# Behavioural Design

Jörg Kienzle & Alfred Strohmeier

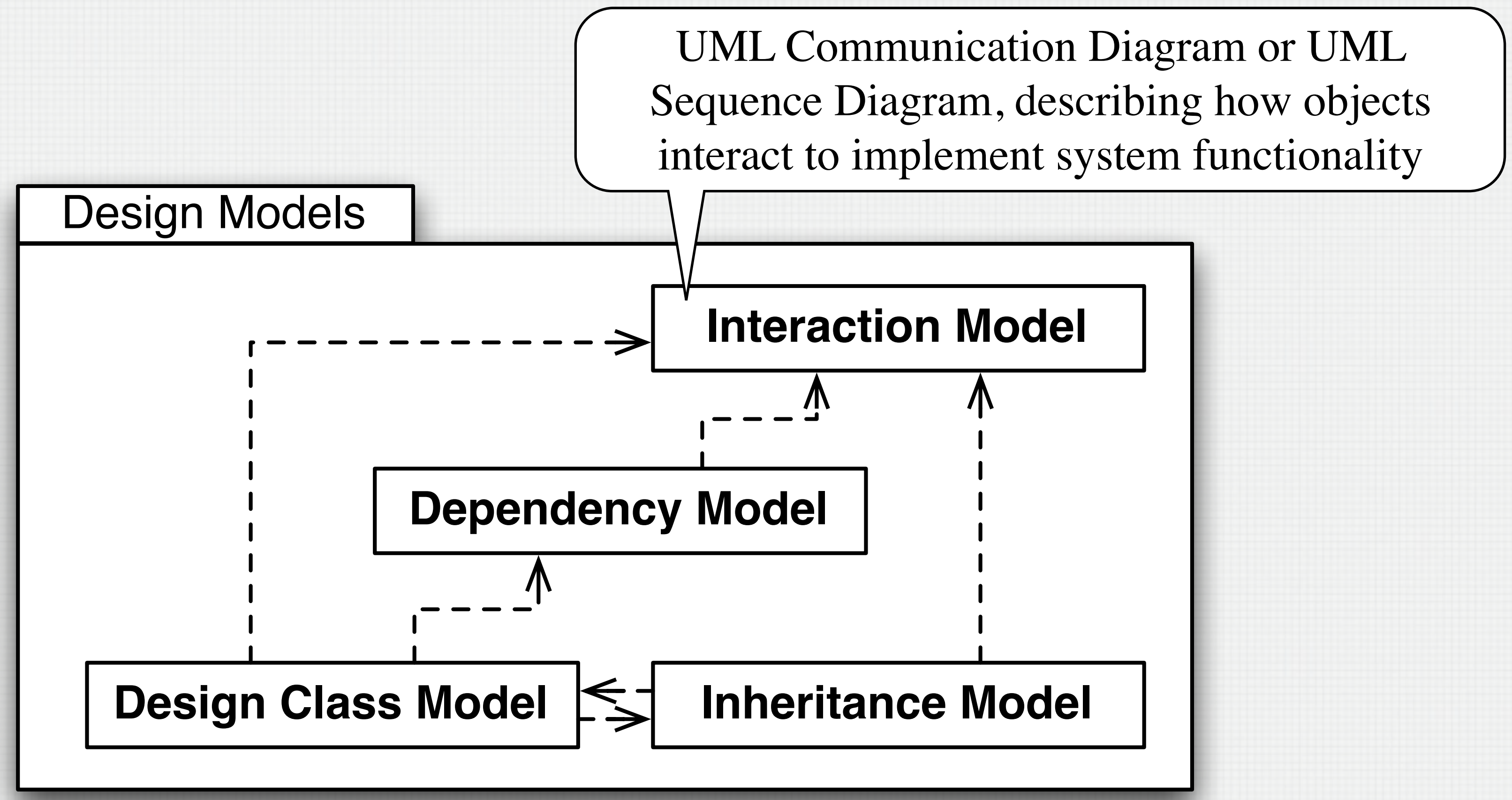
## Behavioural Design Overview

- Purpose and Process of Design
- Interaction Model
  - Communication Diagrams
  - Sequence Diagrams
  - Messages
  - Multi-objects
  - Object Lifetime
  - Process

## Design Purpose

- Develop a system architecture that satisfies the requirements defined in the requirements specification phase
- The designer chooses how the system operations are to be implemented by interacting objects at run-time
    - Different ways of breaking up a system operation into interactions can/should be tried
- The operations are attached to classes
- Provide the foundation for implementation, testing, and maintenance

# Fondue Models: Design


UML Communication Diagram or UML Sequence Diagram, describing how objects interact to implement system functionality
Design Models
Interaction Model
Dependency Model
Design Class Model
Inheritance Model

# Design Models

- Interaction Model

  Describes the Complete System Behaviour

  - The Interaction Model shows how objects interact at run-time to support the functionality specified in the Operation Model.
- Dependency Model
  - The Dependency Model describes dependencies between classes and communication paths between interacting objects.
- Inheritance Model
  - The Inheritance Model describes the superclass/subclass inheritance design structure.
- Design Class Model

  Describes the Complete System Structure

  - The Design Class Model is composed of the contents of all design classes, i.e. their (value) attributes and methods, all the navigable associations between design classes, and the inheritance structure.

## The Design Process in a Nutshell

5. Develop the Interaction Model
    1. Develop communication diagrams for all System Operations
    2. Derive a consistent architecture assigning responsibilities to classes
    3. Revise the communication diagrams, yielding the Interaction Model
6. Develop the Dependency Model based on the Interaction Model
    1. All communication designed in the Interaction Model result in dependencies
7. Develop the Design Class Model
    1. Develop a first version of the Design Class Model based on the Interaction Model
    2. Factor out common properties of classes and build the Inheritance Model
    3. Update and build the final Design Class Model

## Interaction Model (1)

- An operation schema declaratively specifies the behaviour of an operation by defining its effect in terms of the change in conceptual system state and the output messages.
- The purpose of design is to build the object messaging structures that realize the abstract definition of behaviour as stated in the operation schema.
- A communication diagram (or sequence diagram) is constructed for each system operation.
  - The communication diagram shows what objects are involved in the computation and how these objects communicate to satisfy the functional specification.
  - All communication diagrams together form the Interaction Model.

## Interaction Model (2)

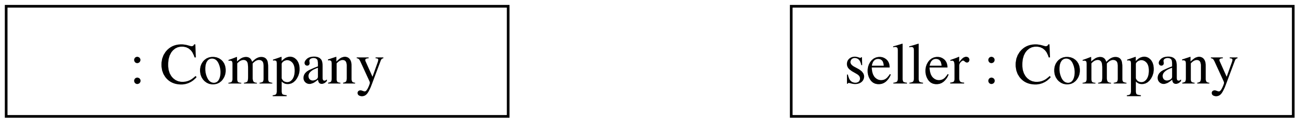
- Communication diagrams are developed incrementally for each system operation.
- The goal is to distribute the functional behaviour across classes in the system.
  - Recall that requirements specification classes do not have a method interface, whereas design classes do.
- As design develops, the interface of classes in the system evolves.
  - During design, responsibilities are assigned to classes, which determines what operations a class should provide.
  - Hierarchical decomposition may be applied to operations.

## Communication Diagram (1)

- A communication diagram is a collection of boxes that are linked by lines and message arrows.
- A box represents a design object, or more precisely a role played by a design object.
- An arrow represents an operation invocation. A message is sent by a sender or client to a receiver or server.
  - Only one arrow has no sender. It corresponds to the system operation or method whose realization is shown by the communication diagram.
- The design object associated with the system operation is called the controller object.
- The other boxes are collaborators.
- The graph is connected: all boxes can be reached by traversing a path starting at the controller.

## Roles

- A role is not an individual object, but a description of an object that may play a part in a collaboration instance
  - Each time the collaboration is instantiated, a different set of objects play the roles
  - The same object may play different roles in different collaborations
  - A role can be unnamed



## Messages (1)

- A message conveys information from one object, the sender, to another, the receiver, to trigger some activity.
- All messages represent synchronous operation invocations, i.e. the sender object waits until the operation execution on the receiver object is complete
- Because it is a call, actual parameters are shown
  - Except for the operation realized by the communication diagram, for which formal parameters are shown.

## Messages (2)

call ::= label decor [“*”] [return-value-list “:=“]
    messagename “(“ argument-list “)”

label ::= integer {“.” integer} [name]

- A Dewey number shows ordering and nesting.
- A name is used to show concurrent threads.

decor ::= “ ’ ” one or several ticks to show a choice

“*” ::= the message is sent repeatedly.

return-value-list ::= name {“,” name}

- The names are actuals for returned entities.

argument-list ::= expression {“,” expression}

- The expressions in the argument list are actual parameters. An expression may use return values of previous messages and attributes of “known” objects.

## Message Syntax

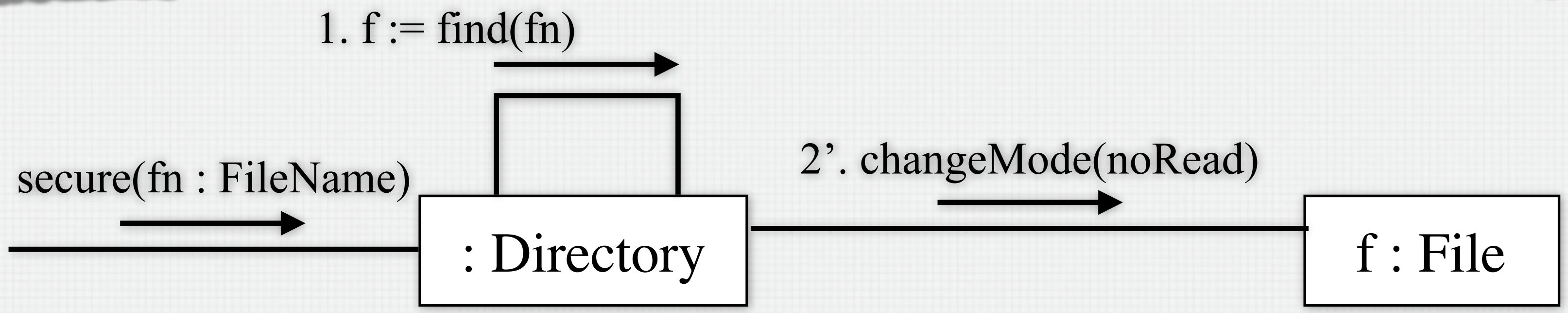
- Messages can be labeled by Dewey decimal numbers to show sequencing: 1, 1.1, 1.2, 1.2.1, 1.2.2, 1.3, 2, 3, etc.
- All method calls at a lower level have to complete before the control is returned to the enclosing method: 1.1, 1.2, 1.3, etc. must complete before control is returned from 1.
- The same number for two messages means that there is no imposed order, or that the order is not shown.
- Ticks or other decorations show exclusive choices: either 3 or 3’ or 3’’. Note that the choice condition is not shown.
- Names show concurrent threads; 4ta concurrently with 4tb.
- A star shows repetition: 3*

# Message Examples

| | |
|---|---|
| • 2: display (x, y) | -- simple message |
| • 1.3.1: p := find (pName) | -- nested call with return value |
| • 3.1*: update() | -- iteration |
| • 2.1: printReceipt (amount) | -- same Dewey number |
| • 2.1: deliverCash (amount) | -- means “in any order”. |
| • 2.1ta: printReceipt (amount) | -- it is also possible to show |
| • 2.1tb: deliverCash (amount) | -- “in any order” by threads. |
| • 2.1’: insufficientFunds | -- either one, |
| • 2.1”: deliverCash (amount) | -- shown by a décor. |

# Communication Diagram Example (1)

- Communication diagram for system operation *secure*
- The controller of the operation is a Directory object.
- The File object f collaborates to perform the operation *secure*.
- The Directory object may send a message changeMode to the file, with the actual parameter noRead.



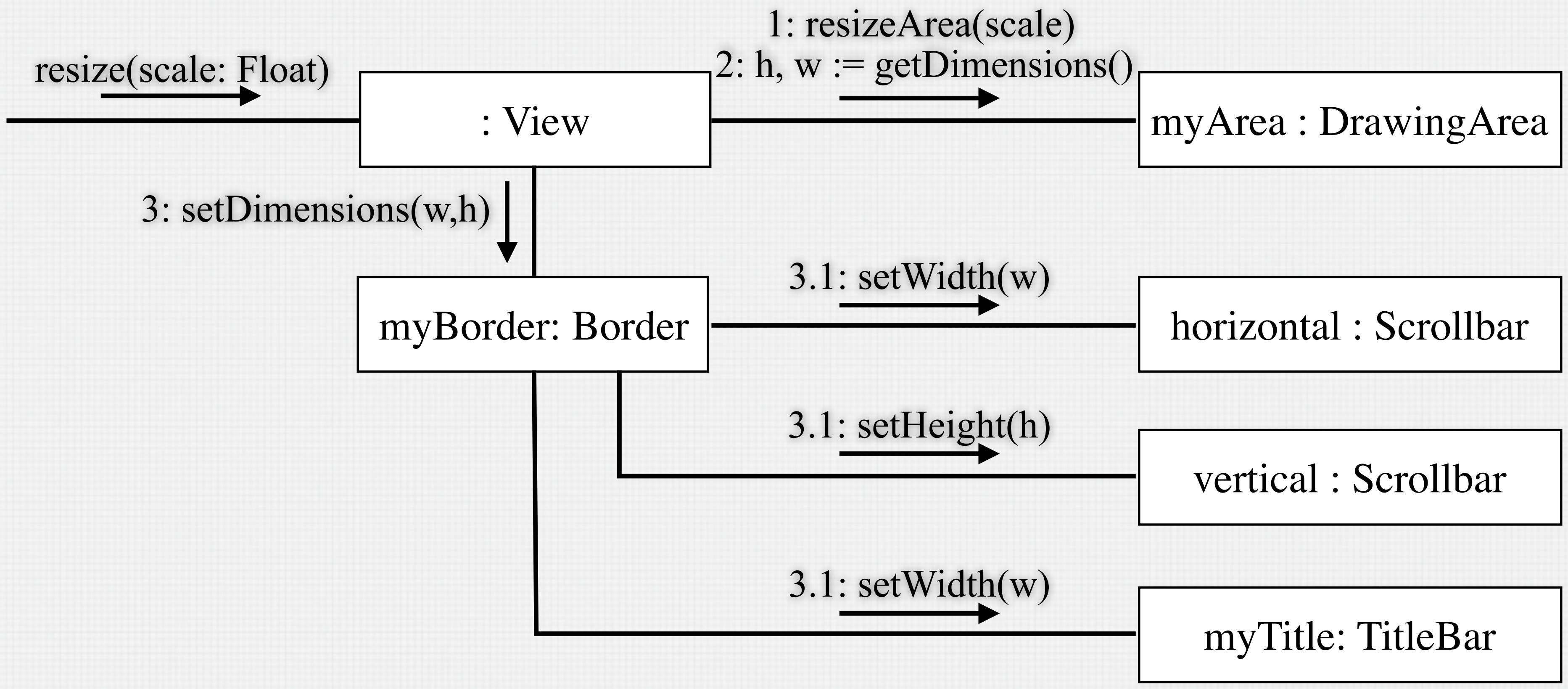
- The Directory class has an operation
  - find(name: FileName) : File
- The File class has an operation
  - changeMode(Mode), where Mode is an enum that includes the value noRead

# Pseudo Code

- It is often useful to complement a communication diagram with pseudo-code, showing not only message passing, but also internal processing and the conditions controlling conditional message passing and iterations.

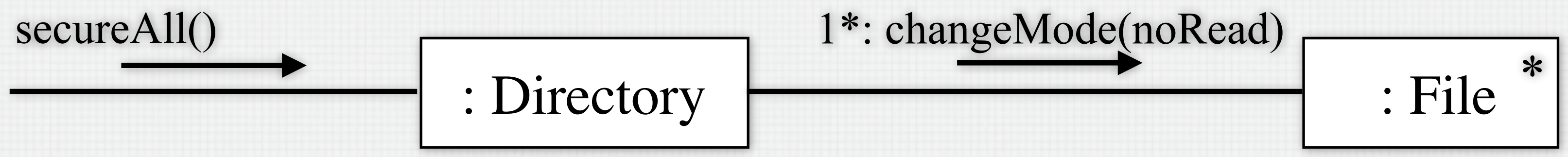
```
operation Directory::secure(fn: FileName) : File
  f: File;
begin
  f := find(fn);
  if f /= null then
      f.changeMode(noRead);
  end if;
end secure;
```

# Message Sequencing Example



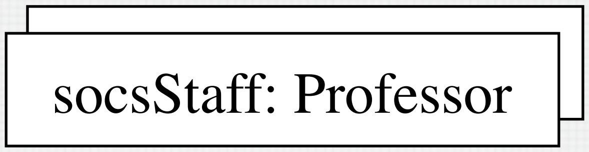
## "Many" Role

- It is possible to have a named role with multiplicity many, denoted by a star.
  - The role then denotes a collection of objects, but the message is sent to the collection members, rather than to the collection as a unity
    - The method is therefore part of the interface of the member class
  - Note that the collection can change with every instantiation of the collaboration.

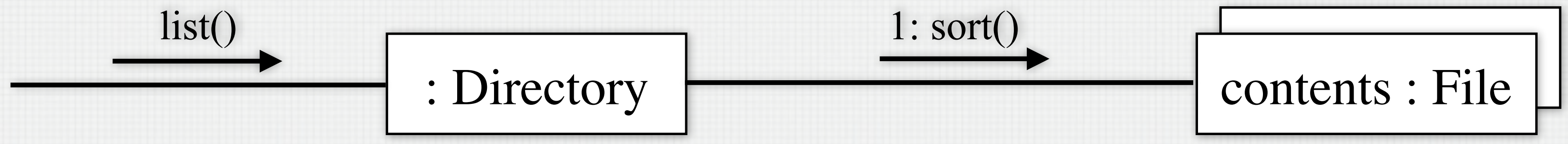


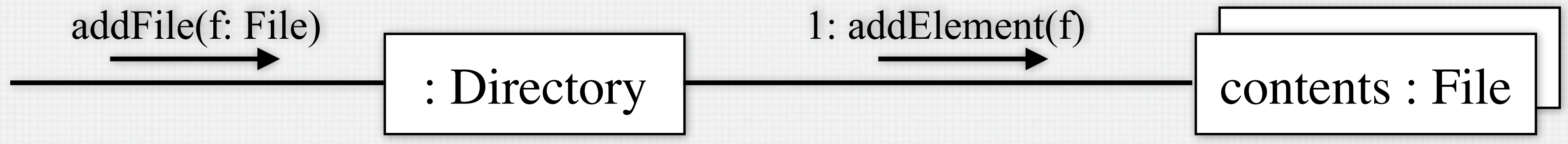
## Multiobjects (1)

- A role which denotes a collection of objects, rather than a single object, is called a multiobject.
- A multiobject is used within a collaboration to show operations that address the entire collection of objects as a unit.
  - The name of the role is that of the collection, but the class name is that of the elements.
    - The message is part of the interface of the collection, and not of the interface of its members.



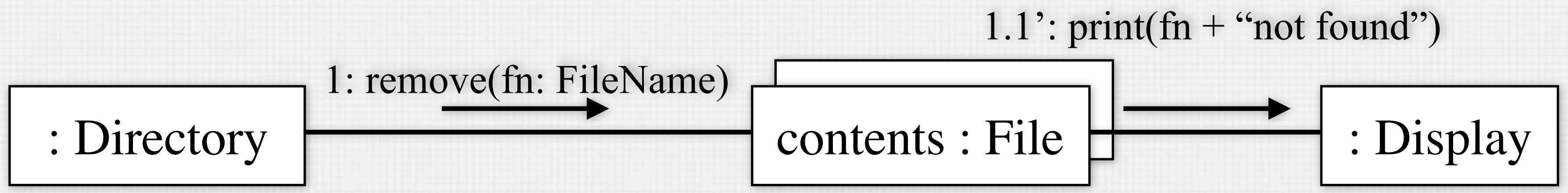
# Multiobject Examples

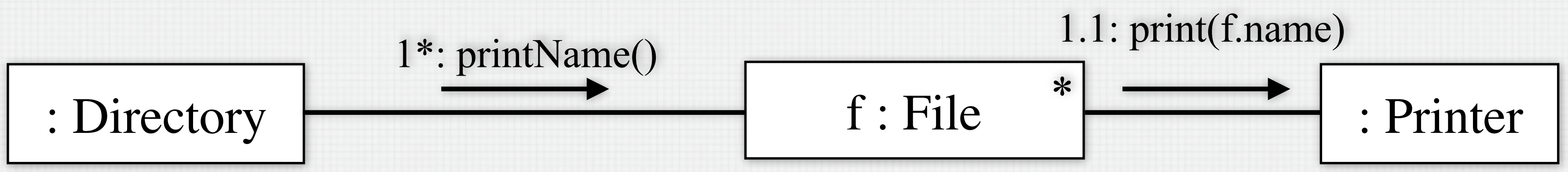




## Multiobjects and Role Many

- A multiobject may in turn send a message. Such a message is sent by the collection.
- Every object in a role of multiplicity many can in turn send a message.

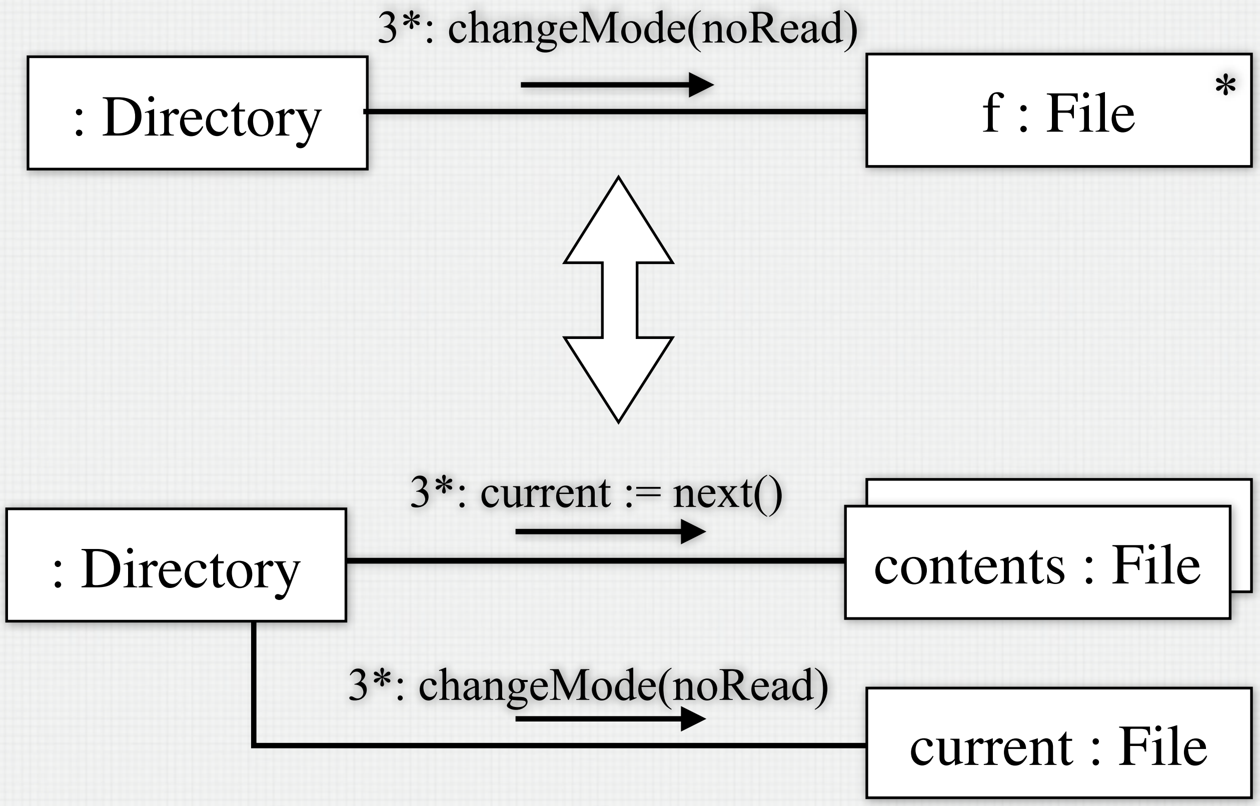




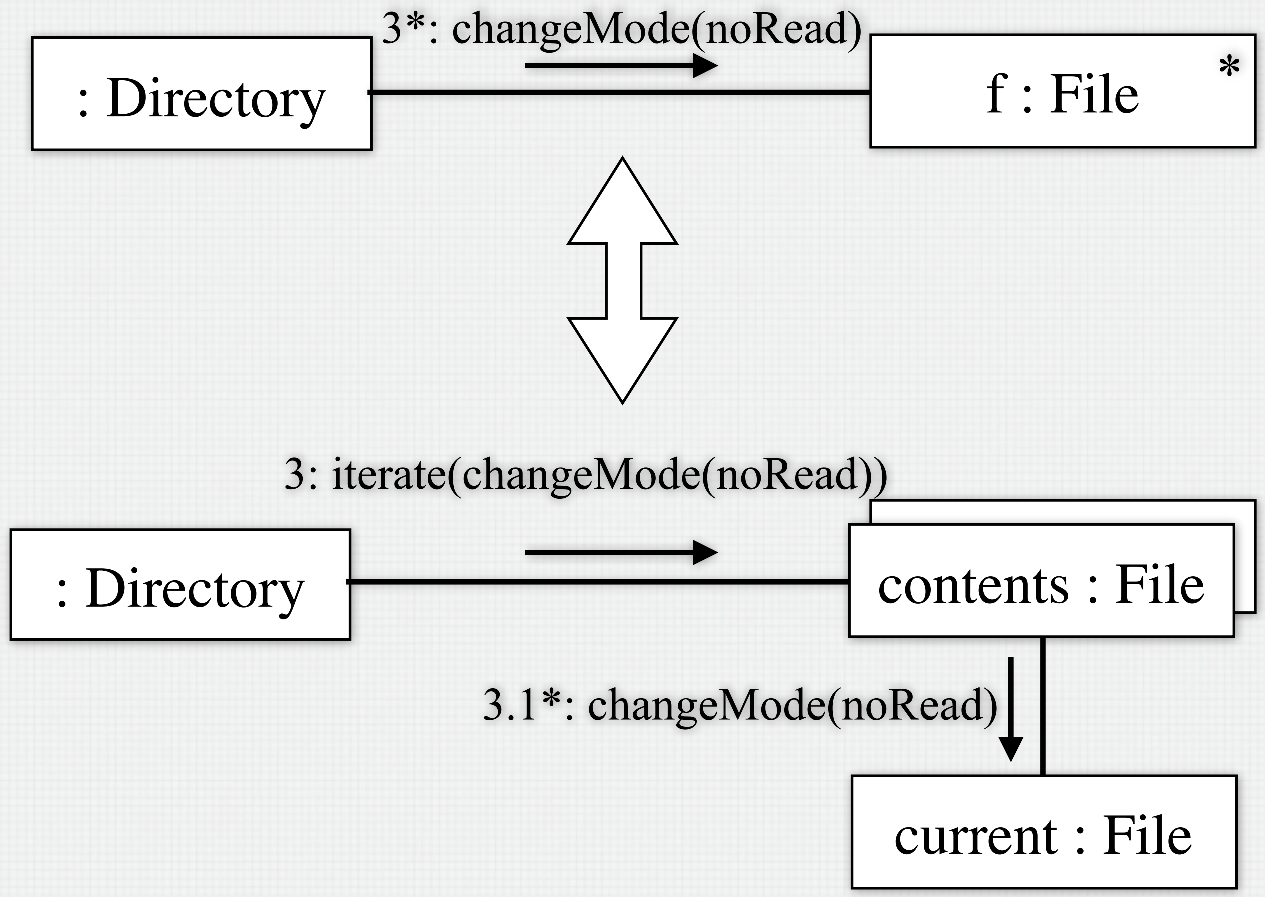
## Multiobjects (2)

- To perform an operation on each object in a collection of objects requires two messages: an iteration on the multiobject to extract links to the individual objects, then a message sent to each object using the temporary link.
- This process is elided on a diagram by combining the messages into one that includes an iteration and an application on each object. The target role name takes a many indicator “*” to show that many links are implied.

# Multiobjects (3)

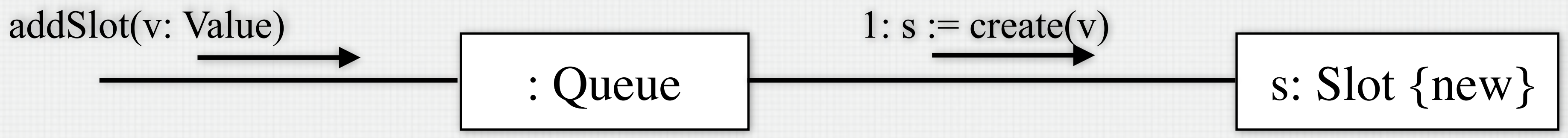


# Multiobjects (4)



## Lifetime of Objects

- Objects without a keyword exist when the operation begins and still exist when it completes
- Objects created during execution are designated as {new}, those destroyed as {destroyed}, and those created and then destroyed as {transient}



## Process: Identify Relevant Objects

- The operation schema is used as a starting point.
- The parameter list, the Scope and New clauses provide objects and classes “used” by the system operation.
- For each output message, a mechanism must be devised that implements the communication with the environment.
  - At least one object must be introduced to deal with the interface to the receiving actor.
- It is often necessary to introduce new objects to represent abstractions of computational mechanisms not identified in the concept model.

## Interaction Model Example (1)

```
Operation:    Bank::openAccount (c: Customer);
Description:  A clerk opens an account for a client;
Scope:        Account, Customer, Owns;
Messages:     Clerk::{AccountNumber;};
New:          newAccount: Account;
Pre:          true;
Post:         newAccount.oclIsNew() &
              newAccount.balance = 0 &
              self.account.number@pre→excludes
                 (newAccount.number) &
              c.account→includes(newAccount) &
              sender^accountNumber(newAccount.number);
```

## Interaction Model Example (2)

- Relevant Objects
  - bank, account, customer
- Which one could be the controller?
  - the account does not exist yet,
  - it’s not the job of the Customer class to open accounts
  - the bank seems to be the only choice,...
- but beware...
  - In a banking system, don’t make “the bank”, i.e., a single global object, the controller of all system operations. “invent” a new class if needed!

## Interaction Model Example (3)

- Think about a possible implementation of the system operation, e.g., by decomposition into other methods

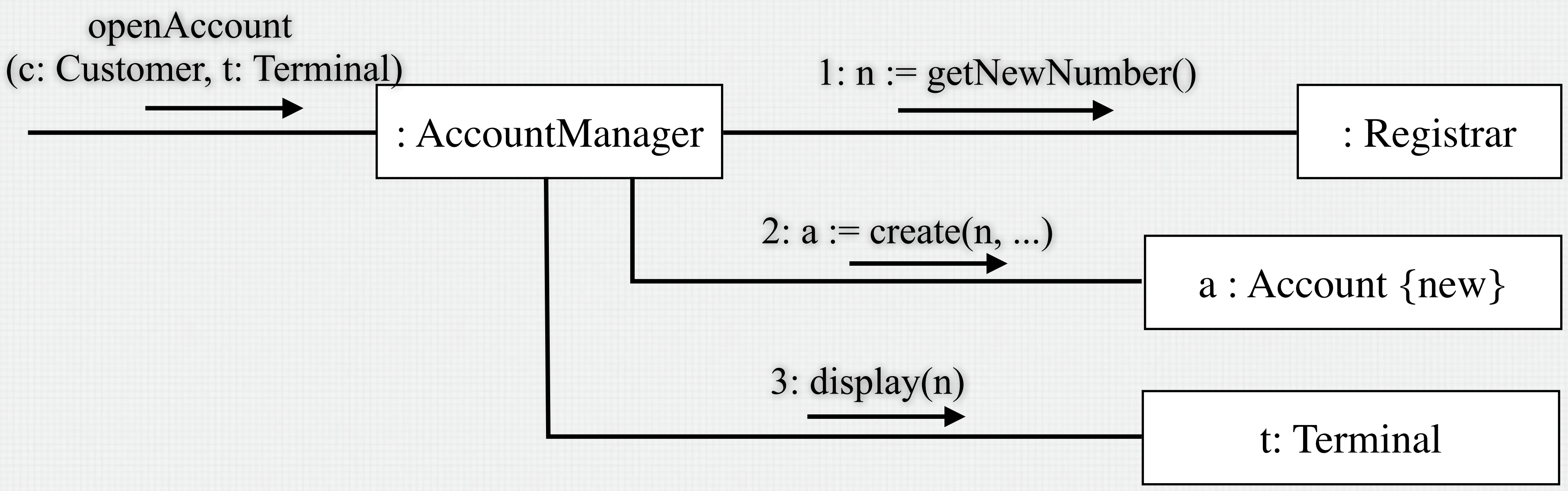
- To open an account, it must first be created.
- An account has a unique number. Who will deliver this? The Account class may have this knowledge, or there may be some Registrar object.
- How can the AccountNumber (nb: Number) message be sent to the clerk? We will suppose that the clerk is sitting in front of a terminal, and that it is enough to display a message on his/her terminal's screen. We therefore add a Terminal class to the system.

# Interaction Model Example (4)

- Make decisions, with examples
- If needed, ask for more information about the application domain / context, e.g., about technical details on how to communicate with the environment
- Assess the different designs.

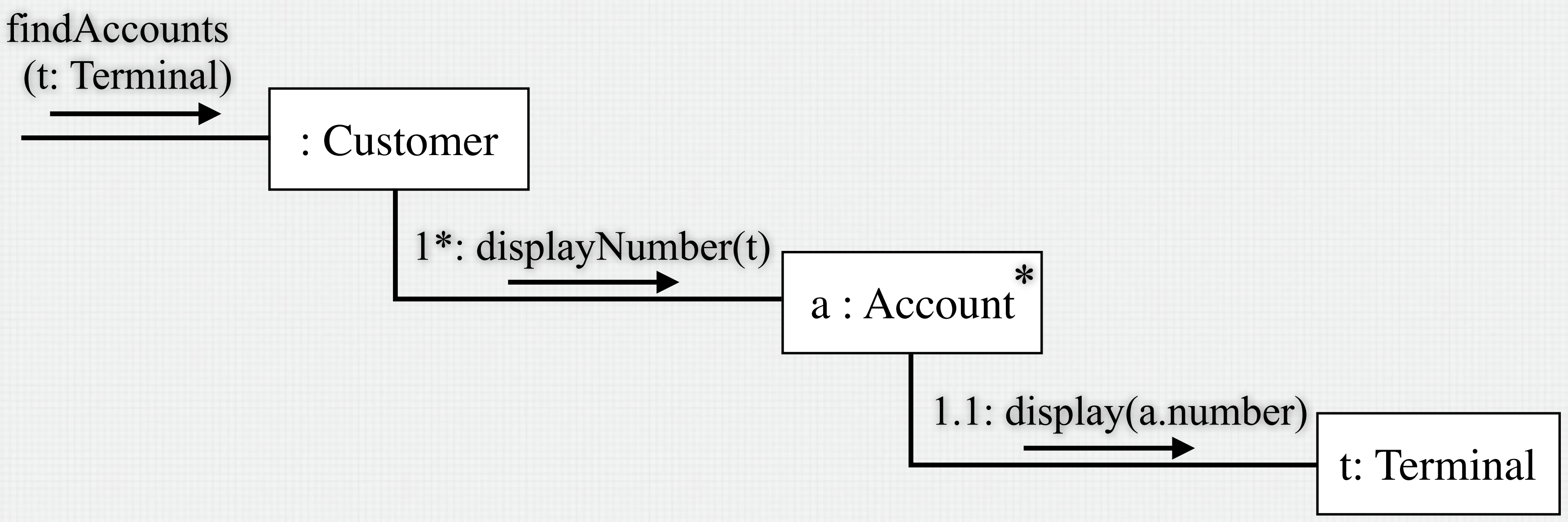
- We decide that we will implement delivery of unique account numbers by means of a central registrar.
- We will have to ask how we can identify the terminal associated with the clerk.
- Let's suppose we are told that the terminals can be referenced, and that the reference is supplied as a parameter to the operation. Note this implements the "sender", supplied implicitly with the input message triggering the operation.

# Interaction Model Example (5)



No explicit use is made of customer c. Can we remove it?

# Interaction Model Example (6)

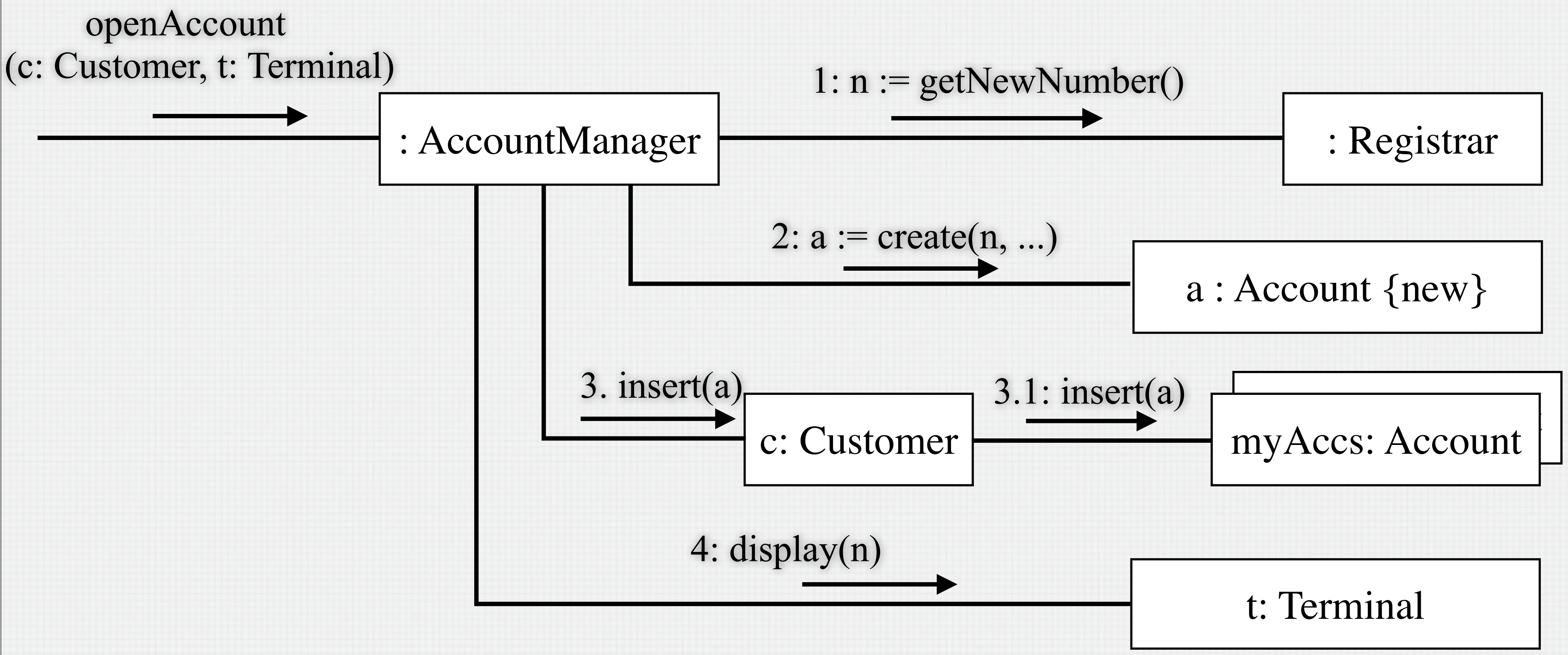


## Interaction Model Example (7)

- Check for Consistency!

- How can a customer know about his/her accounts?
- Possible solution: At creation of an account, the customer who will own the account is notified.
  - ➡ The openAccount communication diagram needs to be revisited. See next slide.
- Note: It is generally impossible to complete communication diagrams and even parameter lists for "create" operations before all services required from the class are known.

## Interaction Model Example (8)

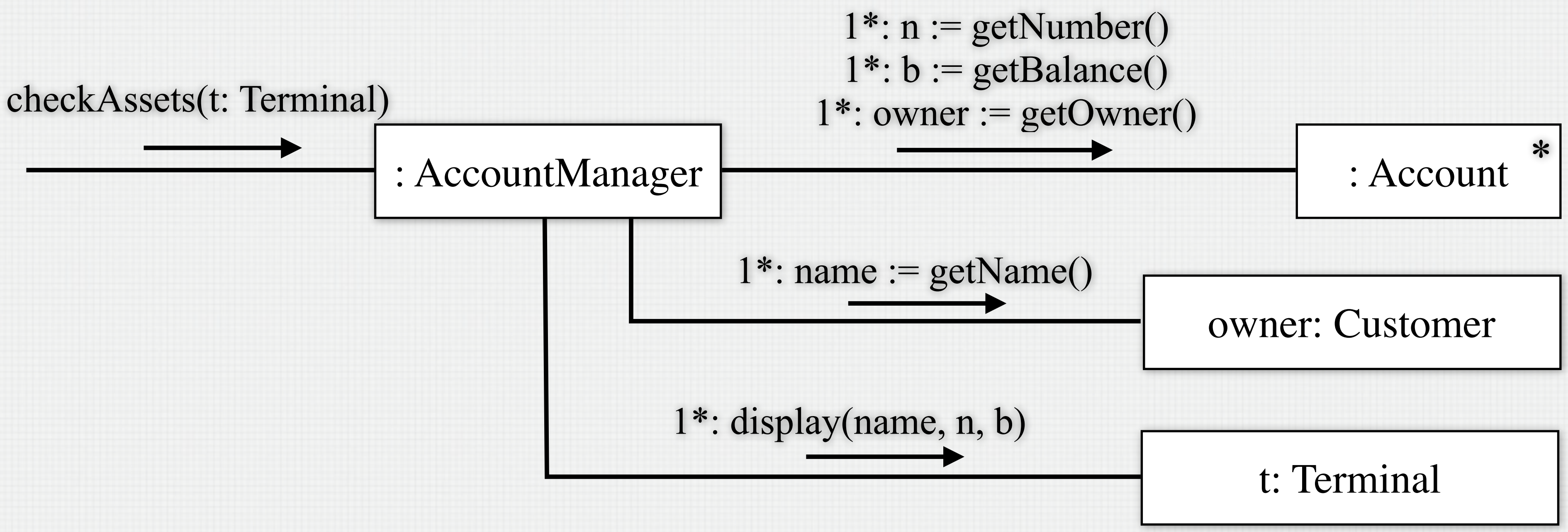


# Check Assets (1)

**type** LineItem = **TupleType**
 {name: String, number: AccNumber, amount: Money}
**message type** CurrentAssets (contents: Sequence (LineItem));

**Operation**: Bank :: checkAssets ();
**Description**: Request issued by a manager. Lists the balances of all accounts, together with the owner's names;
**Scope**: Account; Customer; Owns;
**Messages**: Manager :: {CurrentAssets;};
**Pre**: true;
**Post**: **let** seq: Sequence (LineItem) **in** **self**.account→**forAll** (a | seq→**includes**
  (**Tuple** {name = a.customer.name, number = a.number,
   amount = a.balance})) **and**
   **sender**^currentAssets(seq))
 **endlet**;

# Check Assets (2)

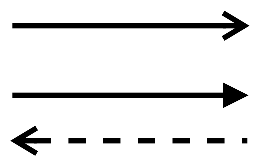


In this example, it is impossible to show the exact control structure by sequence numbers. It can be shown using sequence diagrams or by pseudocode.

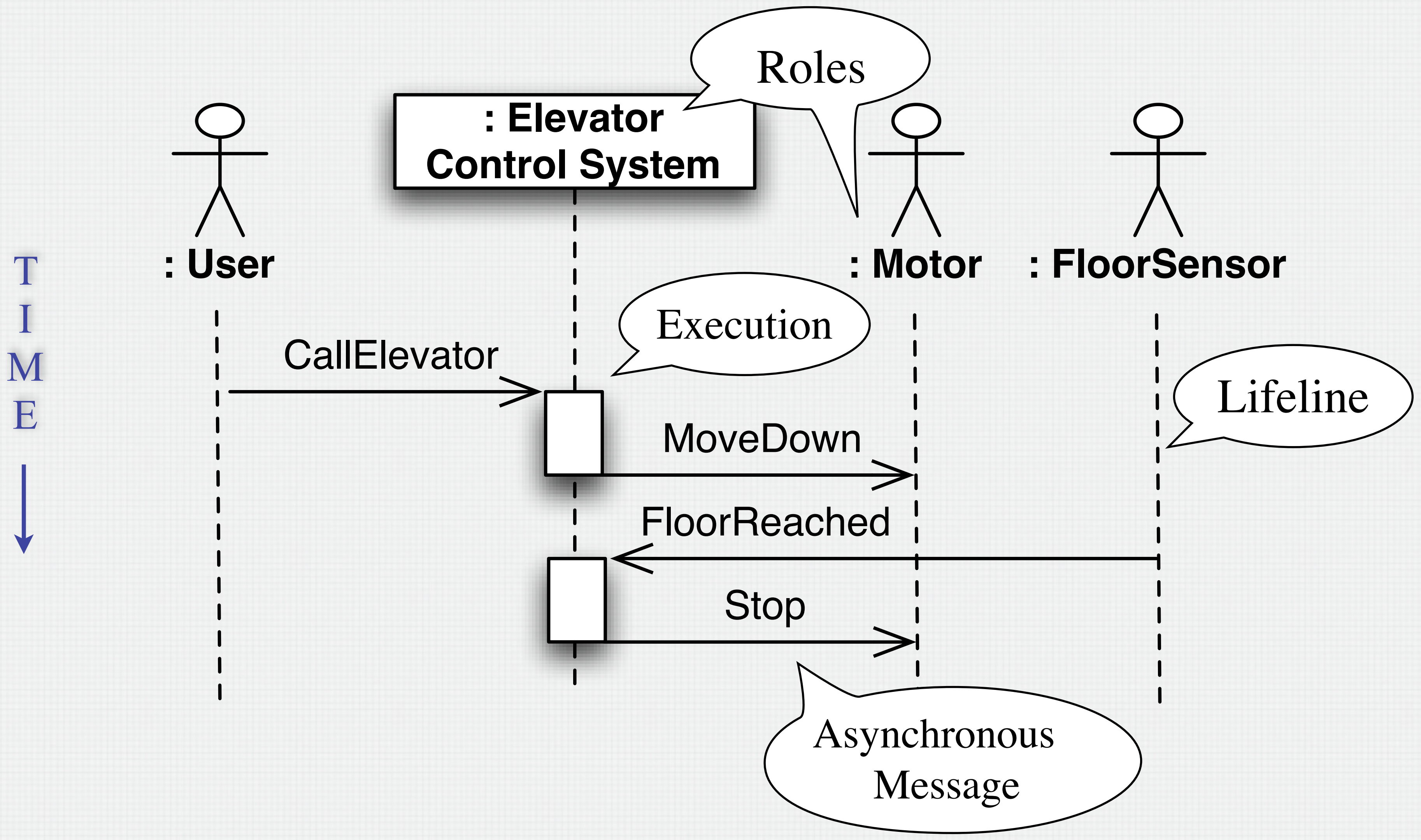
Does this design have an influence on the design of openAccount?

## Alternative Notation: Sequence Diagrams

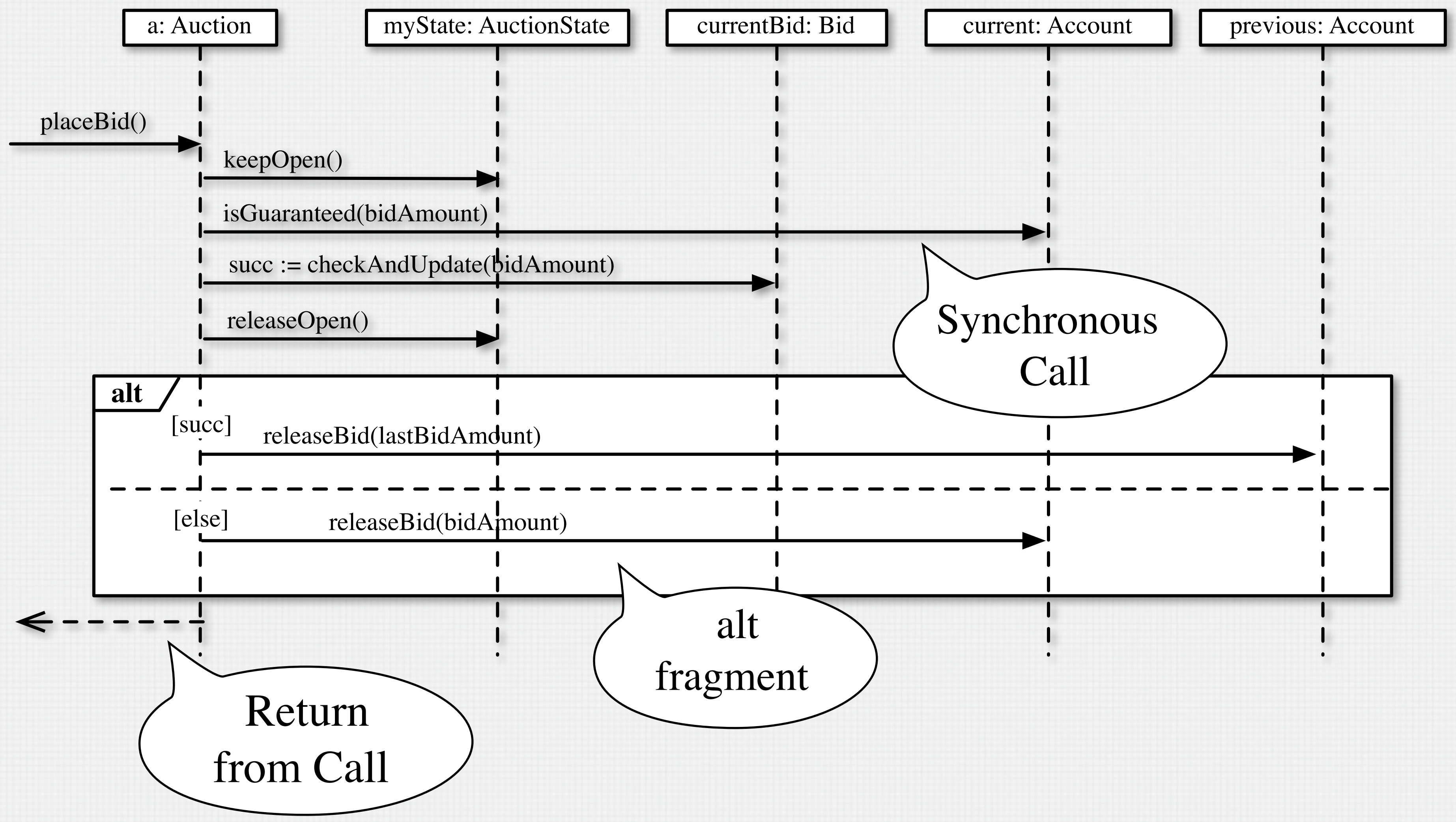
- A sequence diagram focuses on the sequence of messages interchanges between lifelines (objects that exist that play a role)
- It consists of:
  - Several (at least 2) lifelines
  - Messages
    - Asynchronous
    - Synchronous
      - Optionally with reply
    - Fragments
  - Hierarchy is achieved by referencing other fragments
- In Fondue, sequence diagrams can be used for
  - Specifying the Protocol Model (instead of URN or State Diagrams)
  - Specifying the Interaction Model (instead of communication diagrams)



## Protocol Model Sequence Diagram Example



# Interaction Model Sequence Diagram Example

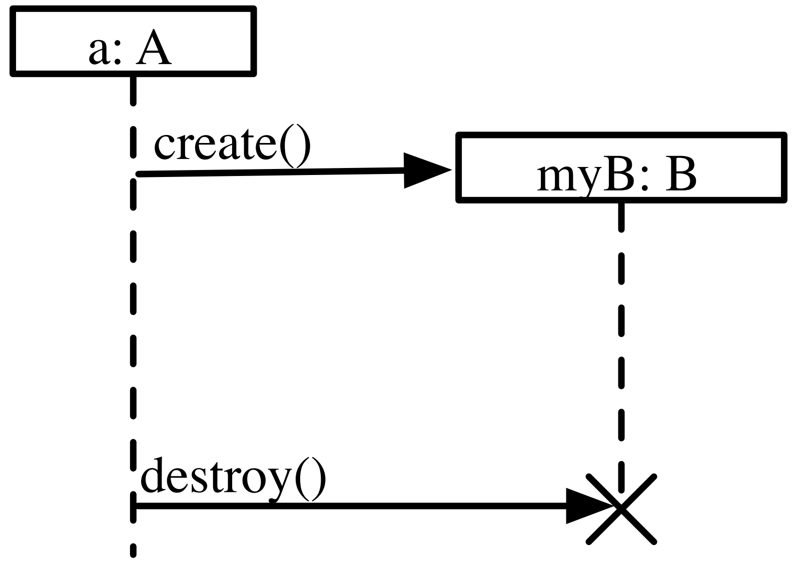


## Different Fragments

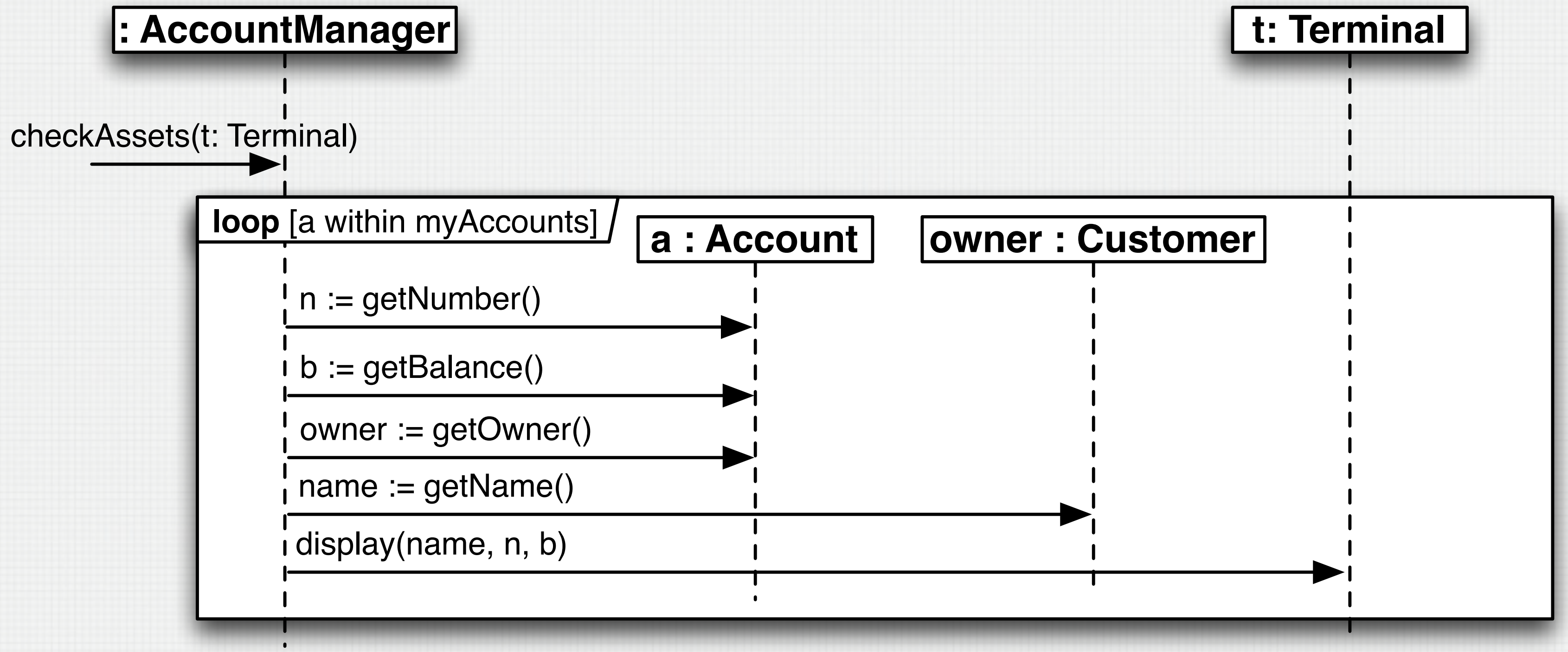
- **ref**: references some other fragment
- **alt**: alternatives with different conditions
- **opt**: executed if condition holds
- **loop**: iteration, either a fixed number of times or as long as a boolean expression is true
- **break**: interrupt the enclosing fragment if the condition holds
- **neg**: sequence that should never happen
- **critical**: a critical section that is never interleaved with other operations
- **seq**, **strict**, **par**, **ignore**, **consider**, **assert**

# Other Features

- Object creation
- Object destruction
- Time constraints
- State invariants
- Continuations



# Check Assets Sequence Diagram



## Interaction Model Process Summary

- Identify relevant objects involved in the computation.
- Establish the role of each object:
  - Identify the controller, i.e. the object responsible for the system operation
  - Identify the collaborators
- Assign responsibilities to objects
- Decide on messages between objects
  - Record how the identified objects interact on a communication diagram / sequence diagram
- Check consistency between communication diagrams / sequence diagrams
  - Responsibilities are key

## Communication Design (1)

- All communication links between the environment and the system must be designed
  - Input messages / output message
    - Network: Network messages, or
    - GUI: mouse / keyboard input, graphical output
    - Device drivers: interface with hardware
  - Conceptual parameters need to be realized
    - Network: serializable objects, or parameters are encoded using IDs / Strings
    - GUI: selection from a list / text, graphics
  - "sender" parameter
    - Network: network ID, IP address, port
    - GUI: window / frame / widget
    - Device drivers: device ID
  - Each class that has an <<id>> association with an actor in the concept model
    - Provides communication interface
    - Stores state to initiate communication, if needed

## Communication Design (2)

- A good way of decoupling output communication (for example using a GUI) is to use the Observer design pattern
- A good way to send and receive communication that triggers functionality over the network is to use the Remote Command pattern

- Decide how to identify object parameters (i.e. instances of classes of the Concept Model) in messages
  - During design, a class will need to be developed that is responsible of mapping from the identification means to the design class implementing the concept and vice versa.
  - For example: the AccountManager maps an account id to an Account instance

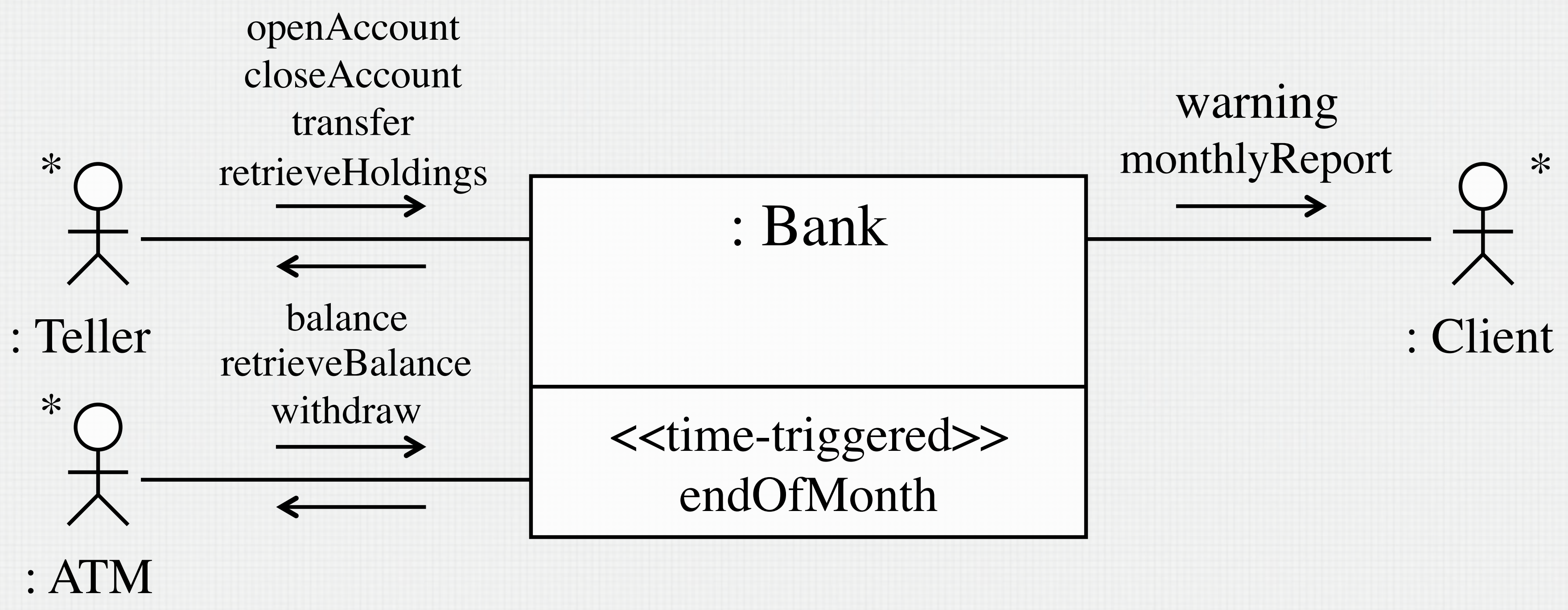
## Persistence

- Is persistence of (domain/business) data important?
  - Include loading / saving functionality
    - System startup loads data
    - System shutdown saves data
- Is persistence and fault tolerance important?
  - Use a database
  - Classes can still be used to encapsulate domain/business data
    - Classes additionally provide operations that read from / write to the database
      - Use a framework such as Enterprise Java Beans

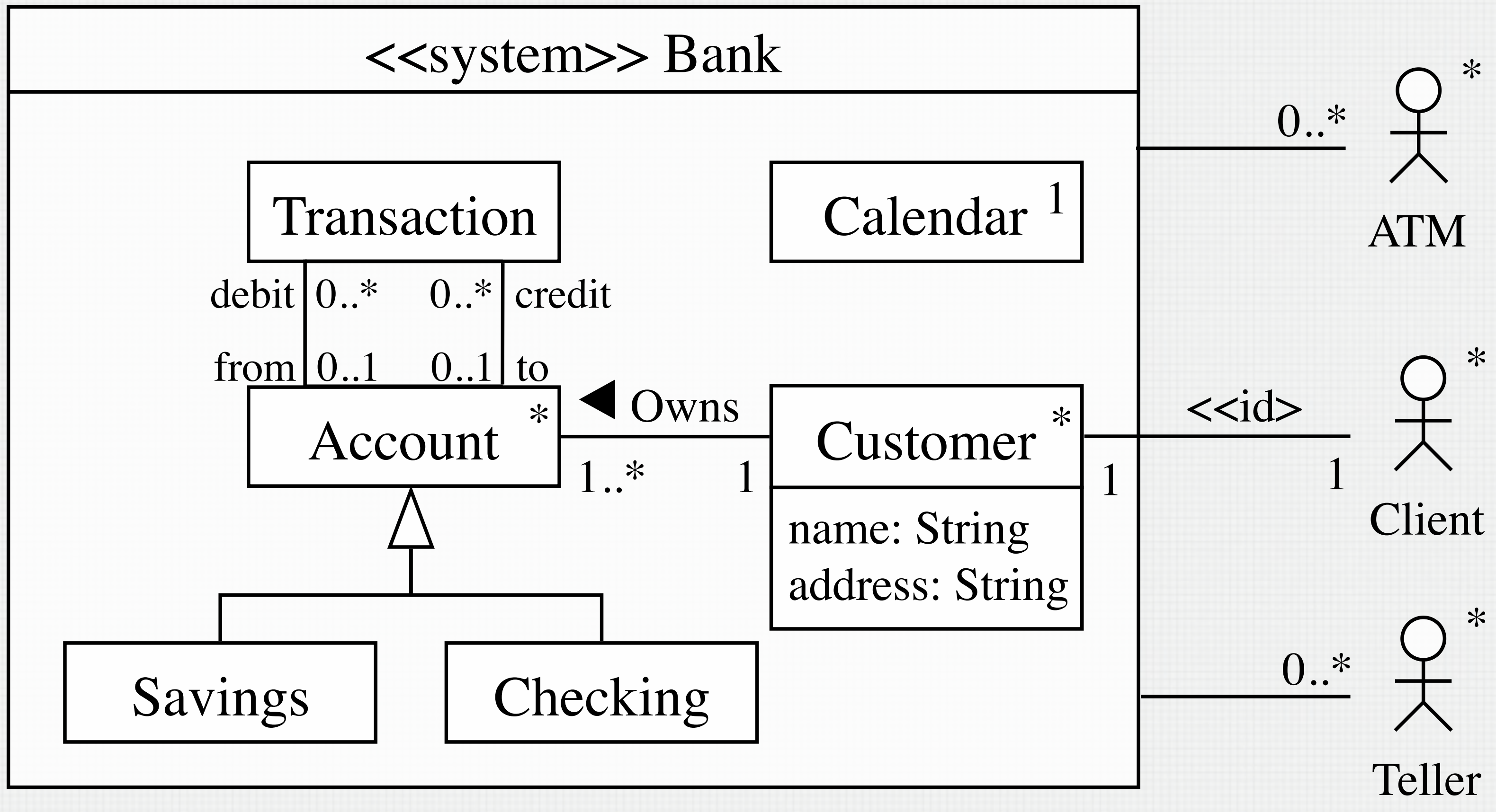
# Questions?

# Bank Question (1)

- The (partial) Environment Model and Concept Model of the Bank System are given below

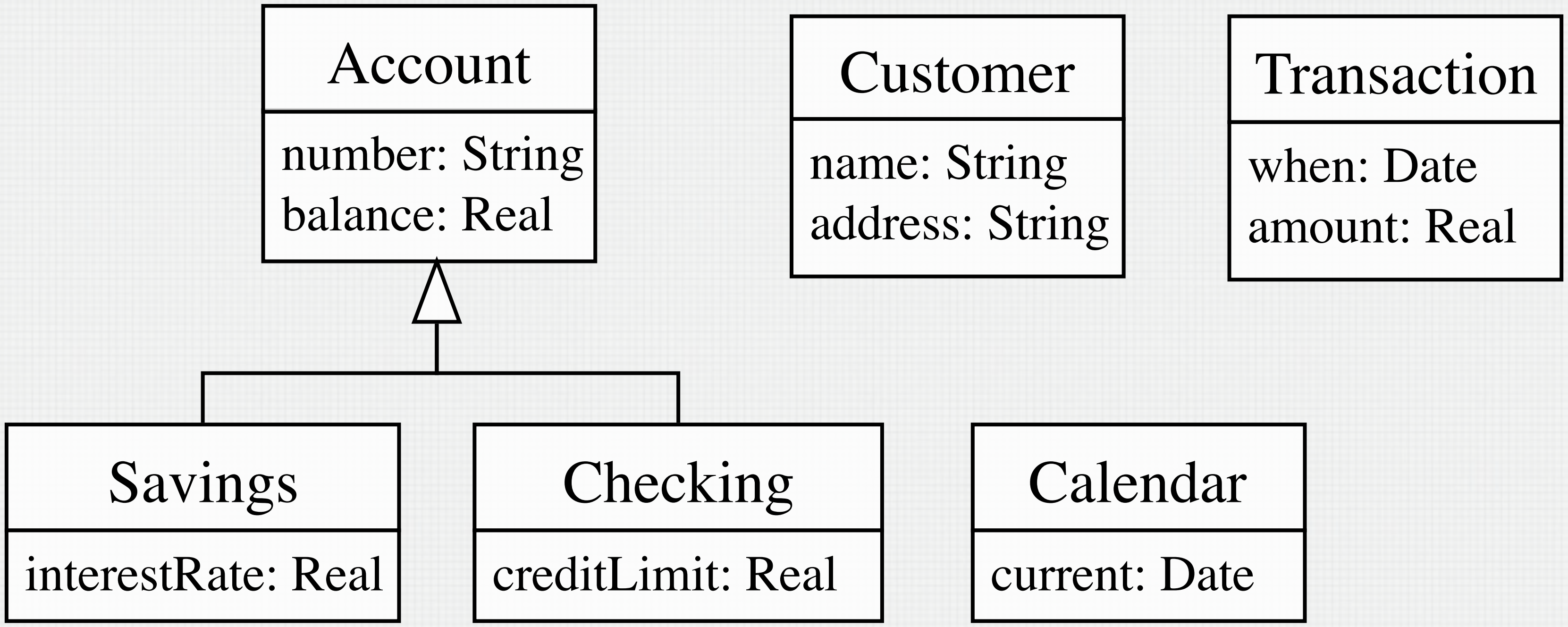


# Bank Question (2)



**context** t: Transaction:
**inv**: t.to→**size**() = 1 **or** t.from→**size**() = 1

## Bank Question (3)



## Bank Question (4)

- Retrieve Holdings
  - Let’s consider the system operation which retrieves the holdings of a customer’s accounts from the Bank system, i.e. the sum of all the balances of her/his accounts. The main effect of the operation is to notify the amount of the holdings to the teller of the bank.
  - Provide an operation schema for this system operation.
  - Develop a communication diagram for the operation.
- Monthly Verification
  - When monthly verification is performed, the owners of all accounts having a negative balance are warned.
  - Provide an operation schema for the monthly verification operation.
  - Develop a communication diagram for the operation

## Transfer Operation Schema (1)

**Operation**: Bank::transfer (source: Account, dest: Account, amount: Money);

**Description**: Moves an amount of money from one account to another one. Money is transferred only if the debited account is not overrun: a savings account must have sufficient funds, and for a checking account the credit limit must not be exceeded;

**Scope**: Account; Transaction; Calendar; Customer;

**Messages**: Teller::{Receipt; Overrun_e;}; Client::{Receipt;};

**New**: trans: Transaction;

**Aliases**: sourceHasSufficientFunds: Boolean =
(source.**oclIsTypeOf**(Savings) **and** source.balance ≥ amount)
**or** (source.**oclIsTypeOf**(Checking) **and**
source.balance + source.creditLimit ≥ amount);

## Transfer Operation Schema (2)

```
Post:  if sourceHasSufficientFunds then
           source.balance = source.balance@pre - amount &
           dest.balance = dest.balance@pre + amount &
           trans.oclIsNew() &
           trans.date = self.calendar.current &
           trans.amount = amount &
           trans.from = source & trans.to = dest &
           sender^receipt (source, Direction:: debit, amount) &
           dest.customer.client^receipt (dest, Direction::credit, amount)
       else
           sender^overrun_e (source, amount)
       endif;
```

## GenerateMonthly Operation Schema

**type** Direction **is enum** {debit, credit};
**type** Transaction **is TupleType**
{acc: AccountNumber, trAmount: Money,
trTimestamp: Date, trDir: Direction, balance: Money};
message (type) declaration
MonthlyStatement(contents: Sequence(Transaction));

**Operation**: Bank :: generateMonthly();
**Description**: At the end of the month, each customer receives a monthly statement that all the transactions and the balances of all his accounts.
**Scope**: Account; Customer; Transaction;
**Messages**: Customer :: {MonthlyStatement};
**Post**: **self**.customer→**forAll**(c |
**let** trList: Set(Transaction) **in**
c.myAccounts→**forAll** (a | a.credit→**forAll**(t | trList→**includes**
(**Tuple** {acc = a.number, trTimeStamp = t.when, trDir = Direction::credit, balance = a.balance})) **and** a.debit→**forAll**(t | trList→**includes**
(**Tuple** {acc = a.number, trTimeStamp = t.when, trDir = Direction::debit, balance = a.balance}))) **and**
**sender^**monthlyStatement(trList→**sortedBy**(trTimestamp))
**endlet;)**

# Bank Design Question

1. Design a collaboration diagram or sequence diagram for the *Transfer* operation. Don't forget that all transactions have to be recorded somehow, since at the end of each month statements are sent by postal mail to all clients showing the transactions performed for their accounts during the last period.
2. Propose a collaboration diagram for *GenerateMonthlyReports*. We propose you to implement the communication with the client by sending him/her a letter by postal mail. Make sure that each statement is complete:
   - a. It can be sent by postal mail.
   - b. It can be understood by the client: for each transaction, the statement provides its date, the involved account, the amount and the direction of the movement (credit or debit).