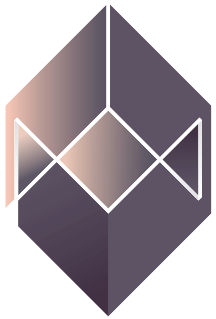
The rCOS Method of Component-Based Model Driven Design

Volker Stolz, Zhiming Liu



HØGSKOLEN I BERGEN
BERGEN UNIVERSITY COLLEGE

Course Plan

1. Introduction
2. UML-Based Model Driven Development
   (a) Use-case driven requirements modelling and analysis
   (b) Design by OO design patterns
   (c) Code generation
3. rCOS
   (a) rCOS Model of Interfaces and Components
   (b) rCOS Model of Object-Oriented Design
   (c) Use-case driven design process in rCOS
   (d) CoCoME case study

INTRODUCTION

1. Aims of the course
2. The state of the art
3. An overview of rCOS

Aims

Deal with two challenges in software engineering

1. inherent complexity of Software Projects
   - multiple aspects: structural, functionality, interaction, security, timing, distribution, mobility and general QoS
   - most aspect are interrelated and assurance consistency is hard
2. ensure correctness of software systems
   - formal modelling, design, verification and validation

State of Practical SE

Deal with complexity with *component-based and model driven* development through

- UML-like multi-view modelling of different aspects
- Separation of design and validation of different concerns by design patterns, object-oriented and component-based designs techniques

No rigorous theories and tools for specification, verification and validation

Verification and Validation

- Requirements specification are a formal version of requirements specification.
- Establishment of a property implies that this property holds as long as the assumptions hold.
- Assumptions are specifications or constraints on the behavior of environment and system elements.

Informal Definitions of V&V Methods

- Testing: determines the correctness of the execution of a program for a given initial condition and input set.
- Static Analysis: Determines program properties such as data-flow paths and control flow paths that can be deduced from the static structure of the program.
- Model Checking: determines the correctness of a temporal property for the executions of a program for all initial conditions and inputs.
- Formal Proof: determines whether a program conforms to a specification of behavior, such as an input/output relation for all executions of the program.
- Runtime Monitoring: dynamically checks whether the execution of a program conforms to a specified condition.

State of Formal Methods

Correctness of different concerns with formal notations and tools

- event-based modelling of interaction protocols and verification by model checking
- pre-post conditions specification of functionality and formal proofs by theorem proving, runtime checking and testing
- state-based modelling of dynamic and timing properties and verification by model checking
- *refinement calculi* for model transformation

- each is researched by a separate research community
- research in verification ignores the impact of design methods on verification
- Refinement calculi do not scale up

Objectives of rCOS

1. Incorporating formal methods and tools of modelling, design and VV into model-driven development processes and environments
   - model different views and analyse correctness of different concerned with different VV techniques and tools
   - automate verified design patterns and strategies to reduce the burden on (automated) verification
2. Provide a semantic foundation for relating the methods and the integrating of tools of VV with those of design

Putting theories, methods and tools consistently together in the design processes

Strands of the Research on rCOS

1. Theory: a modelling language, its semantics, refinement calculus, analysis and verification of models
2. Tool support: an integrated tool suite to support model construction, model transformation and model verification
3. Applications: develop a set of verified case studies, trading system, e-government, etc
4. Knowledge and technology transfer: teach a coherent and comprehensive methodology that begins with design for verification and validation and integrates verification into development process

UML-BASED MODEL DRIVEN DESIGN

1. Software Development Process
2. Use Case Driven Requirement Analysis
3. OO Design
4. Embedded System Design for Integration

CoCoME Case Study

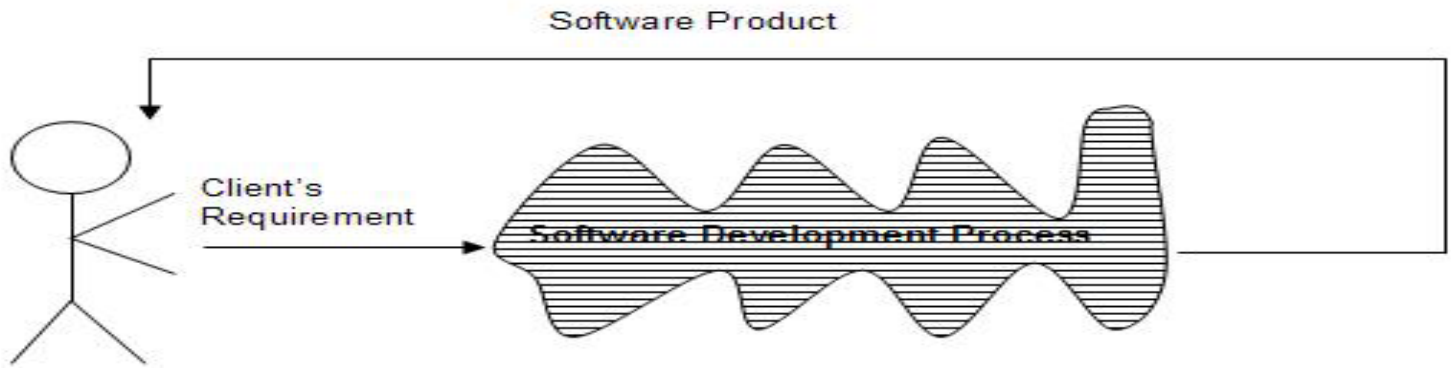
Software Development Processes

Topics

- Software development process in general
- Waterfall model
- Component-based development process

Software Development Process

- A process defines who is doing what, when and how to reach a certain goal
- An engineering project is about how to produce a product in a disciplined process
- A process to produce a software product is called a SDP
- SE is about producing software products in a disciplined process



Client's Requirements

- define the goal of the process
- prepared by the client and set out the services that the system is expected to provide
- include functional requirements and non-functional constraints

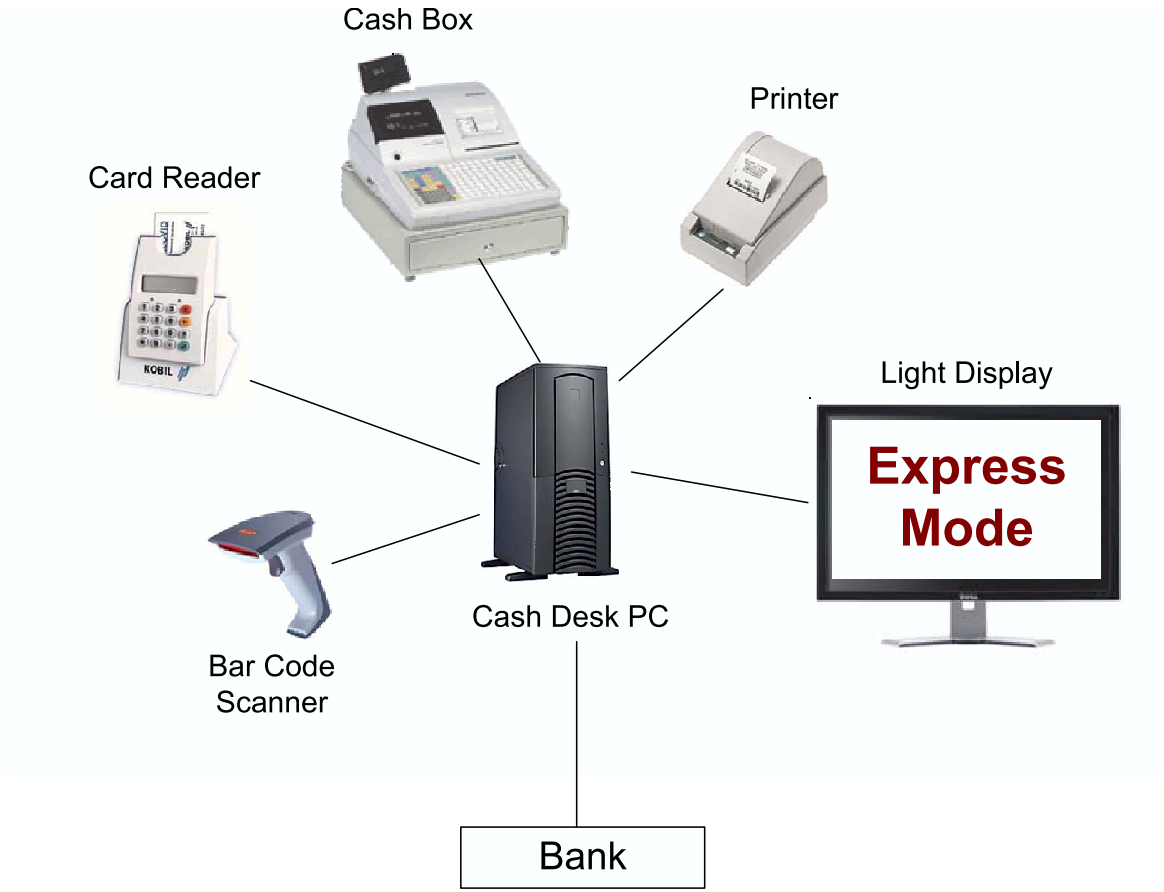
CoCoME: create a trading systems to be used in an enterprise of supermarkets.

- Goal: increased checkout automation, to support faster, better and cheaper services and business processes
  - quick checkout for the customer
  - fast and accurate sales analysis
  - automatic inventory control
  - ...

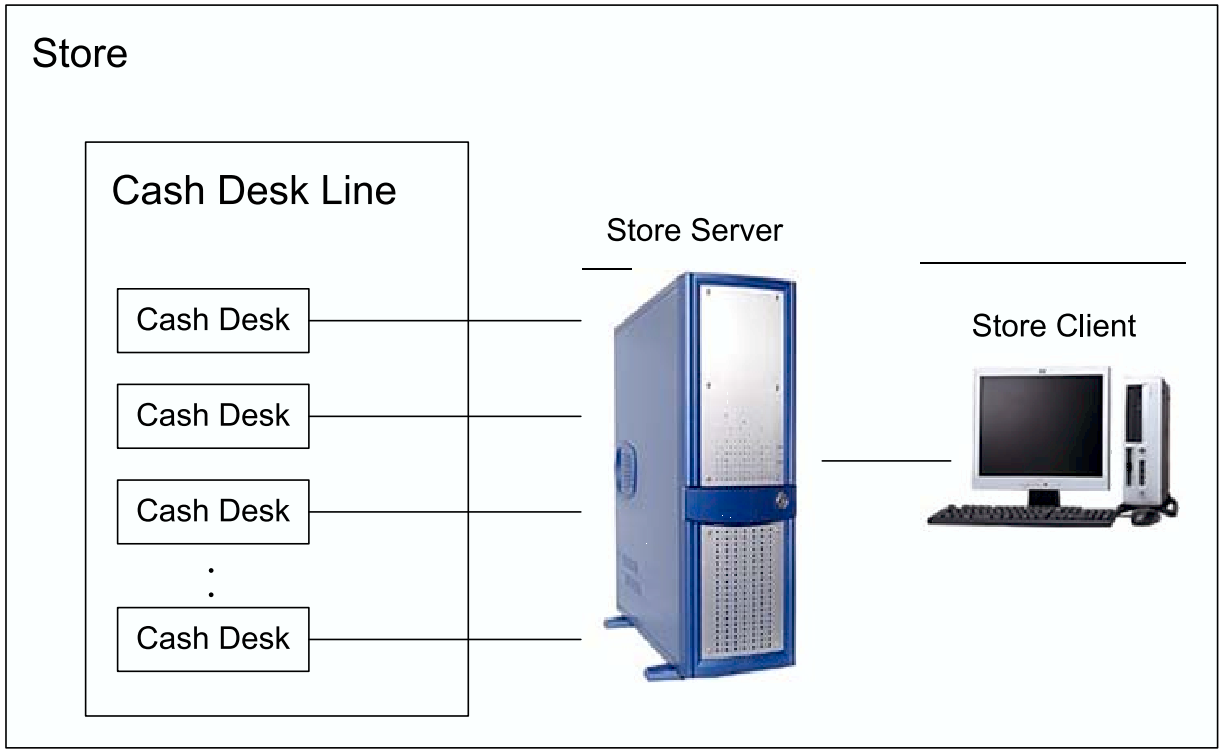
Requirements of CoCoME

- about 20 pages long
- description of hierarchy of components
  - cash desk: cash box, bar code scanner, card reader, printer, lights, cash desk PC
  - store: a line of cash desks, store server
  - enterprise: a network of stores, enterprise server
- interactions with the bank and supplier
- functional requirements: 8 use cases: process sale ...
- extra functionality: QoS

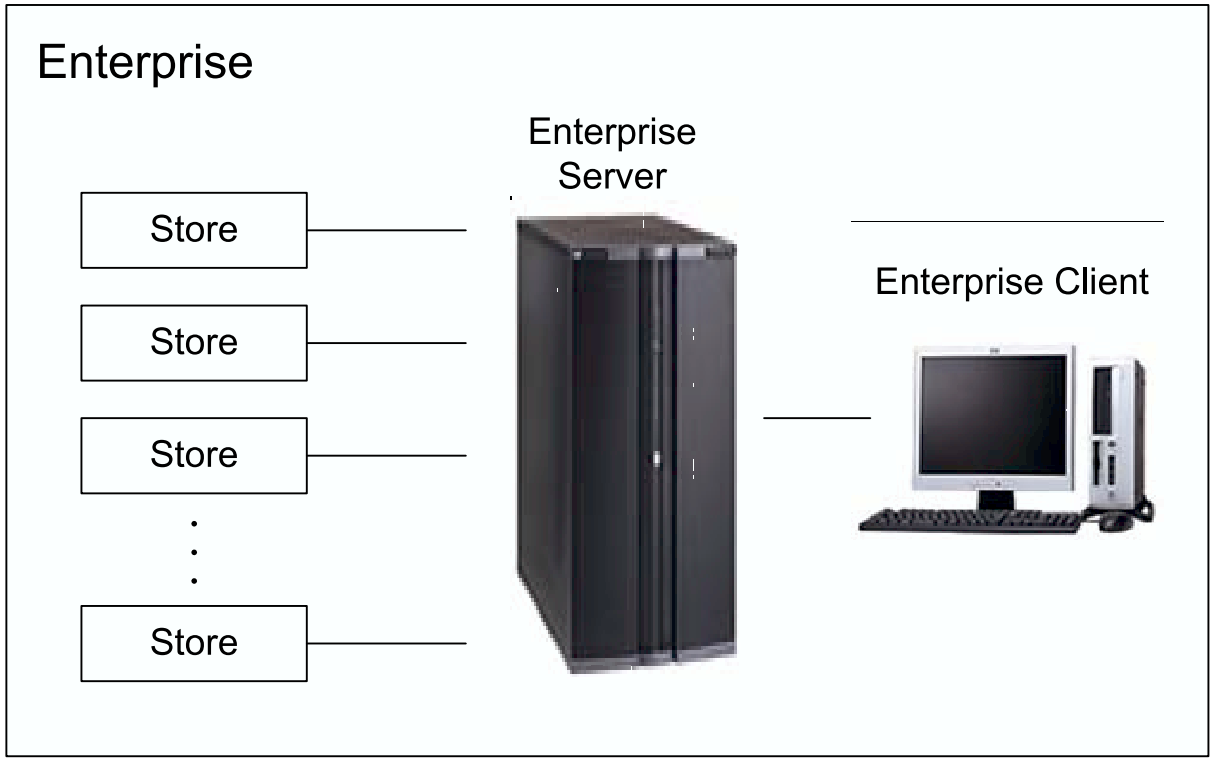
Cash Desk


Cash Box
Printer
Card Reader
Light Display
Express Mode
Cash Desk PC
Bar Code Scanner
Bank

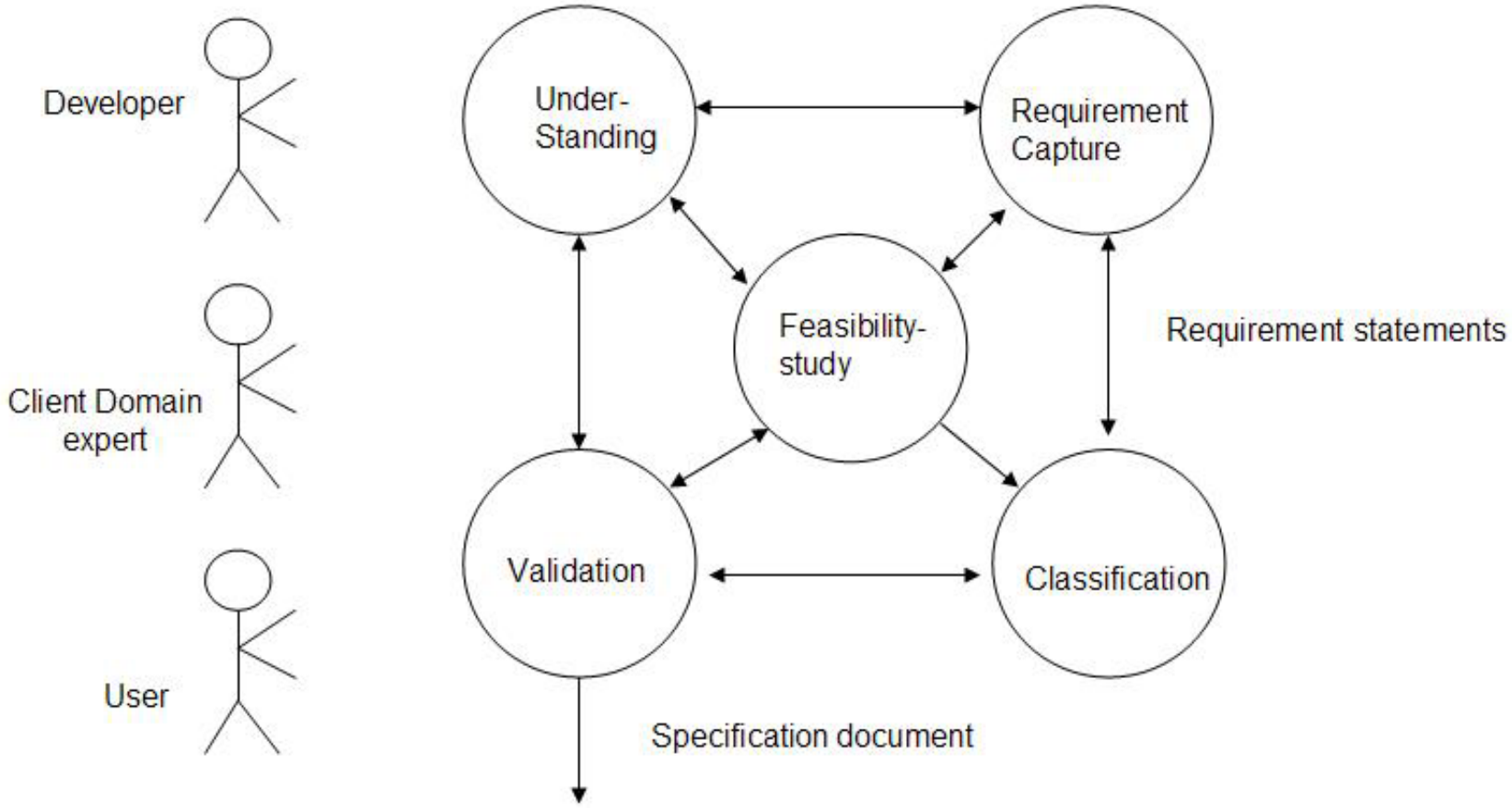
Store


Store
Cash Desk Line
Cash Desk
Cash Desk
Cash Desk
Cash Desk
Store Server
Store Client

Enterprise


Enterprise
Enterprise Server
Store
Store
Store
Store
Enterprise Client

RA Activities and Artifacts

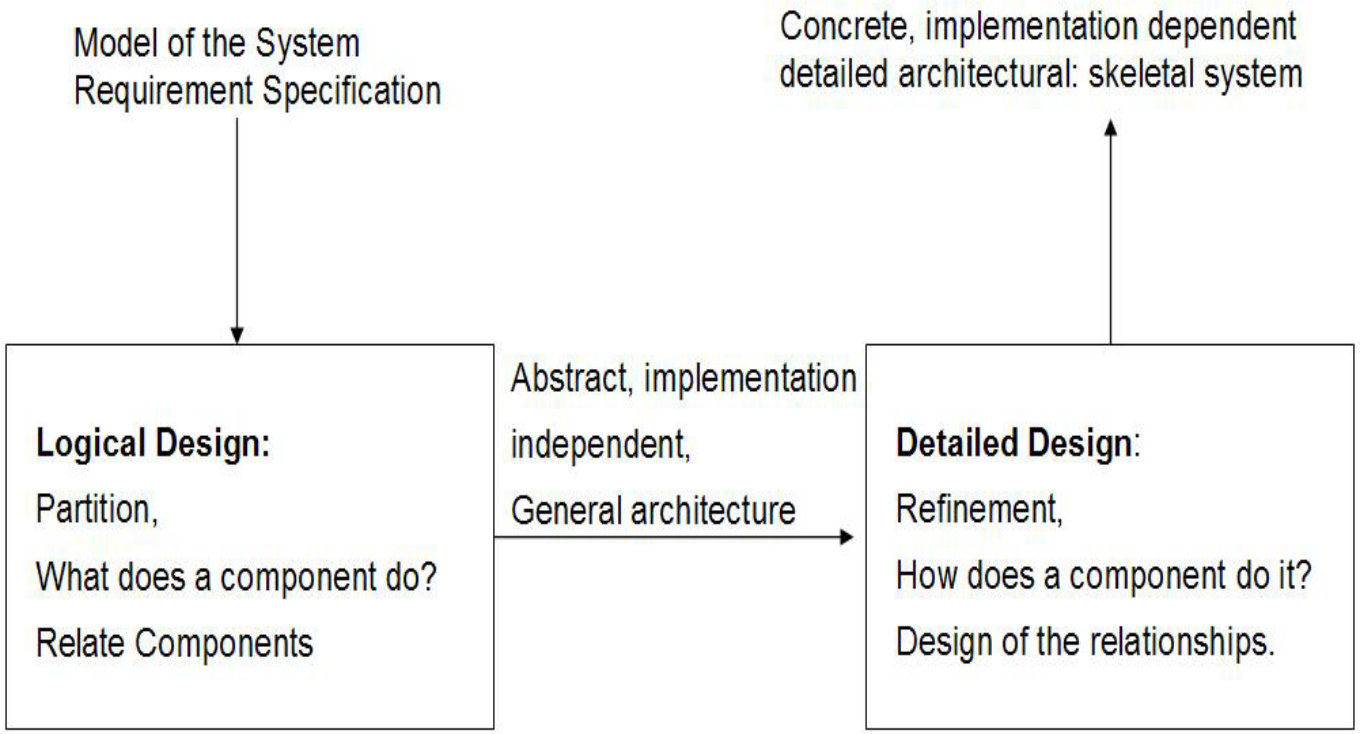


Design

The requirements specification undergoes two consecutive design processes

1. architectural (or logical) design,
2. detailed design

Logical Design



Detailed Design

Refines the components specifications to produce a detailed design model which describes

- the design of the functions of each component;
  how the component does the function
- the design of the interfaces for the components:
  how components interact with each other

This model can be seen as a skeletal system ready for implementation.

Coding and Testing

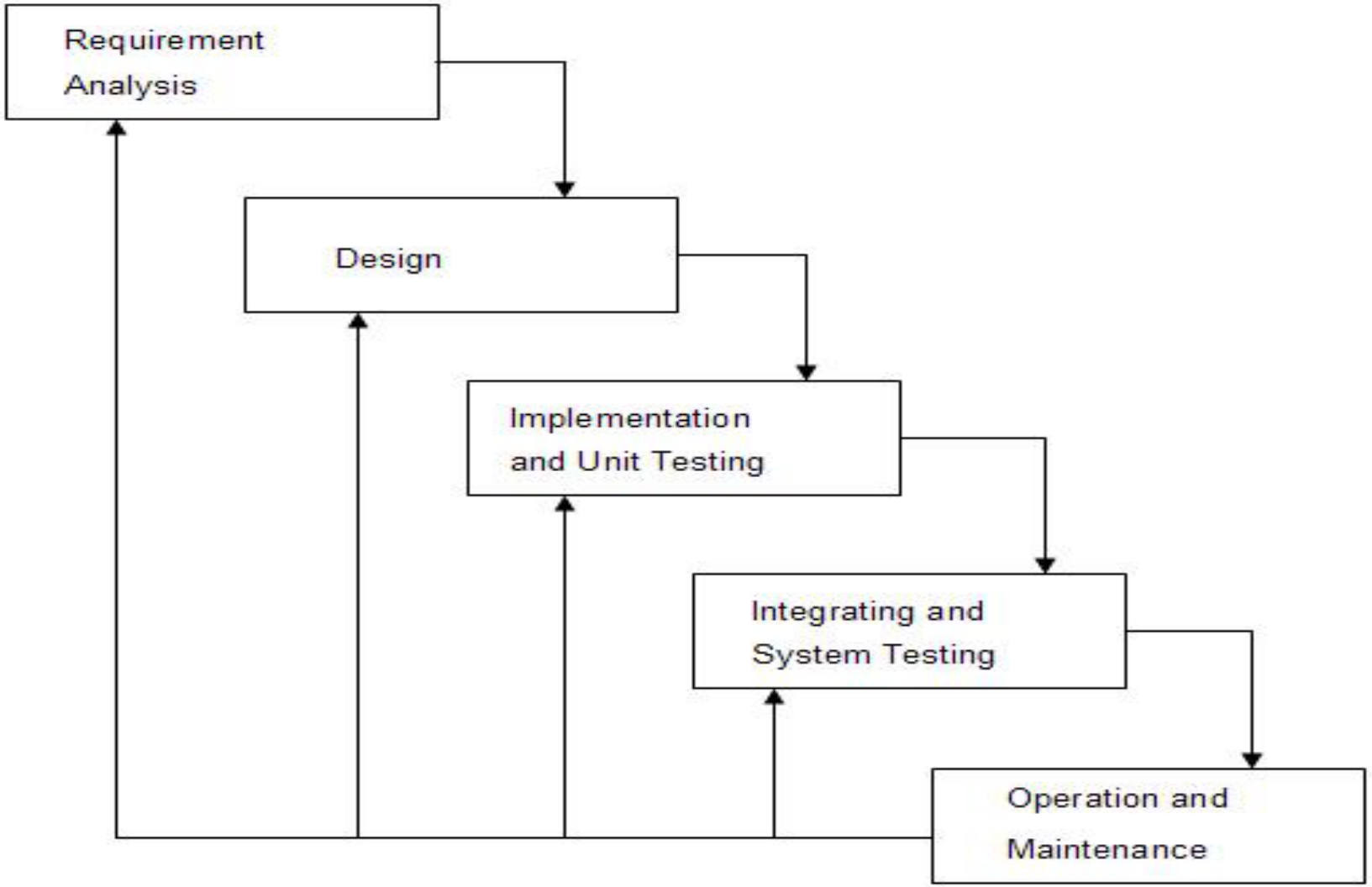
- each of the components from the design is realized as a program unit
- each unit then must be either *verified* or *tested* against its specification obtained in the design stage

- Test plan must be made according to the decomposition in the logical design specification
- Test cases must be designed to cover functions of all components

Integration and System Testing

- individual program units of the components of the system are combined
- system is tested as a whole
- system is tested by the client (acceptance testing)
- This phase ends when the product is accepted by the client.
- Test plan must be made according to the requirement specification
- Test cases should be designed to cover all the crucial services required

The Waterfall Model



Model Driven Development

- each phase of RA, Design and Implementation is based on construction of models that capture the main decisions
- rigorous MDD is based on construction of models that are verifiable – not just stylised that can be used to produce nice looking graphs and charts

Requirements Modelling and Analysis

Modelling the real application world

- business processes $\longrightarrow$ use cases
- real world concepts $\longrightarrow$ classes
- real world objects $\longrightarrow$ objects
- real-world structure $\longrightarrow$ component interactions and class diagrams

Software development = transforming models of real world to models of digital world

Use Case Driven RA

Objectives

- understanding the domain to capture requirements
- create and represent functional and nonfunctional specification

Topics

- use cases for describing domain processes
- use-case diagrams and use case sequence diagrams

Main Artifacts: Requirements model (specification)

- description of the use cases (useful but not rigorous)
- use-case diagrams and use case sequence diagrams (more precise, but not formal)

Use Cases

Informally speaking

- a use case is a story or a case of using a system by some users to carry out a business process (or a task)

A bit more precisely speaking,

- a use case describes the possible sequences of events of some types of users using some part of the system functionality to complete a process
- such a type of users is called an actor

Remarks on Use cases

Use cases describe

- the functional requirements of the system from the actors perspective
- what do the actors do to use the system for realising an application task
- each use case only uses part of the functionalities of the system – component
- actors form the external environment of the system

Actors

When performing use cases

- communicate with the system by sending messages to and receiving messages from the system
- communicate with each other to exchange information

To capture use cases

- identify application processes
- identify actors involved in the use cases
- identify the *communication actions* that actors take when perform a use case

Focus on the actors that directly communicate with the system – direct actors

Example

When perform **process a sale** with the trade system

- two actors must be involved: **Customer** and **Cashier**,
- the following sequence of actions must be performed:
  1. The Customer arrives at a **cash desk** with **items** to purchase.
  2. The **Cashier** records the purchase **items** and collects **payment**.
  3. On completion, the Customer leaves with the items.

Each use case is a complete course of events in the system, seen from a user's perspective

Incremental Analysis of Use Cases

- first write a high level use case description to obtain initial understanding of the overall process
- A high-level use case describe a process very briefly with actors and abstract actions that the actors perform

A high level use case can be presented in a structured format

Use case:	**Name of use case** (use a phrase starting with a verb)
Actors:	List of actors, indicating who initiates the use case.
Purpose:	Intention of the use case
Overview:	A brief description of the process
Cross References:	Related use cases

Example

Use case:	**Process Sale with Cash Payment**
Actors:	Customer (initiator), **Cashier**
Purpose:	Capture a **sale** and its **cash payment**
Overview:	A **customer** arrives at a CashDesk with **items** to purchase. The **cashier** records the purchase **items** and collects a **cash payment**. On completion, the Customer leaves with the items.
Cross References:	

Extended Use Cases

Taking an high level use case and analyse it in details to write an extended use case

- what input data does an actor provide to the system in performing an action
- what output data does an actor receives after an action
- what does the system do when an actor performs an action: **what data to be updated, created, to be checked or found**
- what are the **main course of actions** and when exceptions may occur and what to do to handle them
- the precondition for the use case to start, and the invariant property to be preserved by the actions

Information in Extended Use Cases

- precondition: conditions on under which the use case start
- typical course of events: describes the general pattern of interactions carried out
- alternative courses of events: describes the patterns of interactions carried out under exceptional conditions.
- invariant: conditions to be preserved by actions – safety

An interaction between an actor and the system shows

- what an actor asks the system to do and with what input
- what should be done by the system to respond: what should be updated, checked, or output.

identification and definition of functions, data, classes/objects and their relations

Structured Representation of Use Cases

High Level Description;

Precondition: // mostly ignored in informal analysis

Invariant: //mostly ignored in informal analysis

Typical Course of Events

Actor Action	**System Response**
Actions of the actors	Responses of the system

Alternative Courses

- Alternatives that may arise at *line_number*. Description of exception.

Process Sale With Cash Payment

Typical Course of Events

	Actor Action	System Response
1.	This use case begins when a Customer arrives at a Cash Desk with items to purchase.	
2.	The Cashier records the identifier from each item.	
	If there is more than one same item, the Cashier can enter the quantity as well.	
3.		Determines the item price and adds the item information to the running sale's transaction.
		The description and price of the current item are presented.

4. On completion of the item entry, the Cashier indicates to the Cash Desk that item entry is completed.
5 Calculates and presents the sale total.
6. Cashier tells the Customer the total.
7. The Customer gives a cash payment, possibly greater than the sale total.
8. The Cashier records the cash received amount.
9. Shows the balance due back to the Customer. Generate a receipt.

10. Cashier deposits the cash received and extracts the balance owing.

    Cashier gives the balance owing and the printed receipt to the Customer.

12. Customer leaves with the items purchased.

11. Logs the completed sale.

Alternative Courses

- Line 2: Invalid identifier entered. Indicate error.
- Line 7: Customer didn't have enough cash. Cancel sales transaction.

precondition, invariant?

Assumptions

Make clear the assumptions made when analysis an use case

- Cash payments only, no inventory maintenance
- No tax calculations, no coupons
- Cashier does not have to log in; no access control
- No record maintained of a customer and her/his buying habits
- No control of the cash drawer.
- Name and address of store and date and time of sale are not shown on the receipt; Cashier ID and CashDesk ID are not shown on the receipt

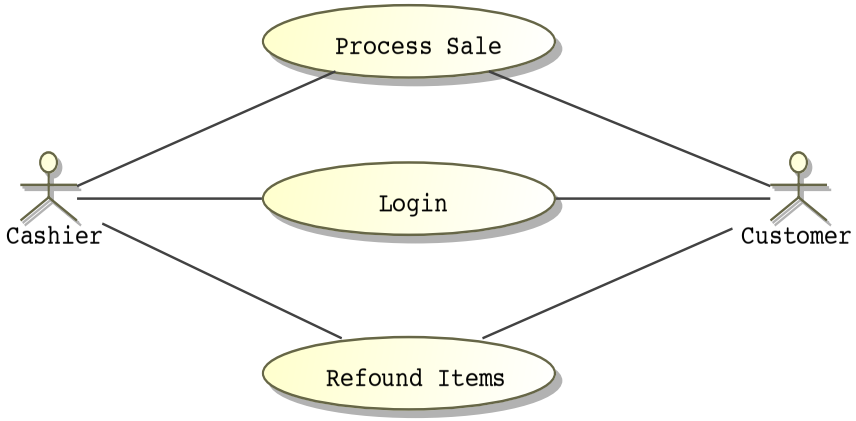
Always begin the development with building a simple system and then let it evolve

Expanded Version of Withdraw Money

1. This use case begins when a Customer arrives at a bank.
2. Customer inputs the identifier.
3. Checks the identification, and shows the description of the choice of services.
4. Customer indicates to withdraw some money.
5. Shows that Customer should indicate the amount and account from which to withdraw.
6. Customer enters the account and amount.
7. Deducts the amount from the account and dispenses the money.
8. On completion, Customer leaves with the money.

Use-Case Diagrams

A use-case diagram describes part of the *requirements model*, illustrating a set of use cases, actors, and their relationships



Do not draw a use case diagram which contains only one use case; use UDs to group and relate use cases.

Decompose Use Case

Pay by Cash

Typical Course of Events

	Actor Action	System Response
1.	Customer gives a cash payment, possibly greater than the sale total.	
2.	Cashier records the cash tendered.	
3.		Show the balance due back to the Customer.
4.	Cashier deposits the cash received an extracts the balance owing. Cashier gives the balance owing and the printed receipt to the Customer.	

Alternative Courses

- Line 3. If cash the amount tendered is not enough, exception handling
- Line 4: Insufficient cash in drawer to pay balance. Ask for cash from supervisor.

Using exactly the same techniques in the creation of Pay by Cash,

we can create two use cases Pay by Credit and Pay by Cheque.

Compose Use Cases

1. This use case begins with a Customer arrives at a CashDesk checkout with items to purchase.
2. Cashier records the identifier from each item.

   If there is more than one of the same item, Cashier can enter the quantity as well.
3. Determines the item price and adds the item information to the running sales transaction.

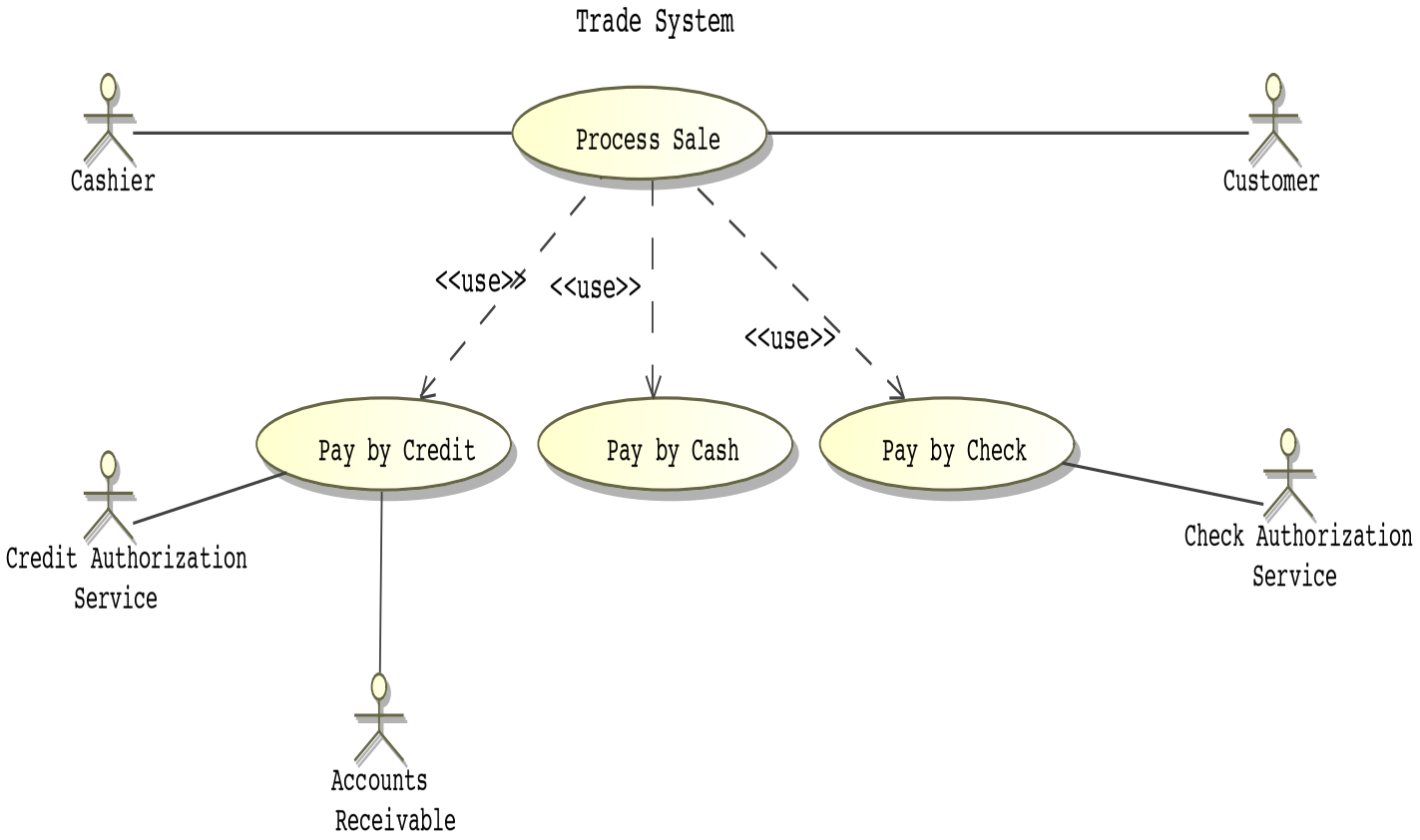
   The description and price of the current item are presented.
4. On completion of the item entry, Cashier indicates to the CashDesk that item entry is completed.
5. Calculates and presents the sale total.
6. Cashier tells Customer the total

7. Customer chooses payment method:
(a) If cash payment, **initiate** *Pay by Cash*.
(b) If credit payment, **initiate** *Pay by Credit*.
(c) If cheque payment, **initiate** *Pay by Cheque*.

8. Logs the completed sale.
9. Prints a receipt.

10. Cashier gives the printed receipt to the Customer.
11. Customer leaves with the items purchased.

Relating Use Cases



Warning

Not always right to combine arbitrary two list of interactions to make a use case

- Do not make “Borrow a Book” and “Return a Book” into one use case.
- Do not make “Add a member” and “Remove a member” into one use case
- Think whether they can be carried by an operator/user in one window and in one go.
- Think about them in terms of business processes

A use case is an end to end complete story using the system in carrying out a business process

CONCEPTUAL CLASS MODEL

Objectives

- understand the concept of classes
- identify concepts in the problem domain
- characteristics of objects
- associations between classes
- attributes of classes
- class diagrams and their use for describing a conceptual model

Output: A full class diagram that models the concepts, objects, their attributes and structures in the problem domain.

Modelling real world concepts, objects, their properties and relations

Classes Model Concepts

Identification of classes in the domain and creation a conceptual model are the most typical activities in OOA

- A conceptual model illustrates meaningful concepts in the problem domain
- Informally, a concept is an idea, thing(s), or a class of objects
- a class represents a concept

Formally, a concept is represented in terms of its symbol, intension, and extension:

- symbol: words representing a concept, used when talking about the concept
- intension: the definition of a concept
- extension: the set of instances to which the concept applies

Examples

1. symbol: Module, intention: a course offered as a component of a degree, extension: M111, M206 ...
2. symbol: Student, intention: a person who takes modules at a university, extension: John Smith, James Brown, Peter Butcher ...
3. symbol: Sale, intention: represents the event of a purchase, extension: sale1, sale2 ...

Attributes of Concepts

1. Student: name, age, student number ...
2. Module: title, code, credit ...
3. Sale: date, time, total ...

Object-Oriented Terminology

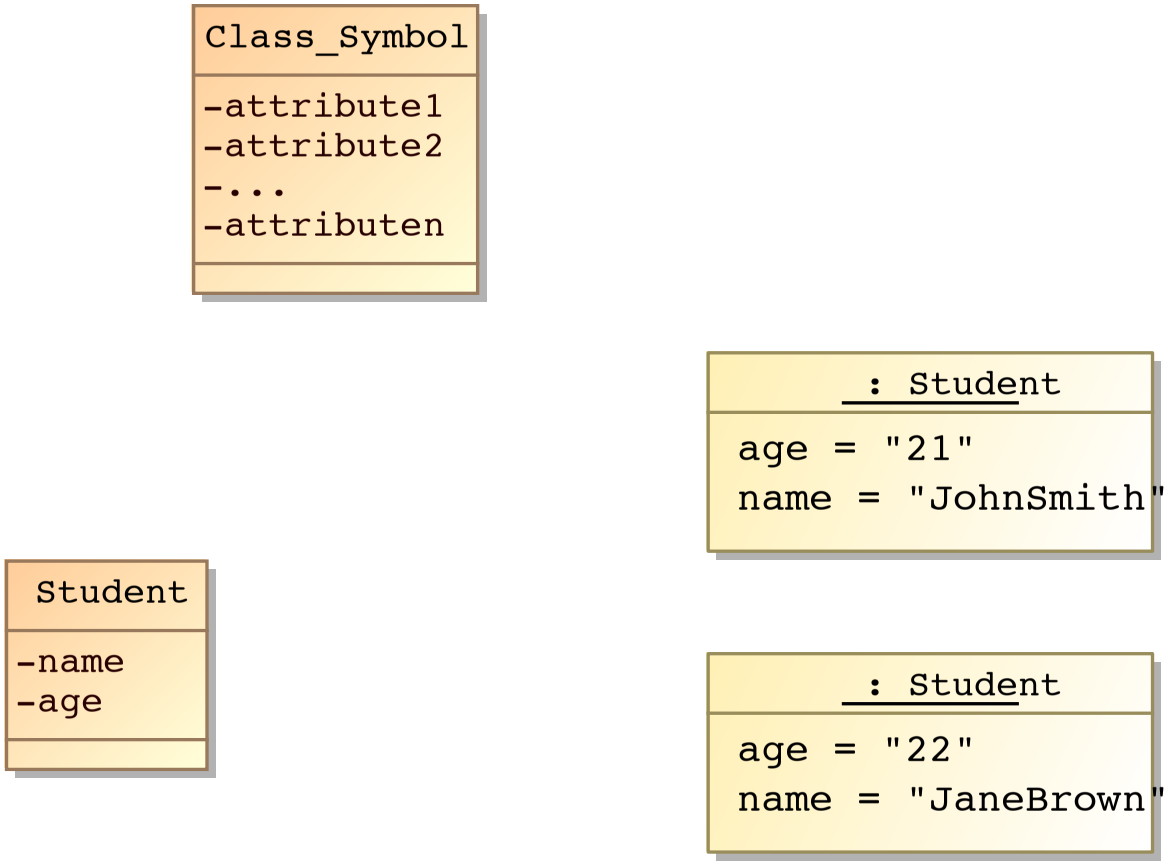
- a class represents a concept
- an object of a class is an instance of the corresponding concept
- each object has a value for each attribute
- A class defines a set of objects which have common types of properties

Examples

- an instance of Student: name = John Smith, age = 21 ..
- an instance of Module: title = rCOS, code = M206, credit = 10
- an instance of Sale: date = 20.03.2008, time = 1500, total = 103

This is not easy to read or understand

Modelling Notation in UML



Defining Features of Objects

- object's identity: every object is distinguishable from every other, even if two objects have exactly the same properties
- object's states: every object has some property at any moment of time
- object's persistence: every object has a life time (static nature of the system)
- object's behaviour: an object may act on other objects and/or be acted on by other objects (dynamic nature of the system)

An object may be in different state at different and may behave differently in different states

Characterisation of Objects and Classes

- an object is an entity with identity, state and behaviour
- a class is a description of a collection of objects, sharing structure, behavioural pattern, and attributes

Example

- a car is an entity, has an identity and a life time, it can be repaired, owned, driven by a person, it can transport other objects from one place to another
- all cars share structure, behavioural pattern, and attributes
- Car is the class defines all the car objects

Capture Classes

There are usually two strategies to identify concepts

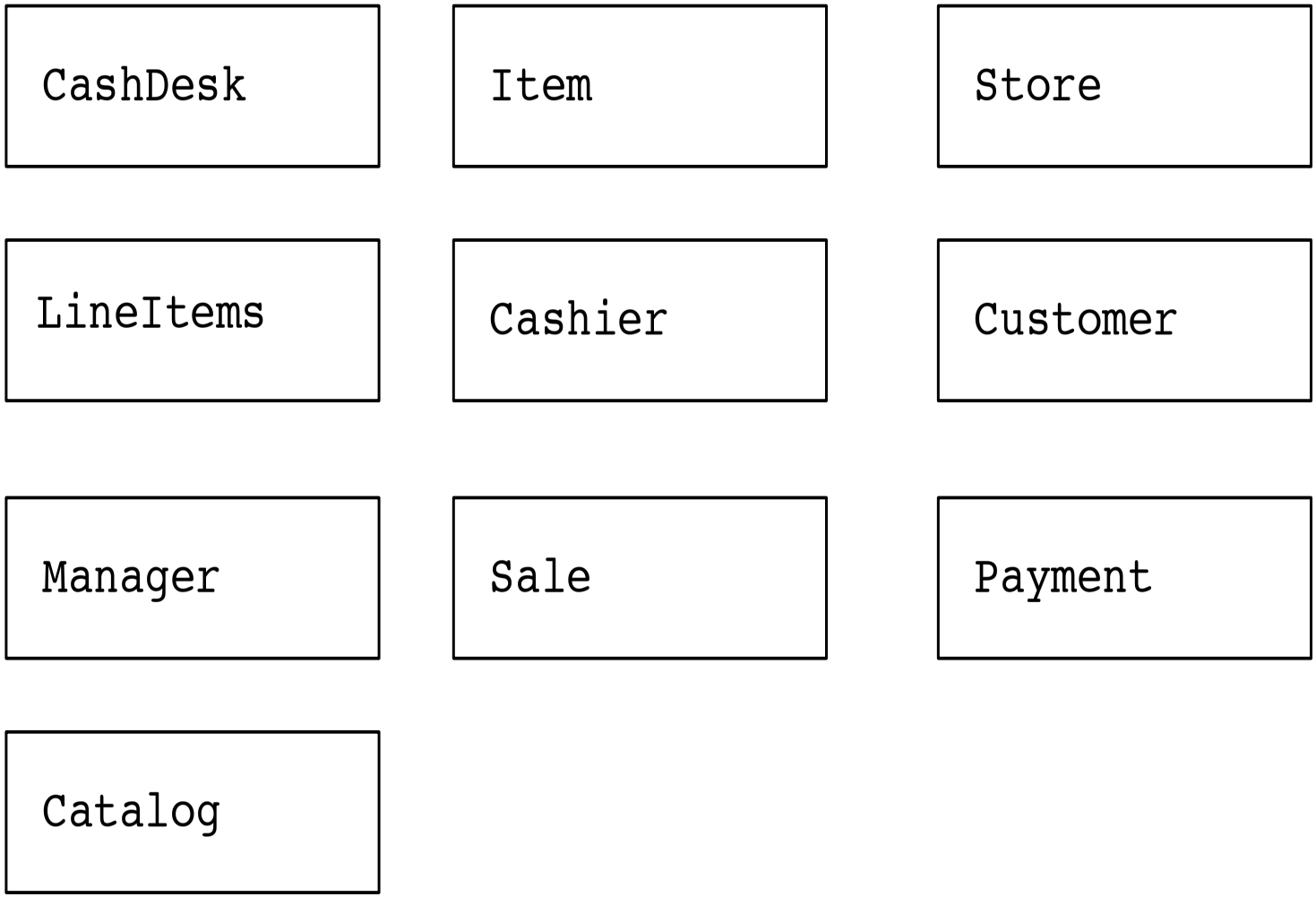
- Identify the noun phrases in the textual descriptions of a problem domain
  - the client's requirements descriptions, the initial investigation reports
  - The expanded use cases
- Find concepts with concept category lists
  - physical or tangible objects, such as cash desk, airplane, car, house, sheep, etc.
  - places such as store, office, airport, etc
  - roles that a person in an organisation, such as cashier, pilot, professor, student, etc.

Remarks

- mechanical noun-to-concept mapping is not possible,
- words in natural languages are ambiguous, and
- nouns may include concepts which are about either *attributes*, *events*, *operations* which should **not** be modelled as *classes*.

Rigorous analysis is needed, and formal specification helps a great deal

Initial CD for CoCoME



The Need-to-Know Principle

- there are many noun phrases, such as receipt and price, that are not included
- there is no indication from the use case that there is a need to store, manipulate, or use a receipt
- the defining features of objects, such as identity do not apply to price

The first is about the need-to-know policy and the second is about

features of objects and attributes

Associations

- understand the concept of “associations”
- understand the structure of the problem domain in terms of associations
- understand class structure in terms of generalization
- understand object structure in terms of aggregation and general association
- identifying associations between classes
- notation for associations

Output: Creation of a conceptual model in the form of a set of class diagrams with classes and object structures

Overview

- component with totally independent classes and objects do not have meaningful behaviour
- objects need to be related so that they can interact and work with each other to provide useful services
- the structure of a component is determined by the relationships between objects and classes
- there are two kinds of structures: structures between classes and structures between objects.

Class Structure – Generalization

- generalization structures gather the common properties and behavioural patterns of different classes into more general classes
- specializations partition a classes into subclasses which share some common properties or behavioural patterns
- a generalization structure is a relation between two or more specialized classes and one general class
- generalization: a general class (called super class) describes properties common to a group of specialized classes (subclasses)

generalisation/specialisation: *reuse, abstraction, refinement, poly-morphism*

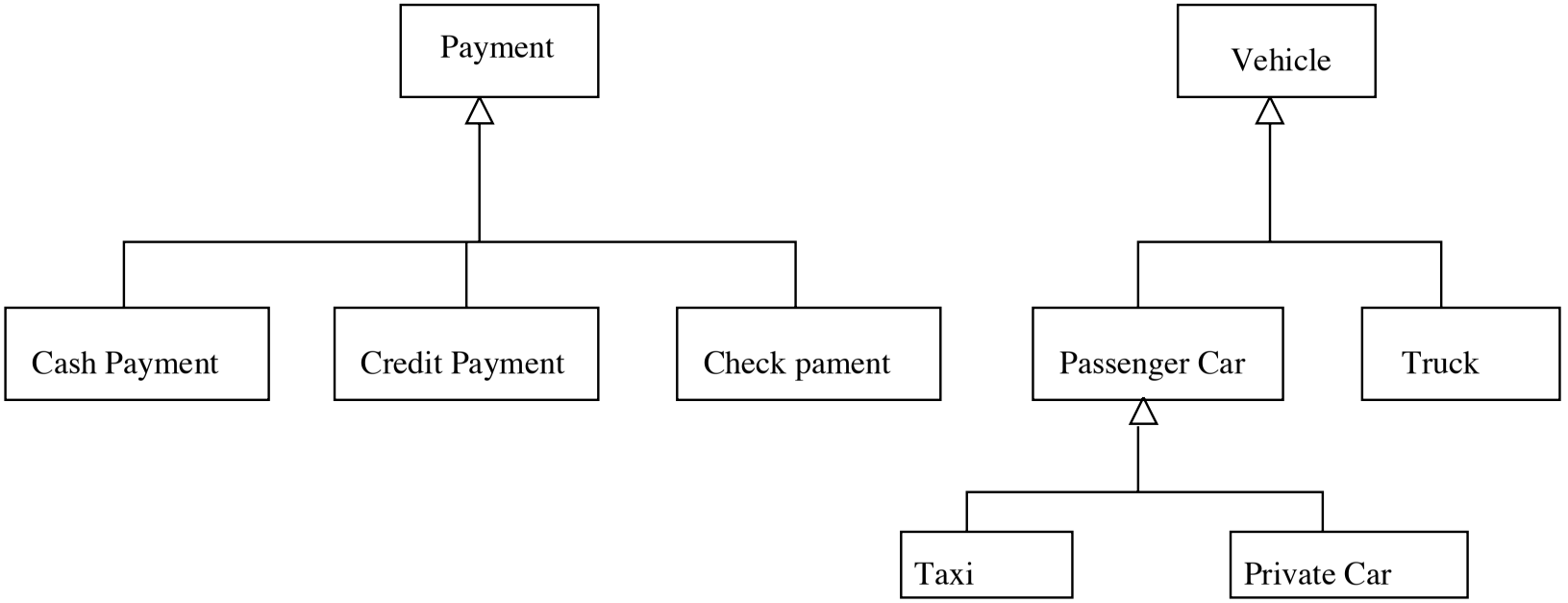
Characterisation of generalisation

A class defines a set of objects and

> *all members of a subclass are members of its super class*
>
> $$Subclass \subseteq Superclass$$

This is also termed as the Is-a-Rule for testing a correct subtype

Examples and UML Notation



Super-sub classes in a Bank System?

Associations

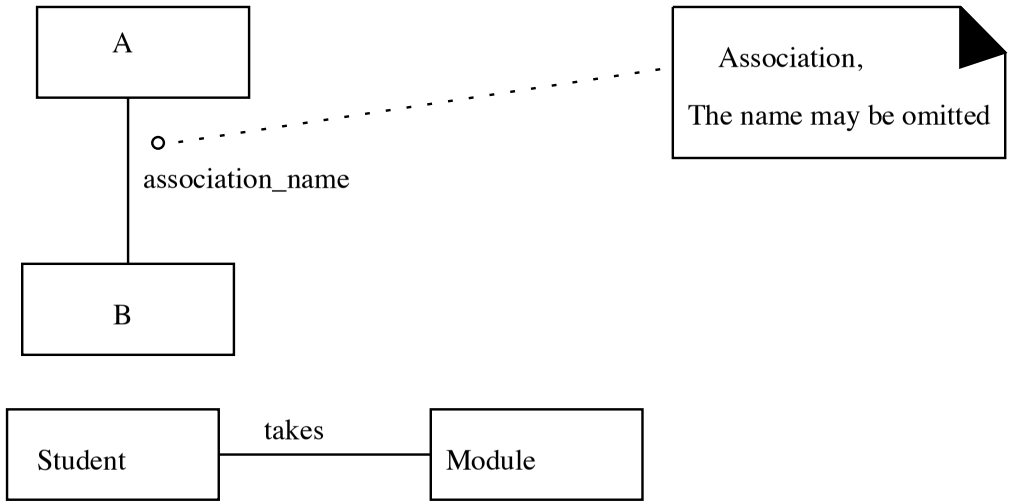
- objects must be related to each other to interact and collaborate with each other
- object interactions are to realise use cases
- two objects can interact only when they are “related” or “linked” at the time of the interaction
- an association between two classes is a type of links between some objects of the classes
- an association between two classes is a temporal relation between the objects of the two classes

Formally

- a class is a type of objects
- a class defines a state variable that takes a set (a heap) of objects
- an association R between two classes A and B defines a variable of type $A \times B$
- $a : A$ and $b : B$ are linked by R if the pair (a, b) is in R

- does R have to link every object of A with an object of B?
- does R have to link an object of A with exactly one object of B?
- can two classes be related by more than one association?
- Can there be association which link objects of the same class?

Example and UML Notation



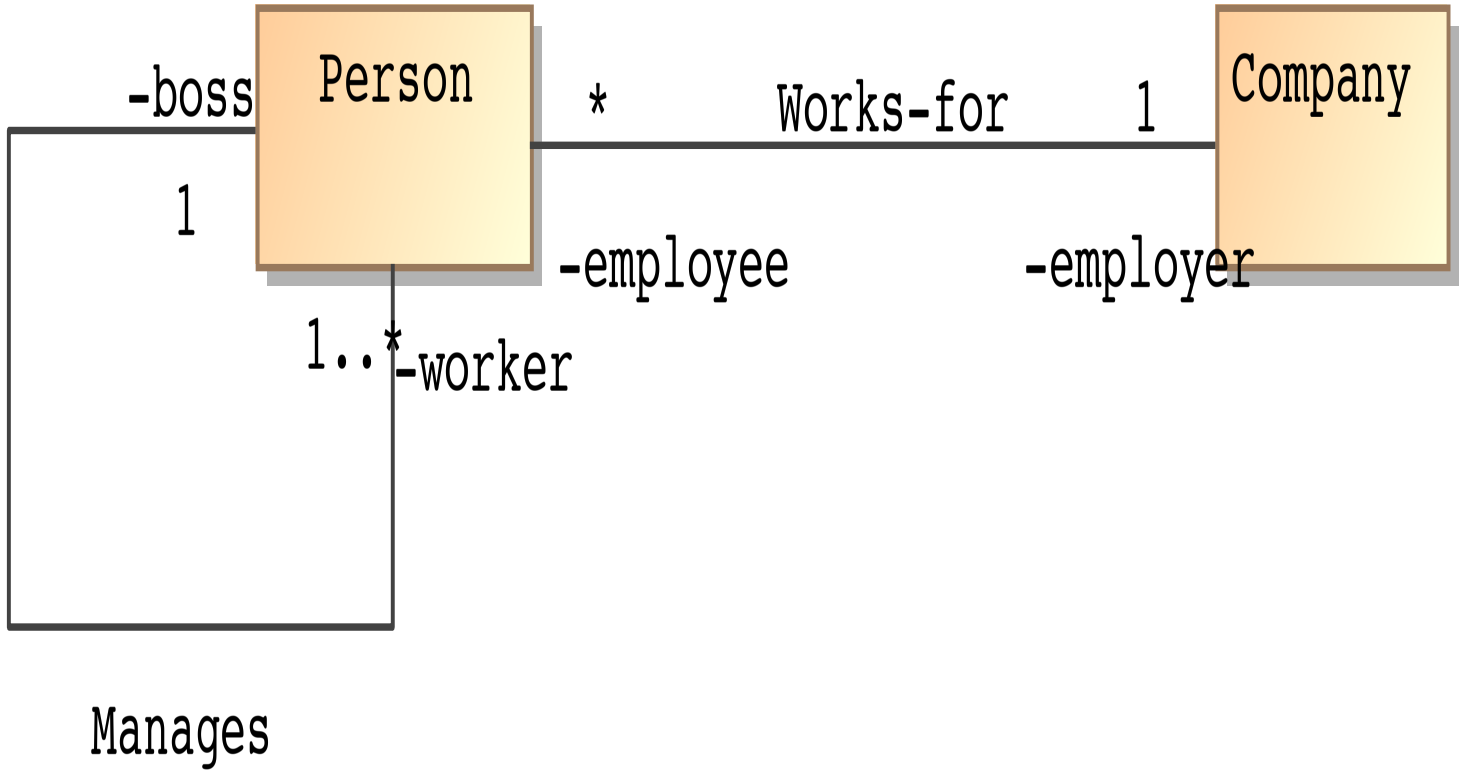
Need-to-Know Policy: a relationship that needs to be maintained, manipulated, needed for interaction

Roles of Associations

- each end of an association is called a role of the association.
- A role optionally have
  - a name
  - a multiplicity

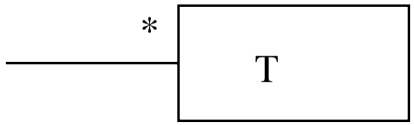
- role names are not very much significant at RA,
- but useful for an association which relates objects in the same class

Example and UML Notation

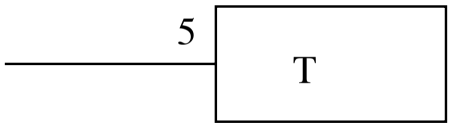


revise this model using generalization-specialization?

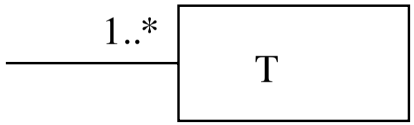
Multiplicities in UML



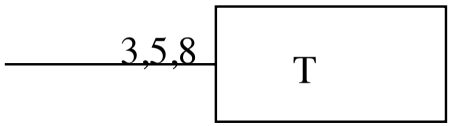
zero or more; "many''



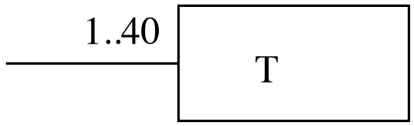
exactly five



one or more; ''at least one''

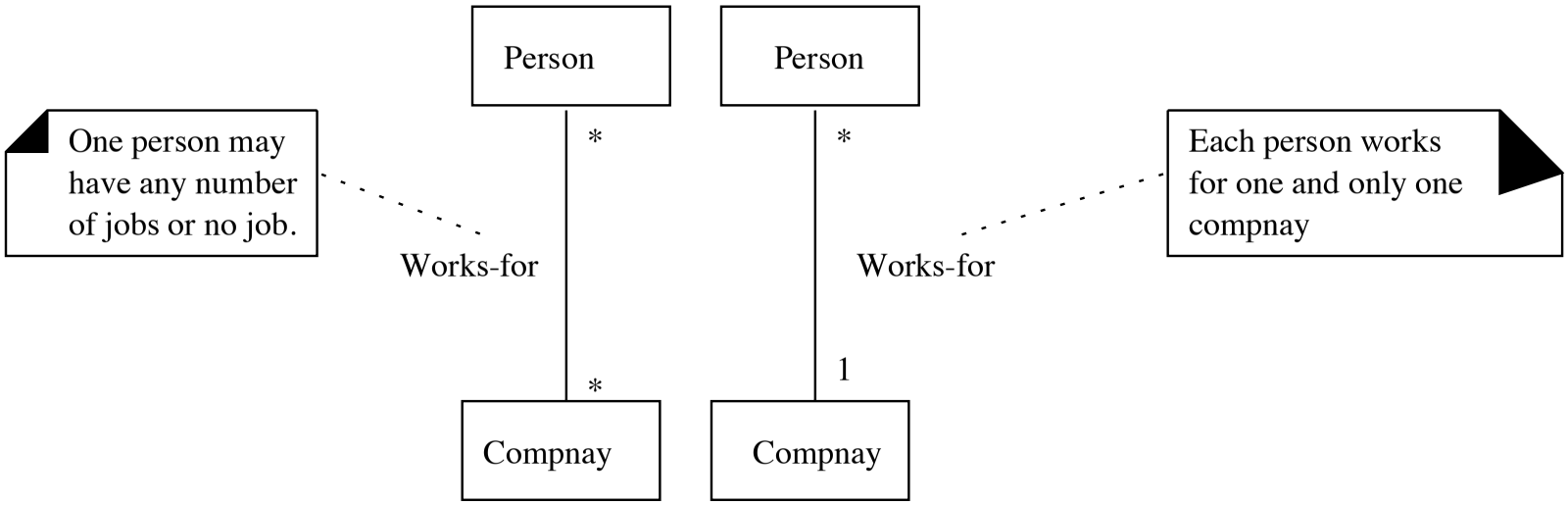


exactly three, five or eight



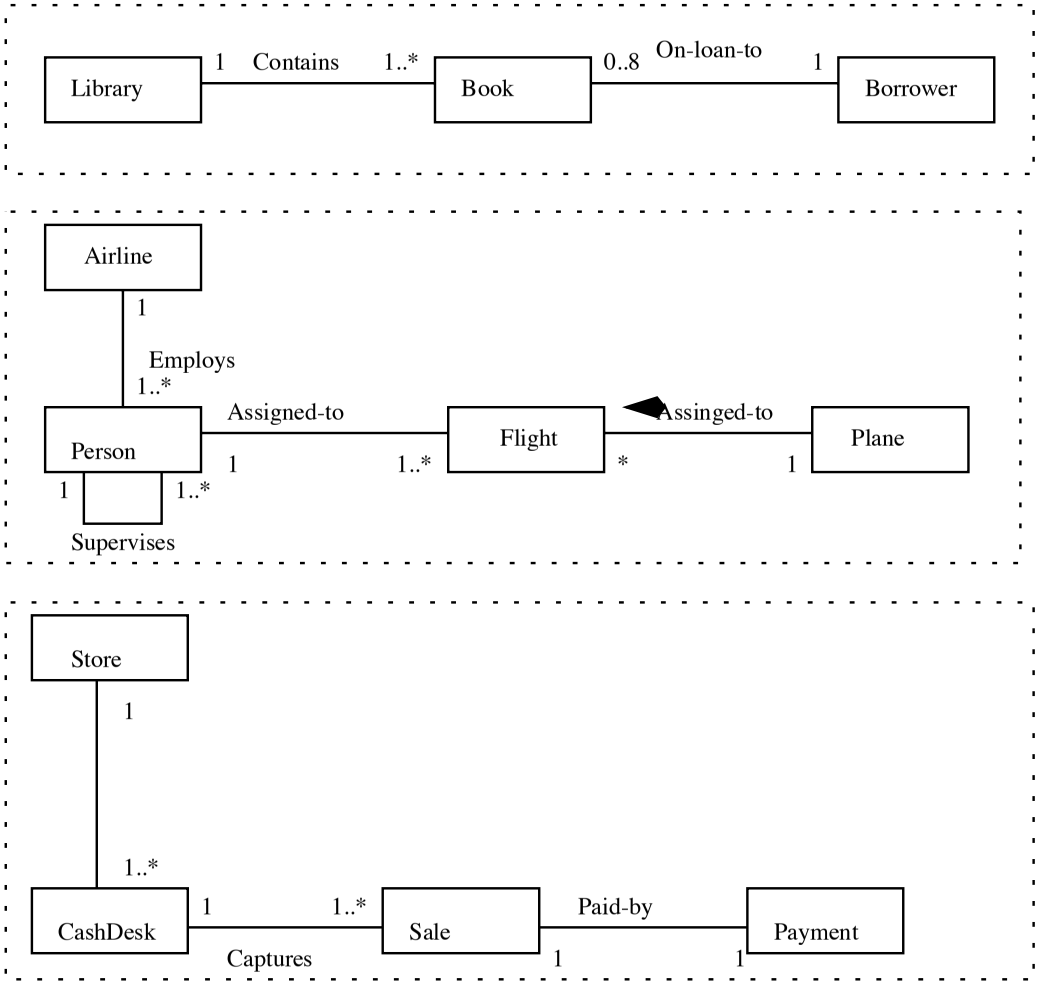
one to forty

Example



Determining multiplicity often exposes hidden assumptions

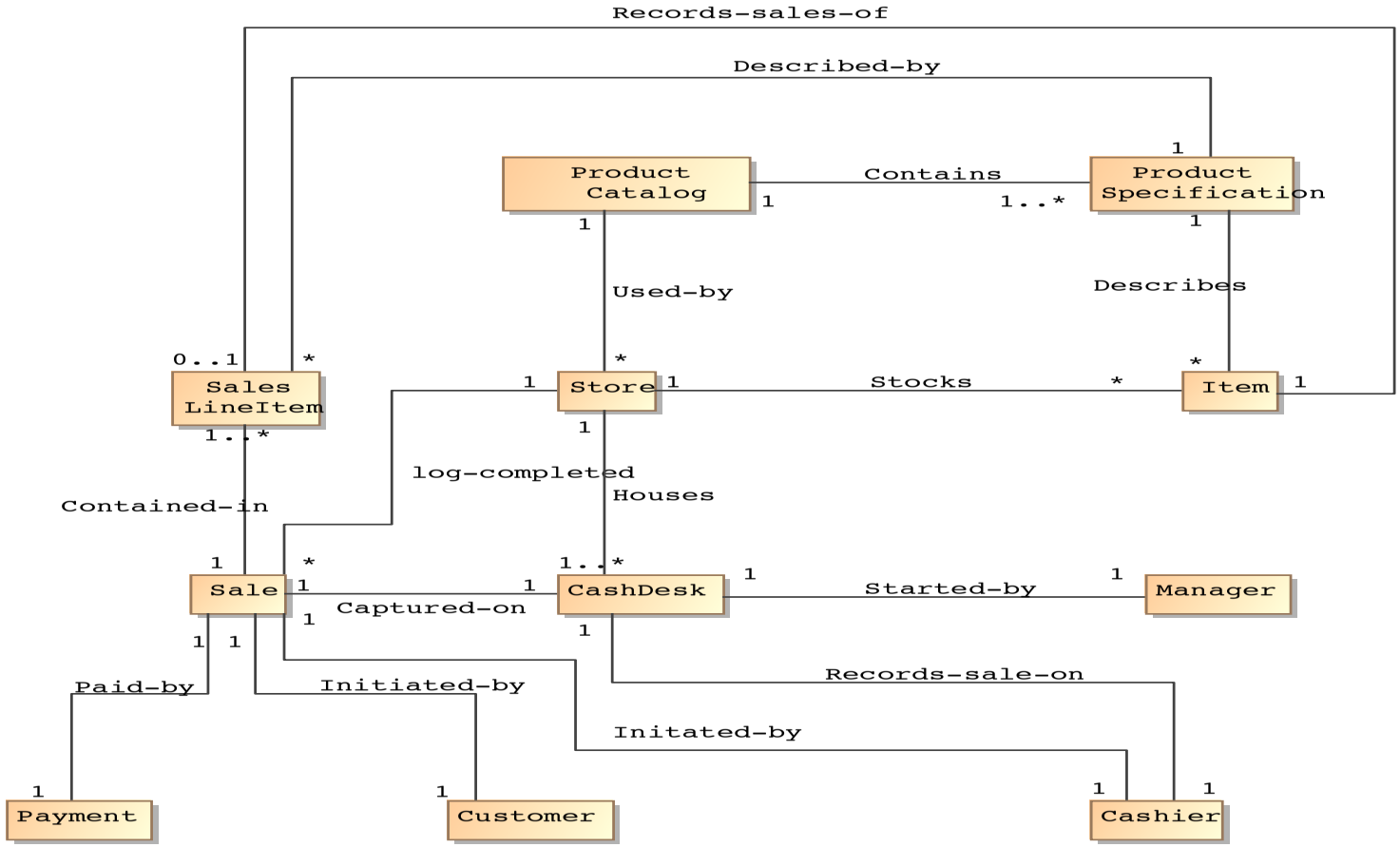
More Examples



Exercises

1. In the domain of the airline, model two relationships between a Flight and Airport, Flies-to and Flies-from:
2. Model the facts that a car is owned by one owner, and a car can be used by any number of people.

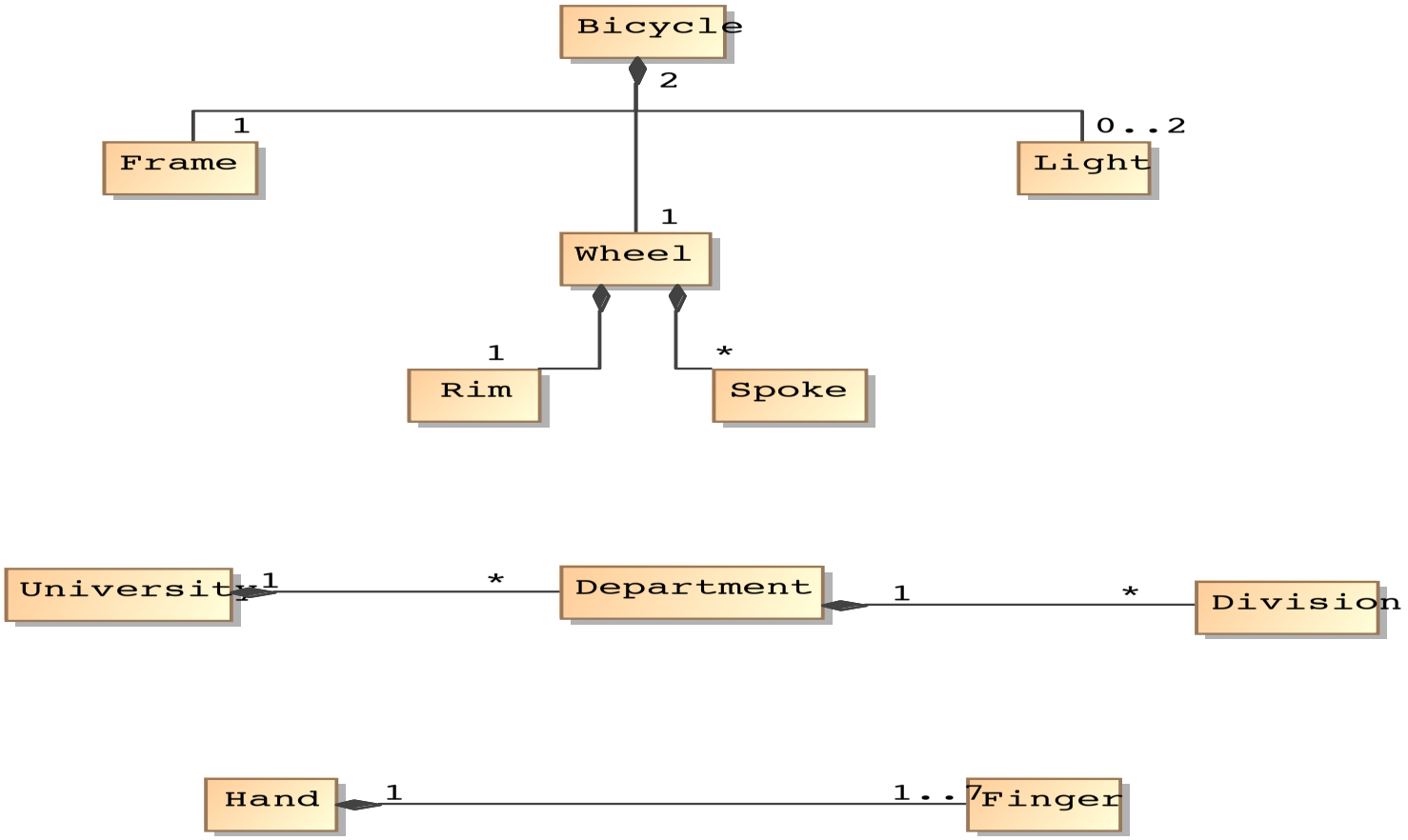
Conceptual Model for CoCoME



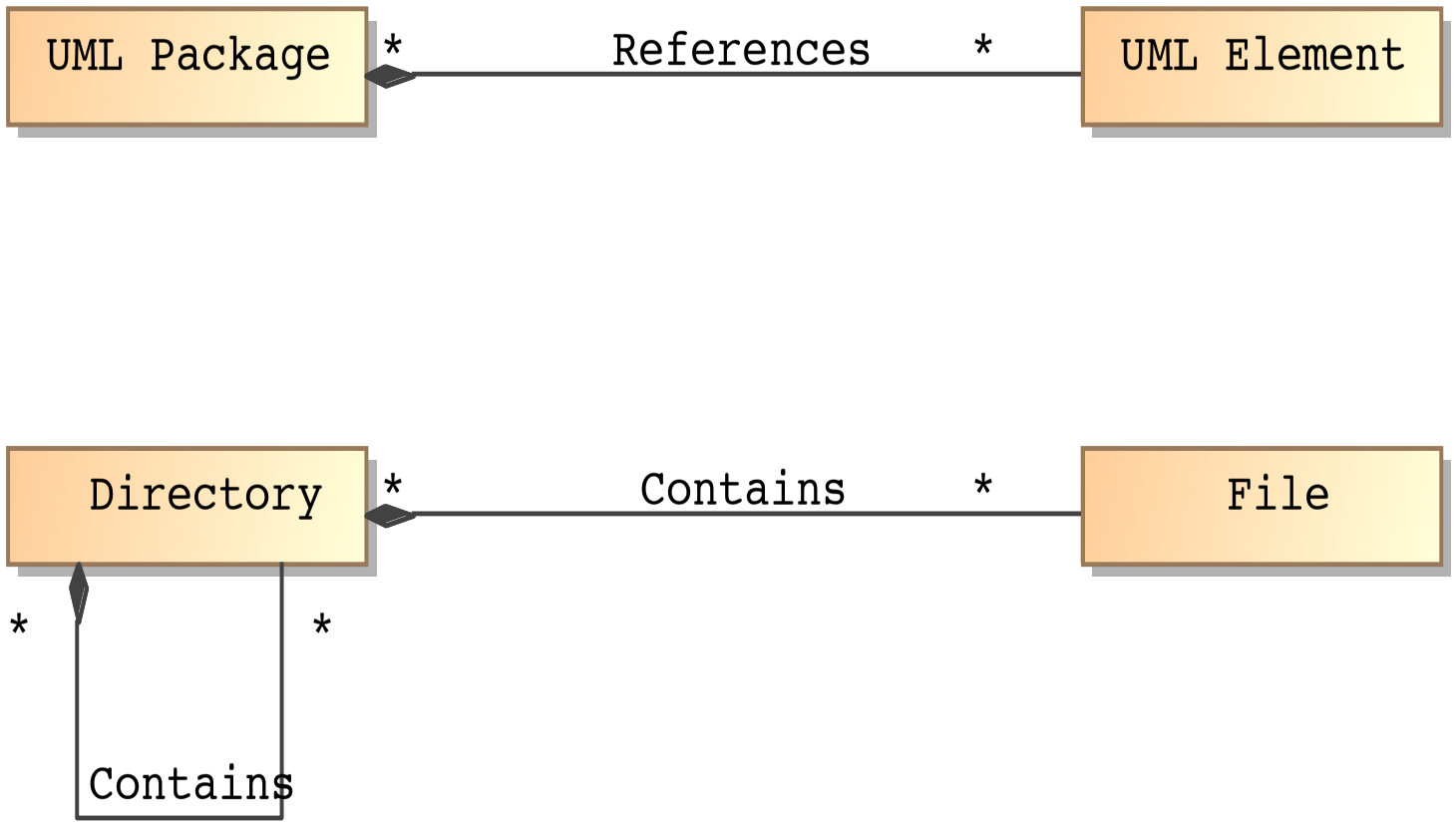
Aggregation

- an aggregation an association used to model a whole-part relationship between objects
- an aggregation is a composition if the multiplicity at the composite end is at most one
- an aggregation is a shared aggregation if the multiplicity at the composite end is more than one

Examples of composition



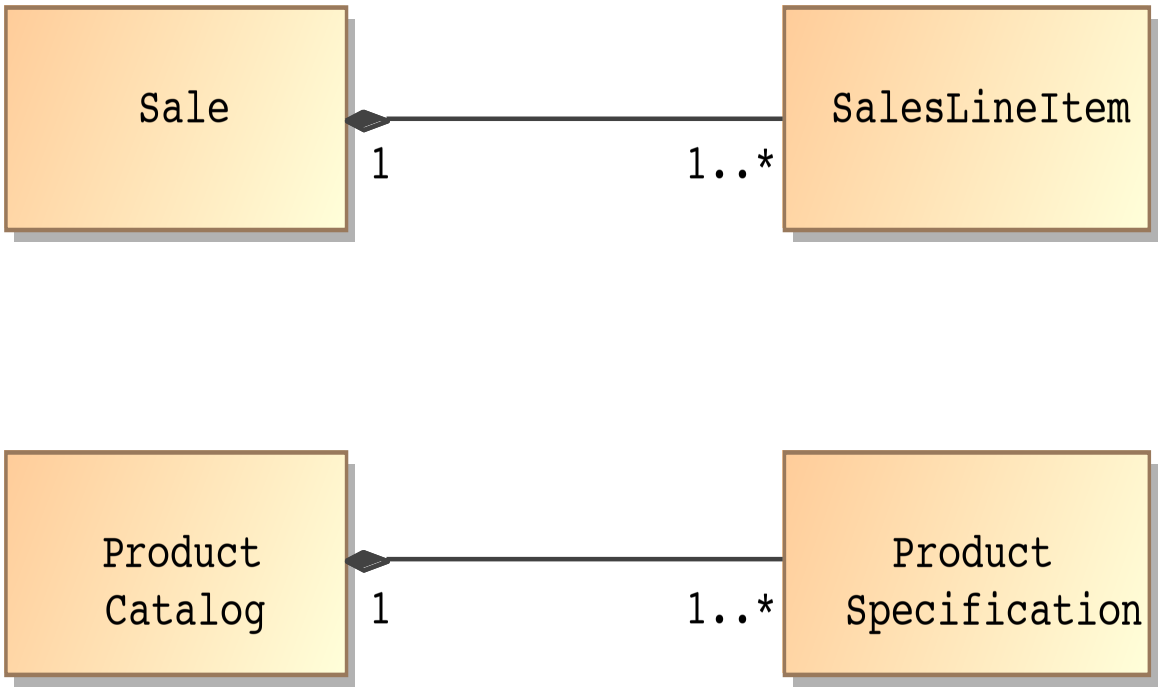
Examples of Shared Aggregation



Common Properties of Aggregations

- the lifetime of the part is bound within the lifetime of the composite, i.e. there is a create–delete dependency of the part on the whole.
- properties of the composite propagate to the parts, such as its location
- operations applied to the composite propagate to the parts, such as destruction, movement, recording.

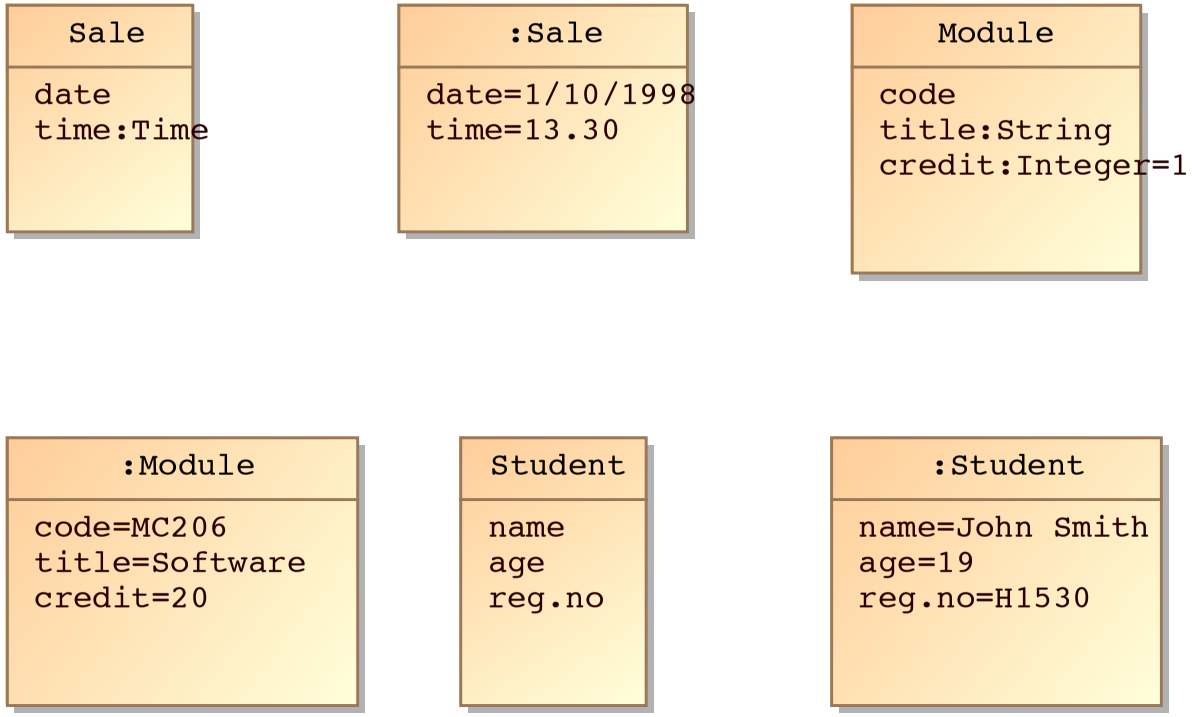
Aggregation in CoCoME



Attributes of Classes

- an instance of a class has meaningful properties
- an attribute of a class is the abstraction of a single property of objects of the class
- at any time, an instance has a logical value of an attribute of its class, called the attribute of the object at that time
- one object has exactly one value for each attribute at any given time, which can be recorded, modified, and passed among objects

Examples and UML Notation



Attributes of Classes

- complete: capture all relevant attributes about a class,
- fully factored: each attribute captures a different property of an object,
- mutually independent: the values of the attributes of an object are independent of each other, i.e. try to avoid derived attributes,
- relevant to the requirements and use cases: focus on those attributes for which the requirements suggest or imply a need to remember information.

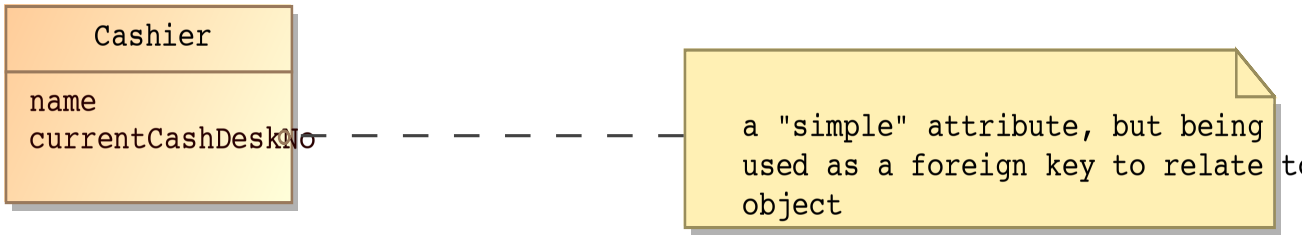
formal modelling is effective for analysing these features

Attributes, Objects, Associations

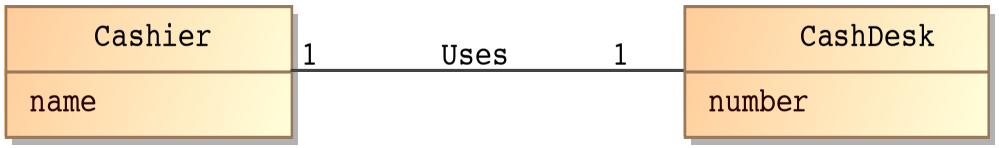
common mistake: represent something as an attribute when it should have been a concept or an association

- attributes are of simple data
- the defining features of objects do NOT apply to attributes
- attributes are NOT for inter-object communication or navigation

Worse



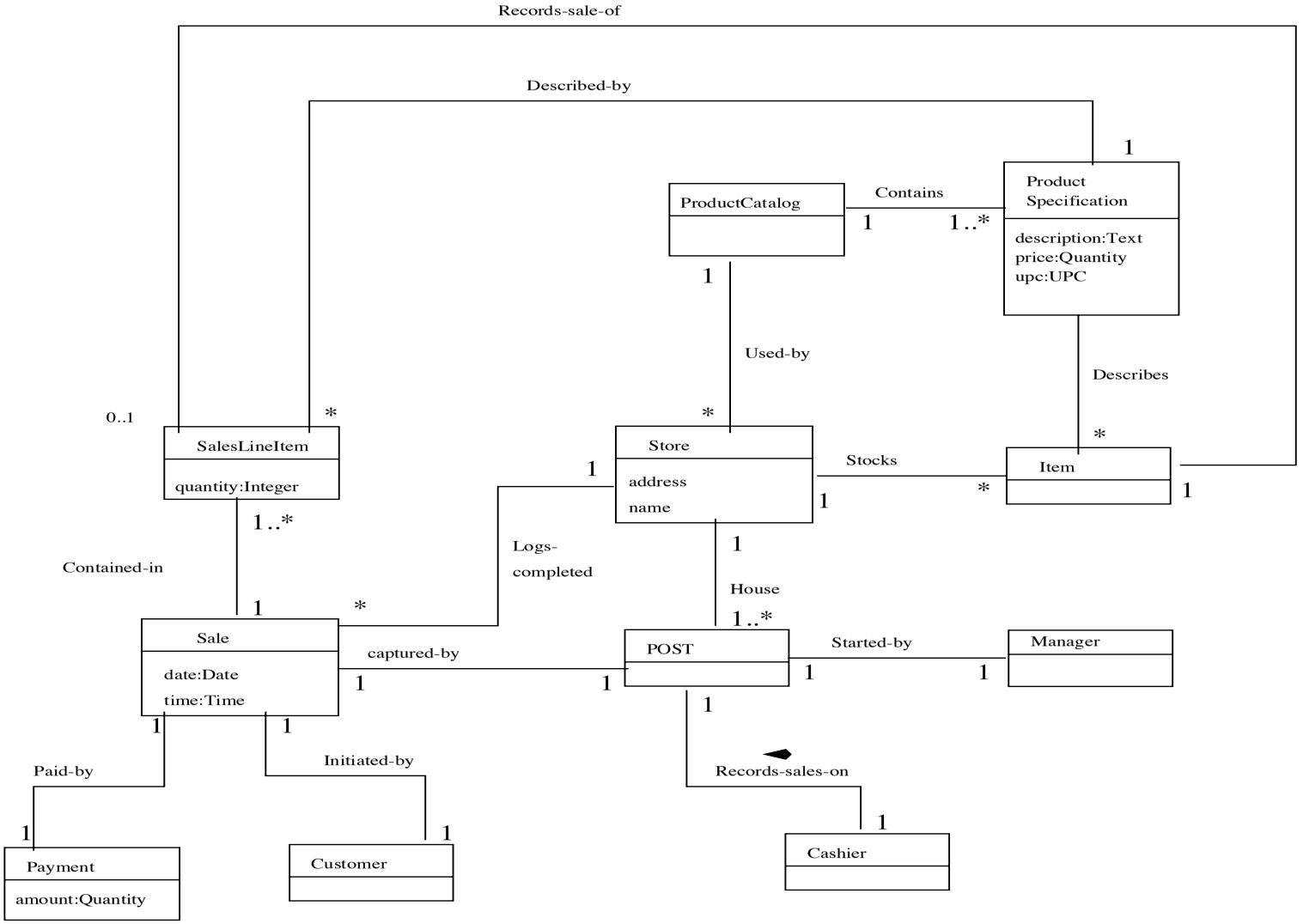
Better



Class Diagram for RA

- defines the data types and class structures of the components
- static OO architecture of a component
- define the allowed state space and some state constraints of the component

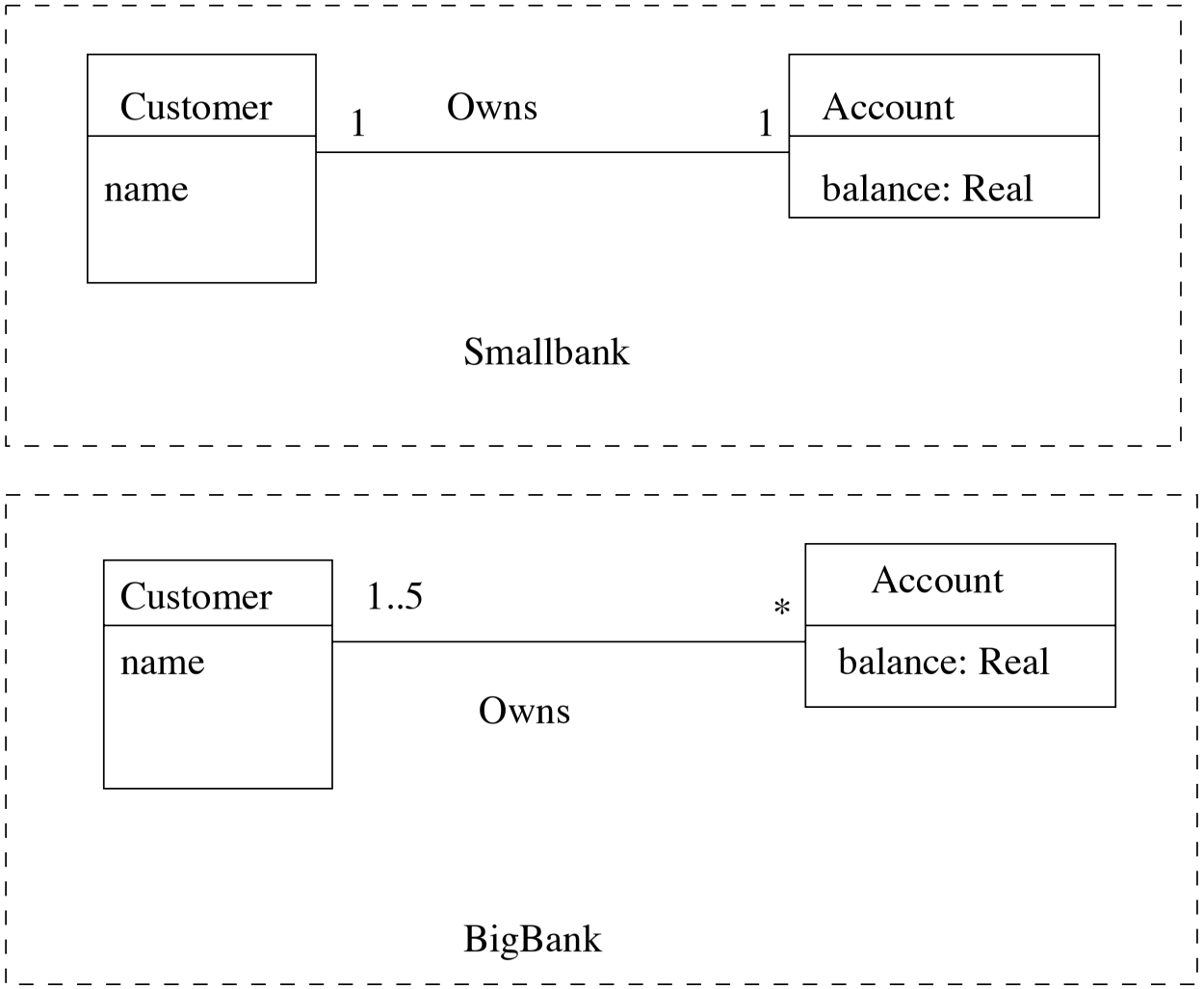
Adding Attributes to Class Diagram



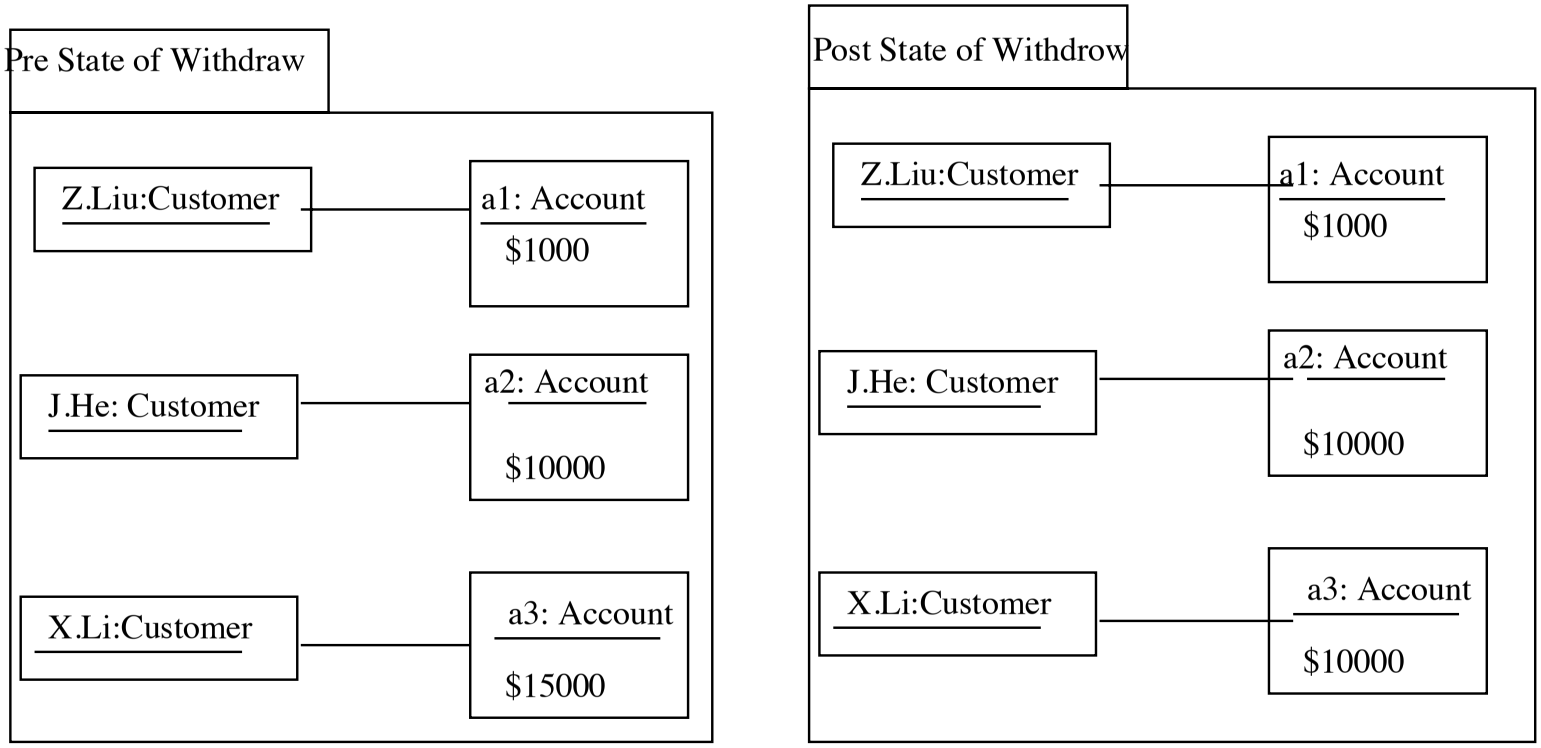
States of a Use Case (Component)

- a state of a component at a moment of time defines
  - the existing objects of each of the classes,
  - the values of the attributes of these objects,
  - the existing links between objects
- the class diagram defines the state space of the component
- a state is represented in UML by an object diagram

Example: Banks

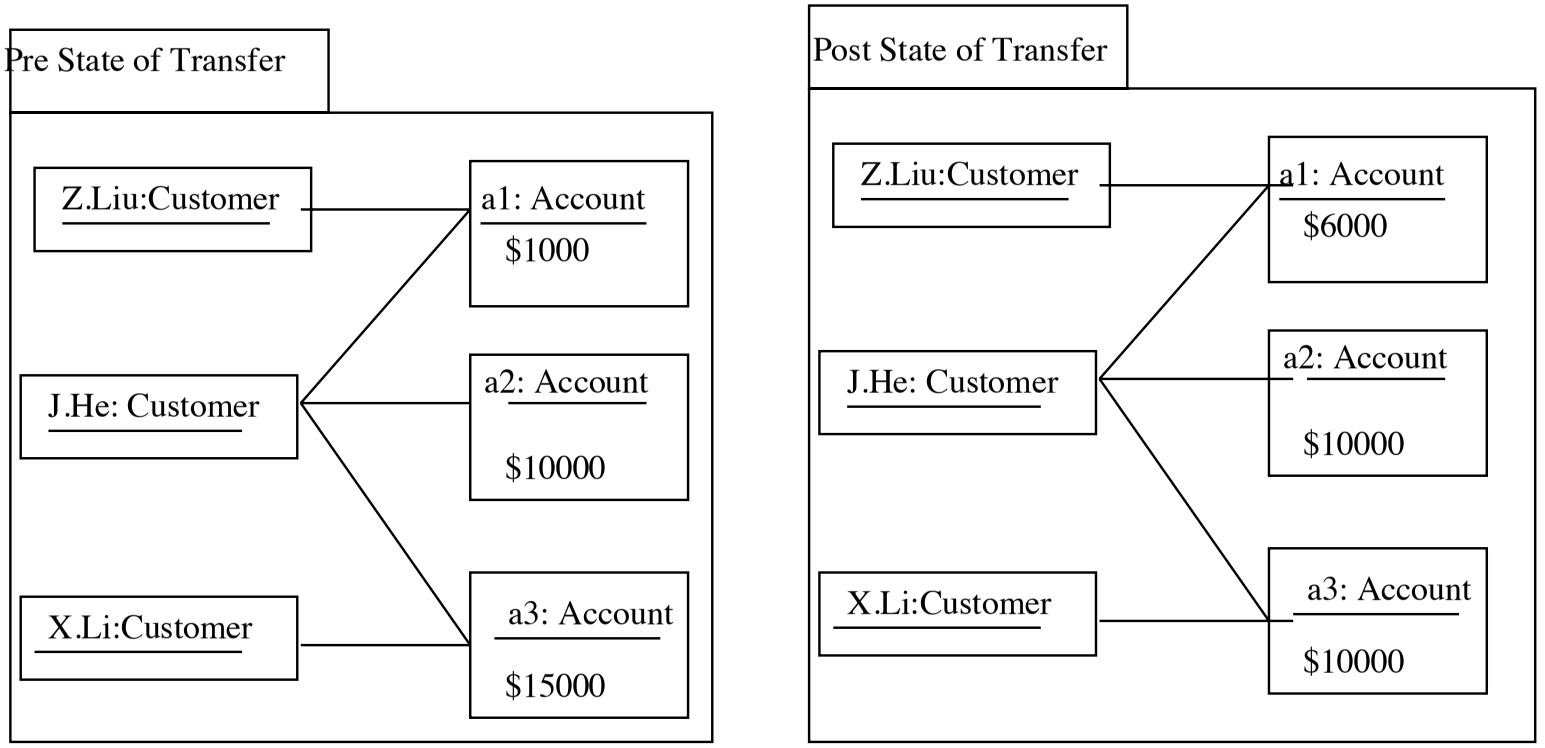


Object Diagrams



OBD of SmallBank or BigBank?

More Object Diagrams

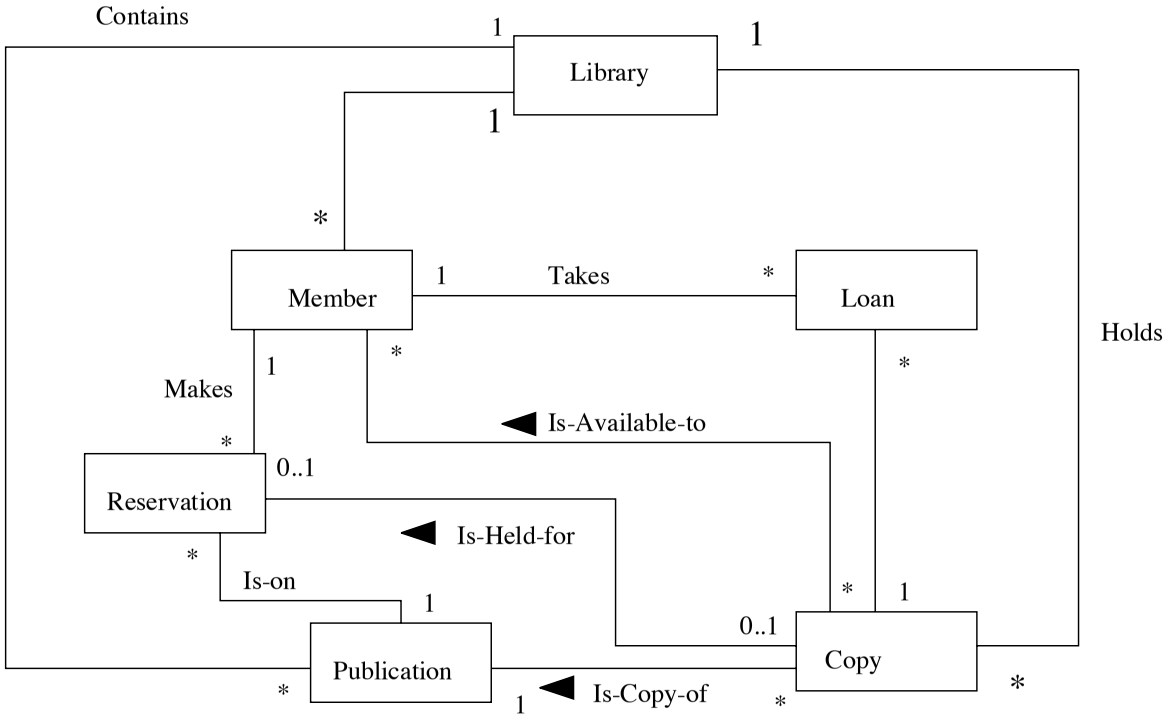


OBD of SmallBank or BigBank?

State Constraints

- an object diagram is a state of the component at a moment of time
- classes, associations, multiplicities and attributes in a conceptual model impose constraints on the set of allowable states
- can a conceptual model express all the possible constraints?

Consider the Library System



a copy c that Is-Held-for a reservation r must be a copy of the publication that is reserved by the reservation r?

Add State Constraints

- diagrams not expressive enough for all possible state constraints
- add informal textual or formal specification
  - $\forall c : Copy, r : Reservation, p : Publication \bullet$
    c Is-Held-For $r \wedge r$ Is-on $p \Rightarrow c$ Is-Copy-of p
  - Is-Held-for $\circ$ Is-on $\subseteq$ Is-Copy-of
- state constraints are important for the system design and need to be checked for every step in every use case

Interaction Protocol

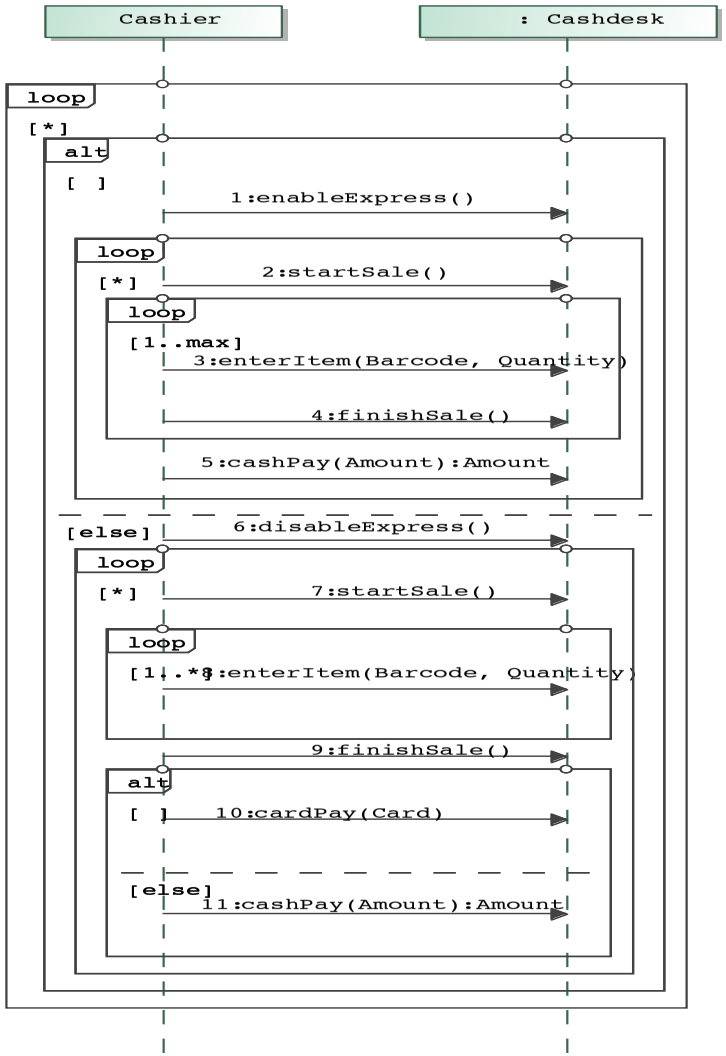
- identifying provided methods by component
- create sequence diagrams for use cases
- define contracts of provided interface methods

Output: use case sequence diagrams, contracts of provided interface of component

Use Case Uses a Protocol

- a use case describes how uses interact with the component
- actors generate event to the component to request some operation in response
- the component carries out the operation requested
- in OO, synchronization of events and their responses is method invocation
- the order in which actors call the operations is important,
- difference executions of the process may call different sequence of operations
- a use case describes the interaction protocol that is all the sequences of events that actors are allowed to generate when using the component
- cf. the use case of process a sale

Modelling Protocol



Formally...

Trace in regular expression *tr*(SD_{uc1})

$$
\begin{aligned}
(\quad & \mathit{enableExpress}()(\mathit{startSale}()\ \mathit{enterItem}()^{(\mathit{max})} \\
& \mathit{finishSale}()\ \mathit{cashPay}())^* \\
+ \quad & \mathit{disableExpress}()(\mathit{startSale}()\ \mathit{enterItem}()^+ \\
& \mathit{finishSale}()\ (\mathit{cashPay}()\ +\ \mathit{cardPay}()))^*\ )^*
\end{aligned}
$$

Are regular languages expressive enough for describing all use case protocols?

What are the existing formalism for specification and analysis,

CSP, Automata...?

Provided Interface of Component

- the set of methods required by the actors *event*$(x; y)$
- these methods must be provided in the provided interface of the component

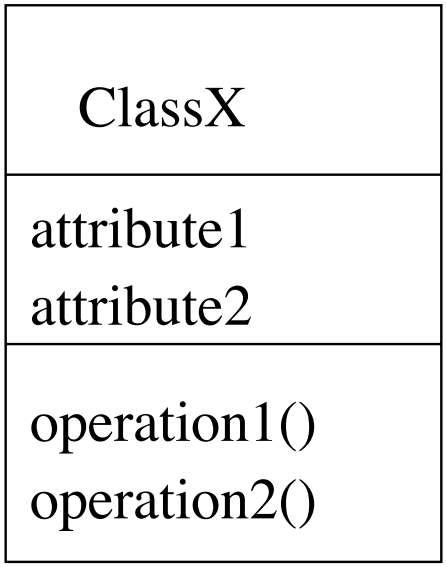
- *startSale*(), *enableExpress*()
- *enterItem(upc,quantity)*, *finishSale*()
- *cashPay(amount)*, *cardPay(card)*

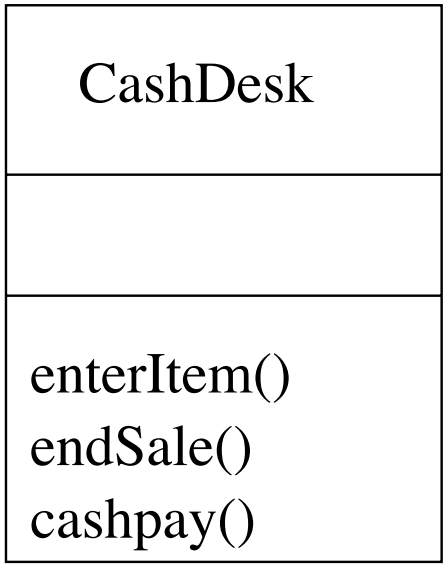
Remarks:

1. names of methods should be expressed at the level of intent – abstraction
2. use verb phrases to name methods
3. e.g. use *enterItem*() and *finishSale*() rather than *enterKeyPressed*() and *enterReturnKey*()

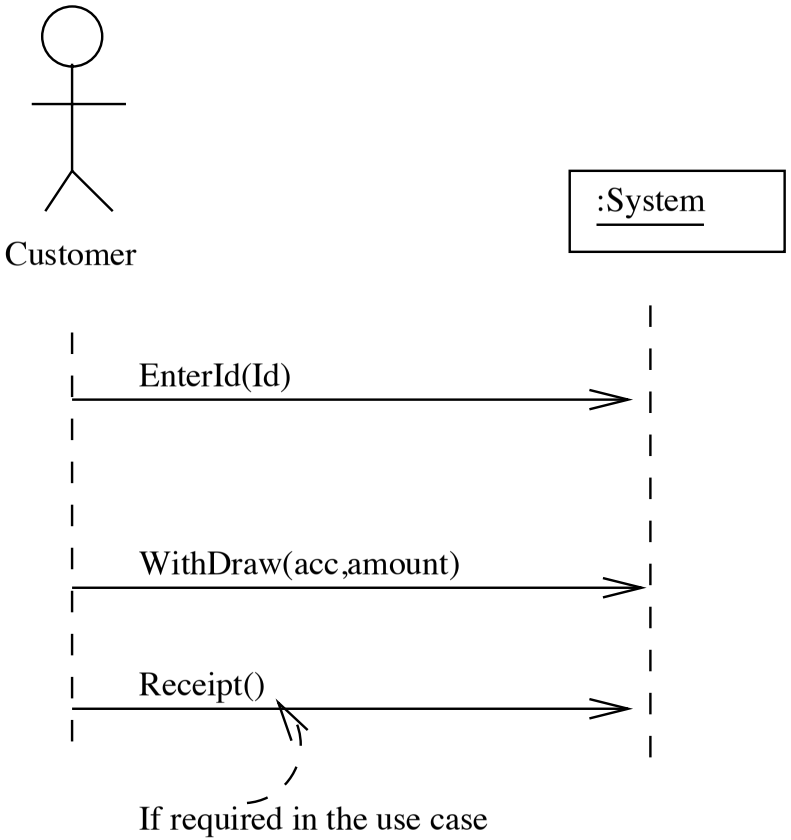
Record provided Methods

- for each use case, we give a a use-case handler class or interface class of the component
- use the UML notation for a class to record the methods

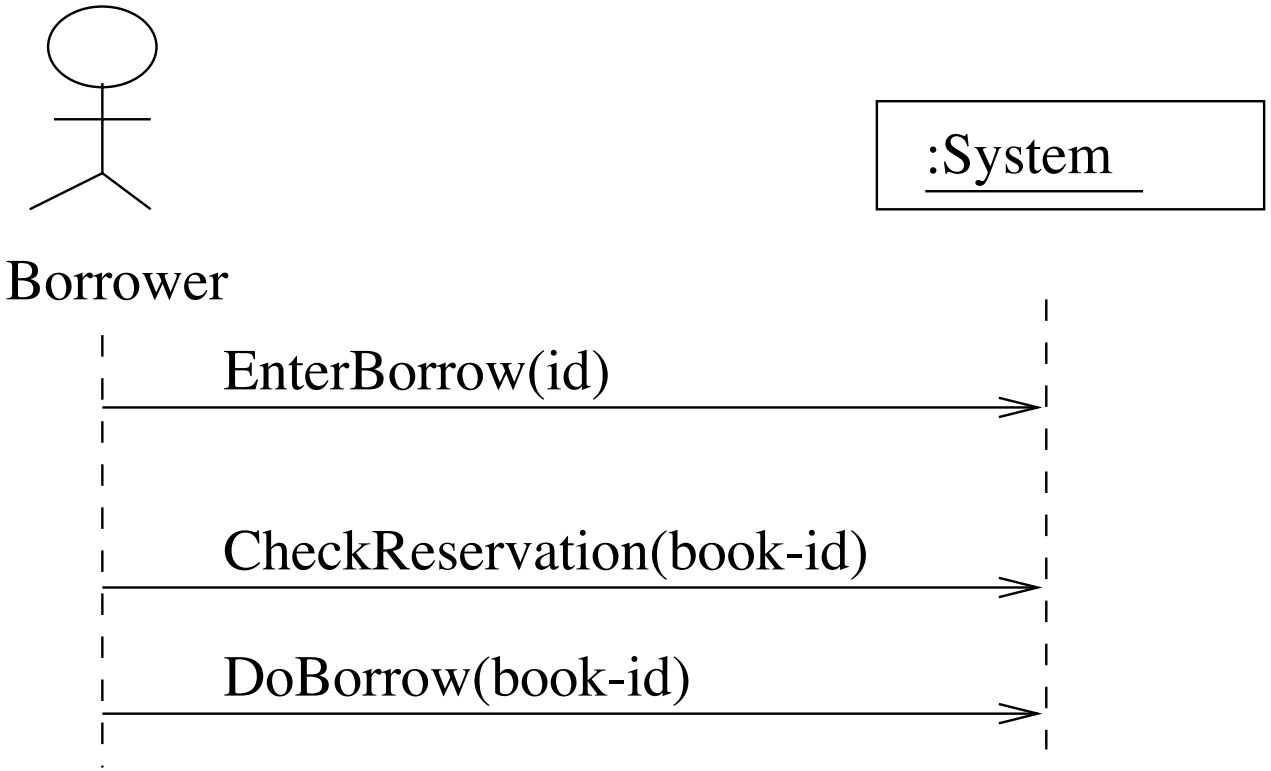




Example: Withdraw Money



Example: Borrow a Book



CONTRACTS OF PROVIDED METHODS

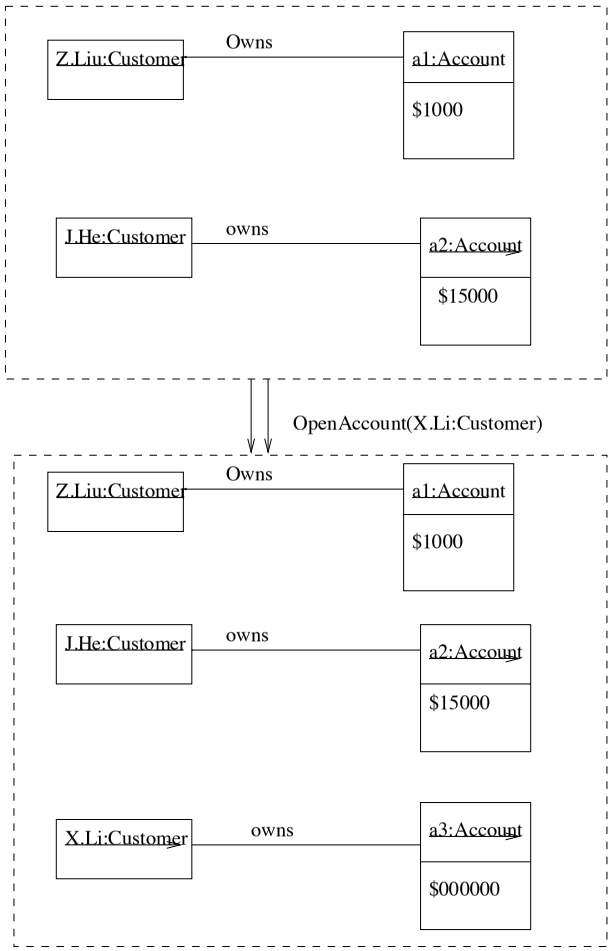
- use sequence diagram describes the interaction view and identifies methods required by actors
- not describe the functionality of the methods
- functionality of a method is understood in terms of state changes of the component

States and Behaviour

- a state of component at a time consists of the objects of classes existing, values of attributes of these objects, and links between the objects at that time
- a state at a time is a snapshot of the component execution
- a state is an *object diagram* of the class diagram
- behaviour is about when and which methods can be invoked and what state change can be made when a method is invoked and executed

How to specify the behaviour of a component?

Example: bank system:



How to specify the functionality of a method?

Pre and Post Conditions

The notation is pre- and post-conditions or Hoare Triples

$$\{\textit{Pre-Condition}\}m()\{\ \textit{Post-Condition}\}$$

- pre-conditions are the conditions that the states is assumed to satisfy before the execution of the operation
- post-conditions are the conditions that the state has to satisfy when the execution operation has finished.
- partial correctness semantics and total semantics of Hoare Triples

Examples

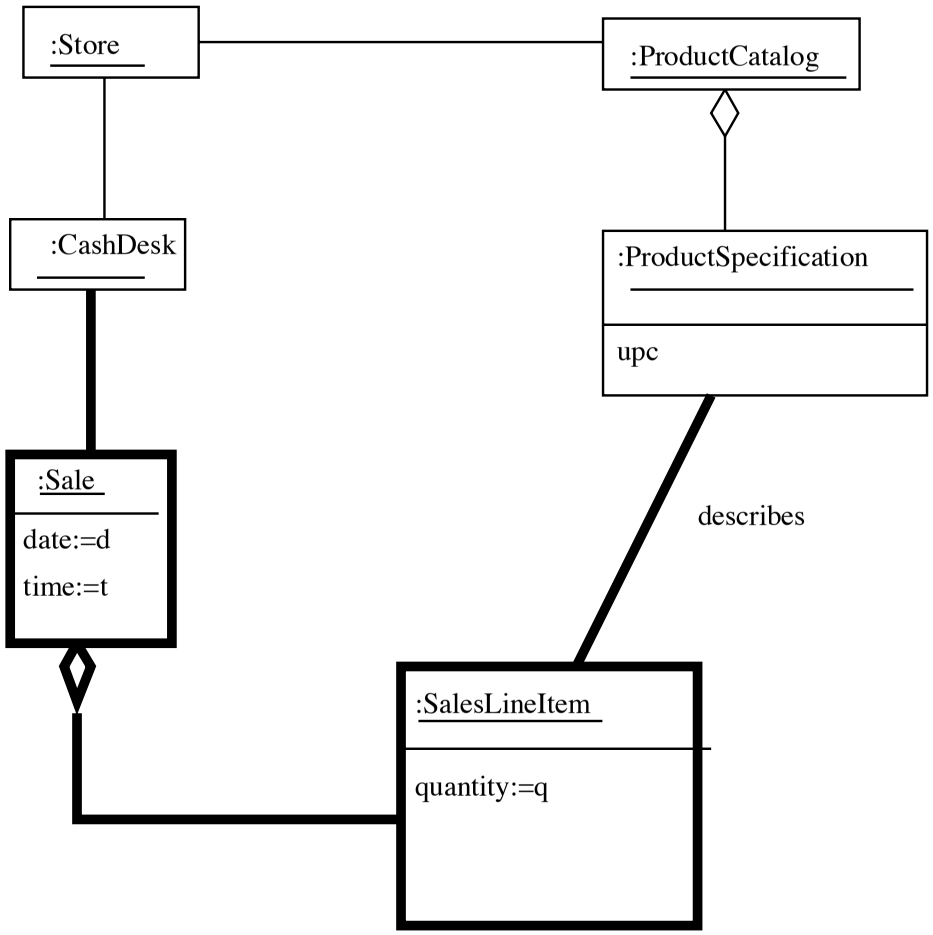
- the precondition of *openAccount*? — depends on the conceptual model!
- the pre and post conditions for *deposit*(a, m)?
- the pre and postconditions of *transfer*(a_1, a_2, m)?

Example

Consider *enterItem(upc:UPC, quantity:Integer)*:

- Precondition: *UPC is known to the system*.
- Postcondition:
  - If a new sale, a *Sale* was created .
  - If a new sale, the new *Sale* was associated with the *CashDesk*.
  - A *SalesLineItem* was created.
  - The *SalesLineItem.quantity* was set to *quantity*.
  - The *SalesLineitem* was associated with the *Sale*.
  - The *SalesLineItem* was associated with a *ProductSpecification*, based on *UPC* match.

Example: state change by *enterItem*



Remarks

- many possible pre- and post-conditions for a method, but focus on the following post conditions
  - instance creation and deletion; attribute modification; and associations formed and broken

  and the precondition about things that are important to check the execution of the operation

For formal verification, complete and precise specification needed,

and thus abstraction is essential

Contracts vs Class Diagrams

- Contracts of operation are expressed in the context of the class diagram
  - What instances can be created or deleted?
  - What associations can be formed or broken?
  - What attributes can be modified?

Informally Documenting Contracts

	Contract
Name:	Name of operation, and parameters.
Responsibilities:	An informal description of the responsibility this operation must fulfill.
Type:	Name of type (concept, software class, interface).
Cross References:	System function reference, use cases, etc.
Note:	Design notes, algorithms, and so on.
Exceptions:	Exceptional cases
Pre-conditions:	As defined
Post-conditions:	As defined

Contract for *enterItem*

Contract	
Name:	enterItem(upc:UPC, quantity:Integer).
Responsibilities:	Enter an item and add it to the sale. Display item description and price.
Type:	Component
Cross References:	Use Cases: Process Sale
Note:	Use superfast database access.
Exceptions:	If UPC is invalid, indicate an error.
Pre-conditions:	UPC is known to the system.

Post-conditions:

- If a new sale, a $Sale$ was created
- If a new sale, the new *Sale* was associated with the *CashDesk*
- A *LineItem* was created
- The *LineItem.quantity* was set to *quantity*
- The *LineItem* was associated the *Sale*.
- The *LineItem* was associated with the *ProductSpec*

Contract for $finishSale$

	Contract
Name:	endSale().
Cross References:	Use Cases: Process sale
Exceptions:	If a sale is not underway, indicate that it was an error.
Pre-conditions:	The Sale is not finished
Post-conditions:	• $Sale.isComplete$ was set to $true$ (attribute modification).

Here, a new attribute *isComplete* for class *Sale* is discovered.

Contract for *cashPay*

Contract

Name: cashPay(amount: Quantity).

Exceptions: If sale is not complete, indicate that it was an error.
If the amount is less than the sale total, indicate an error.

Post-Conditions:

- A *Payment* was created (instance creation).
- *Payment.amountTendered* was set to *amount* (attribute modification).
- The *Payment* was associated with the *Sale* (association formed).
- The *Sale* was associated with the *Store*, to add it to the historical log of completed sales (association formed).

Contract for Start up Component

Contract

Name: startUp().

Responsibilities: Initialise the system

Post-conditions:

- A *Store, CashDesk, ProductCatalog and ProductSpecification* were created.
- *ProductCatalog* was associated with *ProductSpecification*.
- *Store* was associated with *ProductCatalog*.
- *Store* was associated with *Cashdesk*.
- $CashDesk$ was associated with *ProductCatalog*

Contracts and other Documents

- Use cases suggest the input events to the components and interface sequence diagrams.
- The provided operations are then identified from the interface sequence diagrams.
- The effect of the provided methods is described in contract within the context of the conceptual model.

How to check consistency, how to carry out analysis?

Analysis Phase Conclusion

- *Aims:* emphasises on understanding of the requirements, concepts, and operations related to the system.
- *Characteristic:* focusing on questions of *what* – what are the processes, concepts, associations, attributes, operations.
- *Artifacts:* that can be used to capture the results of an investigation and analysis.

Analysis Artifact	**Questions Answered**
Use Cases	What are the domain processes?
Class Model	What are the concepts, associations and attributes?
Use Case SDs	What are the system input events and operations?
Contracts	What do the system operations do?

DESIGN OF COMPONENT

- The nature of the design phase
- UML notation for interaction diagram
- Patterns for assigning responsibilities to objects
- Use Patterns to create collaboration diagrams
- Design class diagrams

Output of Design: interaction diagrams and design Class diagrams

Progress From Analysis to Design

- requirement model should have been analysed
- the functionality of methods are to be realised by interactions among objects of the components
- the heart of the design is to design these interactions
- the design must be presented as a model from which code can be generated and constructed

Model Object Interactions in UML

- the UML mode of design mainly include a set of interaction diagrams and a design class diagram

  1. collaboration diagrams
  2. object sequence diagrams

  Either can be used to express similar or identical messages interactions.

- the interaction diagrams are the most important for design, and require the greatest degree creative effort
- assurance of correctness of a design , either by construction or verification requires formalism

Object Sequence Diagram

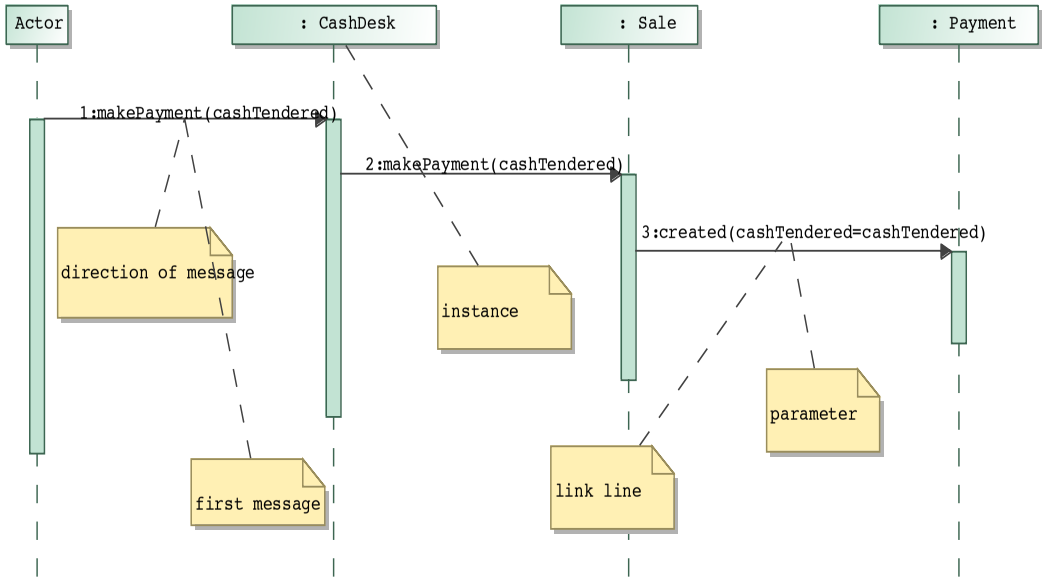
extend interface SD (ISD) to illustrate interactions among objects of components

Interface SD vs Object SD

- ISD models the interaction between the component and its environment (actors)
- OSD shows the interactions between objects inside the component
- use case SD identifies provided methods
- object SD identifies defines methods of classes
- if treat an object as a (sub) component, and the other objects as actors, then the OSD derives a ISD of the subcomponent

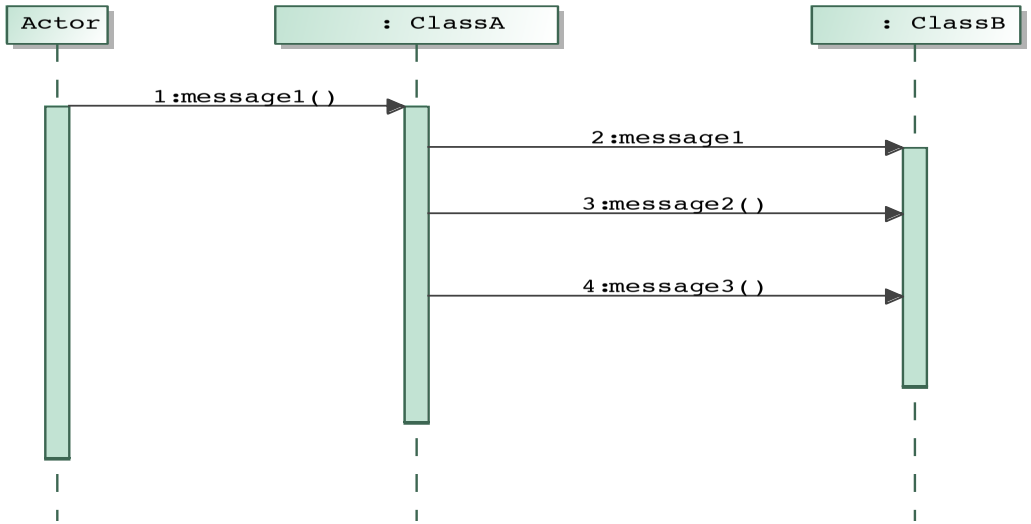
Example

1. Cashier sends request *makePayment(cashTendered)* to *CashDesk*
2. *CashDesk* carries it out by sending $cashPay(cashTendered)$ to $Sale$.
3. *Sale* carries out the task by creating a *Payment*.



Notation in UML

- instances of classes in boxes
- a directed link between two objects represents a instance of an association and visibility
- a message represents an invocation of a method of the target object (server) from the source object (client)



Messages, Links and Associations

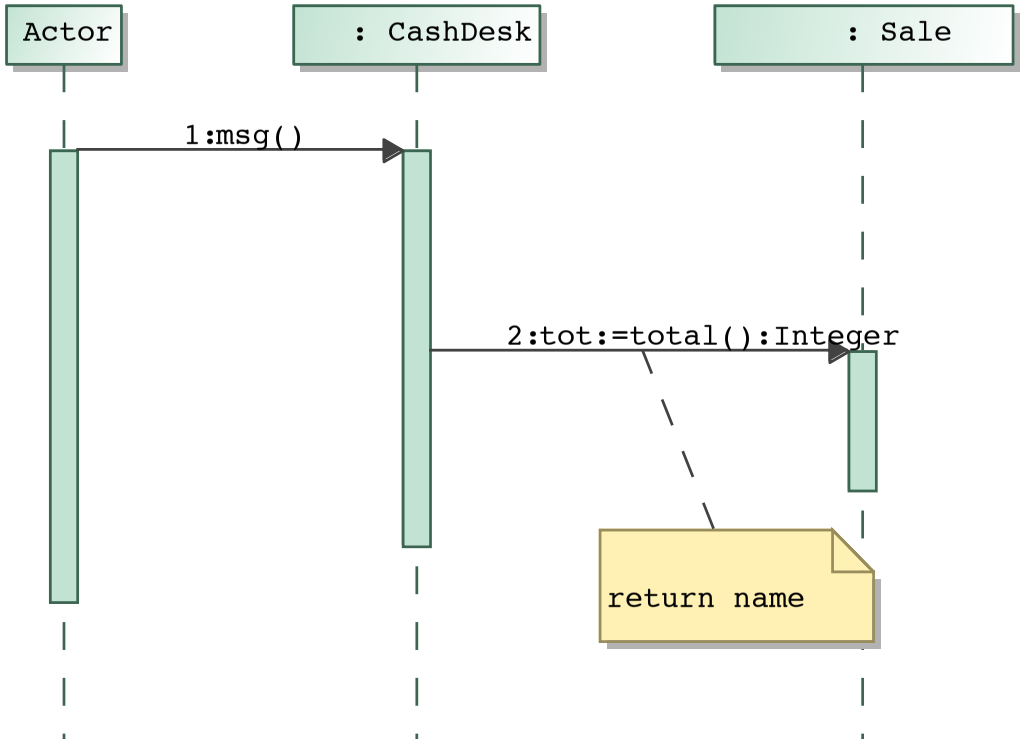
- a message can be passed between two objects only when the two objects are linked
- two objects can only be linked by an association of the classes of the objects
- any method of the server can be invoked by the client when there is link between them

How OSD related to Program?

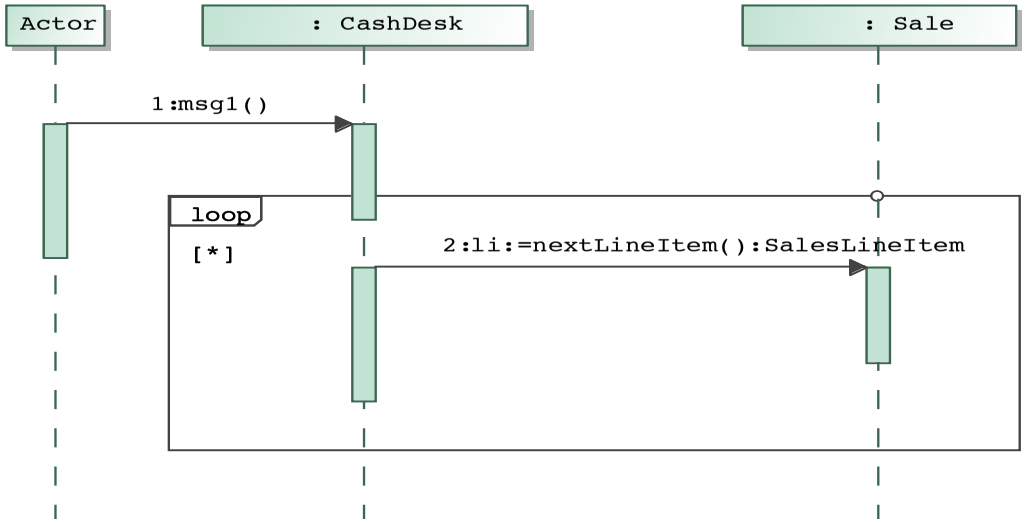
Messages with Return Value

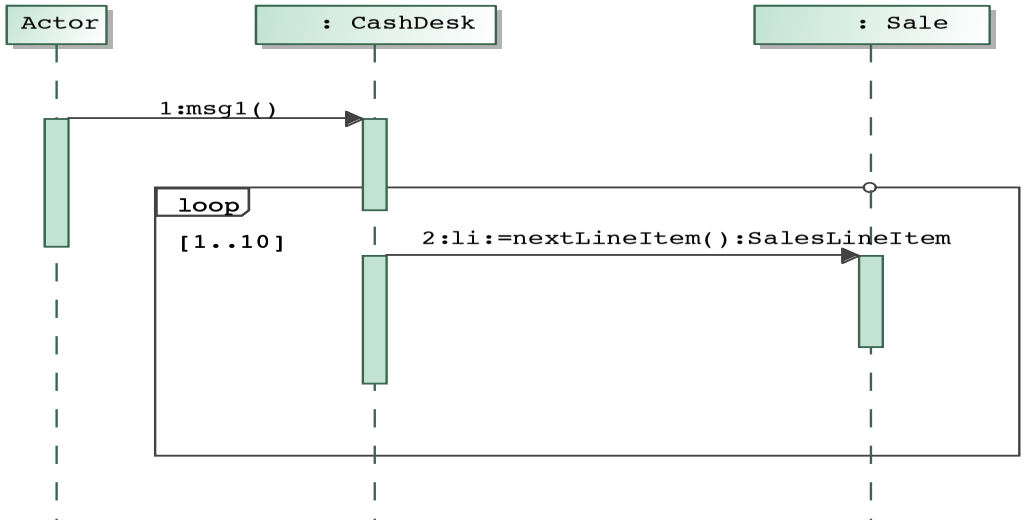
Standard syntax for messages with return value:

```
return := message(p : pType) : returnType
```

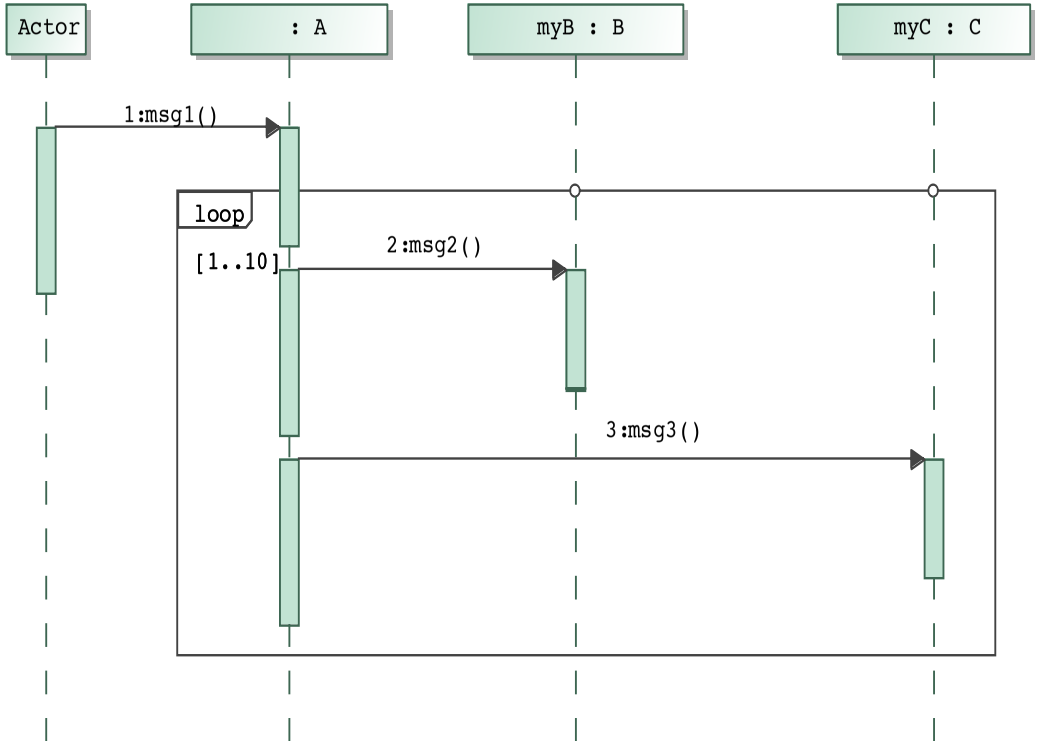


Loops


Actor
: CashDesk
: Sale
1:msg1()
loop
[*]
2:li:=nextLineItem():SalesLineItem

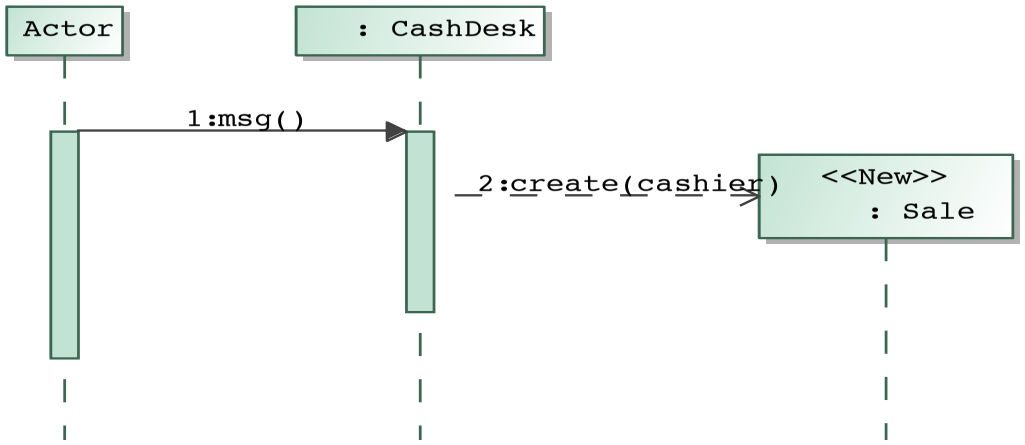

Actor
: CashDesk
: Sale
1:msg1()
loop
[1..10]
2:li:=nextLineItem():SalesLineItem

Multiple Messages in a Loop

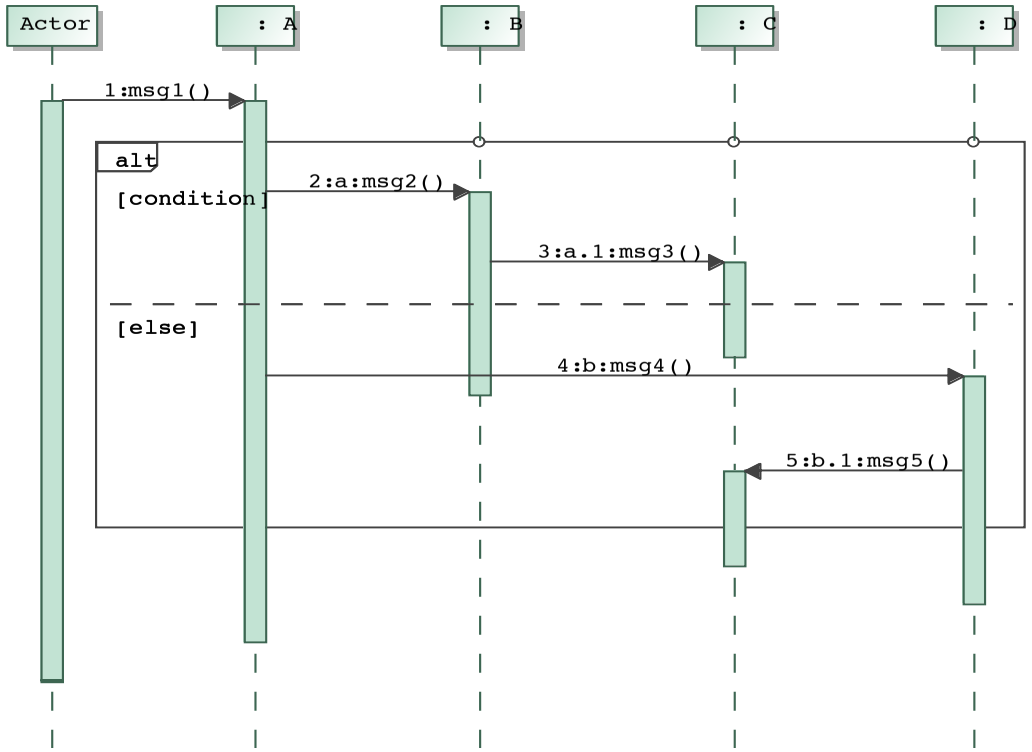


Creation of Instances

- UML use *create* to request for a creation of a new object, which is independent of programming languages
- we can also use *C.New*()
- the newly created instance may include a «*new*» symbol
- message *create*() can take parameters



Conditional Choice



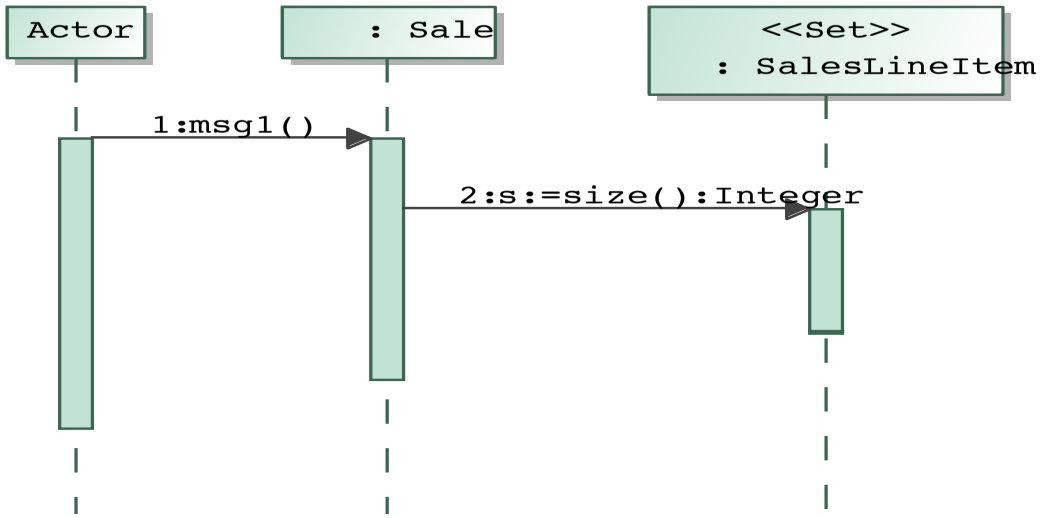
more cases, nondeterministic choices

Multiobjects

- a multiobject represents a set of objects, instance of a container class
- examples: $SetOfSales$; $SetOfSalesLineItems$
- implementation: Array, List, Queue, Stack, Set, Tree, Vector
- a multiobject represents a set of objects at the "*" end of an association
- represent a multi-object by, e.g. *sales* : *Set*(*Sale*), and in diagrams the stereo type «set» is used.

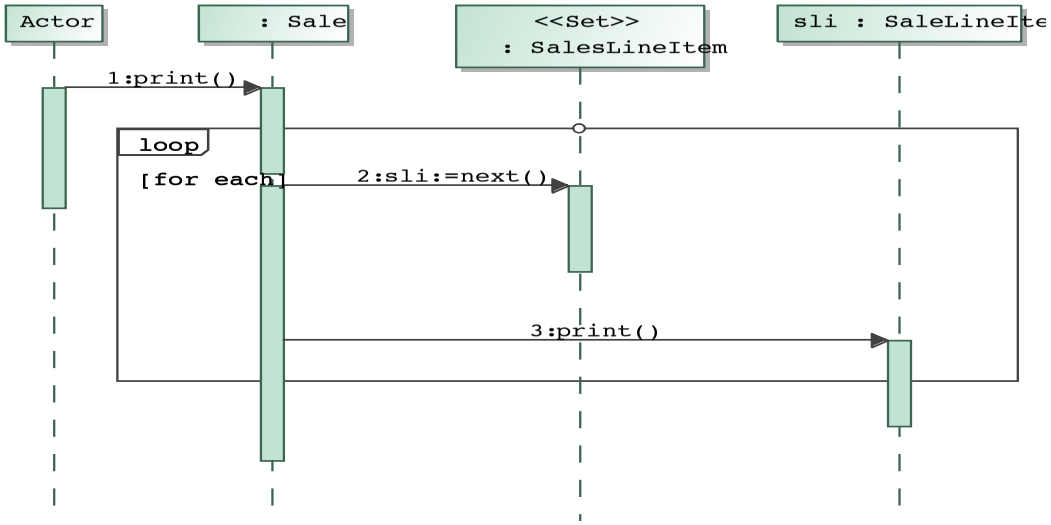
Messages to Multiobject

A message sent to a multiobject is sent to the single object, not broadcast to each element in the multiobject



Method on Each Object in Multiobject

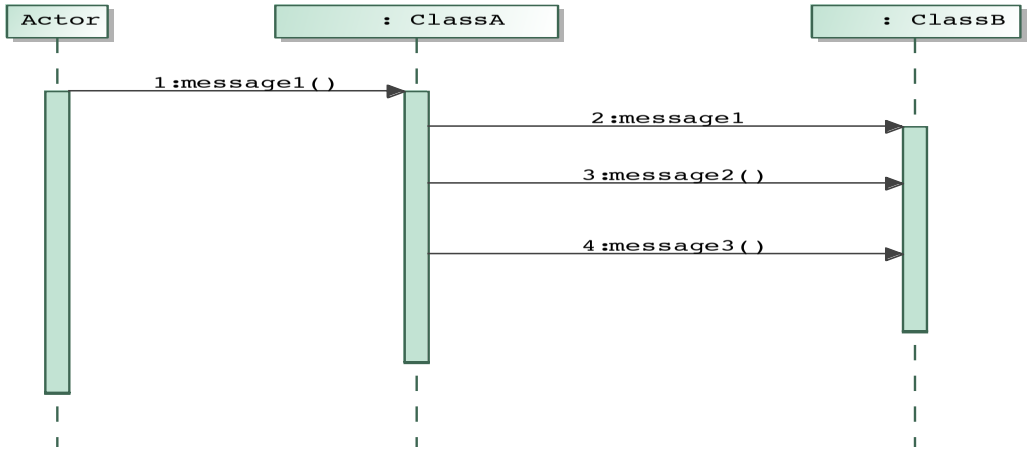
1. an iteration to the multiobject to extract links to the individual objects
2. then a message sent to each individual object using the (temporary) link



Design: what to do in the design phase?

- shift in emphasis from application domain concepts toward software objects
- shift from *what* toward *how*
- interaction diagrams illustrate the decomposition in terms of interactions between objects.

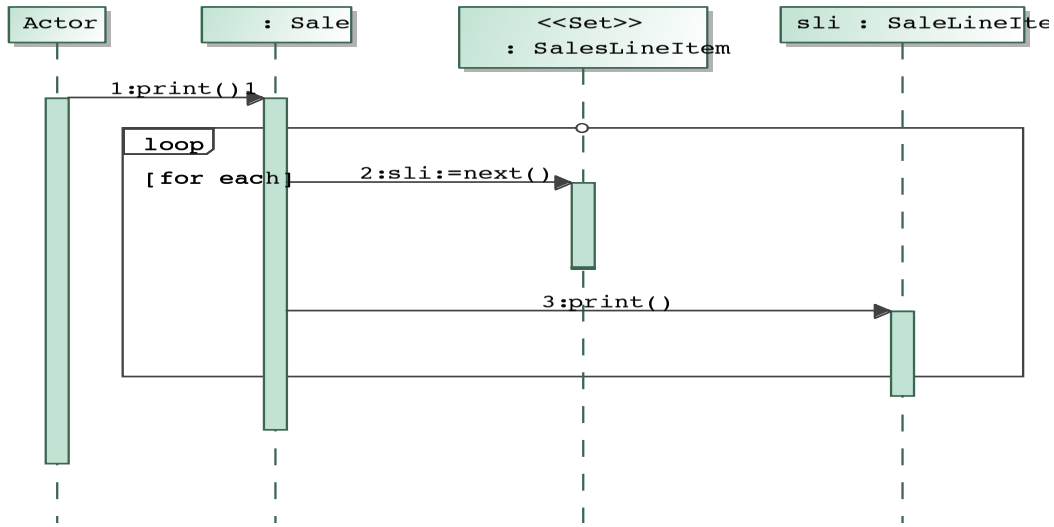
Model Decomposition of Interface Method



- $msg1()$ is decomposed into three methods of ClassB:

```
ClassA:: msg1() {
ClassB.message1();
ClassB.message2();
ClassB.message3()
}
```

Another Example



$print()$ decomposed into loop, printing each SaleLineItem of the Sale:

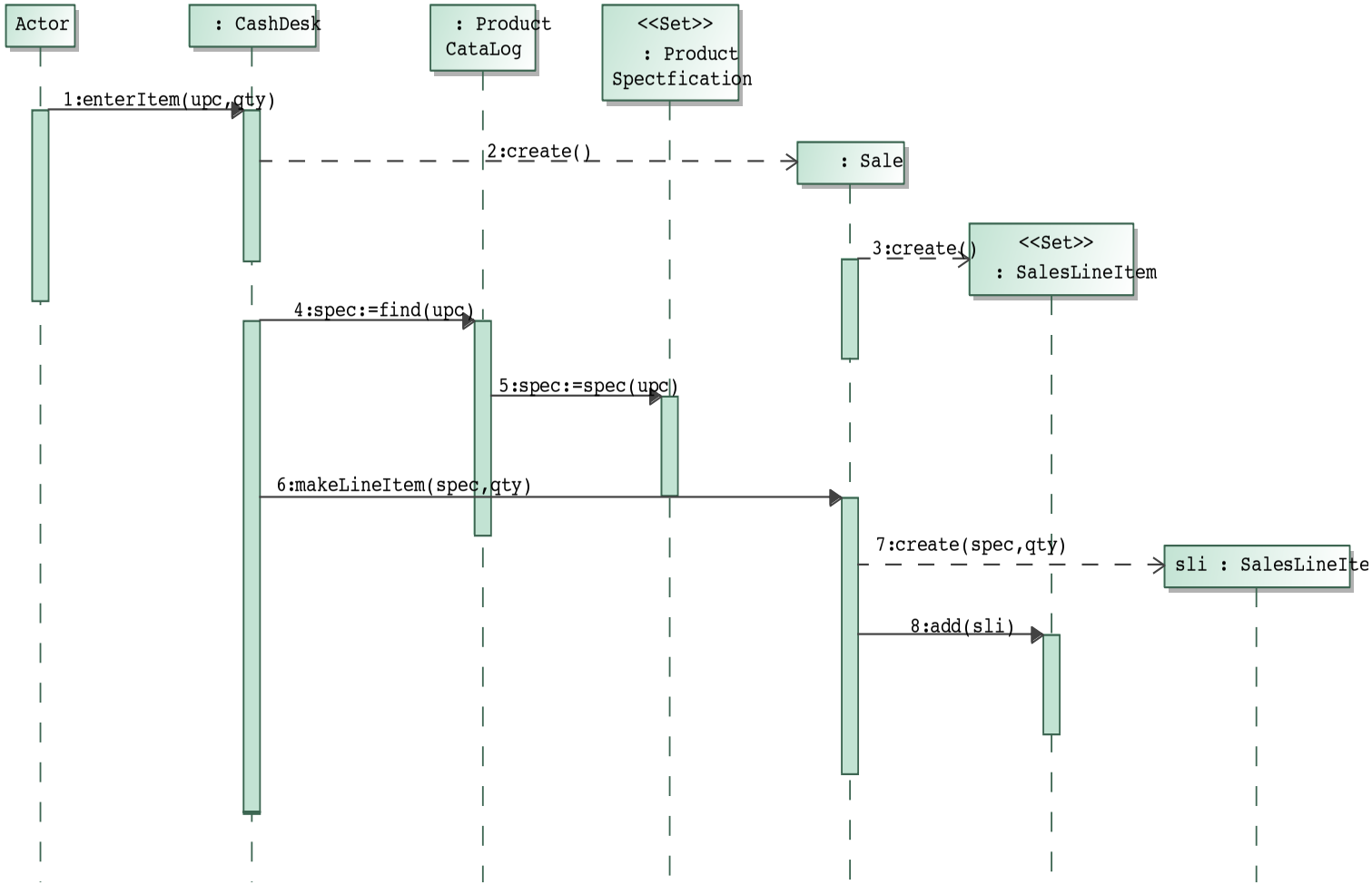
```
Sale:: print() {
  for each SaleLineItem sli do sli.print()
}
```

In Java

A multiobject can be implemented as a vector

```
class Sale {
     private SaleLineItem sli;
     private Vector lineItems = new Vector();
     public void print()
    {
     Enumeration e = lineItems.elements();
     while ( e.hasMoreElements() )
     sli = ((SaleLineItem) e.nextElement());
     sli.print()}
    }
}
```

More Complicated Decomposition



Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
  if (isNewSale() )  /*private method of CashDesk*/
  {
    sale = new Sale()   /*create a Sale*/;
  }
  ProductSpecification spec =
     productCatalog.specification (upc);
  sale.makeLineItem(spect,qty);
}
```

- *new Sale()* is further decomposed
- *makeLineItem(spect,qty)* is further decomposed

Sale:: makeLineItem(spect,qty)

```
class sale { ......

private Vector lineItems = new Vector();

public void makeLineItem
(ProductSpecification spec, int qty)
{lineItems.addElement(new SaleLineItem(spect,qty))
  }
....... }
```

Models Needed for Design

- class model
  - data/objects needed for a task are held by different objects are represented there
  - associations between classes represents knowledge of one object about another
  - objects of the classes participate in interactions illustrated in the interaction diagrams.
- contracts of interface methods: identifies subtasks in post-conditions

Patterns for Assigning Responsibilities

A responsibility (or functionality) is a contract or obligation of an object

1. Doing responsibilities: actions an object can perform:
   - doing something (action) itself
   - initiating an (action) in other objects
   - controlling and coordinating activities in other objects
2. Knowing responsibilities: knowledge an object maintains:
   - know about private encapsulated data
   - know about linked objects
   - know about things it can derive or calculate

Relation between doing and knowing?

Examples

1. We may assign the responsibility for print a $Sale$ to the $Sale$ instance – “a $Sale$ is responsible for printing itself” (doing)
2. We may assign the responsibility of knowing the date of a $Sale$ to the $Sale$ instance itself–“a $Sale$ is responsible for knowing its date” (knowing).

Knowing responsibilities are inferable from the conceptual model

Methods of Objects

- responsibilities of an object are implemented as *methods*
- a method may “act” itself or collaborate with other methods and objects
- e.g. $Sale$ may define a method *print*() that performs printing the $Sale$ instance
  To do so, $Sale$ may have to send a message to $SalesLineItem$ objects asking them to print themselves.

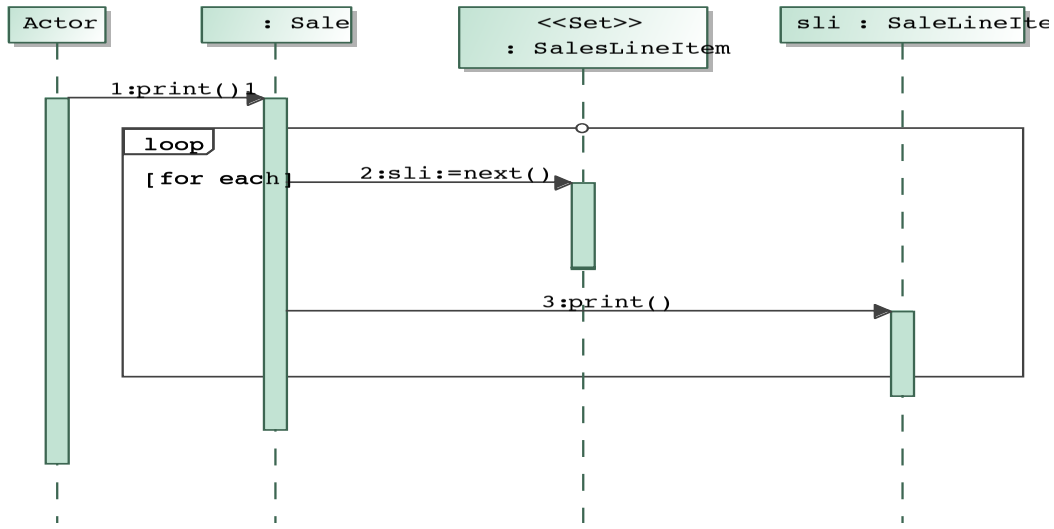
How to assign a responsibility to an object?

Responsibilities Assignment

1. start with the responsibilities identified from the use cases and contracts
2. assign these responsibilities to objects.
3. decide what the objects needs to do to fulfill these responsibilities in order to identify further responsibilities
4. assign further identified responsibilities to objects
5. repeat these steps until the identified responsibilities are fulfilled and a collaboration diagram is completed.

Example

- the contract of $cashPay()$ and the use case, identify the responsibility to print a $Sale$.
- assign the responsibility for print a $Sale$ to the $Sale$ instance – a $Sale$ is responsible for printing itself (doing).
- this responsibility is invoked with a $print$ message to the $Sale$
- fulfilment of this responsibility requires collaboration with $SalesLineItem$ asking them to print



Patterns

- Patterns: general principles and idiomatic solutions that guide the creation of software
- each of these principles or idioms describes a problem(s) to be solved and a solution(s) to the problem(s)
- a pattern is a problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations:

Pattern Name:	name given to the pattern
Solution:	description of the solution of the problem
Problem:	description of the problem that the pattern solves

Expert, Creator, Low Coupling, High Cohesion, and Controller

Expert

Pattern Name: Expert

Solution: **Assign a responsibility to the information expert – the class that has *information* necessary to fulfill the responsibility.**

Problem: What is the most basic principle by which responsibilities are assigned in OOD?

Can we formalise it?

Example in CoCoME

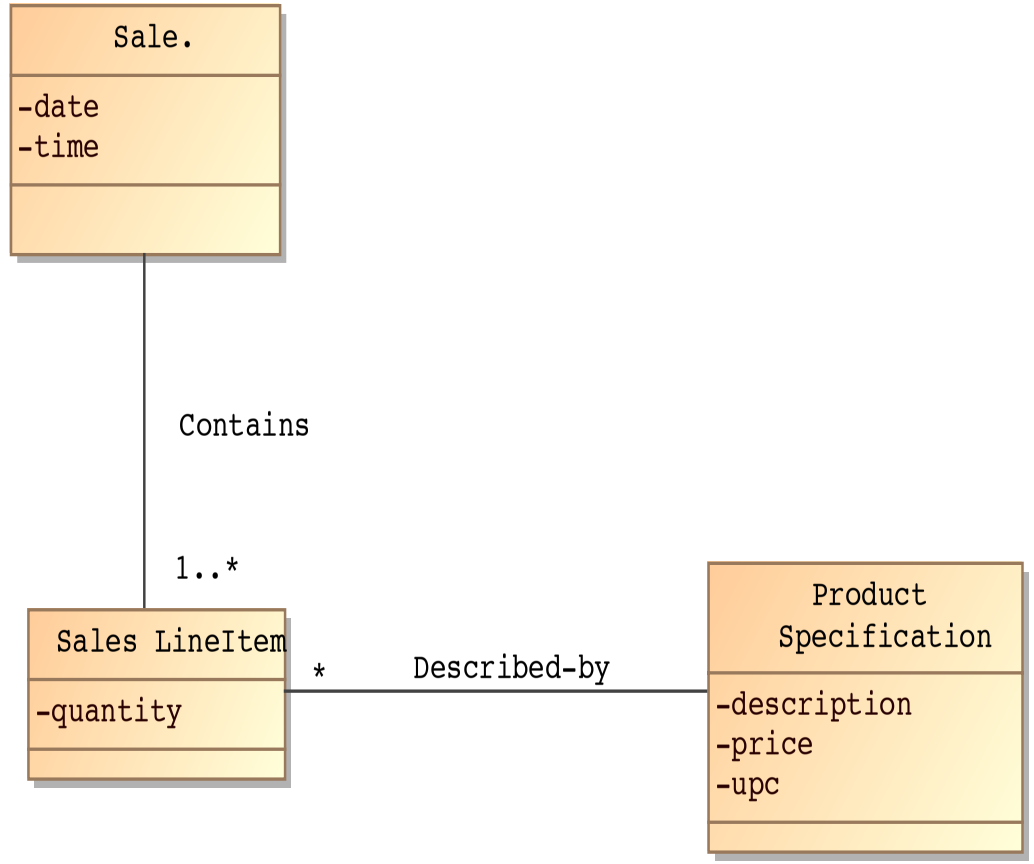
- some class needs to know the grand total of a sale
- to assign responsibilities, we had better to state the responsibility clearly

  Who should be responsible for knowing the grand total of the sale?

- Expert Pattern: we should look for the class which has the information needed to determine the total

Where to look for it?

Class Diagram



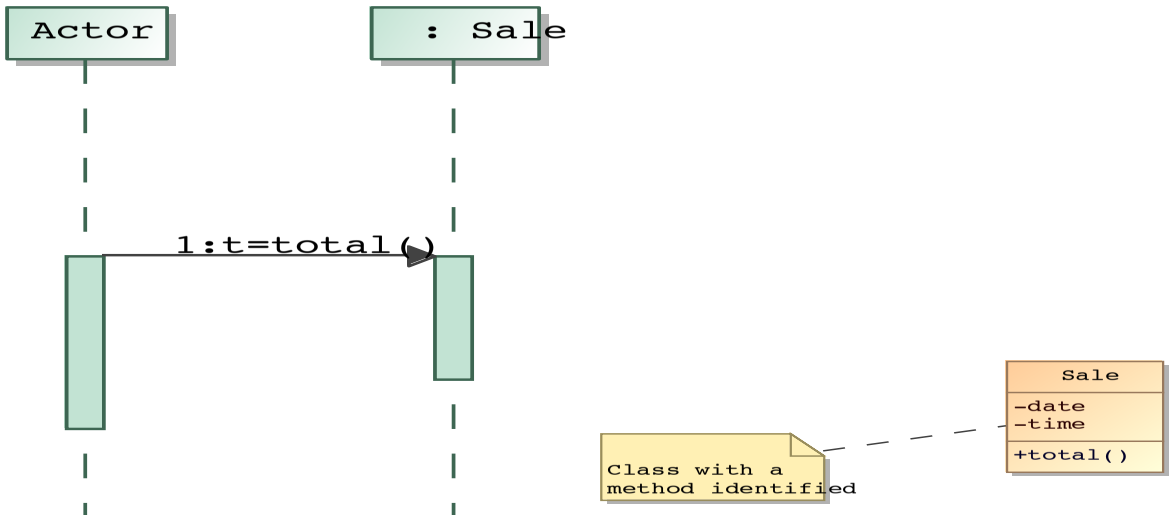
Decide the following

- what information is needed to determine the grand total?
  - all the $SalesLineItem$ instances of the $Sale$ instance, and
  - the sum of their subtotal.
- who knows this information?

Knowledge and skills of algorithms design are needed, or a formulation of total, are needed

The Expert is *Sale*

- assignment of responsibility is done in the context of the creation of collaboration diagrams.
- start working on the OSD related to the assignment of responsibility for determining the grand total to $Sale$:



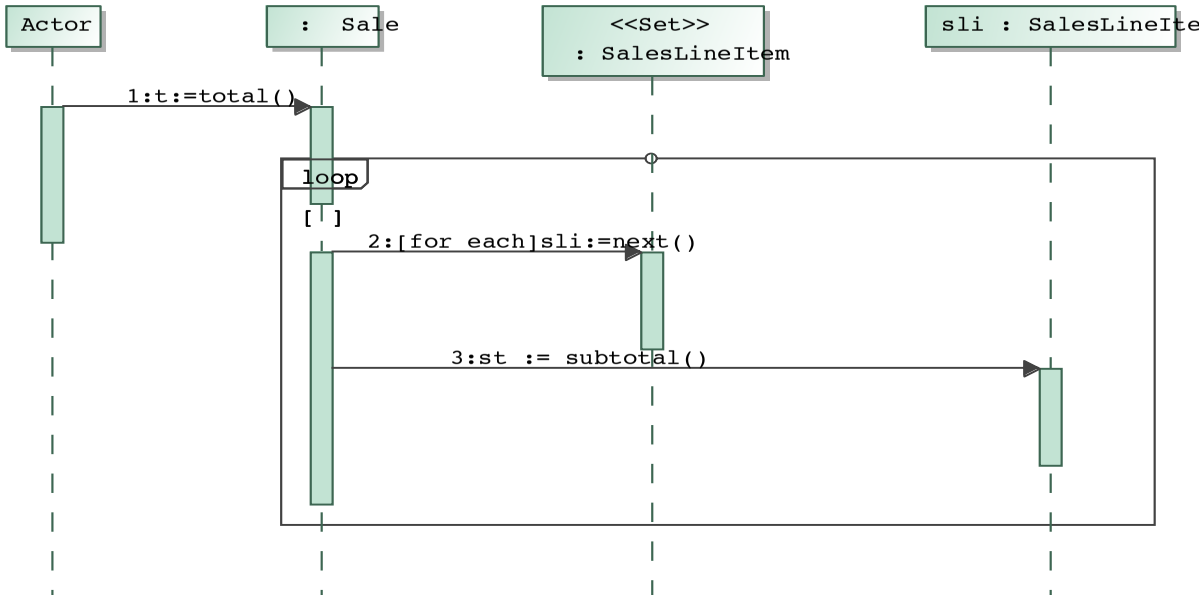
a design class diagram is also created

Continue with the Diagram

- Now, *how does the $Sale$ carry out this responsibility*?
- it needs to get the subtotal of each line item
- *who should be responsible to return the subtotal of a line item*?
- *what is the information needed to determine the line item subtotal*?
- answer: *SalesLineItem.quantity* and *ProductSpecification.price*.
- *who is the information expert for returning the subtotal of each line item?*

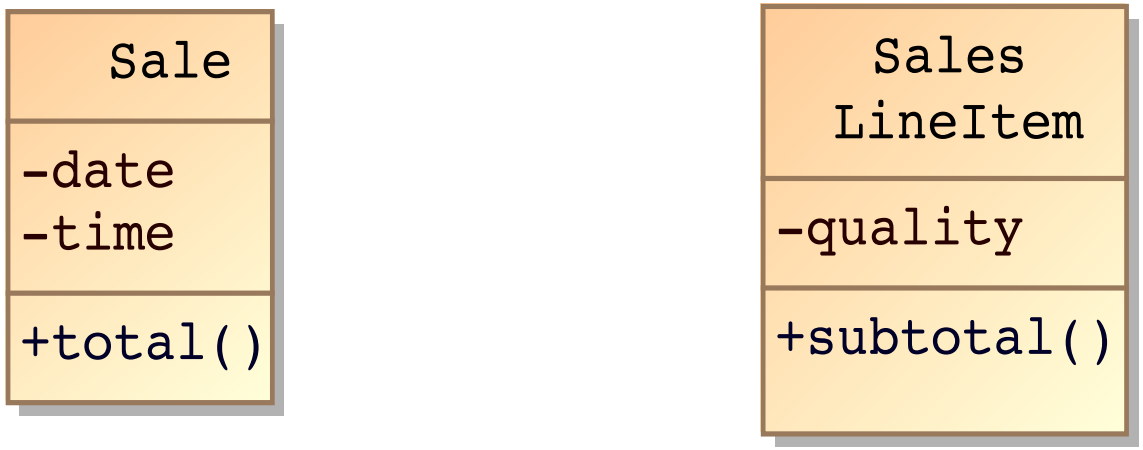
The Diagram

Answer: *SalesLineItem* is the expert for knowing and returning its subtotal, but the sale may contain a number of $SalesLineItem$ objects



design class diagram?

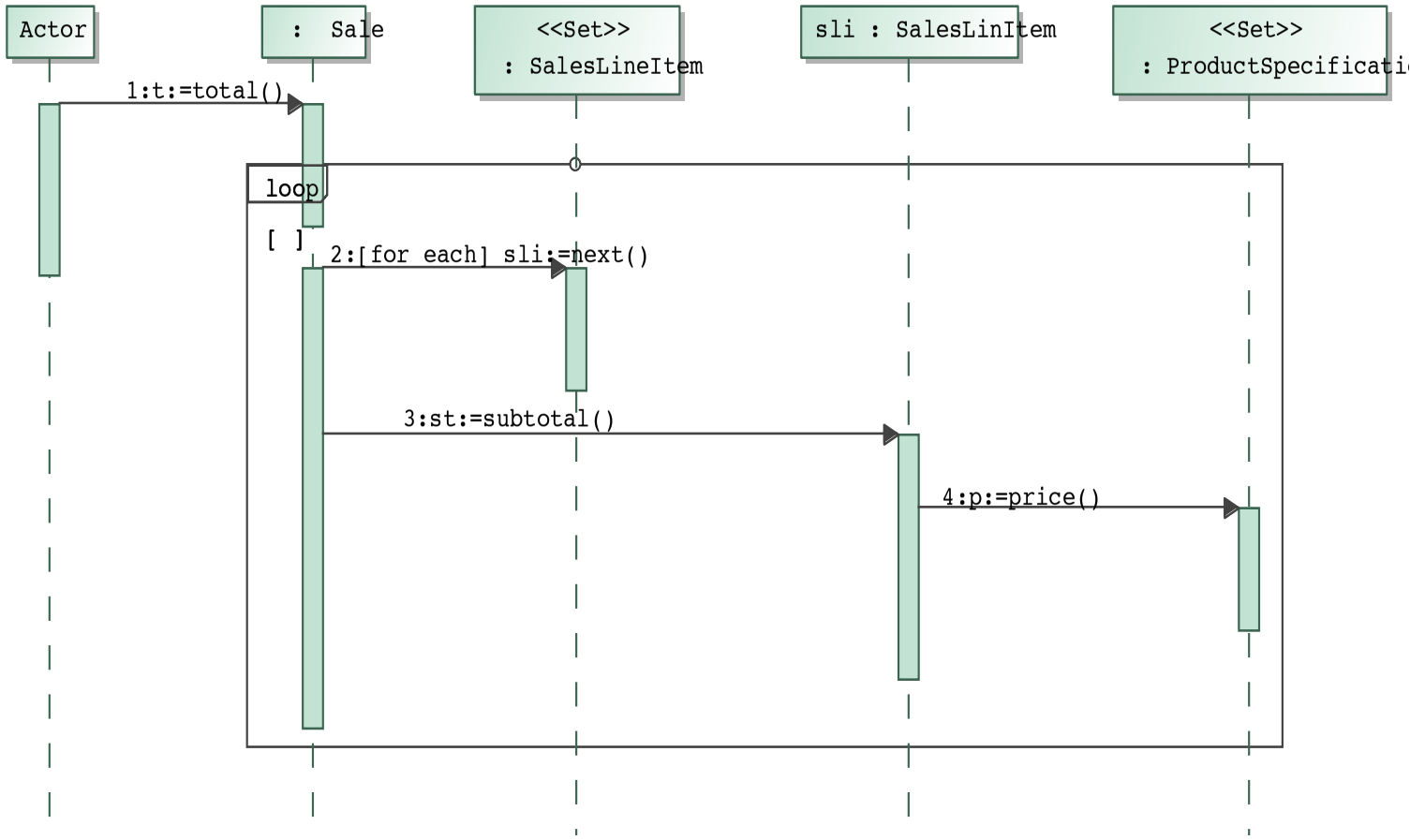
Design Class Diagram



Continue with the Diagram: *subtotal*()

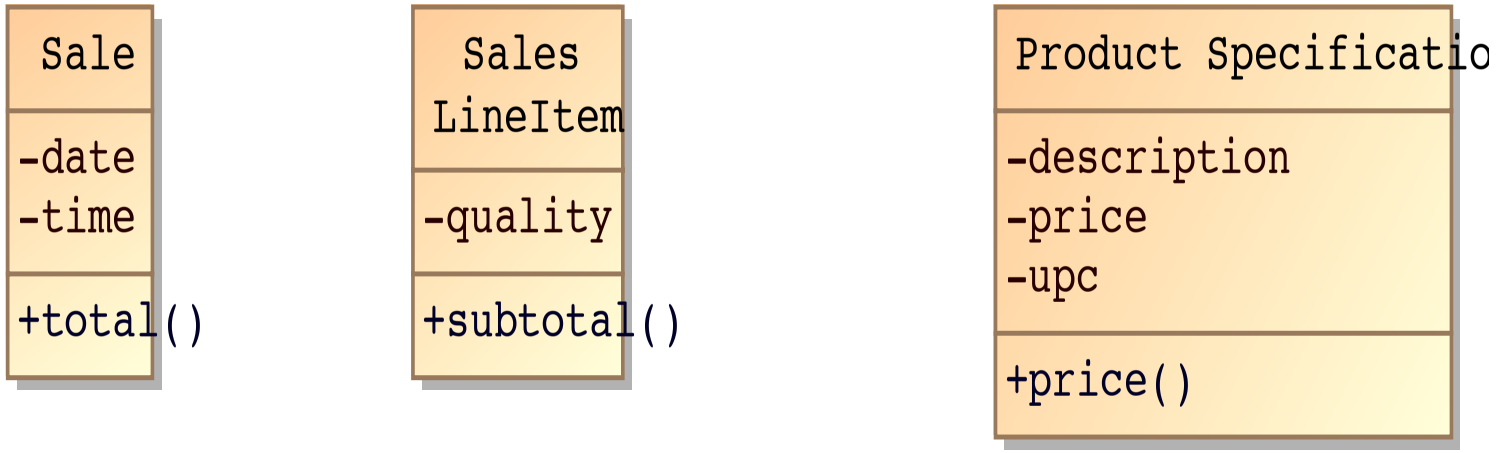
- to carry the responsibility of returning its subtotal, *SalesLineItem* needs to know where to get the price
- *who is the information expert on answering the price*?
- answer: *ProductSpecification*
- a message must be sent to it asking for its price

Complete the Design of *total*()



the design class diagram?

Design Class Diagram



associations are omitted!

Summary: Experts involved Total

Classes	Responsibilities
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

these responsibilities identified and assigned while drawing the OSD

Design Class Diagram

also create a design class diagram while creating a OSD, that refines the conceptual class diagram by

- adding methods to classes, and
- navigation direction of associations – visibility

Remarks on Expert

- expert is used more than any other pattern
- express the intuition that objects do things related to the information they have
- fulfilment of a responsibility often requires information spread across different partial experts.
- then, objects need to interact to exchange information and to share the work
- software object does those operations which are normally done by the domain object it represents (do it myself strategy)

Creator

- the creation of objects is one of the most common activities in an OO
- it is important to have a general principle for the assignment of creation responsibilities
- the object B which invokes the creation of an object A is called the creator of A

Pattern Name: Creator

Solution: **Assign class B the responsibility to create an instance of a class A if one of the following is true:**

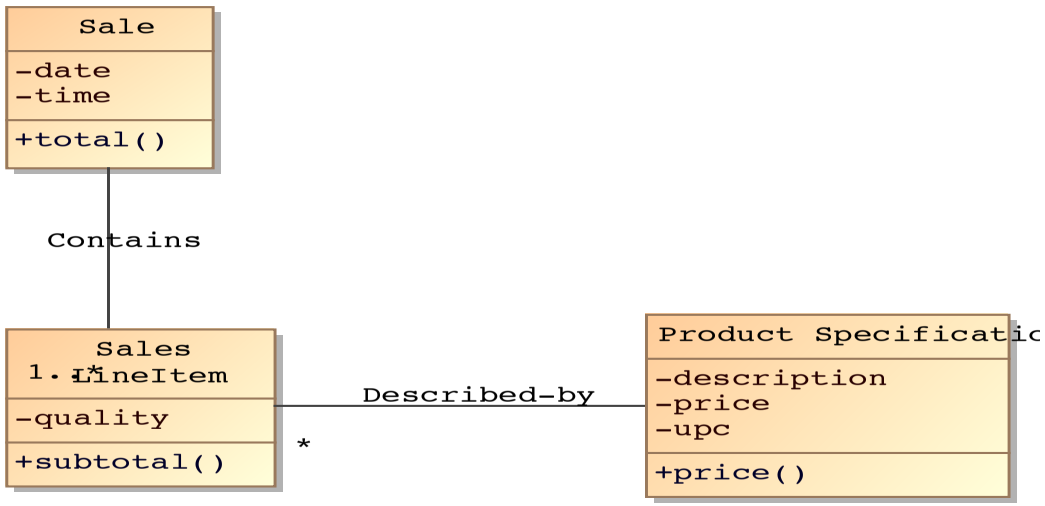
- **B aggregates A objects.**
- **B contains A objects.**
- **B records instances of A objects.**
- **B closely uses A objects.**
- **B has the initialising data that will be passed to A when it is created.**

Problem: What should be responsible for creating a new instance of some class?

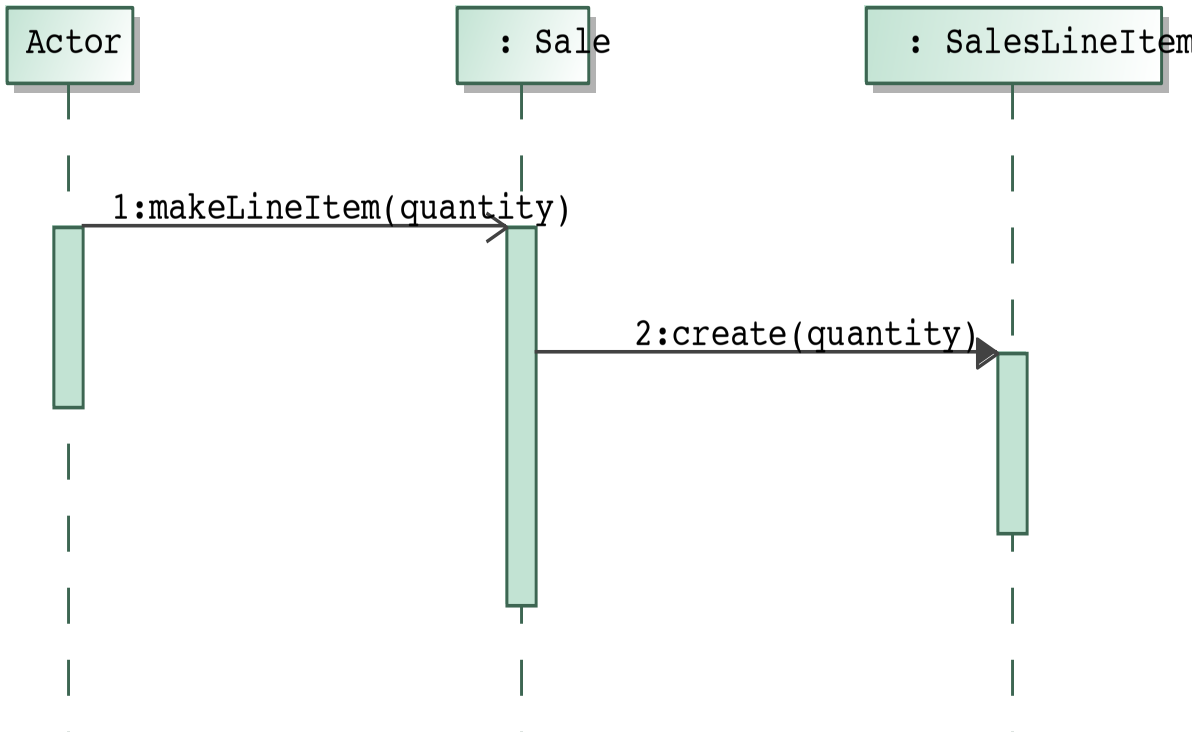
The object that has the direct reference to B is usually the creator of B

Example in CoCoME: *enterItem*()

- who should be responsible for creating *SalesLineItem*?
- by creator, we should look for a class that *aggregates*, *contains*, or *records* $SalesLineItem$



Answer: *Sale*



add *makeLineItem*() method in class *Sale* too

Question: who should be creator of the *Payment* of *Sale*?

Coupling

- coupling is a measure of how strongly one class is connected to or relies upon other classes
- a class with low coupling is not dependent on too many other classes.
- high coupling is not desirable

Low Coupling Pattern

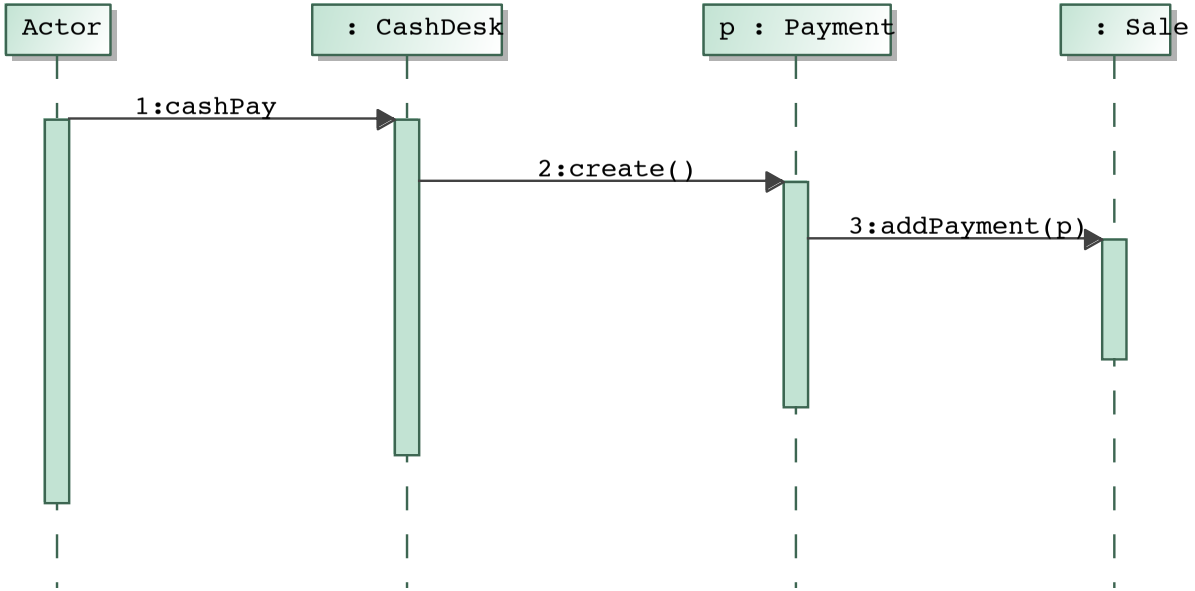
Pattern Name: Low Coupling

Solution: **Assign a responsibility so that coupling remains low**

Problem: How to support low dependency and increased reuse?

- consider the classes $Payment$, $CashDesk$ and $Sale$ in CoCoME
- assume we need to create a $Payment$ instance and associate it with the $Sale$.
- who should be responsible for this, $CashDesk$ or $Sale$?

If the *CashDesk*...



We need an extra link from $CashDesk$ to $Payment$

So $Sale$ is a better choice!

Cohesion

Cohesion is a measure of how strongly related and focused the responsibilities of a class are.

- classes with high cohesion have highly related functionalities, and does not do a tremendous amount of work
- they have a small number of methods with simple but highly related functionality
- classes with low cohesion are not desirable

High Cohesion Pattern

Good OOD is to assign responsibilities to classes that. . .

- are naturally and strongly related to the responsibilities, and
- every class has something to do but does not have too much to do.

Pattern Name:	High Cohesion
Solution:	Assign a responsibility so that cohesion remains high
Problem:	How to keep complexity manageable?

high coupling also contributes to low cohesion!

Example

- $CashDesk$ can take on part of the responsibility for carrying out the $cashPay()$
- But $CashDesk$ is used as handling the interface methods, it would better if it does not act as the creator of $Payment$
- a design that delegates the payment creation responsibility to the $Sale$ would more cohesive

Controller Pattern

Pattern Name: Controller

Solution: Assign the responsibility for handling an input to a class representing one of the following choices:

- Represents the “overall component” (*facade controller*).
- Represents the overall business (*facade controller*).
- Represents something in the real-world that is active that might be involved in the task (*role controller*).
- Represents an artificial handler of all input events of a use case, (*use-case controller*).

Problem: Who should be responsible for handling an external input event?

Design of CoCoME

1. Create a separate diagram for each interface methods whose contracts are defined:

   For each interface method $m()$, make a diagram with it as message to the interface class.

2. If the diagram gets complex, split it into smaller diagrams.

use the contracts and conceptual class diagram, and apply the GRASP

For each use case, we use an interface class as the controller

Design $enterItem$

Recall the contract

Responsibilities: Record sale of an item, display item description and price.

Pre-conditions: UPC is known to the system.

Post-conditions:

- if a new sale, a $Sale$ was created.
- if a new sale, the new $Sale$ was associated with the $CashDesk$.
- a $SalesLineItem$ was created.
- $SalesLineItem.quantity$ was set to $quantity$.
- $SalesLineItem$ was associated with a $ProductSpecification$, based on UPC match.
- $SalesLineItem$ is added to $Sale$

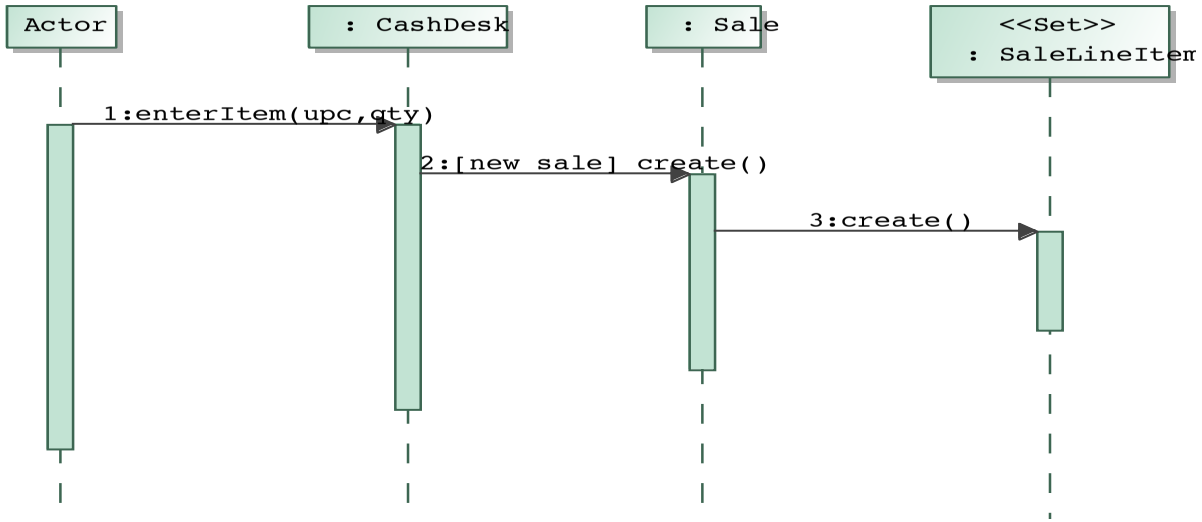
Displaying Item Description and Price

- model objects are not responsible for display of information
- these are the responsibilities of GUI objects
  - access to data known by model objects, and
  - can send messages to model objects.
- but, model objects do not send messages to interface objects in order to keep the system independent of the interface.
- This is a matter of interface design and we leave it for later discussion.

Creating a New $Sale$

The post conditions in the contract of $enterItem$ include:

- If a new sale, a $Sale$ was created.
- If a new sale, the new $Sale$ was associated with the $CashDesk$.



an empty «set» of $SaleLineItem$ is also created by $Sale$ (Creator)

Creating a New $SalesLineItem$

Other post-conditions in the contract for $enterItem()$ include:

- A $SalesLineItem$ was created.
- The $SalesLineItem$ was associated with the $Sale$.
- $SalesLineItem.quantity$ was set to qty.
- The $SalesLineItem$ was associated with the $ProductSpecification$, based on upc match.

These indicate the responsibility to create a $SalesLineItem$ instance. Who should be creator?

$Sale$

- $SalesLineItem$ is then also associated to $Sale$ being added into the collection.
- $CashDesk$ must pass the quantity to $Sale$, which must pass it in the creation message to the $SalesLineItem$.

Finding a $ProductSpecification$

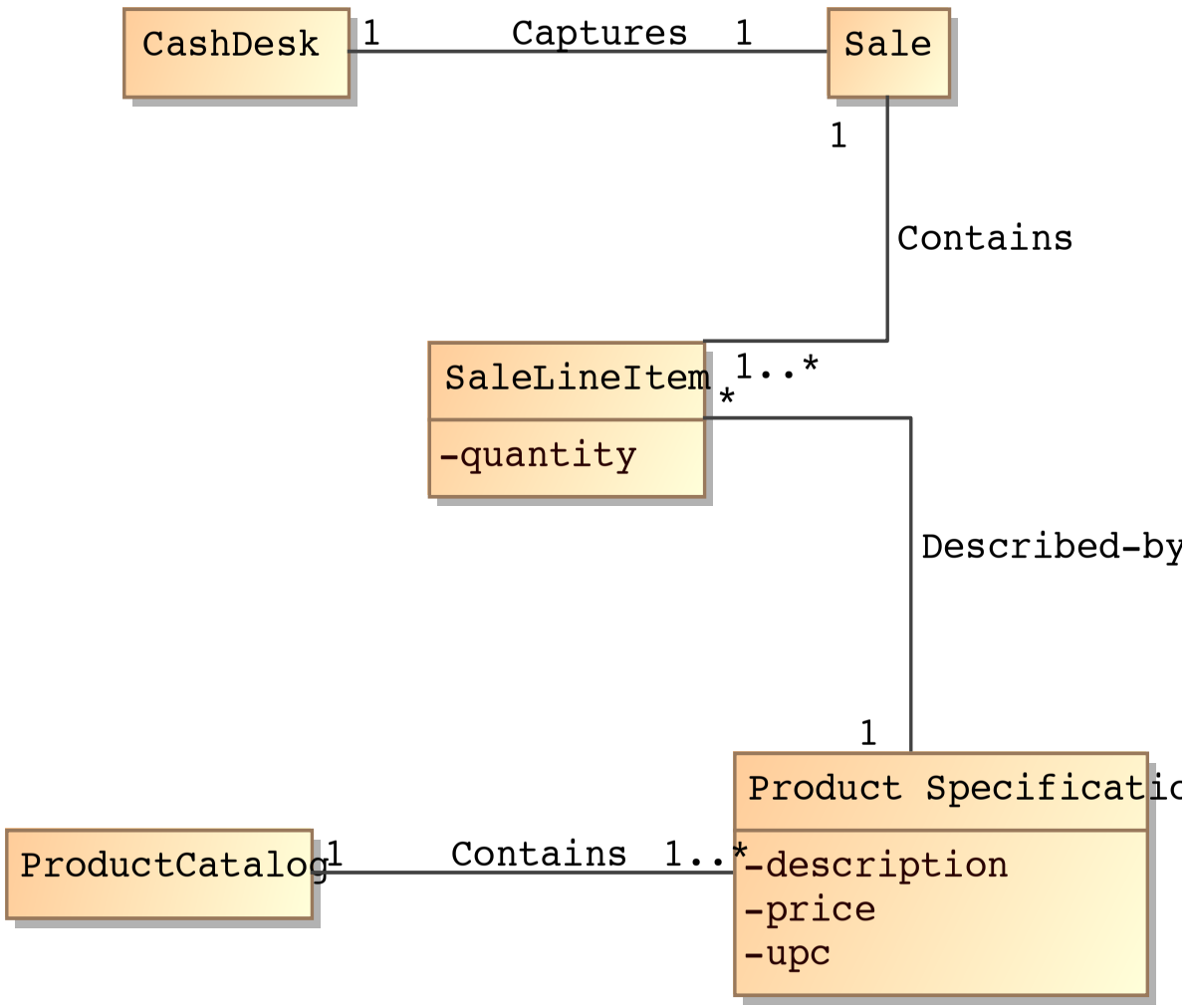
The postconditions of $enterItem$ also requires that

- the newly created $SalesLineItem$ to be associated with the $ProductSpecification$ based on upc match
- message $makeLineItem()$ to the $Sale$ should include the $ProductSpecification$
- where to get $ProductSpectification$ *before* $makeItem(spec, qty)$ is sent to the $Sale$?

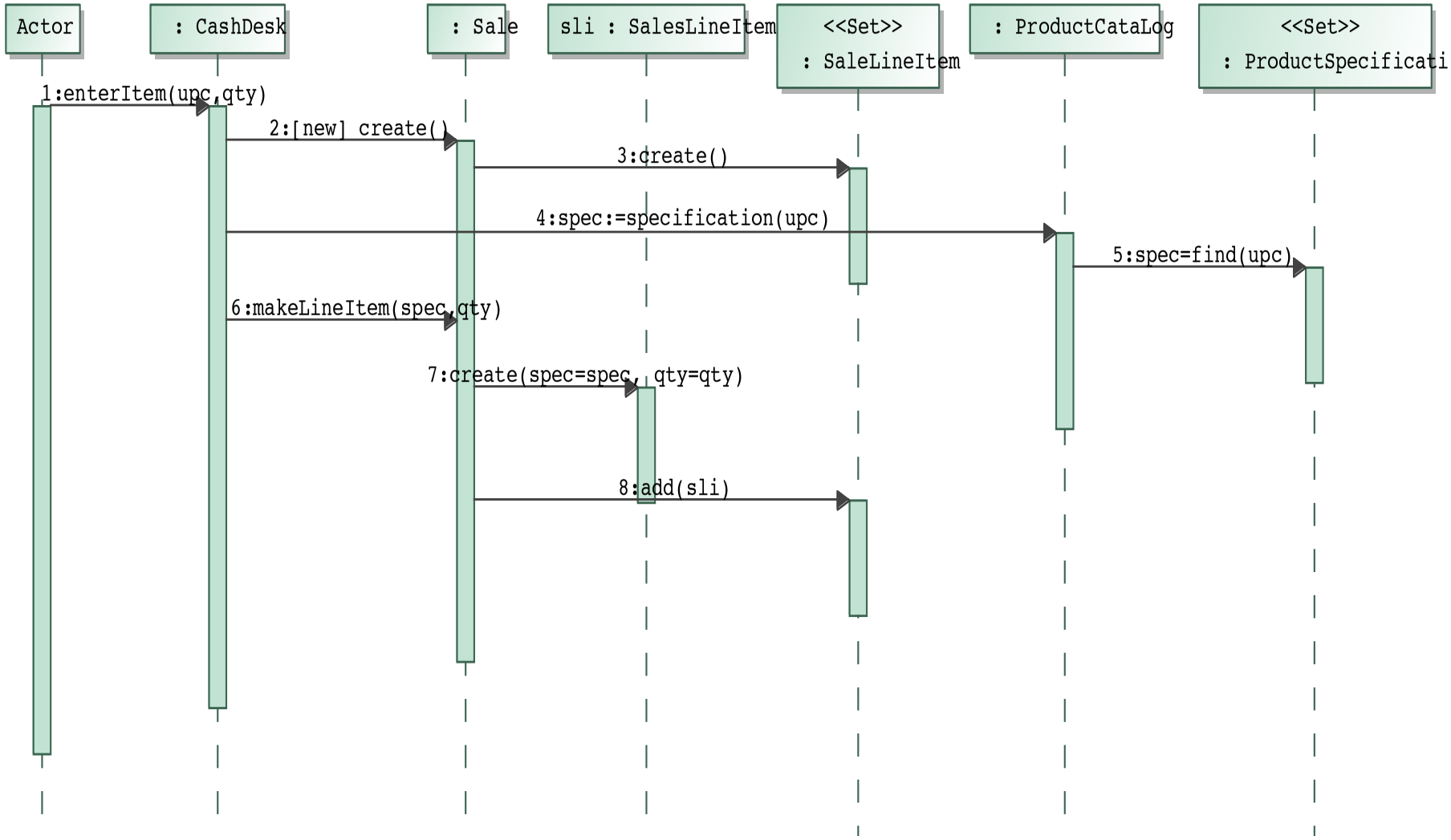
  who should be responsible for knowing and looking up a $ProductSpecification$, based on UPC match?

Answer: The $ProductCatalog$

Part of the Conceptual Model



Complete Design of *enterItem*



Pseudo-code for $enterItem()$

```
CashDesk--enterItem(upc,qty) {
  if new sale then <<new>>Sale.create()
  spec:=ProductCataLog.specification(upc)
  Sale.makeLineItem(spec,qty)
}
```

NB:

- OOD does do top-down functional decomposition
- also, the decomposition is more systematic
- reason: decomposition of the state space has already been largely done when building the class diagram.

Design $finishSale$

Name: endsale().

Responsibilities: Record that it is the end of entry of sale items, and display sale total.

Pre-conditions: UPC is known to the system.

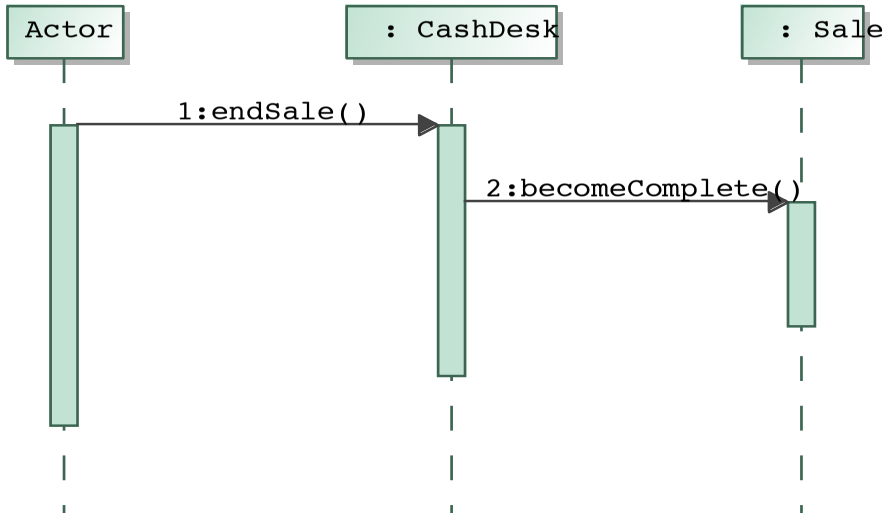
Post-conditions:

- $Sale.isComplete$ was set to $true$ (attribute modification).

Responsibilities: calculate $total()$ and the postcondition

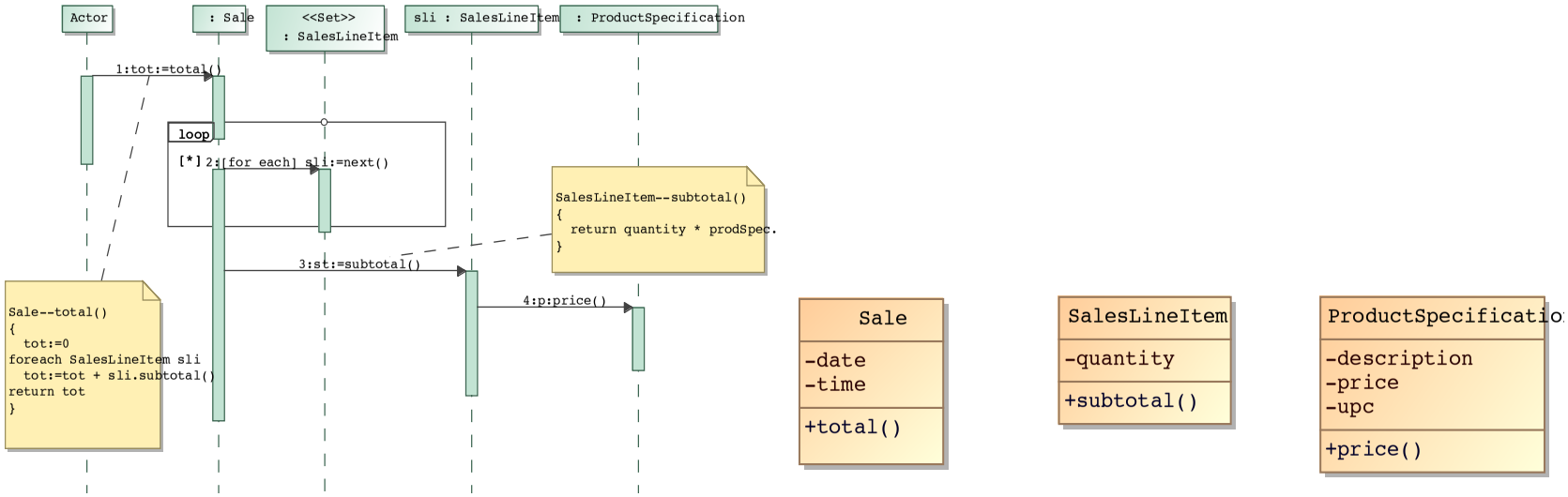
Set $Sale.isComplete$

who should be responsible for setting the $isComplete$ attribute of the $Sale$ to true?



Calculating the *Sale* total

- not design GUI to display the total
- but, design the calculation to be displayed



Who will send the *total* message to the *Sale*?

Design $cashPay()$

Name: cashPay(amount: Quantity).

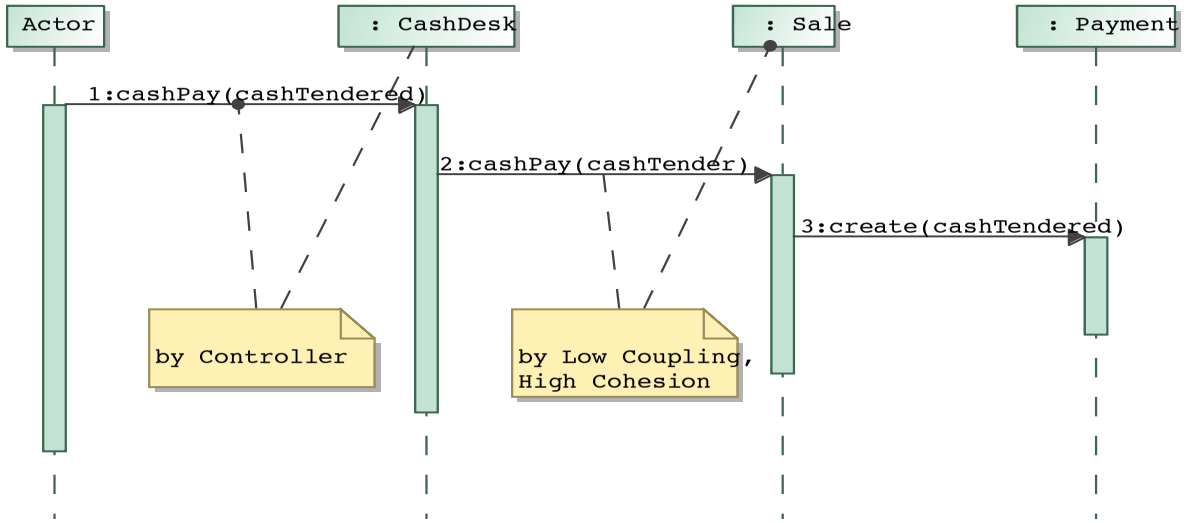
Responsibilities: Record the payment, calculate balance and print receipt.

Post-conditions:

- A $Payment$ was created (instance creation).
- $Payment.amountTendered$ was set to $amount$.
- The $Payment$ was associated with the $Sale$.
- The $Sale$ was associated with the $Store$, to add it to the historical log of completed sales.

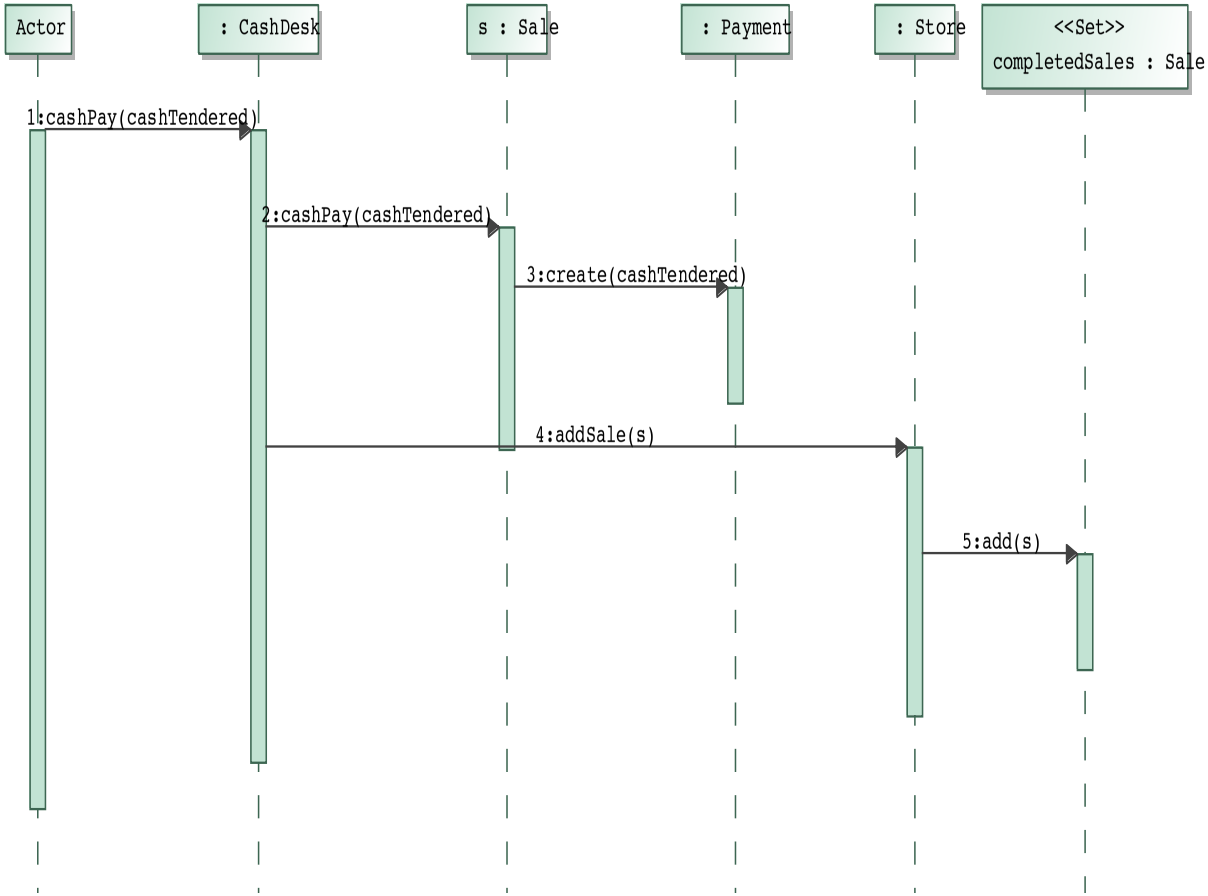
Creating payment

- by creator, low coupling and high cohesion, $Sale$ is the creator of $Payment$



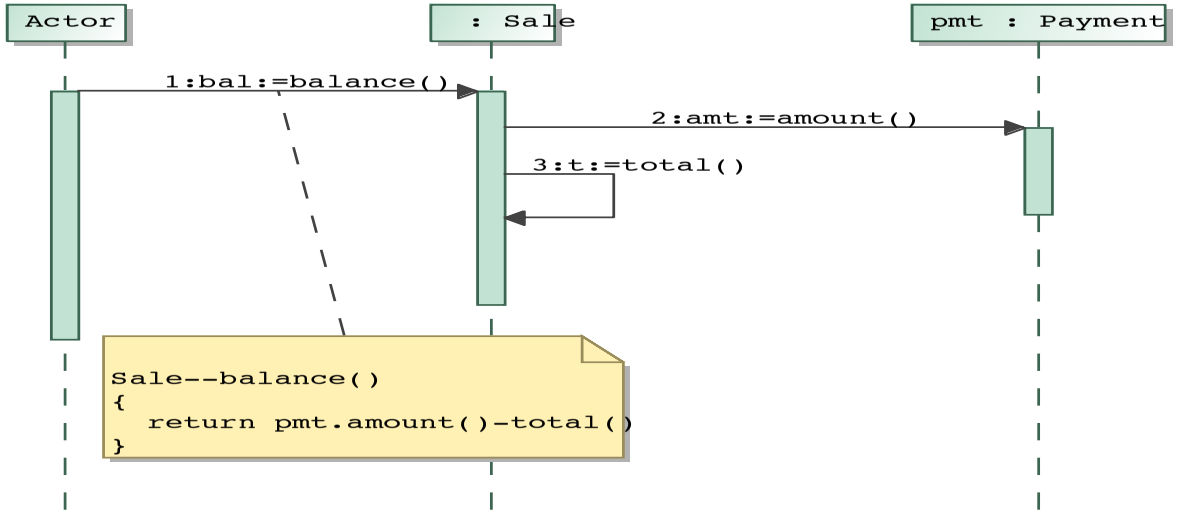
Logging the *Sale*

By expert



Calculating Balance

- by expert, either $Sale$ or $Payment$
- by low coupling, $Sale$ is better.



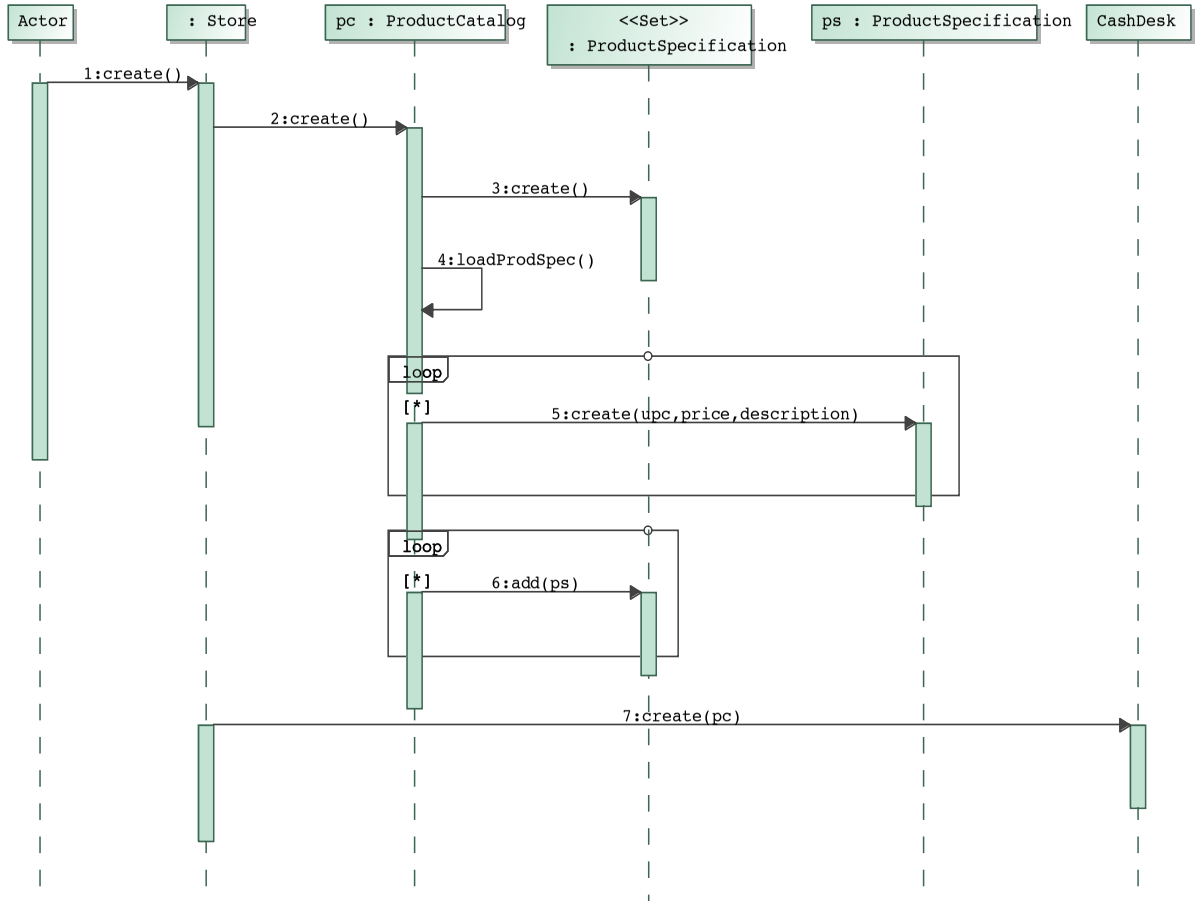
$StartUp$

- all components must be initialized by a $StartUp()$ operation before any other operations can be used
- better to delay the design of $StartUp()$ until after the other interface methods are designed

Post Conditions of $startUp$

- A Store, CashDesk, ProductCatalog and ProductSpecification were created (instance creation).
- $ProductCatalog$ was associated with $ProductSpecification$ (association formed).
- $Store$ was associated with $ProductCatalog$ (association formed).
- $Store$ was associated with $CashDesk$ (association formed).
- $CashDesk$ was associated with $ProductCatalog$ (association formed)

Design *StartUp()*



Conclusions

- OO design = creation of interaction diagrams
- OO implementation = translation diagrams into code.
- OO design is the most technical and demands a lot of skills.
- OO implementation is quite simple, provided that you have got a good design and knows the programming languages.
- incremental and iterative design is supported
- component-based design?
- formal verification and analysis?
- tool support?