Software Requirements Analysis
– Use-Case Sequence Diagram & Operations

Zhiming Liu
zhimingliu88@swu.edu.cn
http://computer.swu.edu.cn
Centre for Research and Innovation in Software Engineering (RISE)
School of Computer and Information Science
Southwest University
Chongqing, China

Aims & Objectives

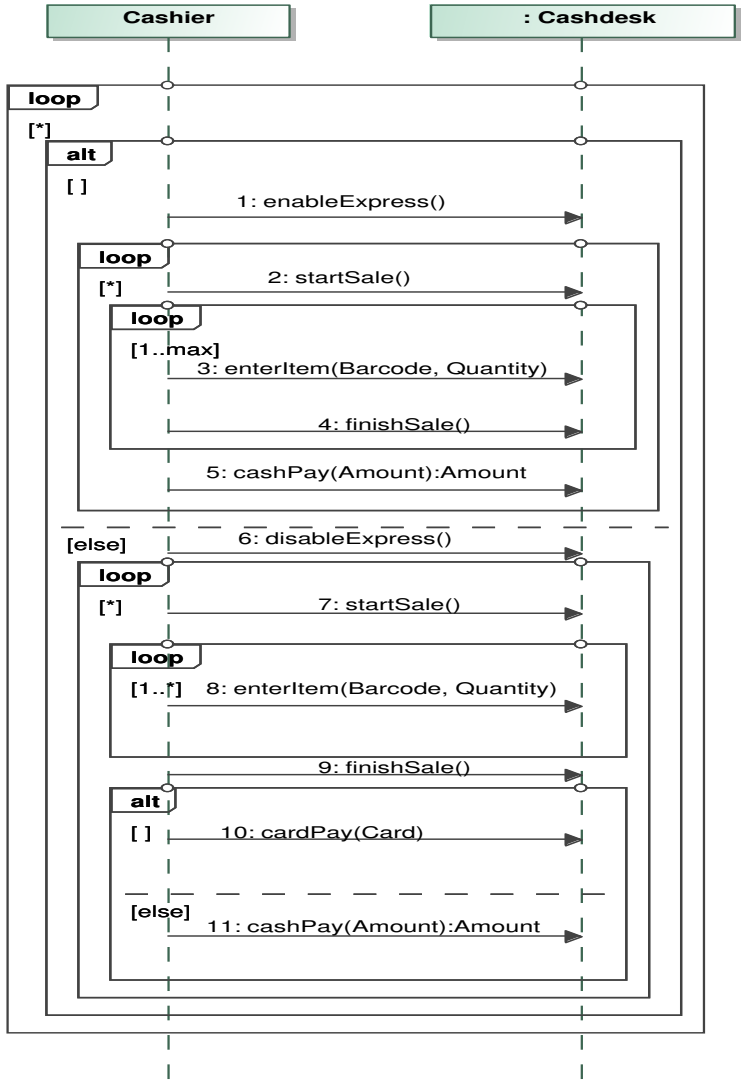
- A use case represents a business process in the application domain – real world
- A use case models the interaction process between the actors and software component – digital world
- We analysis each use case to
  - identify all possible sequences of interaction events, and represent them a sequence diagram
  - Analyse the changes of the system states by the interactions, and represented as contracts of operations
  - **Artefacts: the use case sequence diagrams**, and **the contracts of the use case operations**

Interactions in Use Case

Recall that the typical course of events of a use case describes how actors interact with the component (of the system)

- in each interaction, an actor generates an **events** to the component to request some operation in response
- the component carries out the operation requested
- in OO, synchronization of the two parties in an interaction is **method invocation**
- the **order** in which actors call the operations is **important**,
- different executions of the process may call different sequence of operations
- The use case diagram represents all the sequences of events that actors are allowed to generate in the use case

Sequence Diagram of Process Sale



Remarks

1. In Visual Paradigm, ‘stick figure” is used to represent an actor

Remarks

1. In Visual Paradigm, ‘stick figure” is used to represent an actor
2. Notice the Combined Fragments for "loop", and "alternative"

Remarks

1. In Visual Paradigm, ‘stick figure” is used to represent an actor
2. Notice the Combined Fragments for "loop", and "alternative"
3. Visual Paradigm supports more advanced features

Remarks

1. In Visual Paradigm, ‘stick figure” is used to represent an actor
2. Notice the Combined Fragments for "loop", and "alternative"
3. Visual Paradigm supports more advanced features
4. We learn the details of UML diagrams through practice with the tool

Operations of Use Case

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

Operations of Use Case

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

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

Operations of Use Case

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

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

- We use operation, message, and event to mean the same

Remarks:

1. names of methods should be expressed at the level of intent – abstraction

Remarks:

1. names of methods should be expressed at the level of intent – abstraction
2. use verb phrases to name methods

   e.g. use *enterItem*() and *finishSale*() rather than *enterKeyPressed*() and *enterReturnKey*()

Remarks:

1. names of methods should be expressed at the level of intent – abstraction
2. use verb phrases to name methods

   e.g. use *enterItem*() and *finishSale*() rather than *enterKeyPressed*() and *enterReturnKey*()
3. Parameters can be added, e.g. *enterItem*($id : UPC, qty : Real$),

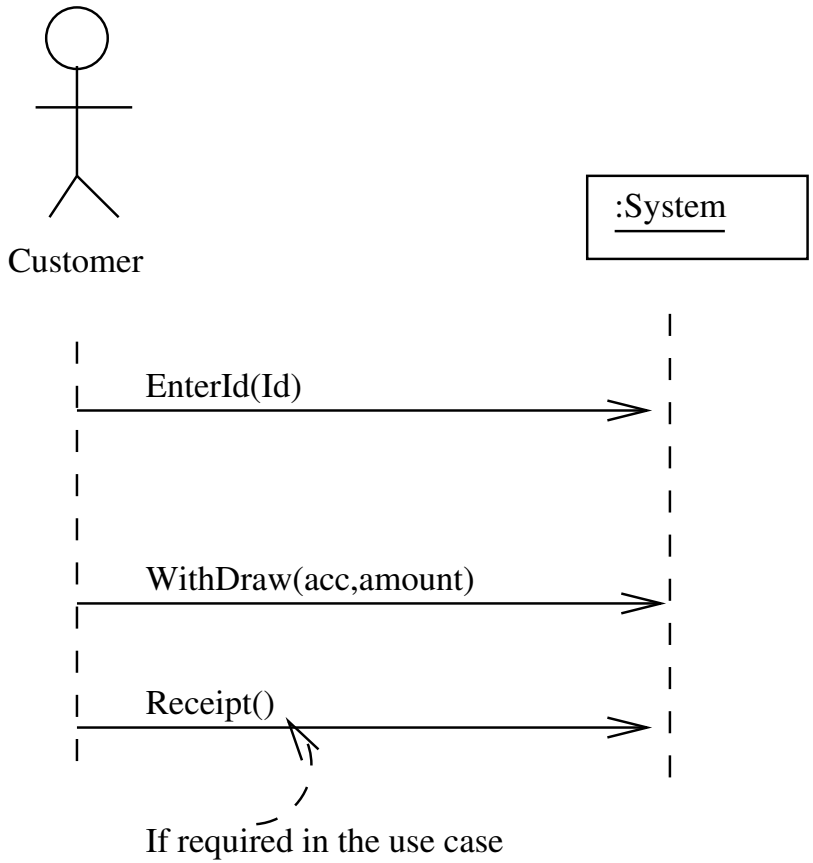
Remarks:

1. names of methods should be expressed at the level of intent – abstraction
2. use verb phrases to name methods

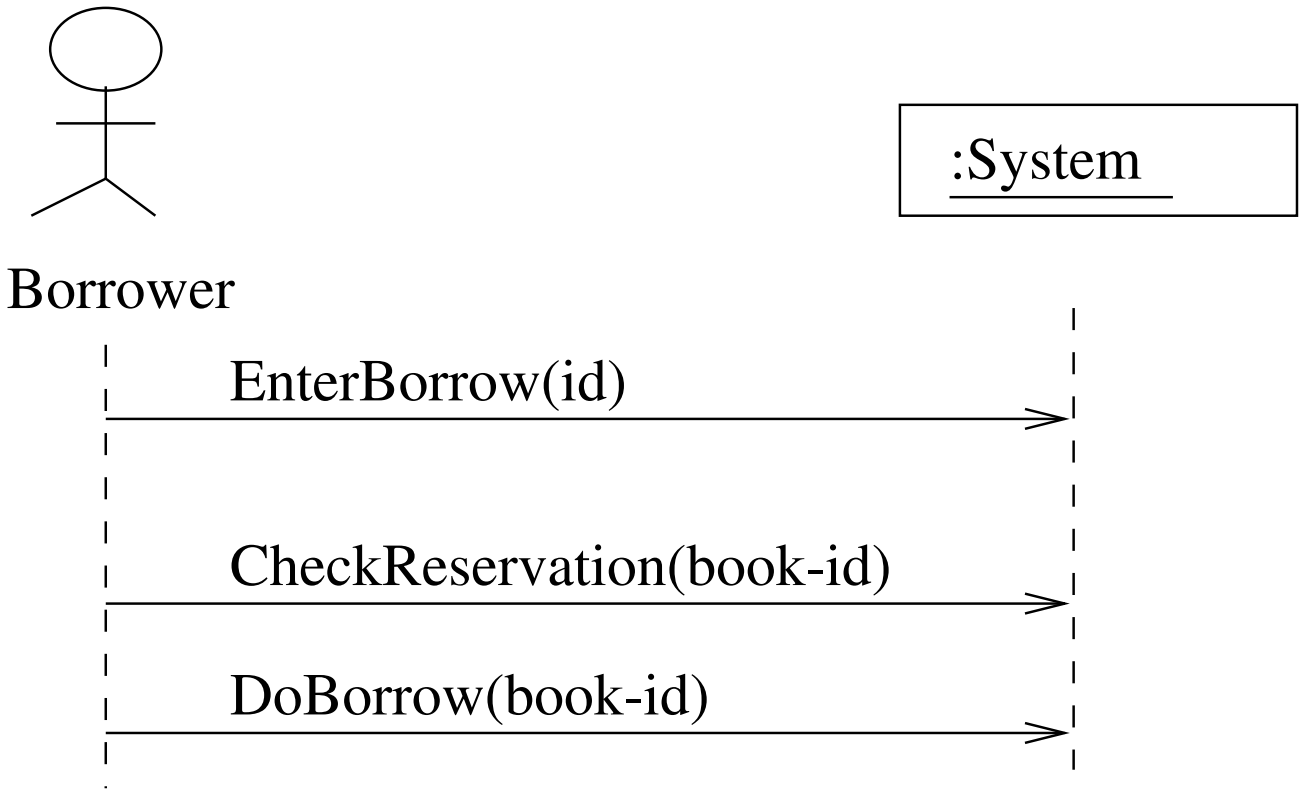
   e.g. use *enterItem*() and *finishSale*() rather than *enterKeyPressed*() and *enterReturnKey*()
3. Parameters can be added, e.g. *enterItem*(*id* : *UPC*, *qty* : *Real*),

   but they are more important in later stages, analysis of operation contracts and design

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 and described in terms of **state changes** of the component

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 and described in terms of **state changes** of the component

We use **contracts of methods** to describe the functionality of methods

States and Behaviour

- A **state** of the system at a time consists of the objects of classes existing, values of attributes of these objects, and links between the objects at that time

States and Behaviour

- A **state** of the system 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 system execution

States and Behaviour

- A **state** of the system 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 system execution
- A state is an **object diagram** of the class diagram

States and Behaviour

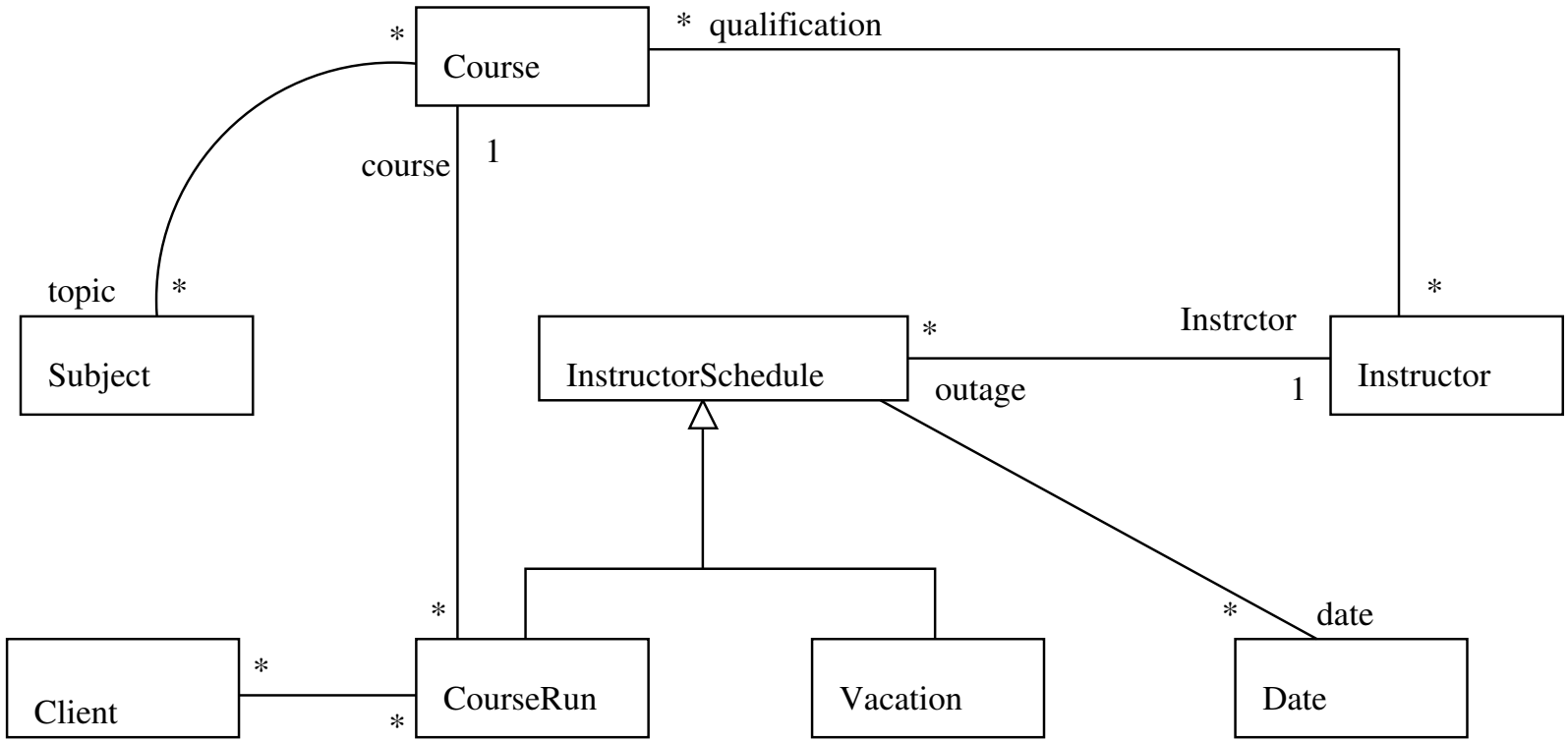
- A **state** of the system at a time consists of the objects of classes existing, values of attributes of these objects, and links between the objects at that time
- A state at a time is a snapshot of the system execution
- A state is an **object diagram** of the class diagram
- **behaviour** is about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

States and Behaviour

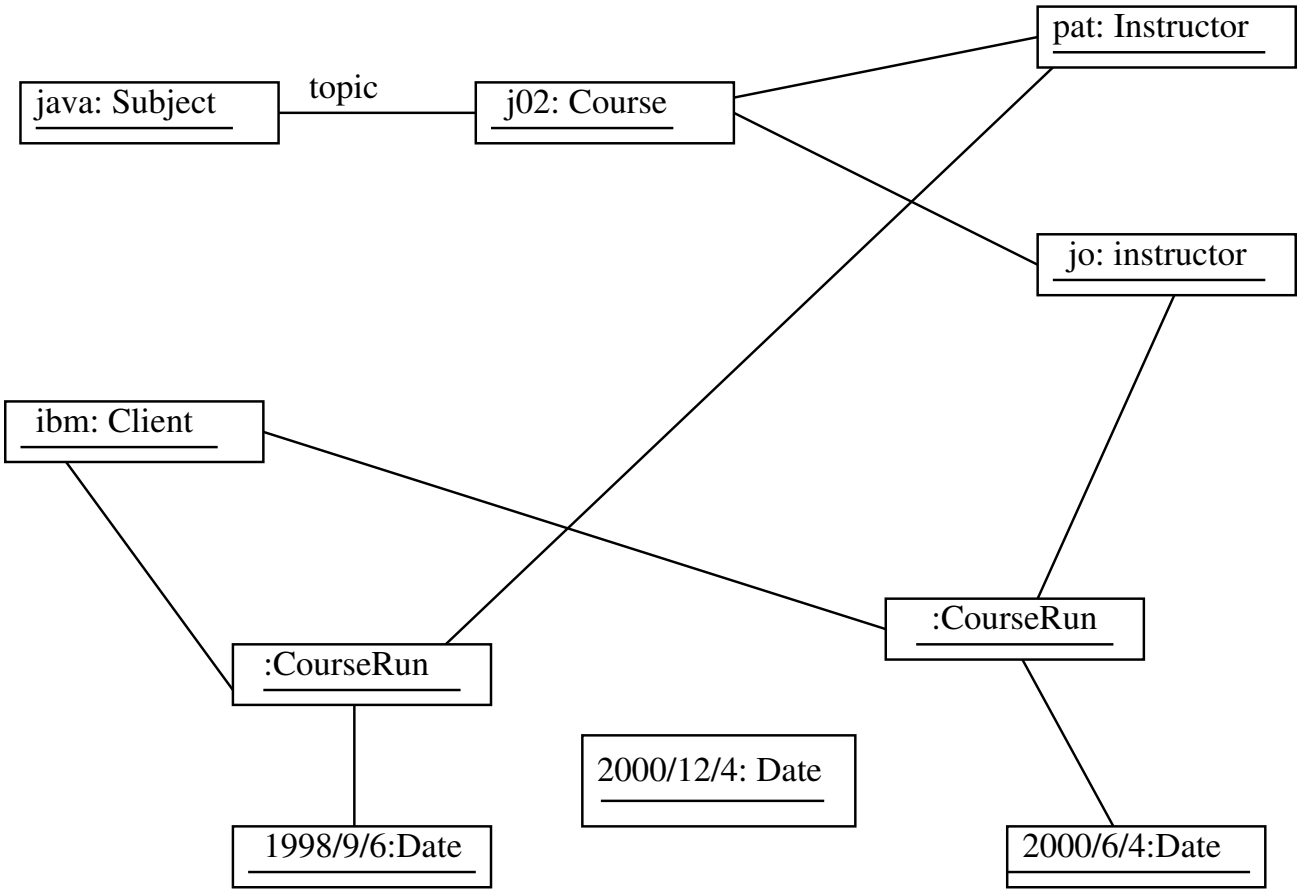
- A **state** of the system at a time consists of the objects of classes existing, values of attributes of these objects, and links between the objects at that time
- A state at a time is a snapshot of the system execution
- A state is an **object diagram** of the class diagram
- **behaviour** is about 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: teaching compony system

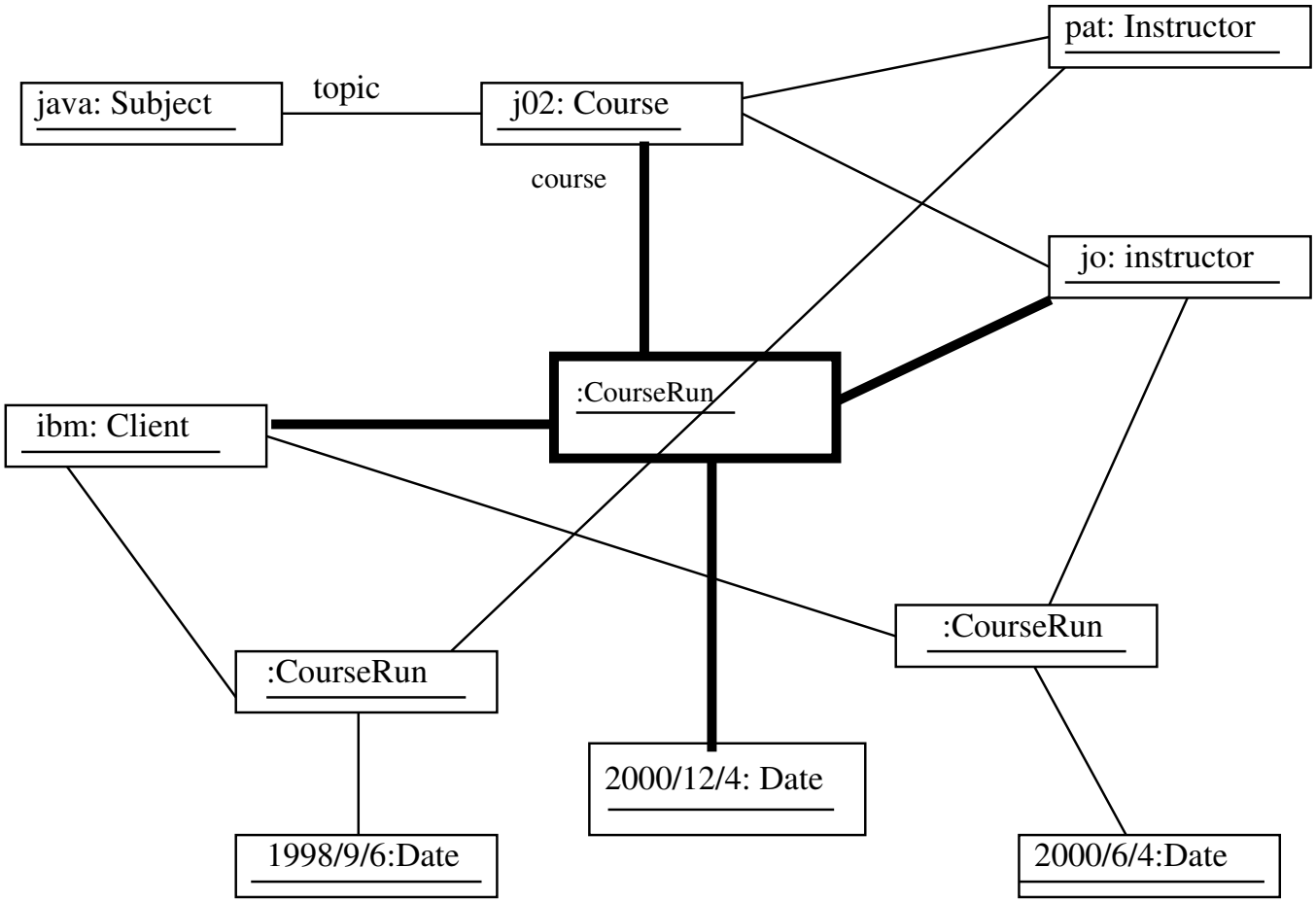


Example: a state of the teaching compony

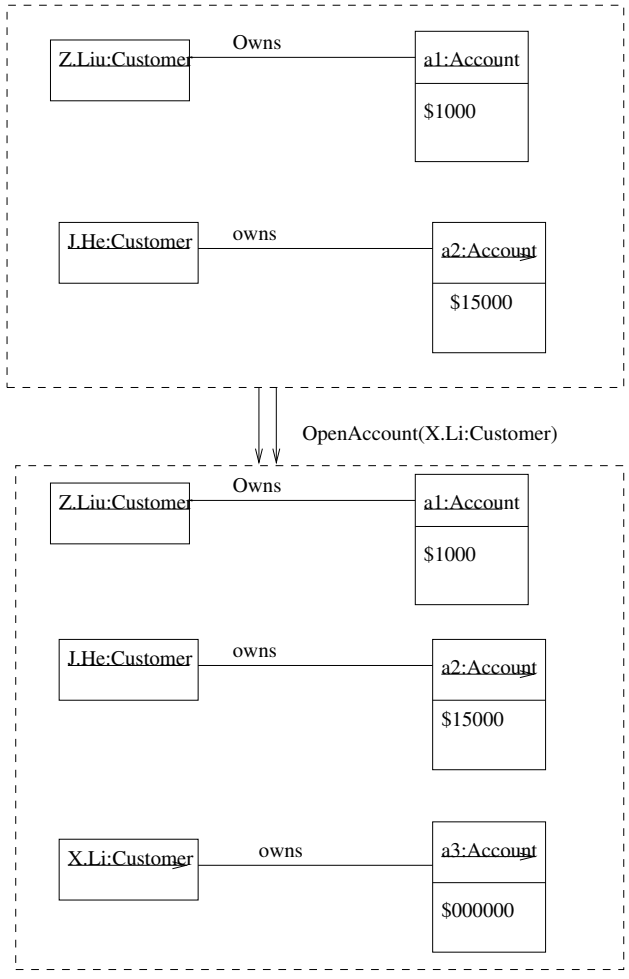


Example: state change

Consider the state after $setUp(java, 2000/12/4, ibm)$



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**

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

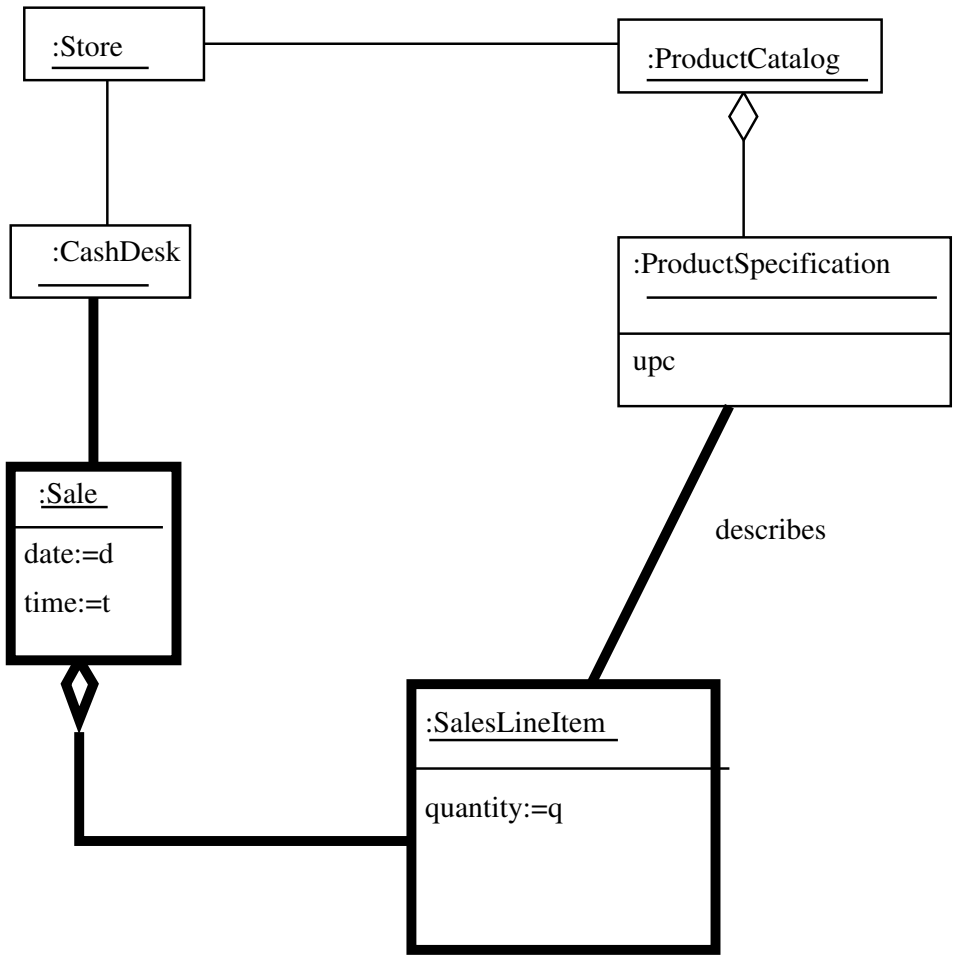
- **pre-conditions** are the conditions that the states are assumed to satisfy **before** the execution of the operation
- **post-conditions** are the conditions that the states have to satisfy **when** the execution operation **has finished**.

Example in POST

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?

Documenting Contracts

Contract

Name: Name of operation, and parameters.

Responsibilities: An informal description of the responsibility this operation must fulfill.

Cross References: System function ref number, use cases, etc.

Note: Design notes, algorithms, and so on.

Exceptions: Exceptional cases. that are sent outside of the system.

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.

Cross References: Use Cases: Process Sale

Note: Use superfast database access.

Exceptions: If upc is invalid, indicate an error.

Pre-conditions: upc is valid

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: finishSale().

Cross References: Use Cases: Process sale

Exceptions: If a sale is not underway, indicate that it was an error.

Pre-conditions:

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: sale is not complete, amount is less than the sale total.

Post-Conditions: sale.isComplete & amount $\geq$ sale.total

Post-Conditions:

- A *Payment* was created (instance creation).
- *Payment.amountTendered* was set to *amount* (attribute medication).
- 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

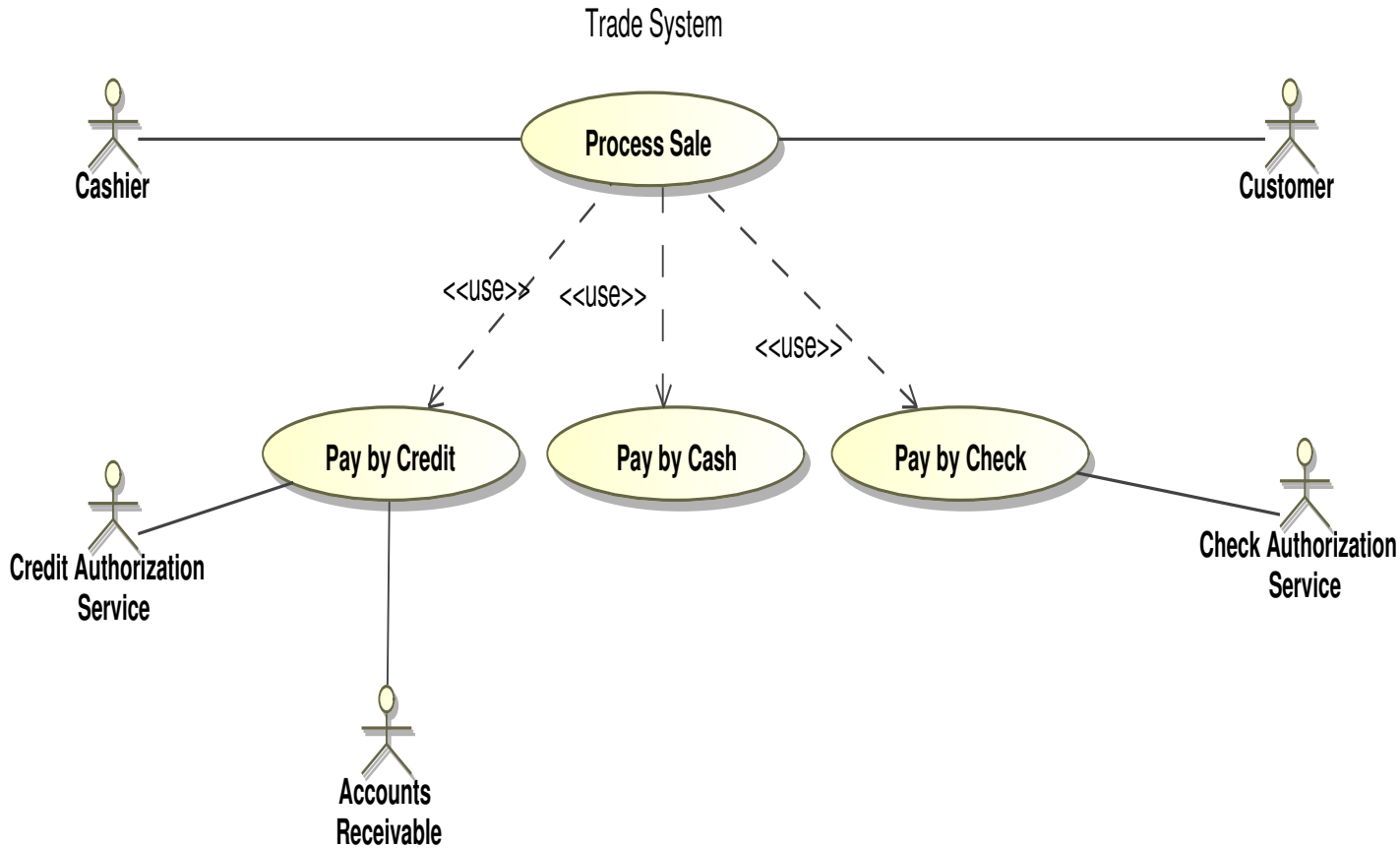
Post-conditions:

- A *Sore*, *CashDesk*, *ProductCatalog and* *ProductSpecification* were created.
- *ProductCatalog* was associated with *ProductSpecification*.
- *Store* was associated with *ProductCatalog*.
- *Store* was associated with *Casdesk*.
- *CashDesk* was associated with *ProductCatalog*

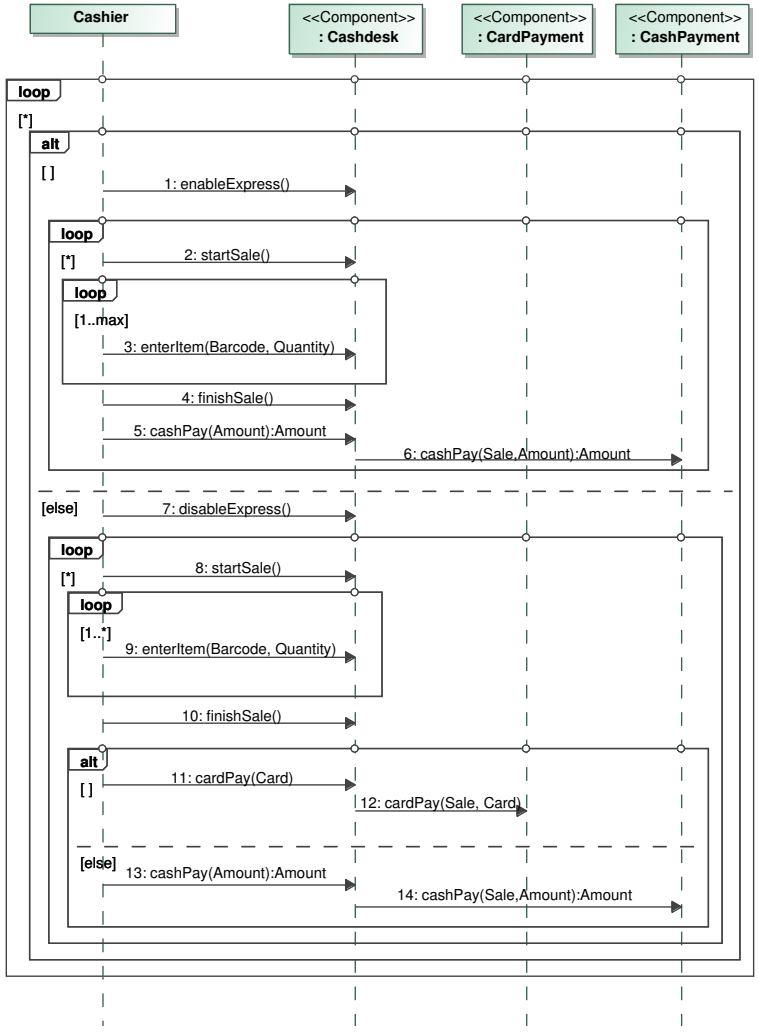
Contracts and other Documents

- 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.

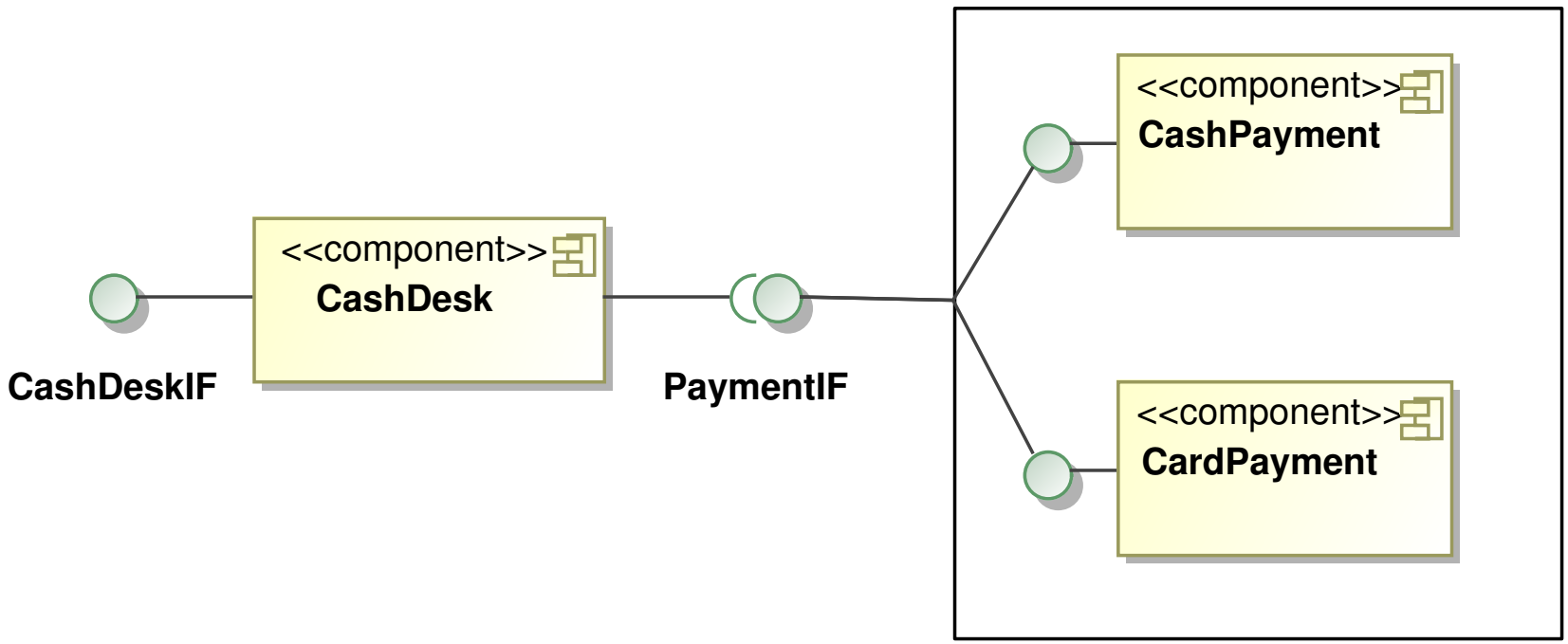
Use Case Decomposition and Composition (Advanced Material)



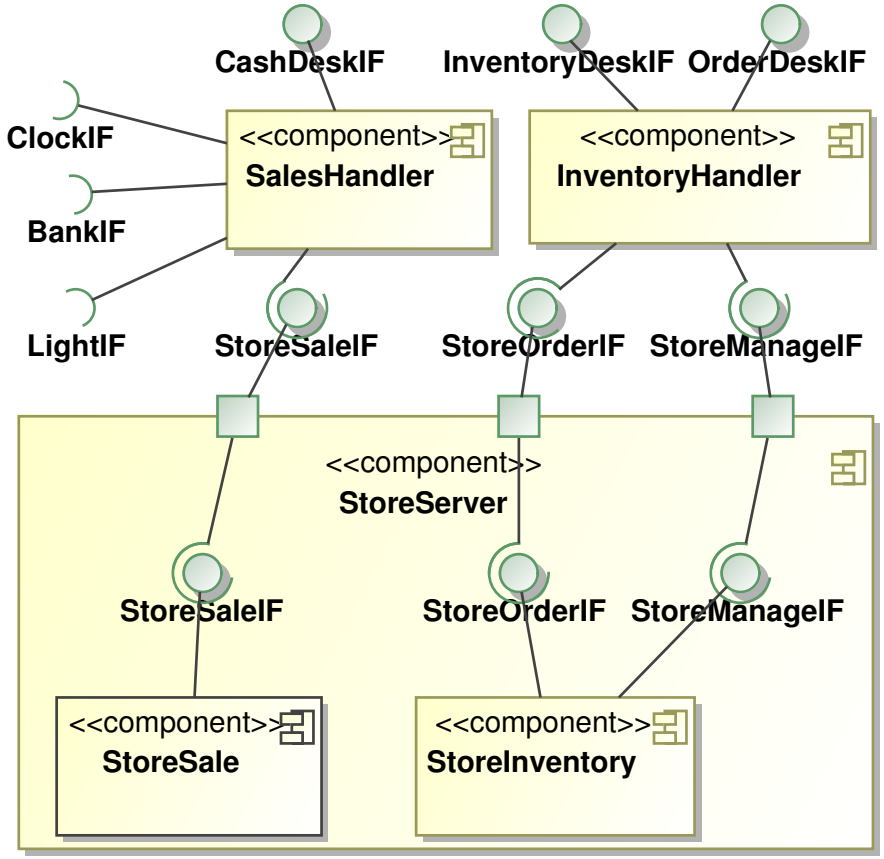
Compose Sequence Diagrams (Advanced Material)



Component Diagram (Advanced Material)



From OOA to CBA (Advanced Material)



How to check consistency, how to carry 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?

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?

Advanced models: Component Diagrams