategory(Category cat):SalesLineItem[*]
Implicit loop
filter(sli -> sli.getProduct().getCategory == cat)
p=getProduct():ProductDescription
catItem=getCategory():Category

sd example (final): compact «Stream» notation

- Collections are often converted to (`stream()`) or from streams (`collect()`). As a shortcut we propose to use the stereotype «Stream» SalesLineItem[*] and leave out these conversions

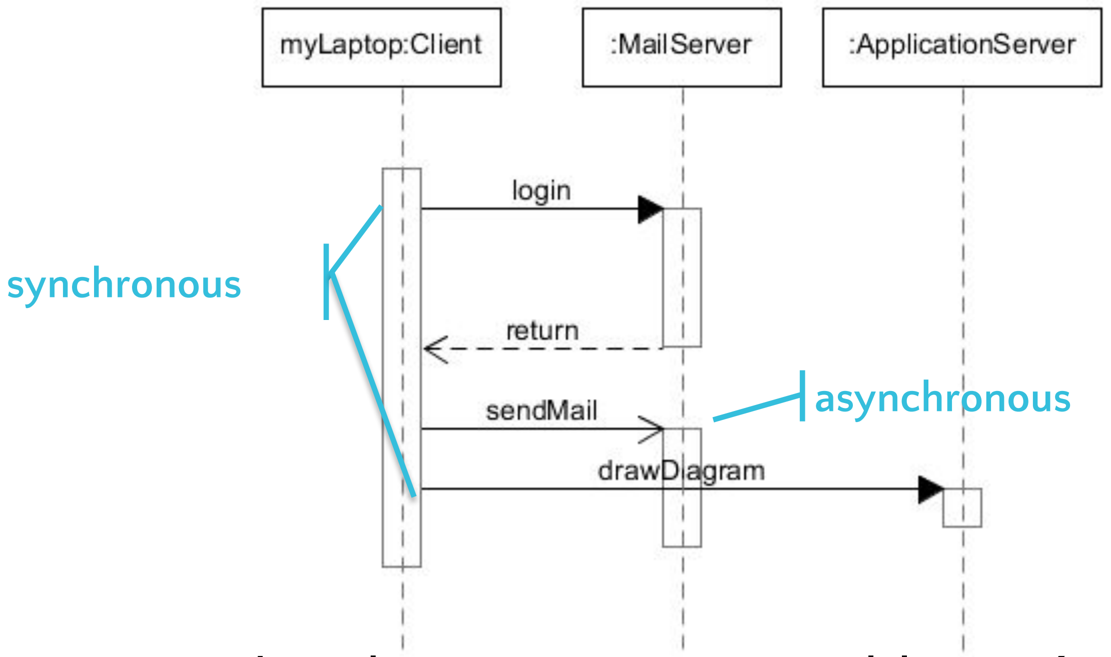

Implicit loop
:Sale
«Stream» lineItems:SalesLineItem[*]
sli:SalesLineItem
p:ProductDescription
filterItemsInCategory(Category cat):SalesLineItem[*]
filter(p -> p.getProduct().getCategory == cat)
p=getProduct():ProductDescription
catItem=getCategory():Category

Polymorphism


Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation
Payment {abstract}
authorize() {abstract}
...
CreditPayment
authorize()
...
DebitPayment
authorize()
...
polymorphic message
object in role of abstract superclass
:Register
:Payment {abstract}
doX
authorize
stop at this point – don't show any further details for this message
Best to use a ref: frame Here for clarity
:DebitPayment
:Foo
authorize
doA
doB
:CreditPayment
:Bar
authorize
doX
separate diagrams for each polymorphic concrete case

Asynchronous message

- **Asynchronous message:** caller does not wait until message is processed
- *open arrowpoint!*



- sendmail can not start until login (synchronous) is done
- *drawDiagram does not have to wait until senMail (asynchronous) is done*