Karlsruhe Institute of Technology

A BY-PRODUCT OF THE TV INDUSTRY

Softwaretechnik II

Oliver Hummel, IPD

Topic 6
(Object-Oriented) Requirements Analysis

SOFTWARE DESIGN AND QUALITY GROUP
INSTITUTE FOR PROGRAM STRUCTURES AND DATA ORGANIZATION, FACULTY OF INFORMATICS

sdq.ipd.kit.edu

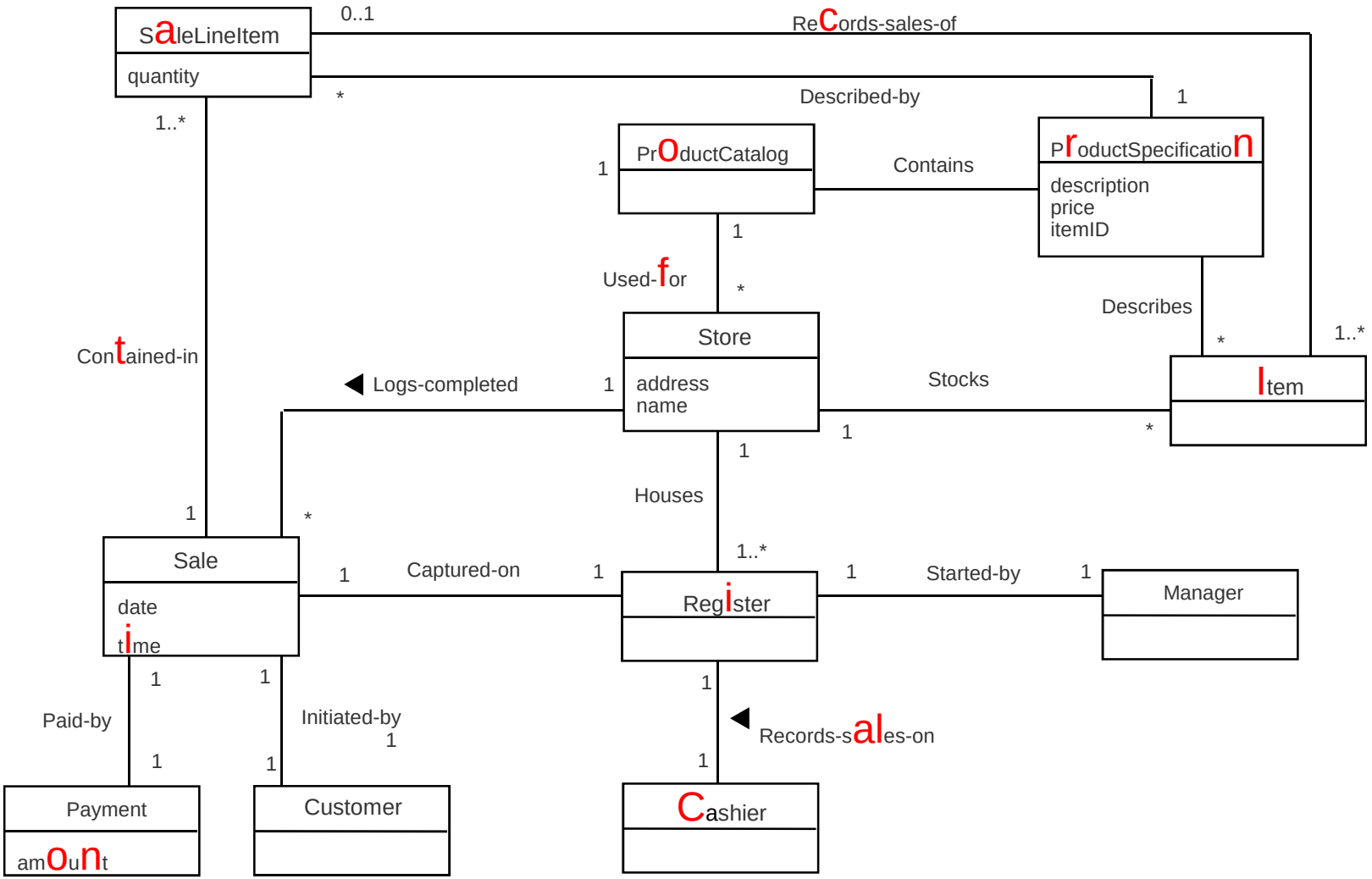
KIT – University of the State of Baden-Wuerttemberg and
National Research Center of the Helmholtz Association

www.kit.edu

Overview on Today's Lecture

- Content
  - System modeling/analysis perspectives and approaches
  - Object-Oriented Analysis
    - System Sequence Diagrams
    - Operation Contracts
- Learning Goals
  - Get an Overview of different analysis perspectives
  - Understand how object-oriented analysis can be carried out
    - with system sequence diagrams
    - and operation contracts
  - Understand the implicit processes that happen in the minds of the “gurus”

Warm Up: Find a Popular TV Series

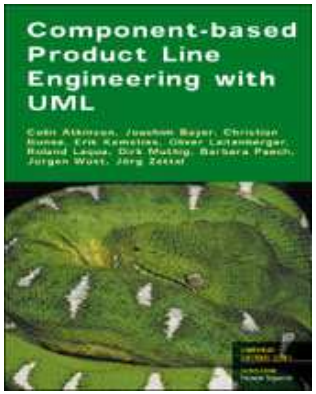


Traditional System Perspectives

- Three perspectives for software systems and their analysis
  - Structural / data perspective
    - static structure of data
  - Functional perspective
    - manipulation of data by functions of the system
    - transformation of input into output data
  - Behavioral perspective
    - describes behavior of the system
    - reaction to external stimuli
      - states
      - state changes
      - output

Perspectives Illustrated

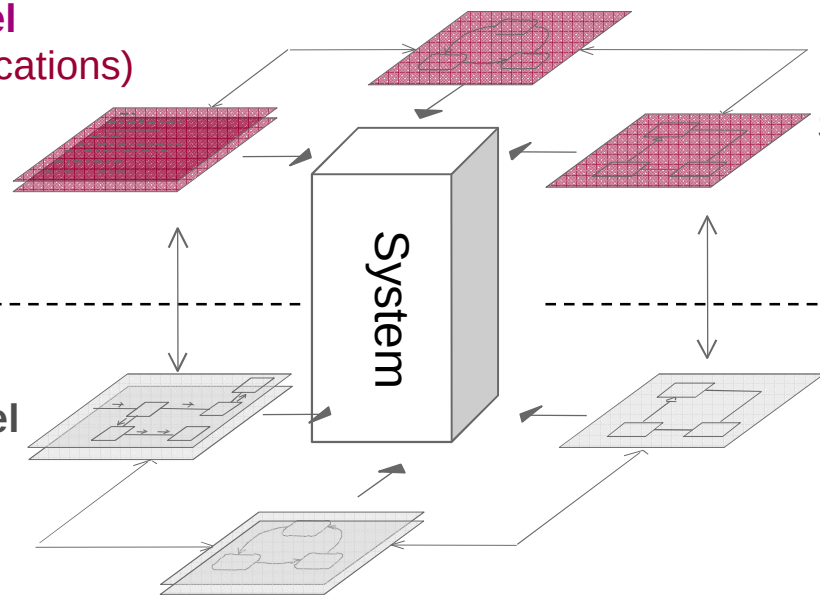
- Avoid doing „Software Design“ during Analysis!
  - define what and not how



Specification *(WHAT?)*

Behavior Model
(UML statechart diagram)

Functional Model
(operation specifications)



Structural Model
(UML class diagrams)

Interaction Model
(UML interaction diagrams)

Structural Model
(UML class diagrams)

Activity Model
(UML activity diagrams)

Realization *(HOW?)*

Objects, Functions and States

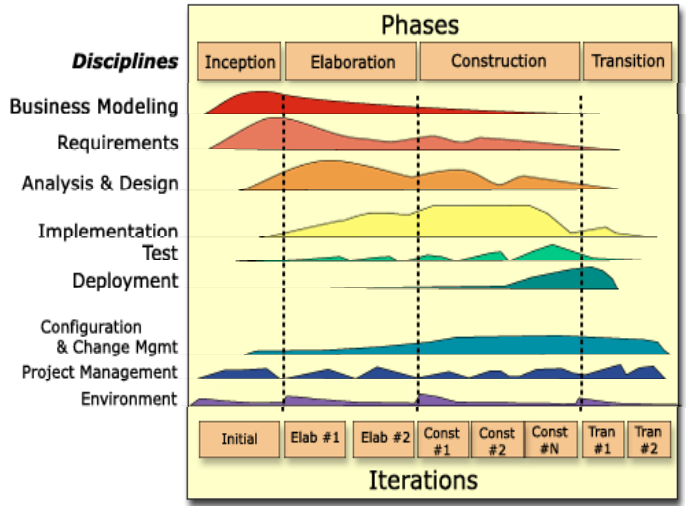
- Object
  - any real-world entity, important to the discusion of requirements
    - with crisply defined boundary
  - often grouped in classes
- Function
  - task, service, process, mathematical function
    - performed in real world
    - ➔ to be perfomed by a system
  - often grouped hierarchically
- State
  - condition of some ‘thing‘, captures history
  - determines behavior in specific circumstances
  - ‘Thing‘ can be system, object or function

Analysis Overview

- Different problems require different solutions
  - *give someone a hammer and every problem starts to look like a nail*
- Various analysis models have been developed over the decades
  - Structural models
    - ER
  - Function models
    - SADT
    - SSA
  - Behavioral models
    - State machines
  - Object-oriented models
    - UML

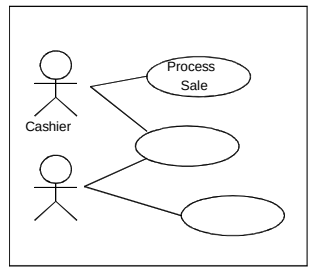
OOA within Agile Modeling

[Larman]



Requirements

Use -Case Model



Use Case Diagram

use case names

Process Sale

1. Customer arrives ...
2. ...
3. Cashier enters item identifier .

Use Case Text

domain objects

ideas for the post - conditions

system events

: Cashier

: System

Operation : enterItem (...)

Post -conditions :
- . . .

system operations

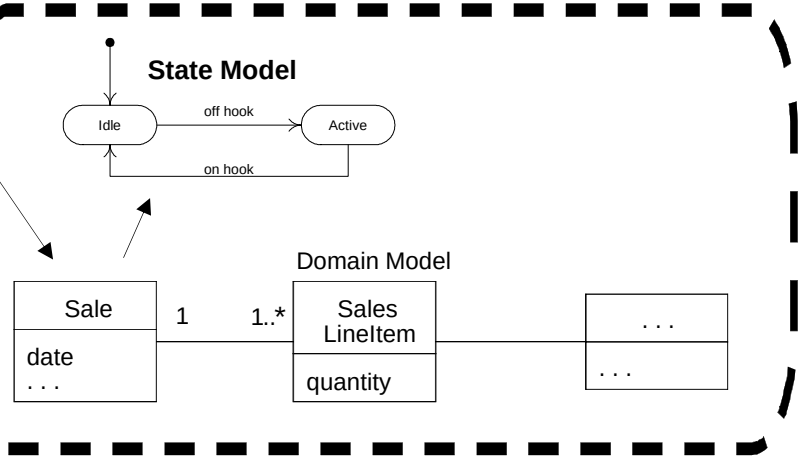
make NewSale ()

enterItem (id, quantity)

Operation Contracts

System Sequence Diagrams

Analysis



Analysis & Business Modeling

System Sequence Diagrams I

- Use cases describe how external actors interact with the software system
  - users generate **events** which request some **operation** to be performed (system operation)
    - e.g. *the customer enters his PIN* that leads to an event

➔ as a starting point for system design it is important to identify the individual system operations

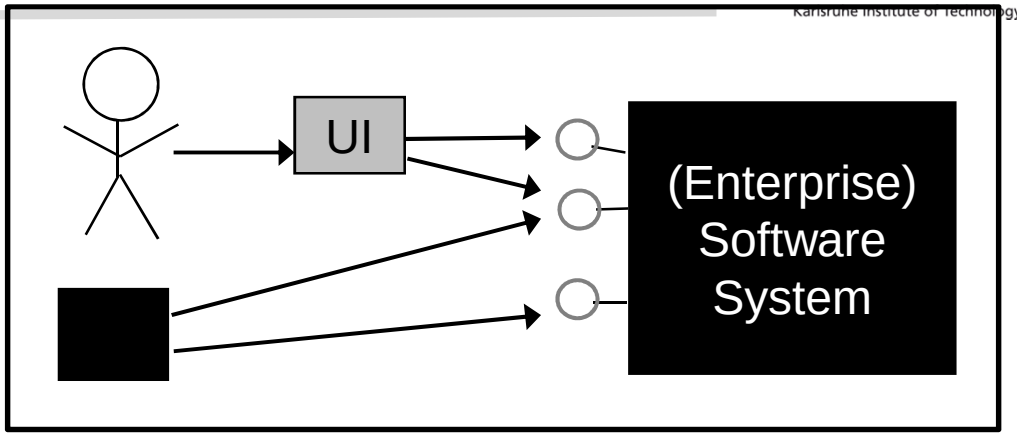
- So-called System Sequence Diagrams (SSD) are useful for this purpose
  - SSDs are based on the notation of (simplified) UML sequence diagrams
    - are a helpful in practice to find system operations
    - and for identifying the data flow into and out of the system

System Sequence Diagrams II

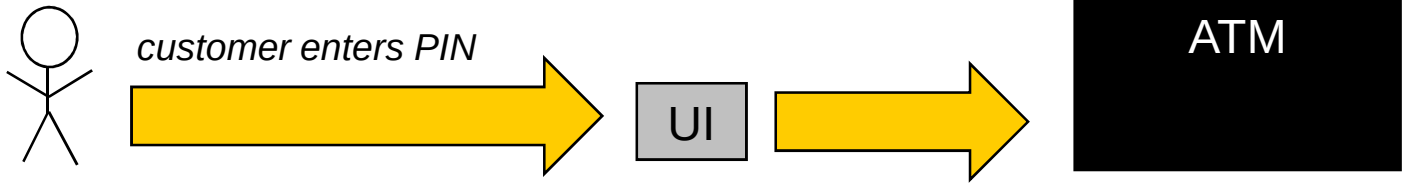
- The emphasis in SSDs is on interactions that cross the system boundary
  - from actors to the system
    - primary and supporting actors can be included in an SSD
  - for a particular scenario of a use case, a SSD shows –
    - the events which external actors generate
    - the order of these events
    - necessary parameters and their types as well as return values
    - inter-system events
      - interfaces of supporting systems are often fixed
- SSDs are typically created for the main success scenario and frequent or complex alternate scenarios
  - in practice typically not all SSDs are drawn
    - unless they are required for some reason
      - *such as e.g. function point counting for effort estimation*

Reminder

1. Input
2. Processing
3. Output



- e.g., in case an ATM?




```
public class PinFrame
   extends JFrame implements ActionListener {

      public void actionPerformed(ActionEvent ae) {
         if (ae.getSource() == OkButton) …
      }
   . . .
}
```

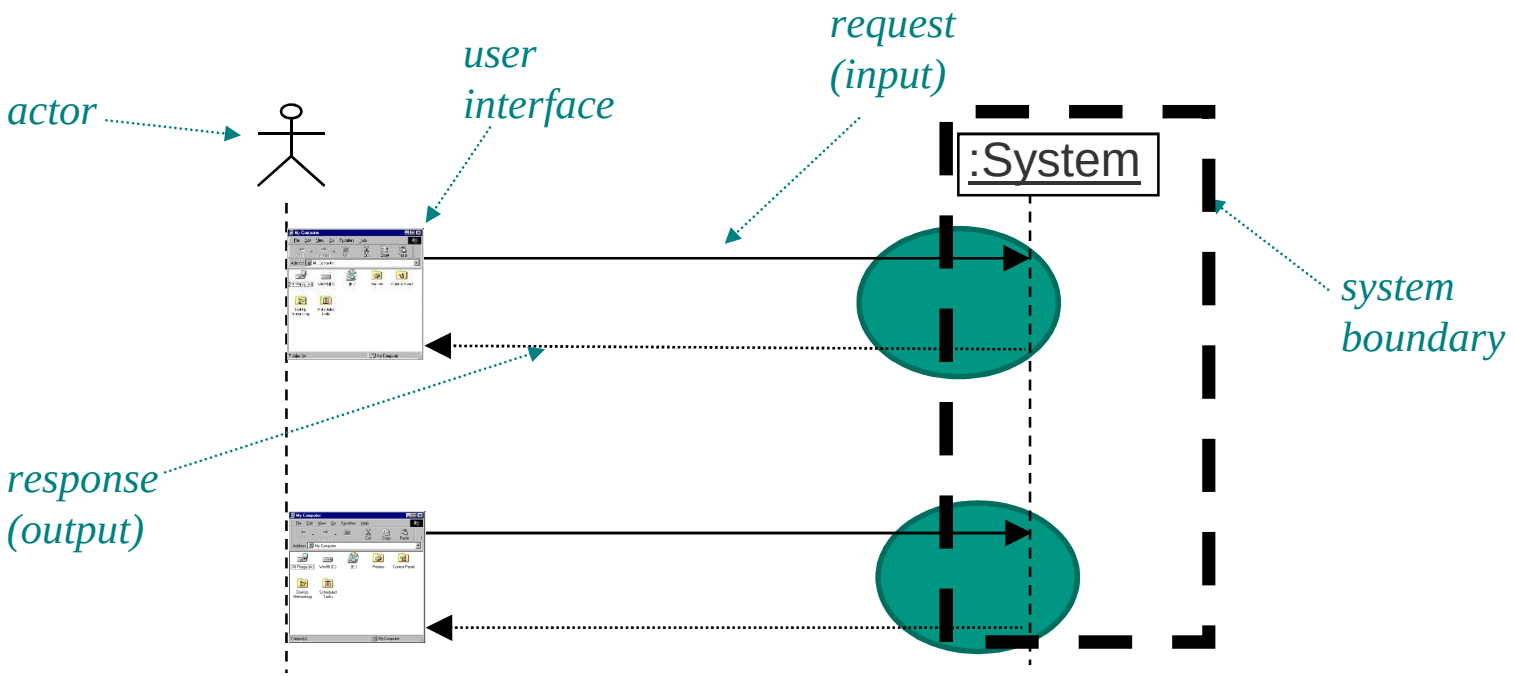
```
public class ATM {
   public … ???

   . . .

}
```

System Boundary

- To identify system events one must be clear on the system boundary
- System events are external events that directly stimulate the software system
  - typically human users interact with the system via a GUI
    - *which is ignored in system sequence diagrams, however*



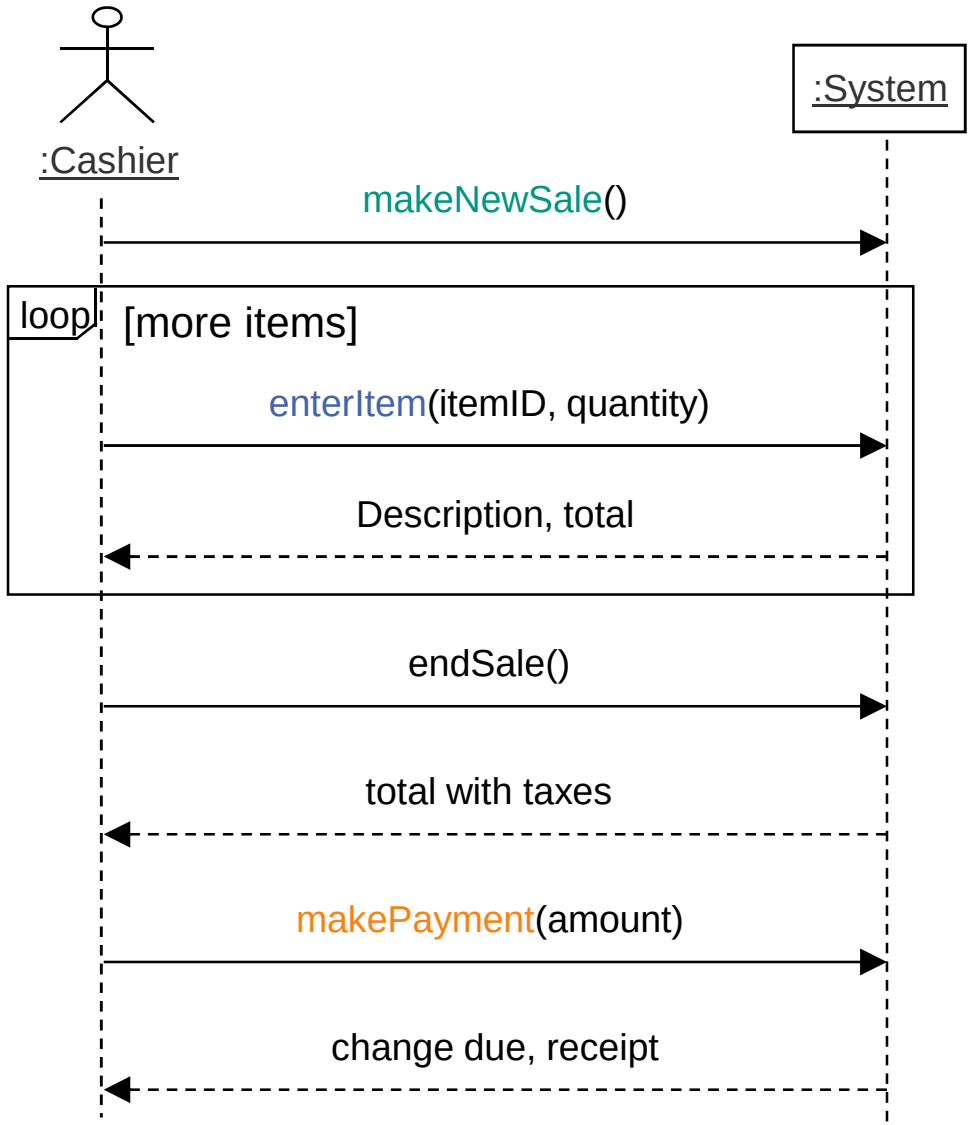
Deriving SSDs from Use Cases

Simple cash-only Process Sale operation

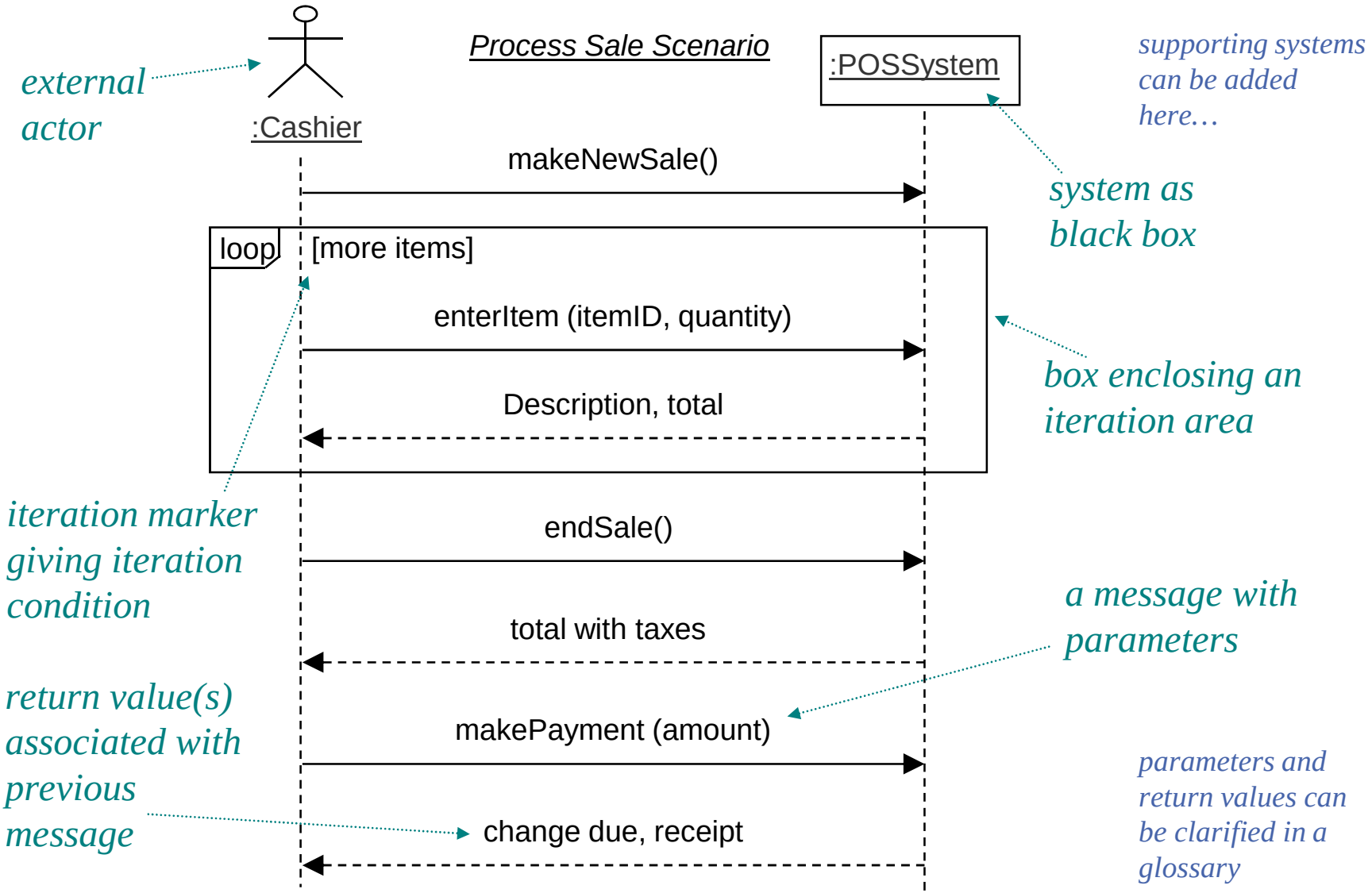
1. Customer arrives at POS checkout with goods and/or services to purchase
2. Cashier starts a new sale
3. Cashier enters an item
4. System records sale line item and presents item description, price and running total.

   Cashier repeats steps 3-4 until indicates done

5. System presents total with taxes calculated
6. Cashier tells Customer the total, and asks for payment
7. Customer pays and System handles payment



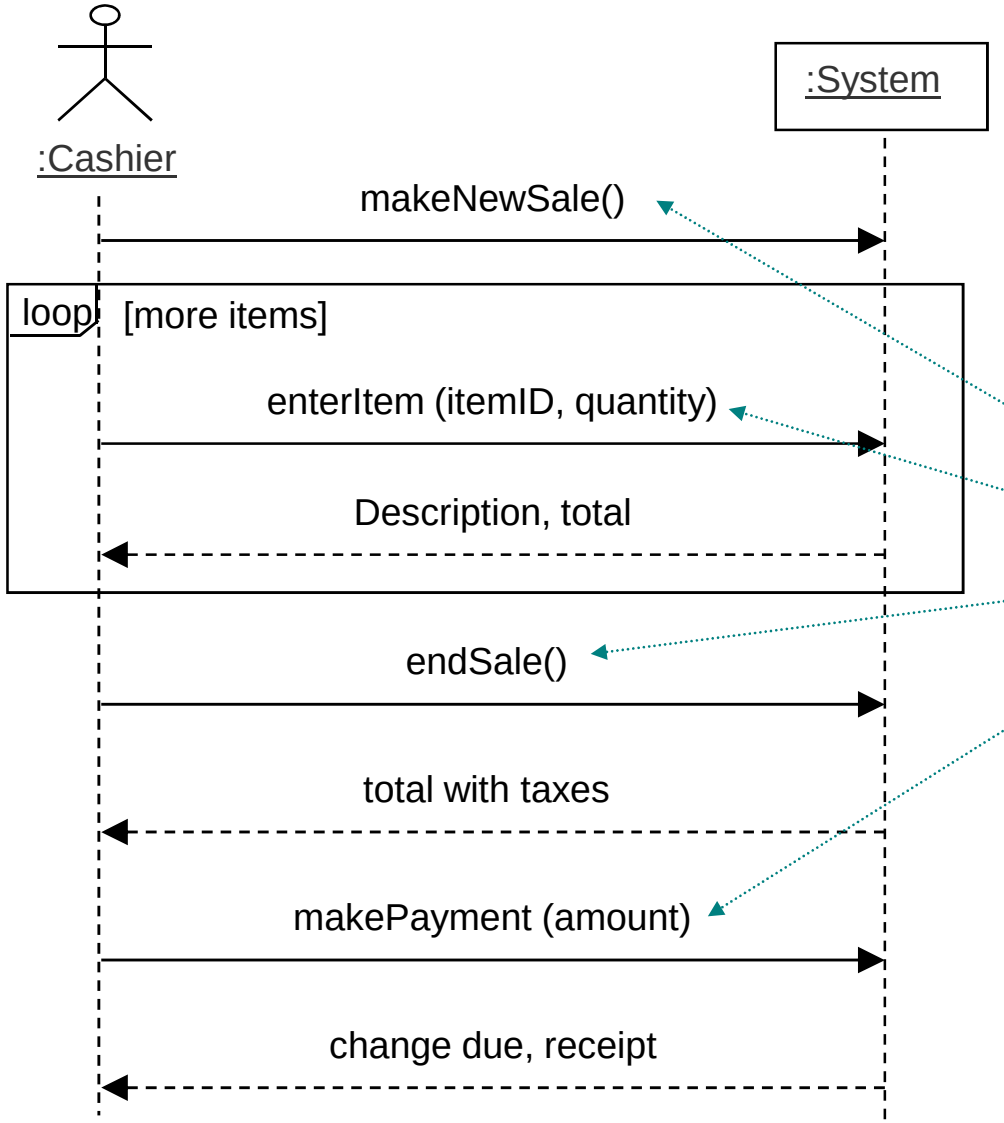
Elements in a System Sequence Diagram



UC1: Geld abheben (Auszug)

- Scope: Geldautomat, Level: User Goal
- Vorbedingung: Der Geldautomat ist betriebsbereit und zeigt den Willkommensbildschirm.
- Success Scenario:
  1. Der Bankkunde führt eine Karte (EC- oder Kreditkarte) ein.
  2. Der Geldautomat liest die Kartendaten, validiert diese und fordert den Bankkunden zur Eingabe der PIN auf.
  3. Der Bankkunde gibt die PIN ein.
  4. Der Geldautomat validiert die PIN mit dem Banksystem und fordert den Bankkunden zur Eingabe des gewünschten Geldbetrags auf.
  5. Der Bankkunde gibt einen Geldbetrag ein.
  6. Der Geldautomat validiert beim Banksystem eine ausreichende Deckung für den gewünschten Geldbetrag auf dem zur Karte gehörenden Konto, veranlasst dort die Abbuchung des Geldbetrags, gibt die Karte sowie den Geldbetrag gestückelt nach Regel 08/15 aus und fordert den Bankkunden zur Entnahme von Beidem auf.
  7. Der Bankkunde entnimmt seine Karte und den Geldbetrag.
  8. Der Geldautomat registriert die Entnahme der Karte sowie des Geldbetrags und zeigt für 20 Sekunden den Verabschiedungsbildschirm, bevor er wieder den Willkommensbildschirm anzeigt.

Identifying System Operations…



system events invoking system operations

… is fortunately simple!

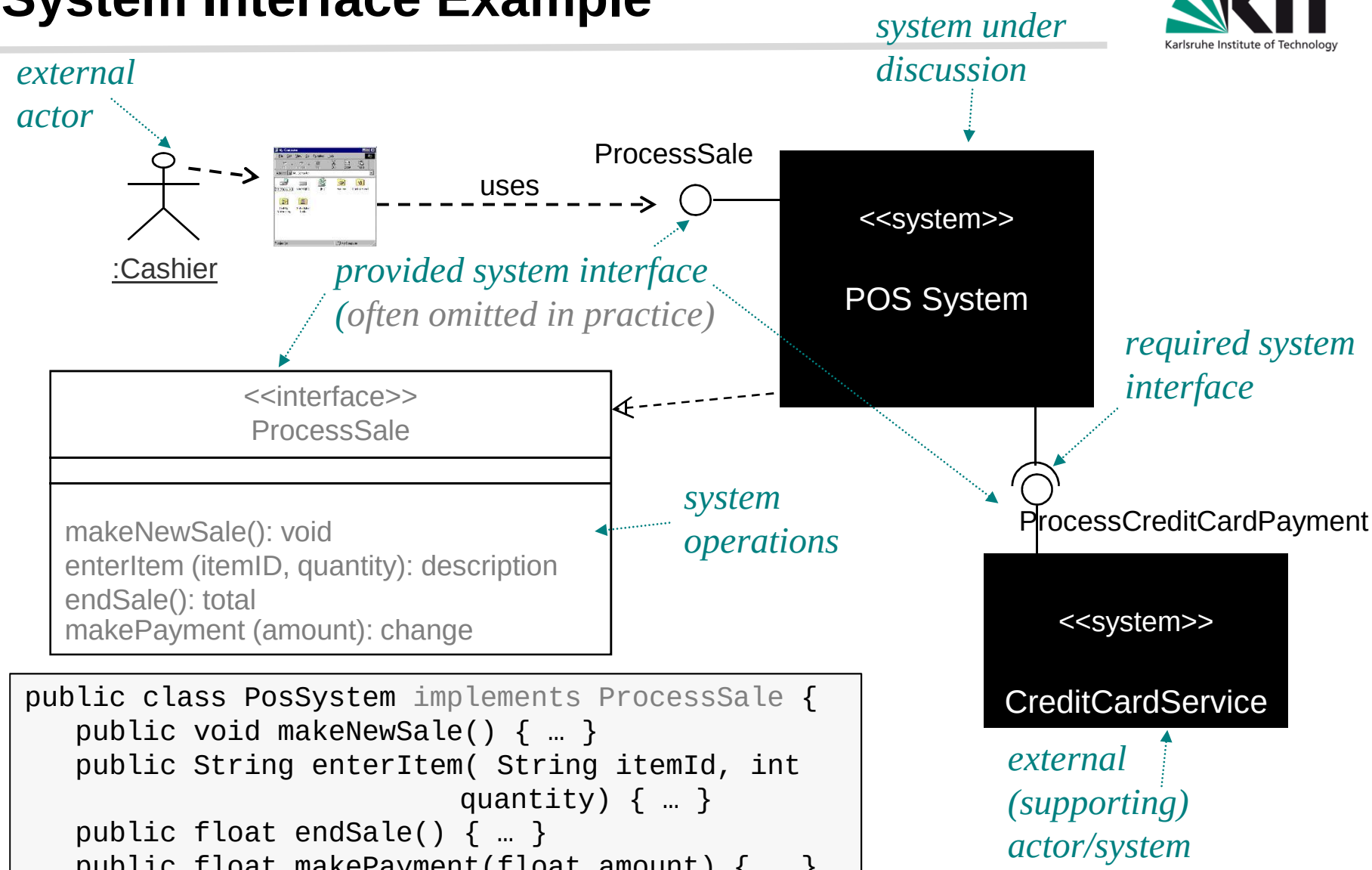
Naming System Events and Operations

- System operations are operations that the system as a black box offers
  - in order to answer resp. satisfy incoming events resp. requests
- *The entire set of system operations, across all use cases, defines the public system interface*
  - viewing the system as a single component or class

- System operations should be expressed at the level of intent
  - rather than at the level of physical devices or interface widgets
    - *as we already know from use case writing*
  - in general, use "talking names" just like in programming

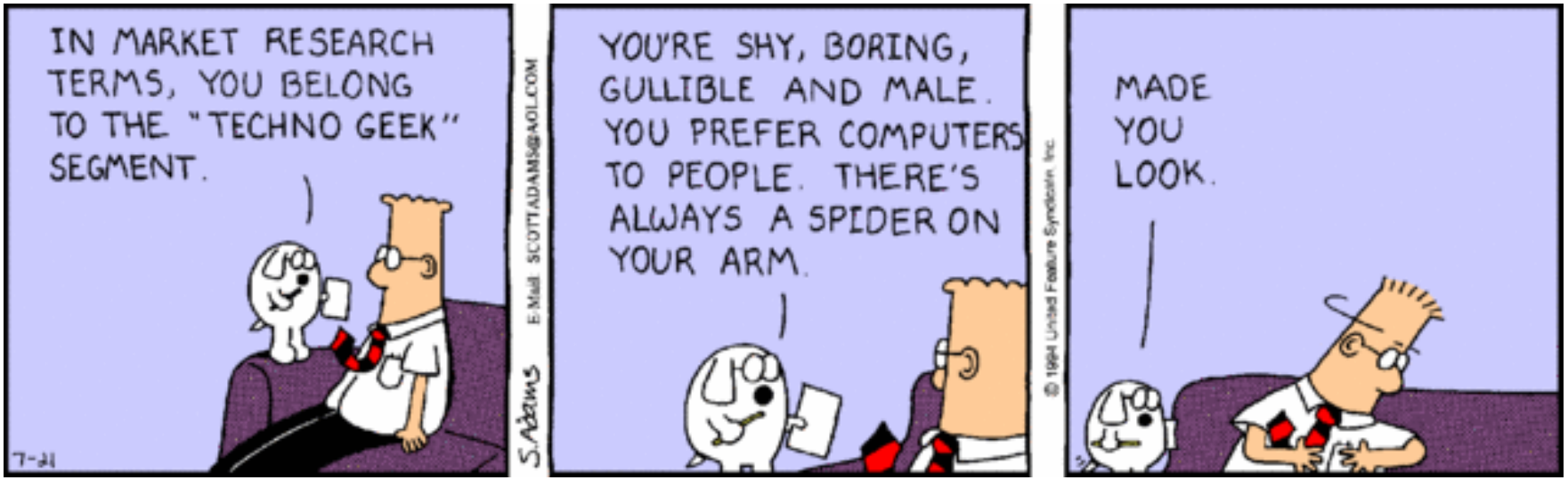
- It adds clarity to start the name with a verb
  - *add…, enter…, end…, make…, get…, is…, set…, retrieve…, delete…*
    - ➔ thus, *enterItem* is better than just *scan*
      - captures intent but is non-committal on realization

System Interface Example




```
public class PosSystem implements ProcessSale {
   public void makeNewSale() { … }
   public String enterItem( String itemId, int
                            quantity) { … }
   public float endSale() { … }
   public float makePayment(float amount) { … }
}
```

Relax for a Minute



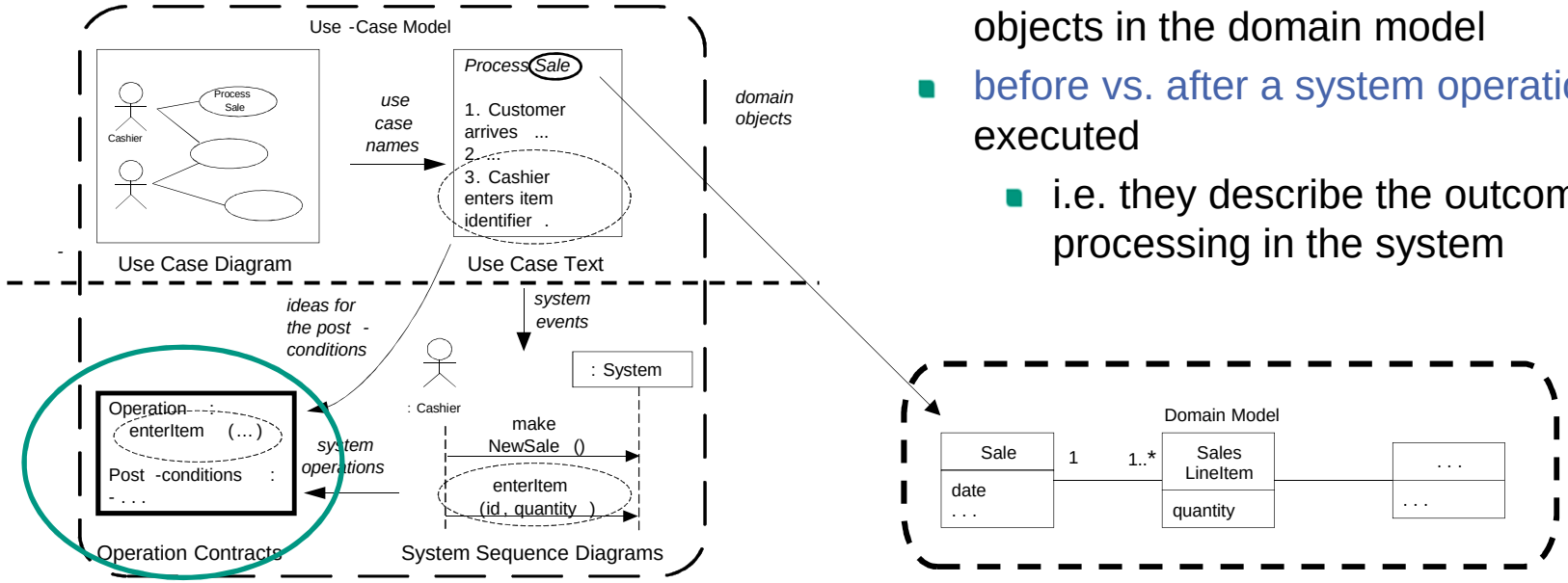
Describing Behavior: Operation Contracts

- Based on the “Design by Contract” approach popularized by Bertrand Meyer
  [http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=161279]
  - provide a more detailed description of system behavior as starting point for system design
- Contracts are expressed in a **declarative state change** fashion
  - focus analytically on *what* must have happened rather than *how* it has to be accomplished
- … are thus helpful for system design that has to deal with the *how*
  - free use cases from becoming overly verbose and detailed

Party	Obligations	Benefits
Client	Provide letter or package of no more than 5 kgs, each dimension no more than 2 meters. Pay 100 francs.	Get package delivered to recipient in four hours or less.
Supplier	Deliver package to recipient in four hours or less.	No need to deal with deliveries too big, too heavy, or unpaid.

Operation Contracts

- Once the system operations are identified, they should be described in more detail
  - with the help of operation contracts
    - In other words, contracts –
      - specify detailed system behavior
        - in terms of state changes to objects in the domain model
        - before vs. after a system operation has executed
          - i.e. they describe the outcome of the processing in the system



Operation Contract Schema

Operation:	name of operation and parameters
Cross References:	(optional) use case(s) this operation can occur within
Preconditions:	Noteworthy assumptions about the state of the system or objects in the Domain Model before execution of the operation. These are non-trivial assumptions the reader should know were made and will not be tested within the logic of the this operation. They are assumed to be true.
Postconditions:	The states of objects in the Domain Model after completion of the operation.

This idea can be helpful in other contexts as well…

for use cases, business processes, methods in objects, etc.

Various Uses in Practice

1. Business Processes
   - normally called Service Level Agreement (SLA) here
   - *example: operation of a web server*
2. Use Cases
   - *example: process DVD rental*
3. System Operations
   - *example: withdraw money with an ATM*
4. (Java) Methods
   - *example: assertions for binary search method*

- *Roommate Agreements ☺*
  - *"If one of the roommates ever invents time travel, the first stop has to aim exactly five seconds after this clause of the Roommate Agreement was signed" [TBBT]*
  - *http://www.youtube.com/watch?v=N0qDy0T5WXM*

Conditions

- Preconditions are usually simple and mostly require the previous execution of another system operation
- Postconditions –
- … describe changes in the state of domain model objects in terms of –
  - **instance creation and deletion**
  - **links (i.e. association instances) formed and broken**
  - **attribute modifications**
- … are expressed in terms of domain model objects
  - the instances, links and attributes which are changed are those defined in the domain model
- … are declarative rather than imperative
  - i.e. they are not actions to be performed
    - but declarations about the domain model objects that are true when the operation has finished
    - i.e. when “the smoke has cleared”

The Spirit of Postconditions

- Postconditions should be expressed in the past tense
  - in order to emphasize that they are declarations about changes that have occurred
  - thus “ a SaleLineItem was created” is better than “Creates a SaleLine-Item”

- A helpful metaphor is to think of a system operation in terms of the system and its objects being presented on a theatre stage
  1. before the operation take a picture of the stage
  2. close the curtains on the stage and apply the system operation
     - *background noises of clanging and screeching etc.*
  3. open the curtains and take a second picture
  4. compare the before and after pictures and express the changes in the state of the stage as postconditions
     - i.e. the changes performed before the curtain was opened

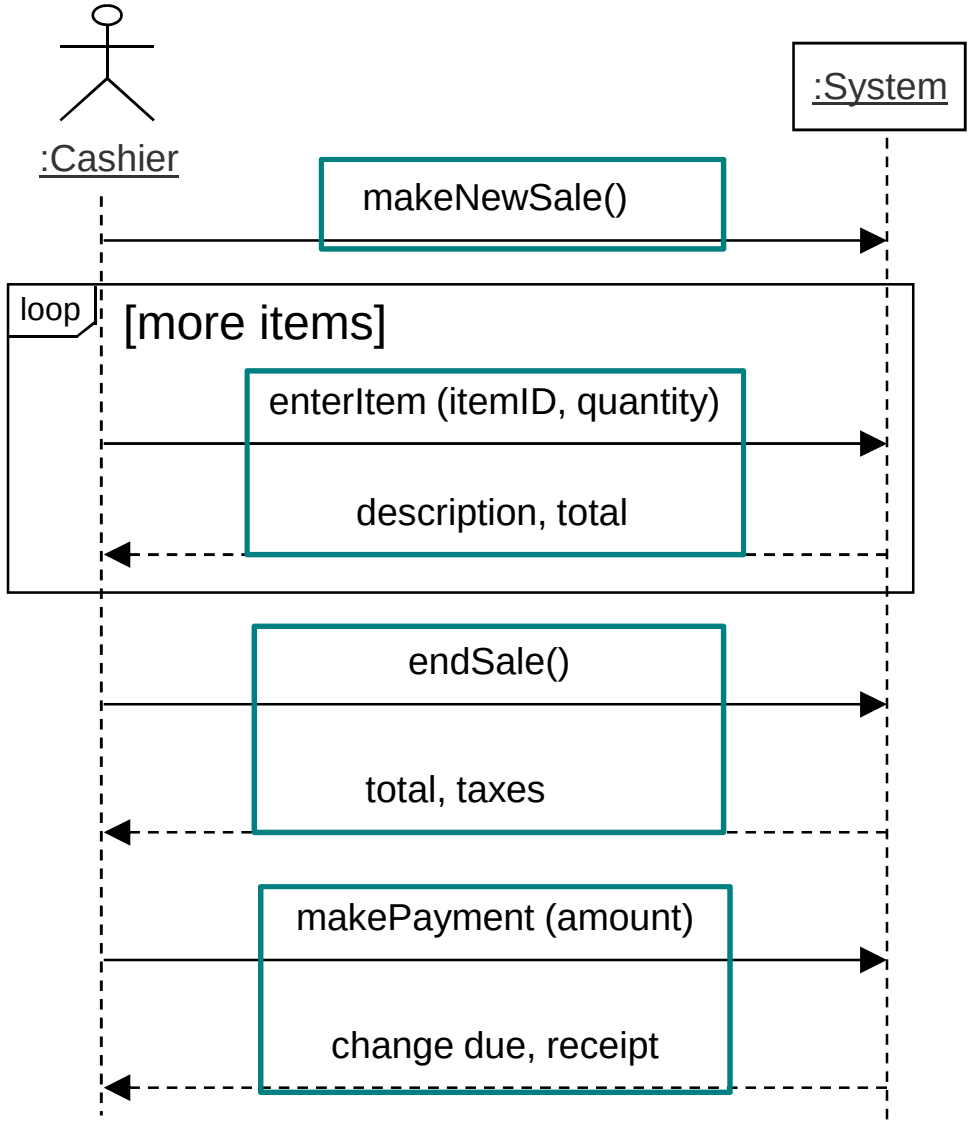
Context: Point of Sale

Simple cash-only Process Sale operation

1. Customer arrives at POS checkout with goods and/or services to purchase
2. Cashier starts a new sale
3. Cashier enters an item
4. System records sale line item and presents item description, price and running total.

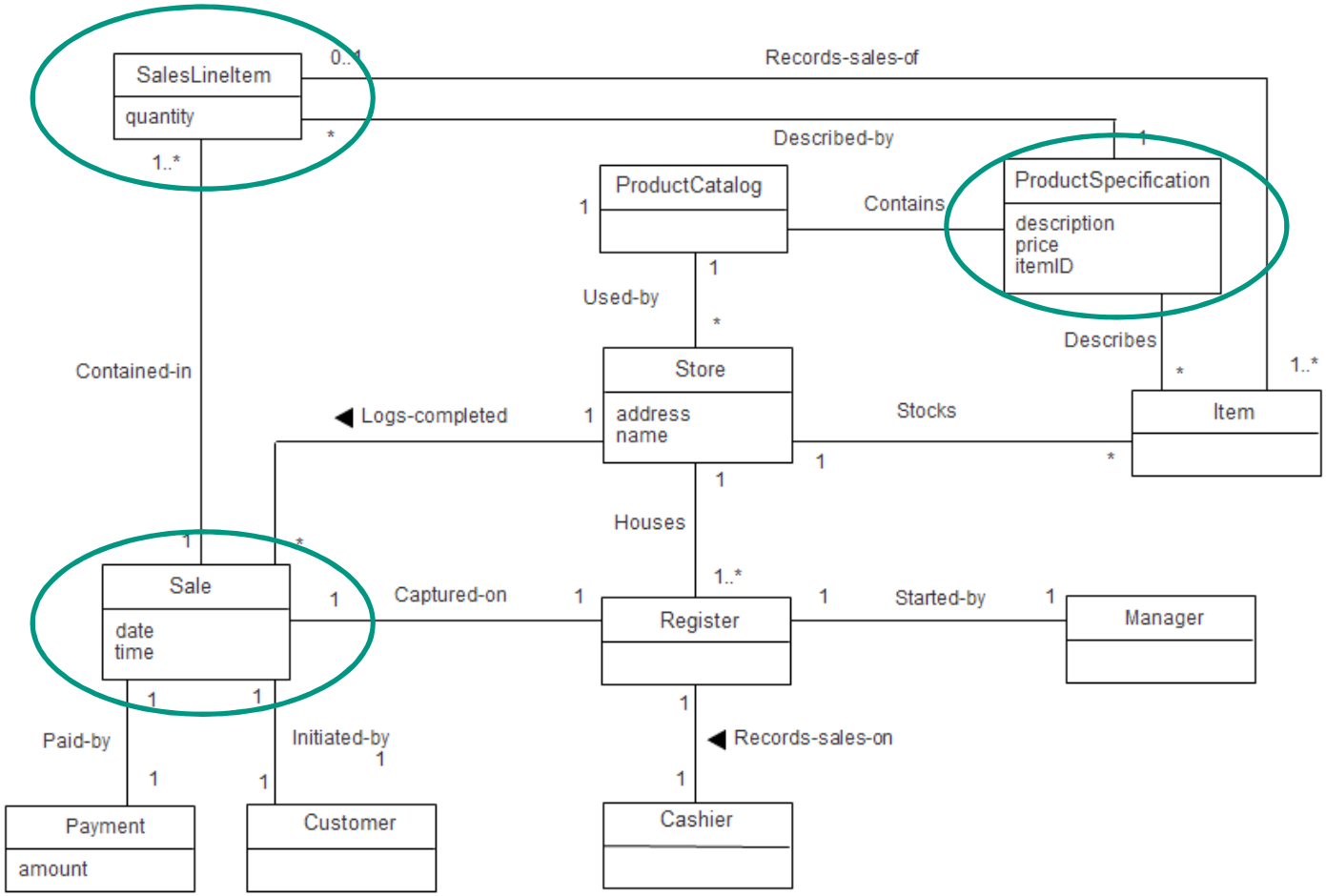
   Cashier repeats steps 3-4 until indicates done

5. System presents total with taxes calculated
6. Cashier tells Customer the total, and asks for payment
7. Customer pays and System handles payment



Reminder

- POS domain model: what needs to happen for enterItem?
  - enterItem (itemID, quantity): description, total



Software Contract Example C02

- Contract C02: enterItem

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

Contracts versus Use Cases

- Use cases are the main repository for requirements of a project
  - they may provide most or all of the detail necessary to know what to do in design
    - in which case contracts can be omitted
  - however, there are situations where the details and complexity of required state changes are awkward to capture in use cases
    - e.g. addNewReservation in an airline reservation system
    - fine-grained details and business-rules could be written up in the use case
      - but make the use case extremely difficult to read

➔ The post-condition format of contracts encourages a very precise analytical language

- supports thoroughness

- **To summarize: contracts are useful when there is complexity**
  - *or inexperienced people onboard*

How Complete Should Contracts Be?

- Contracts may not be needed in a clear context
  - however, their creation is helpful to better understand potential relationships in a domain
    - even an incomplete creation is better than deferring this investigation to design work
      - there developers should be concerned with the design of a solution
        - not with sorting out what needs to happen
      - during analysis the domain experts can be asked
- Contracts should be treated as an **initial best guess** and regarded as something that will not be complete
  - as some of the fine design details will only be discovered during design work
  - avoid "analysis paralysis" caused by over-specifying
- When creating contracts it is common to discover the need to record new elements in the Domain Model
  - ➔ enhance the Domain Model as you gain more understanding

Contract Creation Summary

1. Identify system operations from the SSDs
2. For system operations that are complex and perhaps subtle in their results, or which are not clear in the use case, construct a contract
3. To describe the postconditions, use the following categories
   - instance creation and deletion
   - attribute modification
   - associations formed and broken

- State the postconditions in a declarative, passive tense form
  - e.g. was…
- The most common mistake in creating contracts is forgetting to include the forming of associations
  - e.g. it is not enough that a new SaleLineItem instance was created by the *enterItem* system operation
  - it also needs to be associated with Sale

Contract Example C01

- Contract C01: makeNewSale

Operation:	makeNewSale()
Cross References:	Use Cases: Process Sale
Preconditions:	none
Postconditions:	

Contract Example C03

- Contract C03: endSale

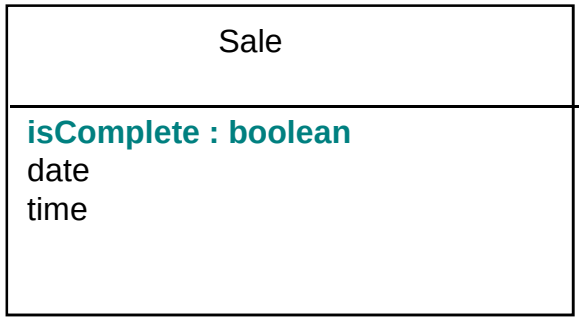
Operation:	endSale()
Cross References:	Use Cases: Process Sale
Preconditions:	There is a sale underway.
Postconditions:	

Example Update to the Domain Model

- There is information suggested in the previous contracts not yet represented in the domain model
  - the completion of item entry into a sale
    - the endSale contract modifies it and makePayment will probably need to test it

➔ One way to represent this information is with an isComplete attribute in Sale

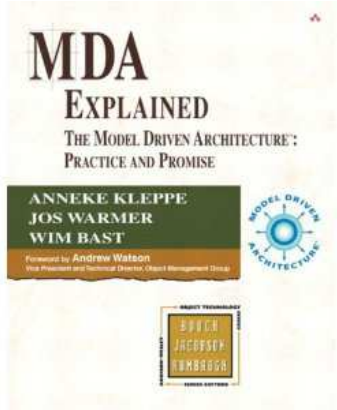
- therefore the Sale class in the domain model might be updated as follows after developing the contract



➔ this may also cause the creation of a *state model for Sale*

Operations, their Contracts and the UML

- The UML specifies operations having a contract
  - including a signature and pre- and postconditions
- Natural language contracts are perfectly acceptable in the UML
  - however, the so-called Object Constraint Language (OCL) offers a more formal way to express contracts
  - beyond the scope of this lecture



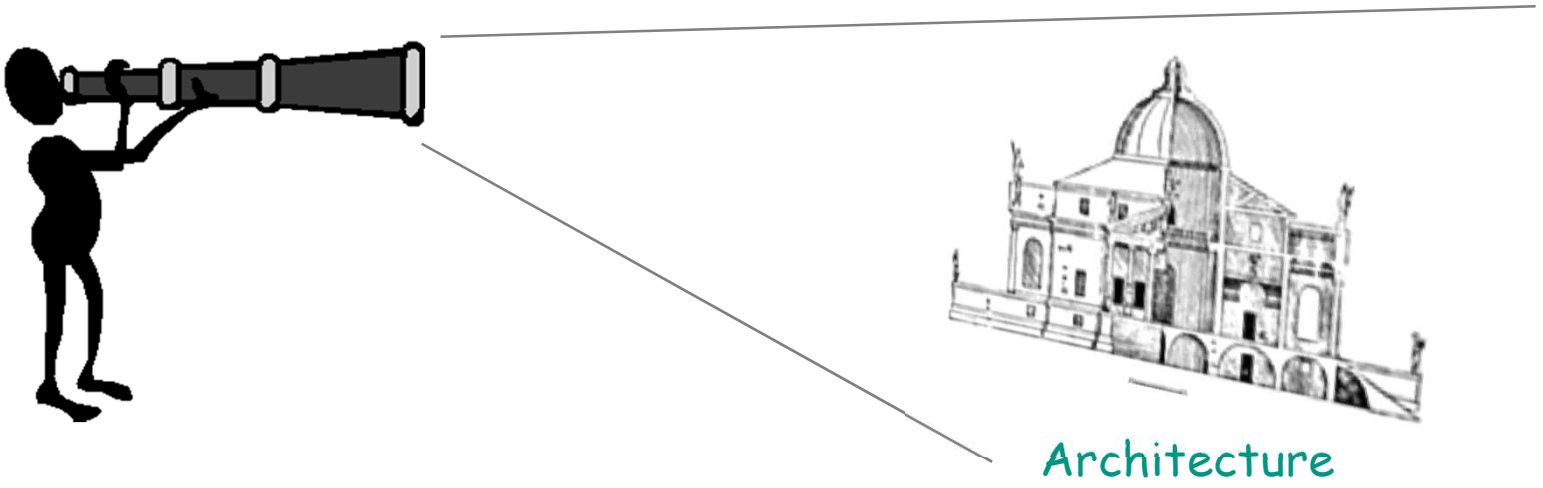
- **Important!** Do not get dragged into a waterfall thinking and try to model the whole system until perfection
  - you will not be able to achieve this
  - **try to model it just good enough**
    - at least in iterative and incremental development approaches
  - ➔ requirements analysis and modeling is rather a matter of hours or a few days per iteration in agile modeling approaches

Conclusion

- Numerous analysis techniques are available today
  - can add a lot of clarity for system architecture and design
  - however, you have to decide which one best suites your needs
    - typically, various views are necessary to describe a system fully

➔ System sequence diagrams and operation contracts can help tame complexity in object-oriented development

- Thank you for your attention!



References

- [Larman] C. Larman
  Applying UML and Patterns (3rd ed.)
  Prentice Hall, 2004