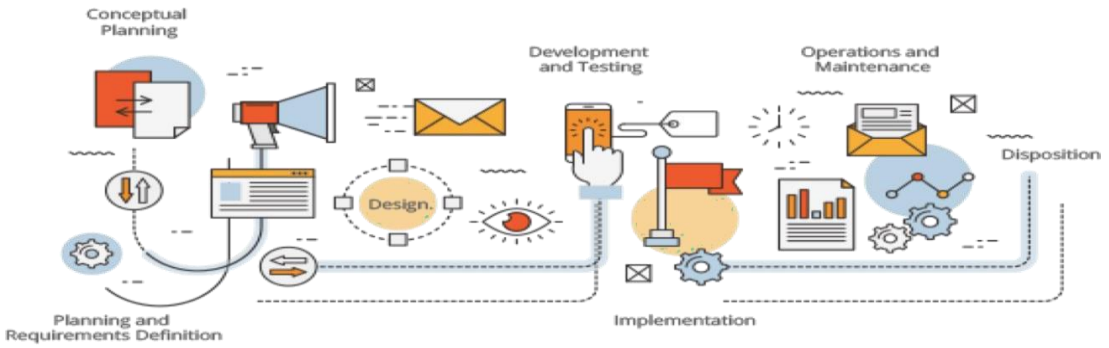
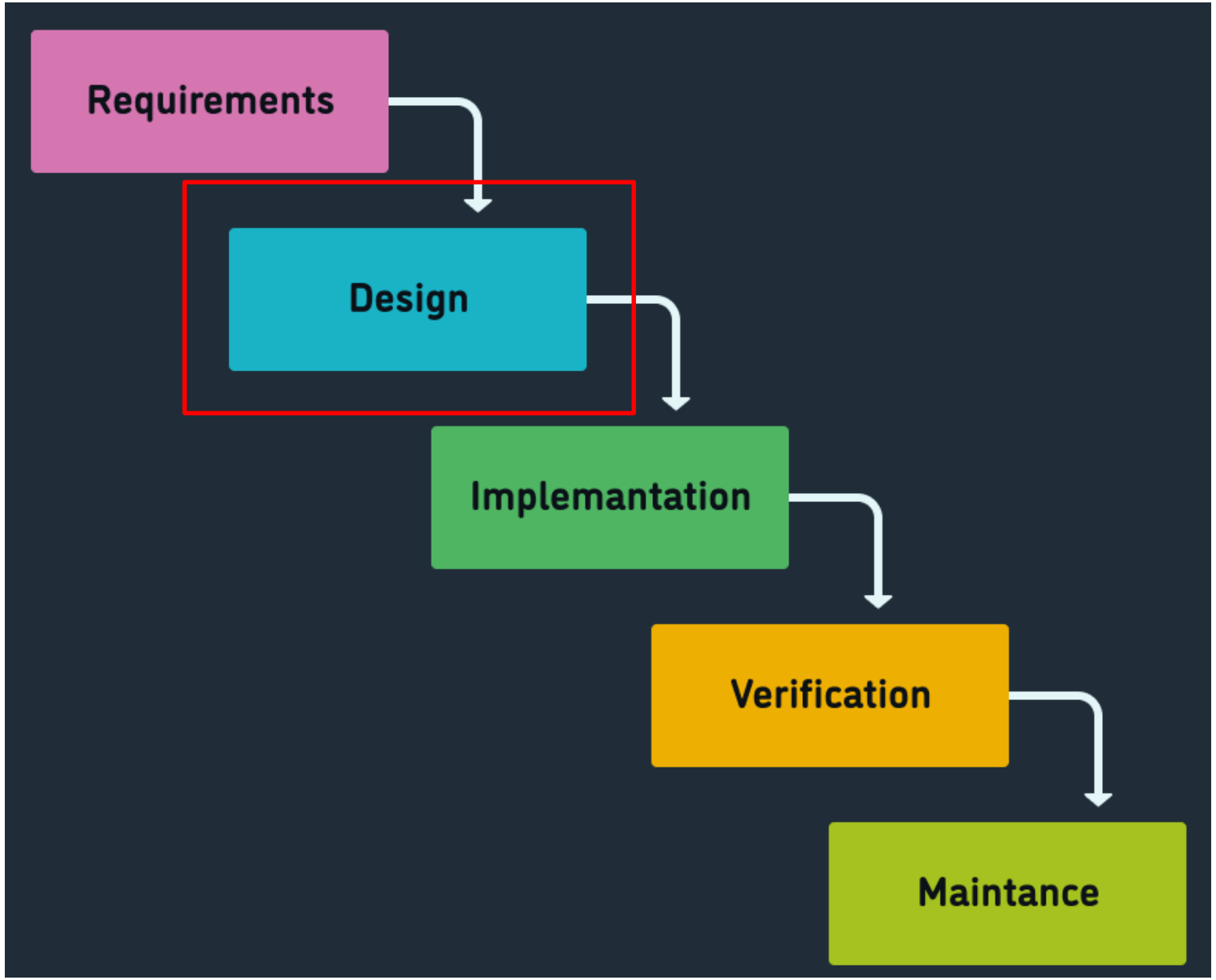
Software Engineering

Requirement Engineering



Phases of the Software Development Life Cycle



Importance of design phase in SDLC

- **Blueprint for Development**: The design phase serves as the blueprint for the entire software development process. It outlines how the system will be structured, how components will interact, and what technologies will be used. This blueprint provides clarity and direction for developers as they begin coding.

- **Risk Mitigation:** Through careful planning and analysis during the design phase, potential risks and challenges can be identified early in the process.

- **Efficiency and Cost Reduction**: Clear and well-thought-out designs reduce the likelihood of rework and unnecessary iterations later in the process, ultimately saving time and reducing development costs.

- **Alignment with Requirements**: The design phase ensures that the proposed solution aligns with the requirements and expectations of stakeholders.

- **Communication and Collaboration:** Through discussions and reviews of design documents, everyone involved gains a clear understanding of the proposed solution and can provide valuable feedback.

How are design ideas communicated in a team environment?

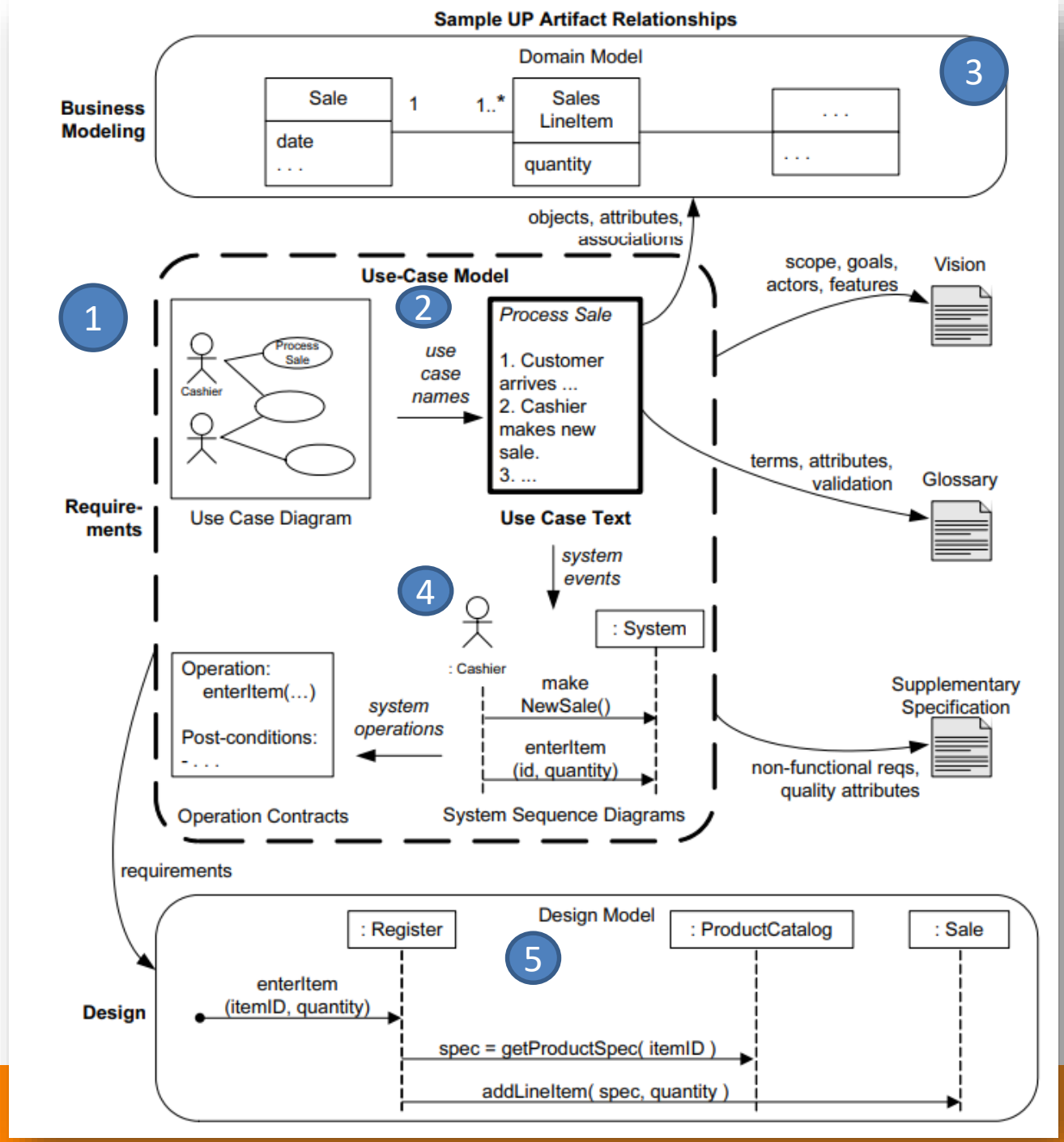
- If the software is large scale, employing perhaps dozens of developers over several years, it is important that all members of the development team communicate using a common language.
- This isn't meant to imply that they all need to be fluent in English or C++, but it does mean that they need to be able to describe their software's operation and design to another person.
- That is, the ideas in the head of say the analyst have to be conveyed to the designer in some way so that he/she can implement that idea in code.
- Just as mathematicians use algebra and electronics engineers have evolved circuit notation and theory to describe their ideas, software engineers have evolved their own notation for describing the architecture and behaviour of software system.
- That notation is called UML. The Unified Modelling Language. Some might prefer the title Universal Modelling language since it can be used to model many things besides software.

What is UML ?

- UML is not a language in the same way that we view programming languages such as ‘C++’, ‘Java’ or ‘Basic’.
- UML is however a language in the sense that it has syntax and semantics which convey meaning, understanding and constraints (i.e. what is right and wrong and the limitations of those decisions) to the reader and thereby allows two people fluent in that language to communicate and understand the intention of the other.
- UML represents a collection of graphical notations to capture requirements and design alternatives.
- UML is to software engineers what building plans are to an architect and an electrical circuit diagrams is to an electrician.
- UML is suitable for large scale projects

Phases of System Development

- Requirement Analysis
  - The functionality users require from the system
  - **Use-case model**
- OO Analysis
  - Discovering classes and relationships
  - Class diagram
- OO Design
  - Result of Analysis expanded into technical solution
  - Sequence diagram, state diagram, etc.
  - Results in detailed specs for the coding phase
- Implementation (Programming/coding)
  - Models are converted into code
- Testing
  - Unit tests, integration tests, system tests and acceptance tests.


Sample UP Artifact Relationships
Business Modeling
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
3
objects, attributes, associations
Use-Case Model
1
2
Process Sale
Cashier
use case names
Process Sale
1. Customer arrives ...
2. Cashier makes new sale.
3. ...
Use Case Diagram
Use Case Text
scope, goals, actors, features
Vision
terms, attributes, validation
Glossary
Require-ments
system events
4
: Cashier
: System
Operation:
enterItem(...)
Post-conditions:
- . . .
system operations
make NewSale()
enterItem (id, quantity)
Supplementary Specification
non-functional reqs, quality attributes
Operation Contracts
System Sequence Diagrams
requirements
Design
Design Model
5
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
spec = getProductSpec(itemID)
addLineItem(spec, quantity)

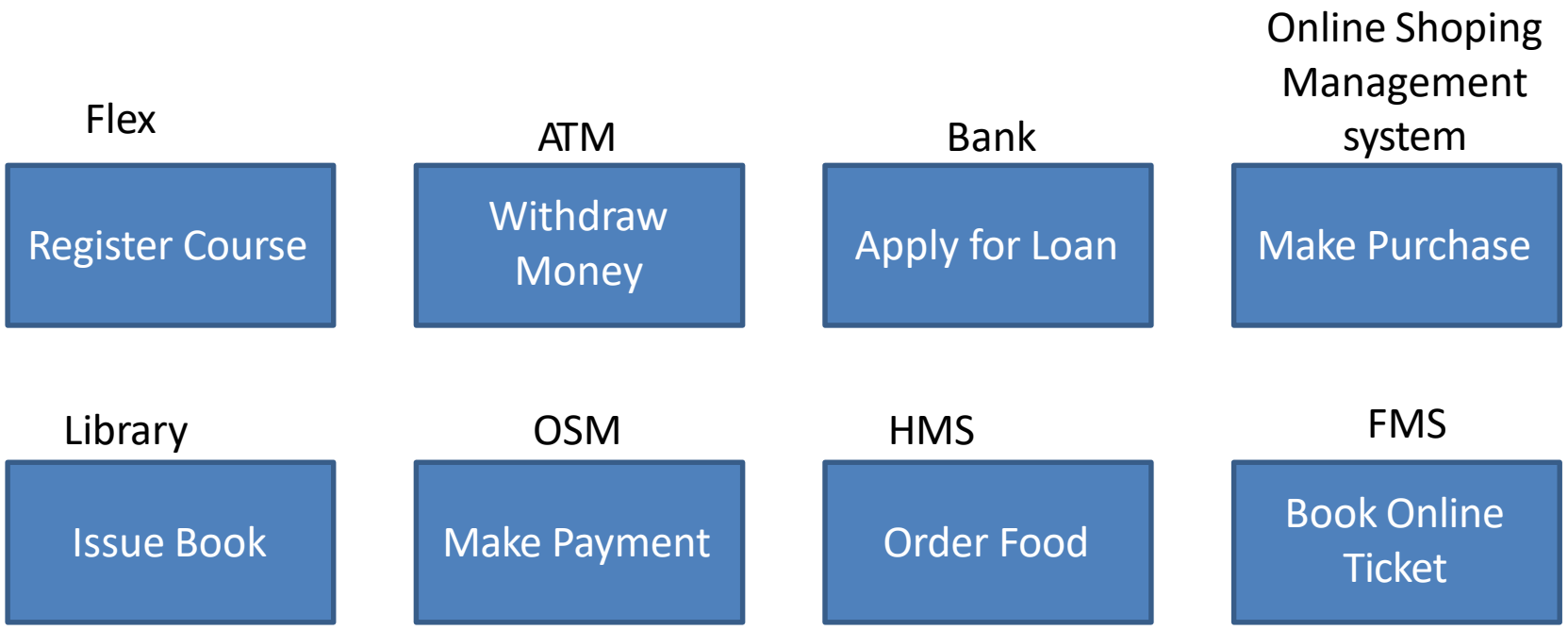
Definition of a Use-Case

- A process or procedure, describing a *user's interaction* with the system (e.g. library) for a *specified, identifiable purpose.* (e.g. borrowing a book).

- As such, each Use-Case describes a step-by-step sequence of operations, iterations and events that document

  - The Interaction taking place.
  - The Measurable Benefits to the user interacting with the system.
  - The Effect of that Interaction on the system.

- It is important to document these use-cases as fully as possible as each use-case captures some important functionality that our system will have to provide in the automated version of the library.


Flex
Register Course
ATM
Withdraw Money
Bank
Apply for Loan
Online Shoping Management system
Make Purchase
Library
Issue Book
OSM
Make Payment
HMS
Order Food
FMS
Book Online Ticket

Use Cases in overall Software Development

- Use cases are a valuable tool in software development for capturing and documenting functional requirements from the perspective of end-users. They describe interactions between users and a system to achieve specific goals or tasks. Here's how use cases are used in software development:

  - Use cases are often used during the **requirement elicitation phase** to understand the needs and expectations of end-users.
  - Use cases serve as a basis for **designing** the system's architecture.
  - Each use case represents a specific scenario that needs to be **validated** to ensure the system functions correctly and meets the user's needs.
  - Use cases serve as a **communication** tool between stakeholders, including developers, designers, testers, and clients.
  - Use cases help in defining and managing the **scope** of the project.
  - In **agile development methodologies**, such as Scrum, use cases can be used as user stories.
  - Use cases serve as valuable **documentation** for the system.

What Tests Can Help Find Useful Use Cases?

- **The boss test**

  “What have you been doing all day?”

  - Your reply “logging in!”
    - Is your boss happy? No value? No good use case!

- **The Elementary Business Process (EBP) test**

  a task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves data in a consistent state

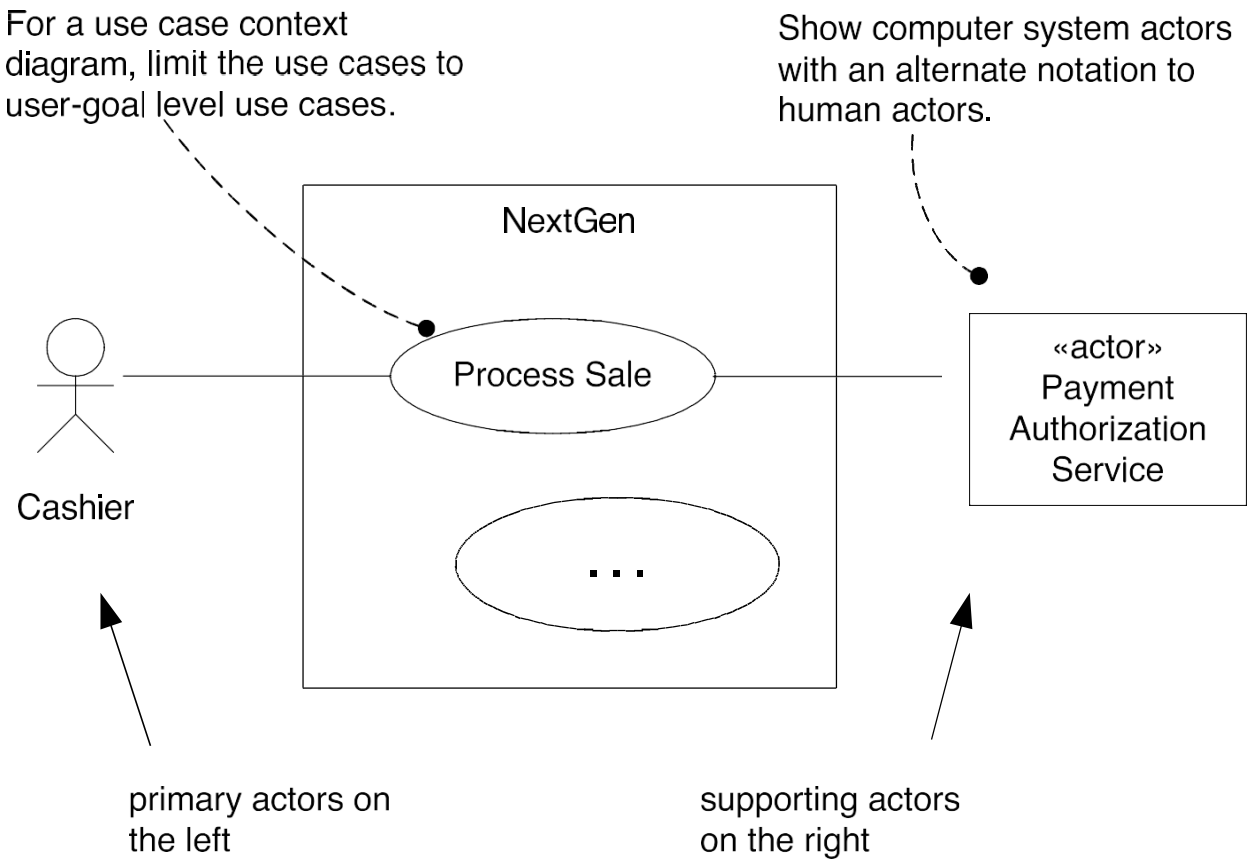
  Good Examples: Approve Credit or Price Order

  Bad Examples: delete a line item or print the document

- **The size test**

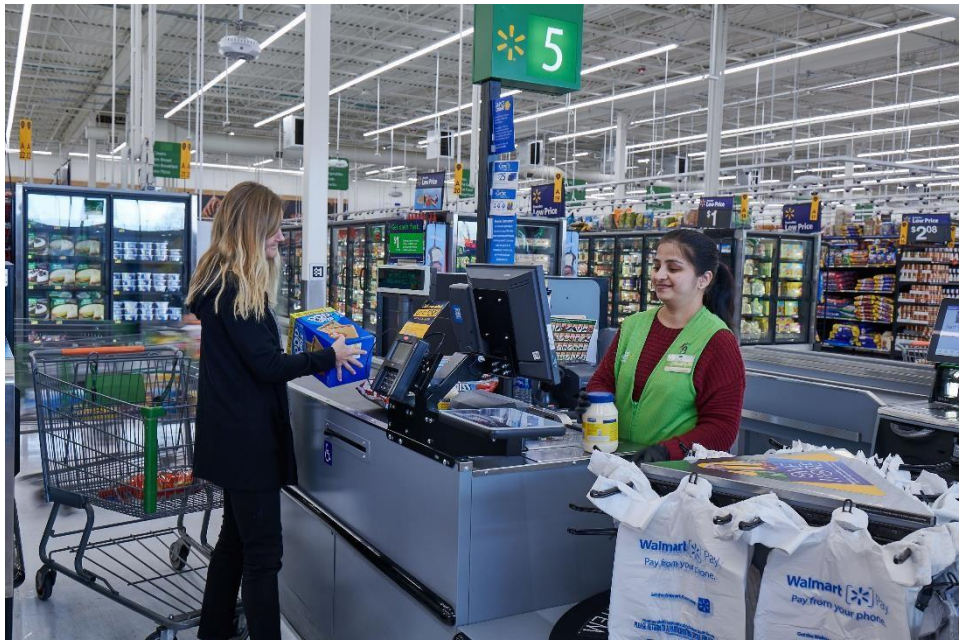
  - Just a single step in a sequence of others -> not good!

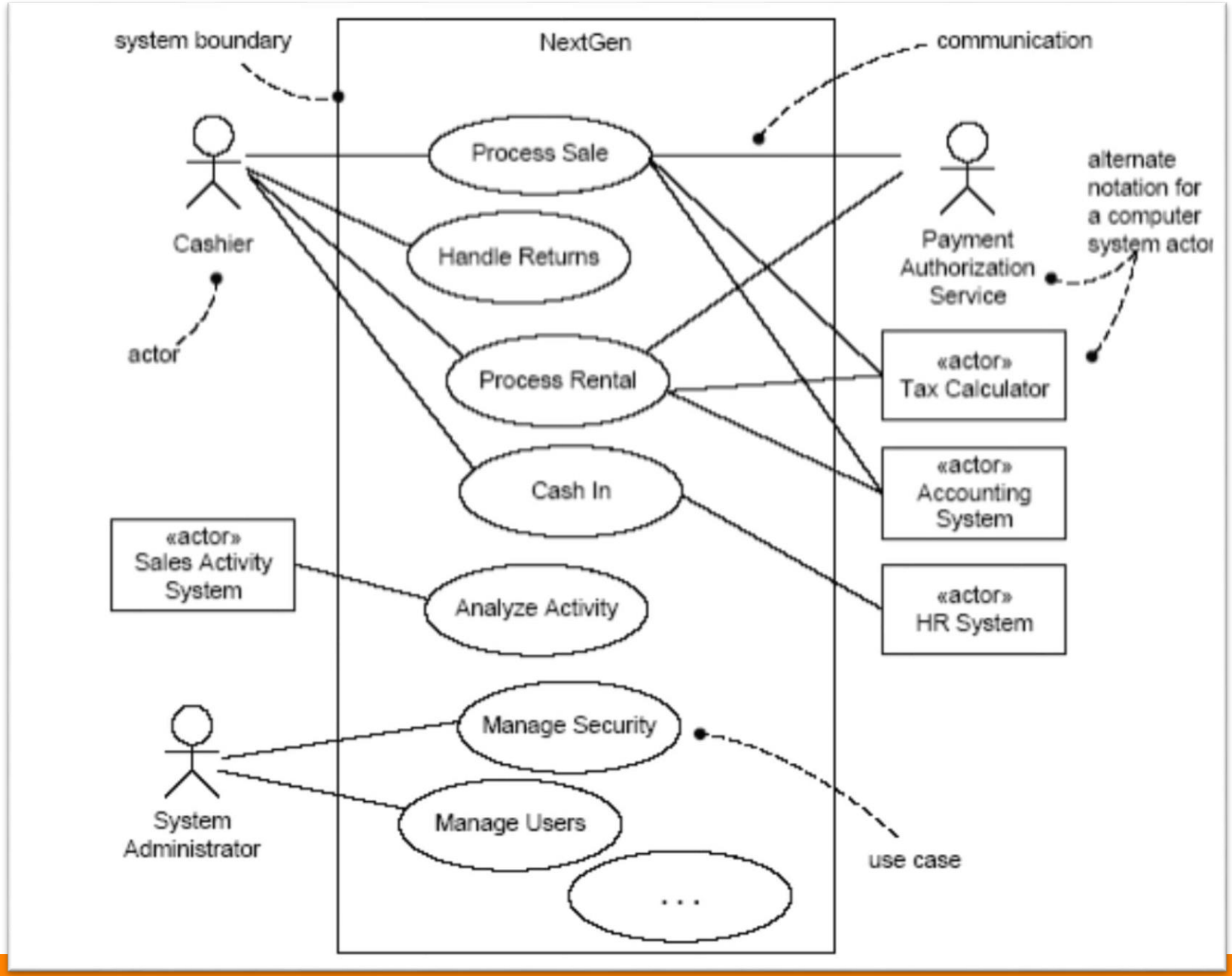
Use Case (Context) Diagrams: Suggested Notation



Point of Sale System

A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.




system boundary
NextGen
communication
Process Sale
Cashier
Handle Returns
Payment Authorization Service
alternate notation for a computer system actor
actor
Process Rental
«actor» Tax Calculator
Cash In
«actor» Accounting System
«actor» Sales Activity System
Analyze Activity
«actor» HR System
Manage Security
System Administrator
Manage Users
use case
. . .

Domain Model

- Why:
  Domain modeling helps us to identify the relevant concepts and ideas of a domain

- When:
  Domain modeling is done during object-oriented analysis

- Guideline:
  Only create a domain model for the tasks at hand

“Good designs require deep application knowledge.”

Curtis’law

Domain Model

- The Domain Model illustrates noteworthy concepts in a domain.
- The domain model is created during object-oriented analysis to decompose the domain into concepts or objects in the real world.
- The model should identify the set of conceptual classes (The domain model is iteratively completed.)
- Also known as Visual Dictionary
- It is the basis for the design of the software.
- The domain model is also called **conceptual model**, **domain object model** or **analysis object mode.**
  - **conceptual is idea, thing, object**

Domain Model

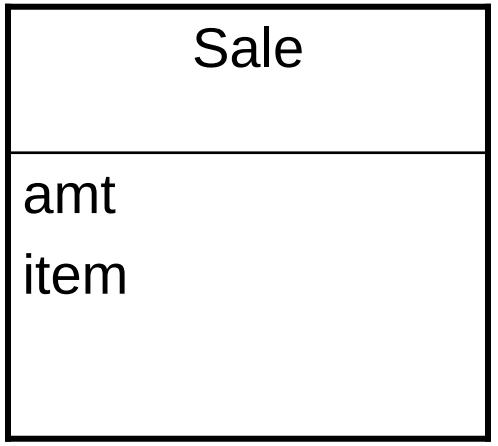
- Illustrates meaningful conceptual classes in problem domain (Analysis phase)
- It helps us identify, relate and visualize important information.
- Represents real-world concepts, not software components
- Software-oriented class diagrams will be developed later (during design phase)
- It provides inspiration for later creation of software design classes, to reduce "representational gap."

To visualize domain models the UML class diagram notation is used.

- However, no operations are defined in domain models
- Only ...
  - domain objects and conceptual classes
  - associations between them
  - attributes of conceptual classes

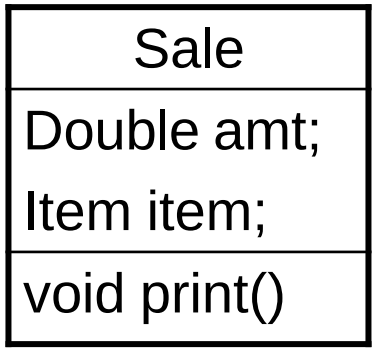
A Domain Model is Conceptual, not a Software Artifact

Conceptual Class:



vs.

Software Artifacts:

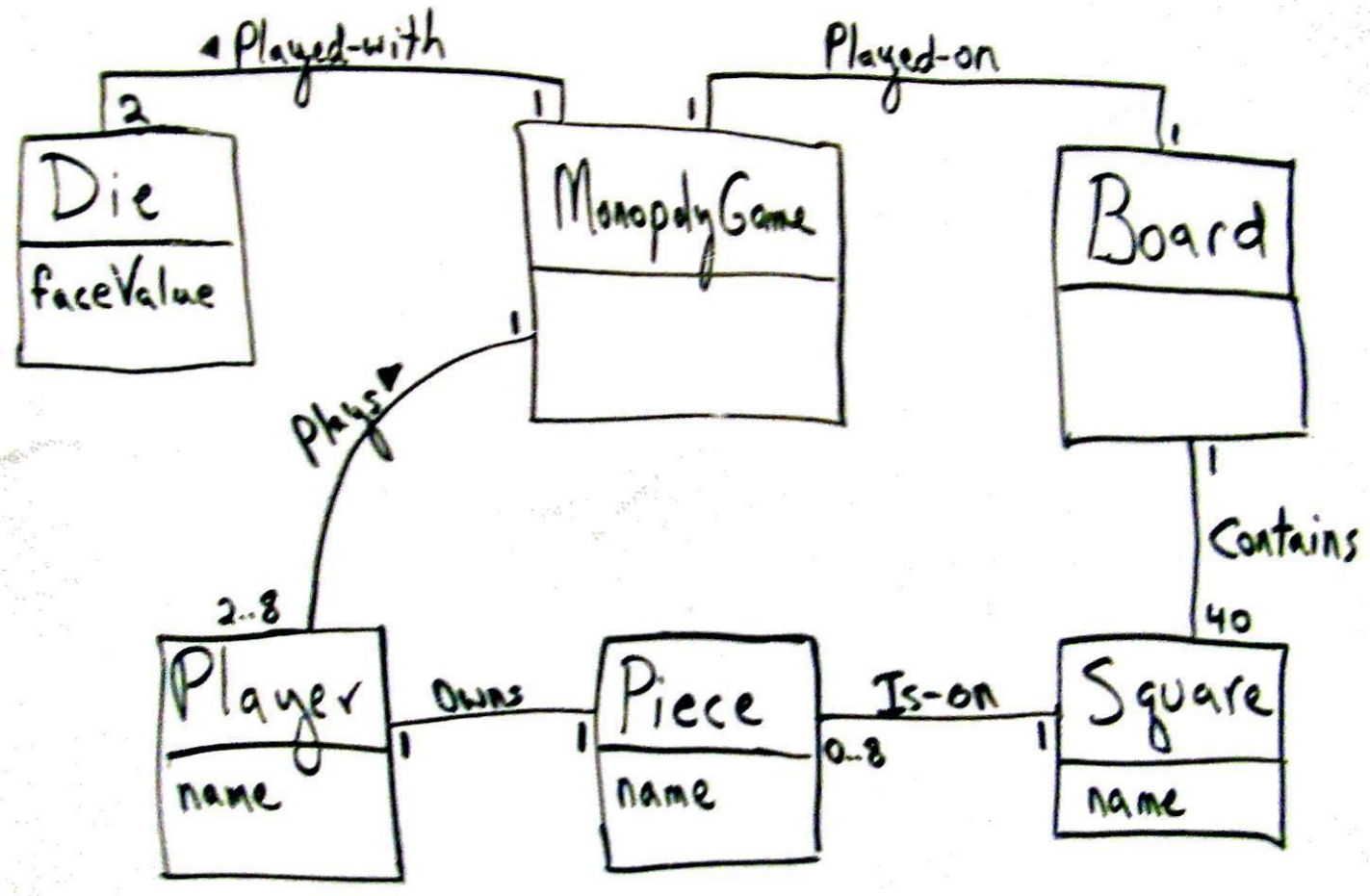


What's the difference?

Monopoly Game domain model (first identify concepts as classes)

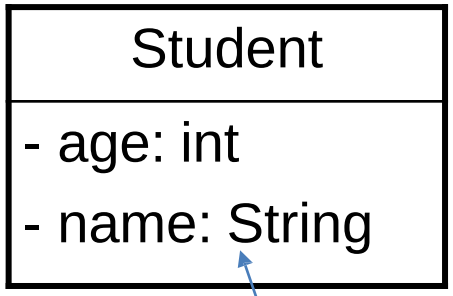
Monopoly Game	Dice	Board
Player	Piece	Square

Monopoly Game domain model
Larman, Figure 9.28



Syntax of Domain Model

Conceptual Class:



Attributes

Association

Registers

Cardinality

Registers

1 *

How to Create Domain Model

1. Find conceptual classes
2. Draw them as conceptual classes
3. Add associations and attributes

Finding Conceptual Classes

1. Reuse or modify the existing model if one exists
   - There are many published, well-crafted domain models.

2. Use Conceptual Category List

3. Identify noun phrases in your use-cases
   - Identify nouns and phrases in textual descriptions of a domain (use cases, or other documents)

Reuse or Modify Existing Models

- There are published, well---crafted domain models and data models (which can be modified into domain models) for many common domains, such as inventory, finance, health, and so forth..

- Knowledge sources are:
  - Interviews
  - Questionnaires
  - Mission Statements
  - Subject Matter Experts (SMEs)

- Reusing existing models is excellent

How to create the domain model?
Use a category list to find the conceptual classes.

Conceptual Class Category	Classes (for the POS system)
Business transactions...	Sale, Payment
Transaction line items...	SalesLineItem
Product or service related to a transaction or transaction line item.	Item
Where is the transaction recorded?	Register
Roles of people or organizations related to the transaction; actors in use cases.	Cashier, Customer, Store
Place of transactions.	Store
Noteworthy events, often with a time or place that needs to be remembered.	Sale, Payment
...	...

Identify noun phrases to find the conceptual classes

- Identify the nouns and noun phrases in textual descriptions of a domain and consider them as candidate conceptual classes or attributes.

"Use Cases" are also an excellent source for identifying conceptual classes.

Main Success Scenario (or Basic Flow):

1) Customer arrives at a POS checkout with goods and/or services to purchase.
2) Cashier starts a new sale.
3) Cashier enters item identifier.
4) System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 2-3 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.
8) System logs the completed sale and sends sale and payment information to the external Accounting (for accounting and commissions) and Inventory systems (to update inventory).
9) System presents receipt.
10) Customer leaves with receipt and goods (if any).

Main Success Scenario (or Basic Flow):

1) Customer arrives at a POS checkout with goods and/or services to purchase.
2) Cashier starts a new sale.
3) Cashier enters item identifier.
4) System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 2-3 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.
8) System logs the completed sale and sends sale and payment information to the external Accounting (for accounting and commissions) and Inventory systems (to update inventory).
9) System presents receipt.
10) Customer leaves with receipt and goods (if any).

List and Draw UML Diagram

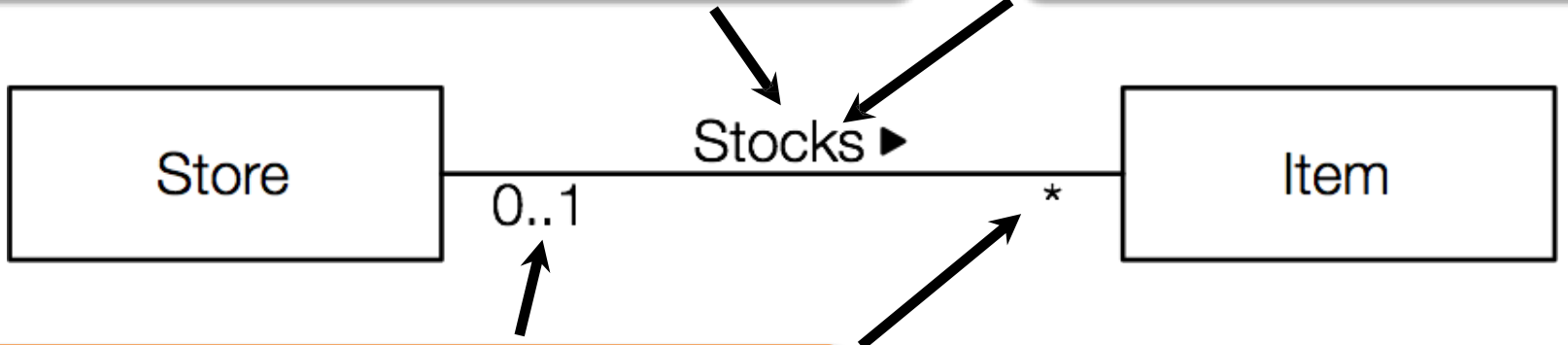
1. Sale
2. Cash Payment
3. SaleLineItem
4. Item
5. Register
6. Amount
7. Cashier

1. Decide which ones are classes and which ones are attributes
2. Add attributes and relation to the identified domain classes

Association Notations

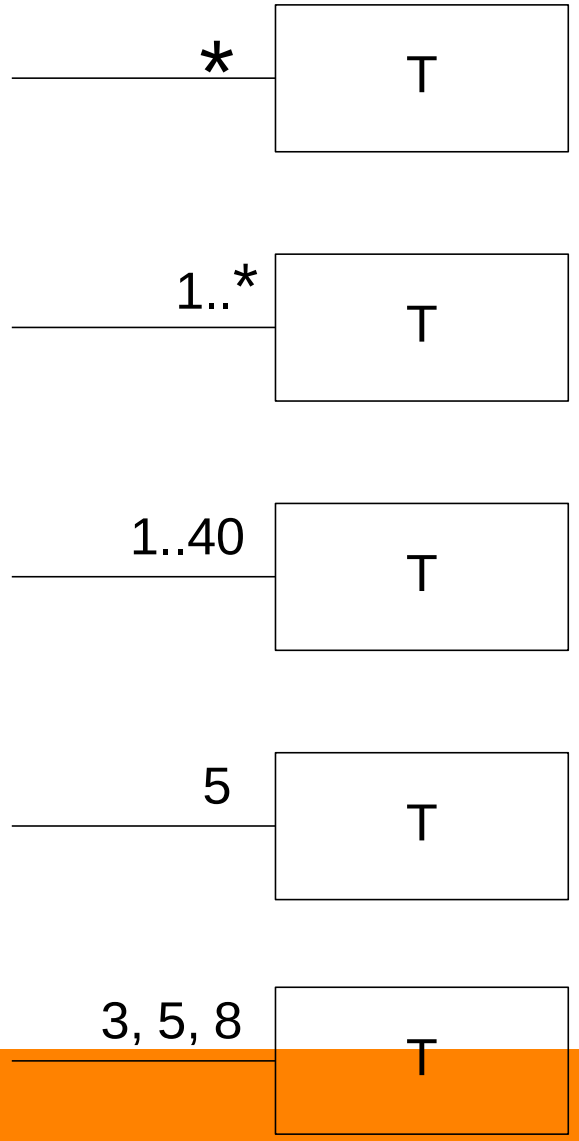
Association name:
- ✓ Use verb phrase
- ✓ Capitalize
- ✓ Typically camel-case or hyphenated
- ✓ Avoid “has”, “use”

Reading direction:
Can exclude if association reads left-to-right or top-to-bottom



Multiplicity (Cardinality):
- ✓ ‘*’ means “many”
- ✓ x..y means from x to y inclusively

Cardinality (or Multiplicity)



zero or more;
"many"

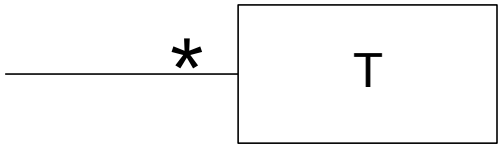
one or more

one to 40

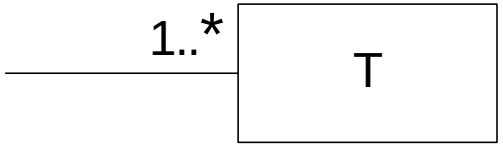
exactly 5

exactly 3, 5, or 8

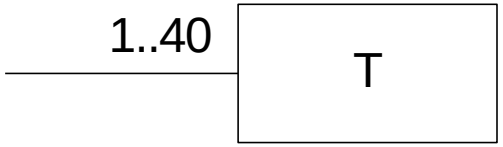
UML: Multiplicity



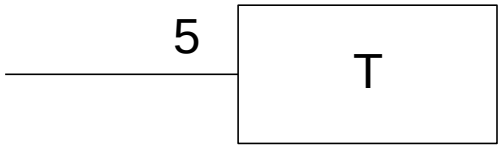
zero or more;
"many"



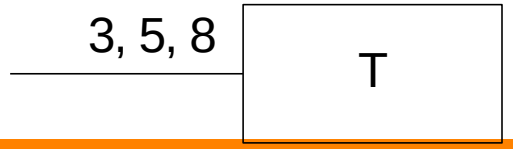
one or more



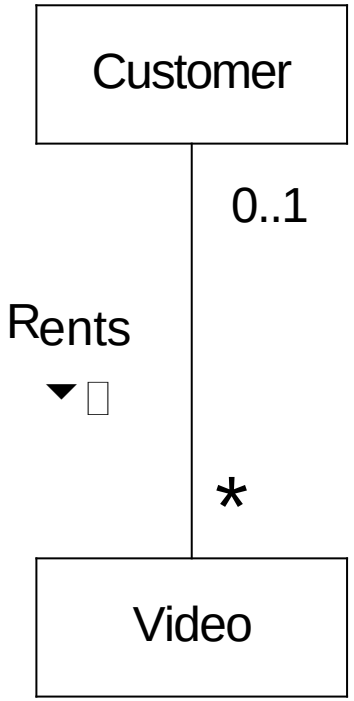
one to forty



exactly five



exactly three,
five or eight

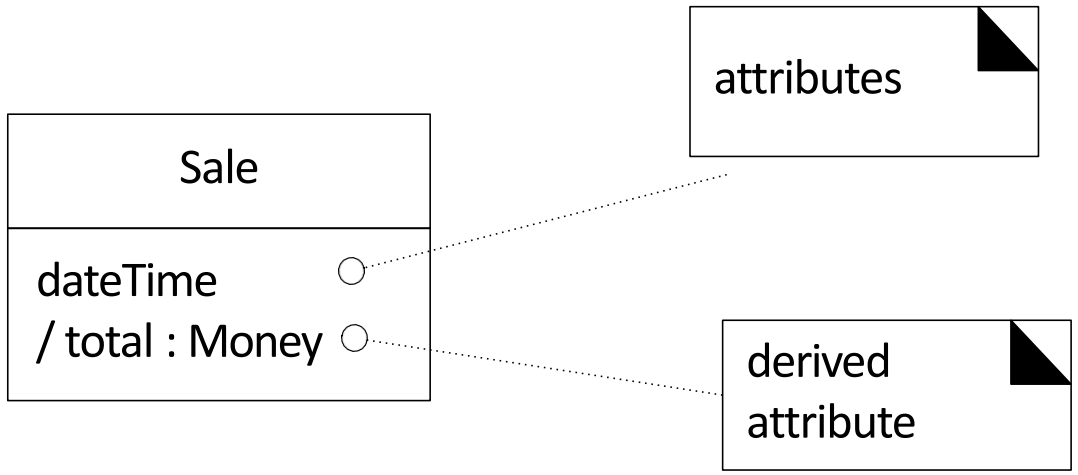


One instance of a Customer may be renting zero or more Videos.

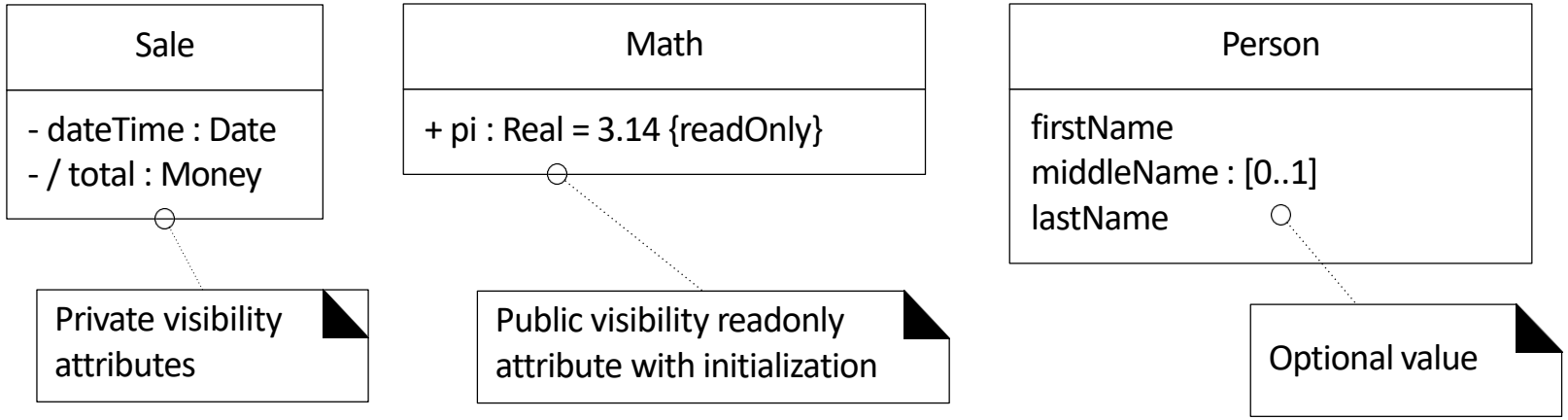
One instance of a Video may be being rented by zero or one Customers.

Attributes

- An object's **logical data value** that must be remembered
  - Some attributes are derived from other attributes



Visibility in Domain Models



The full UML attribute syntax:

[visibility] name : [type] [multiplicity] [= default] [property-string]

e.g. + store : Walmart [1..*]
{readOnly}

EXAMPLES

Point of Sale

Main Success Scenario (or Basic Flow):

1) Customer arrives at a POS checkout with goods and/or services to purchase.
2) Cashier starts a new sale.
3) Cashier enters item identifier.
4) System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

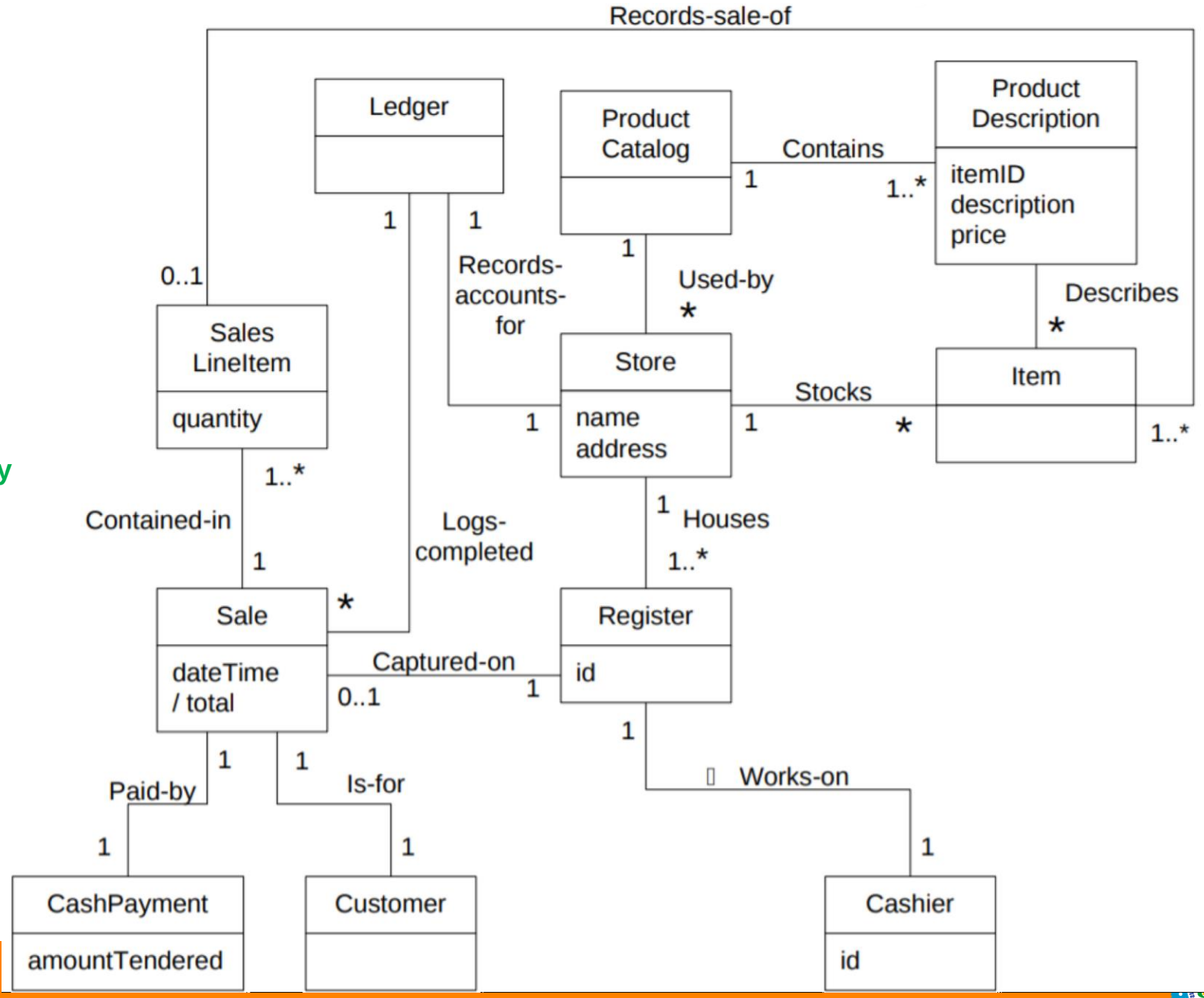
Cashier repeats steps 2-3 until indicates done.

5) System presents total with taxes calculated.
6) Cashier tells Customer the total, and asks for payment.
7) Customer pays and System handles payment.
8) System logs the completed sale and sends sale and payment information to the external Accounting (for accounting and commissions) and Inventory systems (to update inventory).
9) System presents receipt.
10) Customer leaves with receipt and goods (if any).

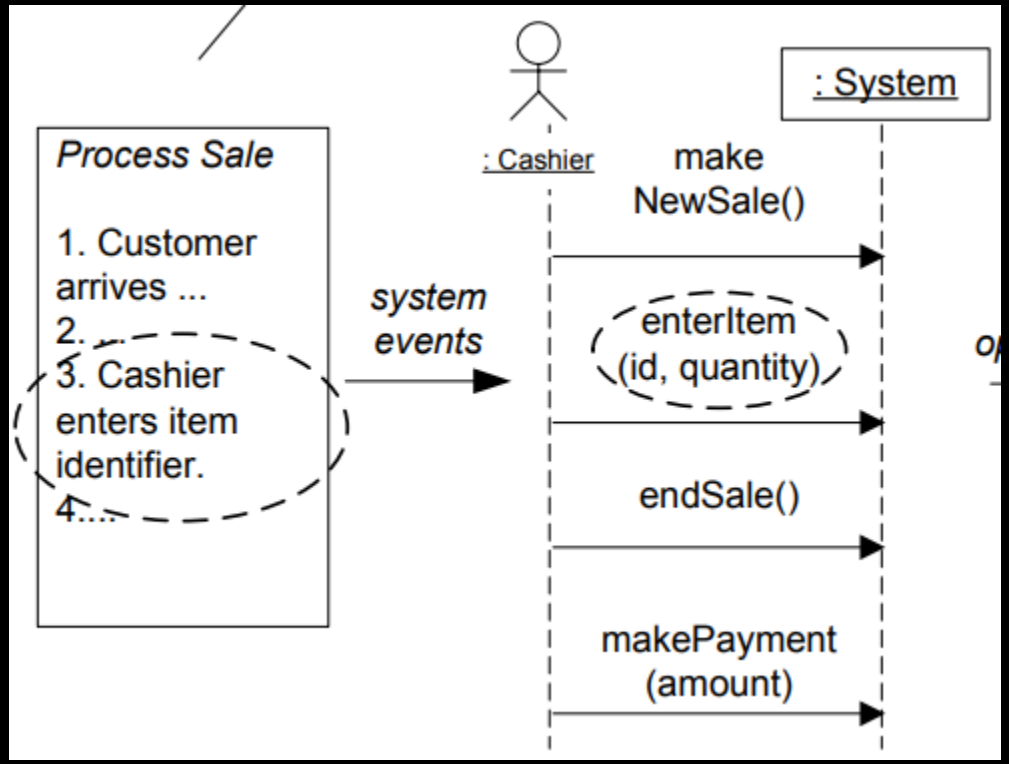
4. Domain Model

Point of Sale

1. Is there any register?
2. Is there any actor?
3. Is there any service or product?
4. Is there any physical place where activity has been performed?
5. Is there any report?
6. Is there any payment mechanism?
7. Is there any description class?



System Sequence Diagram



quest lab

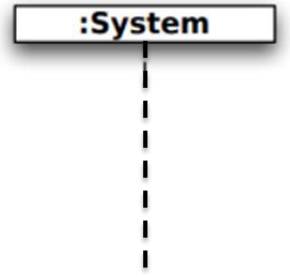
System Sequence Diagrams

System Sequence Diagrams (SSDs) play a significant role in the analysis and design phase of software development, particularly in the context of object-oriented systems.

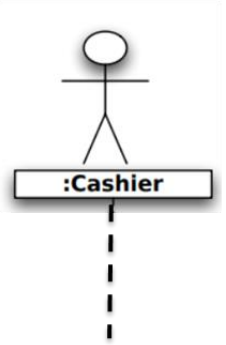
Visualizing System Interactions: SSDs provide a graphical representation of interactions between external actors (users or other systems) and the system under design. They illustrate the sequence of messages or actions exchanged between actors and the system in response to various scenarios or use cases.

What are Elements of an SSD?

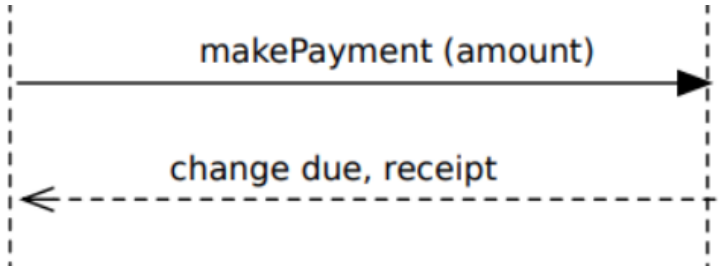
Objects
In a traditional UML, the box depicting a title represents the class or an object but, in System Sequence Diagram, this shape represents the system as the black-box. The black box is a system that is not illuminated to the outside world because they contain inner processes.



Actors
These are the external entities that interact with our system, and the system is created by keeping actors in mind. The interaction can be like interacting with the cashier, customer retrieving amount from the ATM or anything. They are the stick-like figures in the diagram.



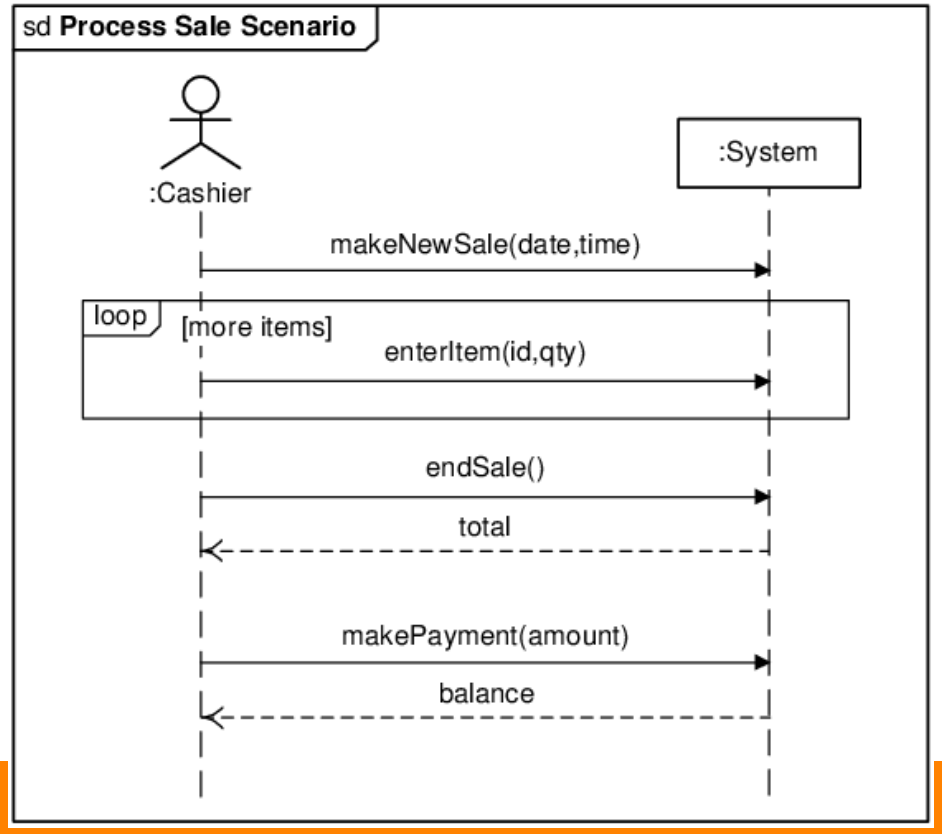
Events
The interaction or communication of an actor or system to the system is an event. Then with the specific event, the system performs the specific task. These events are generated in a sequence or a specific order. The dashed lines aka, the lifelines, are the events in the system sequence diagram. These lines start from the rectangle shape o with an actor.

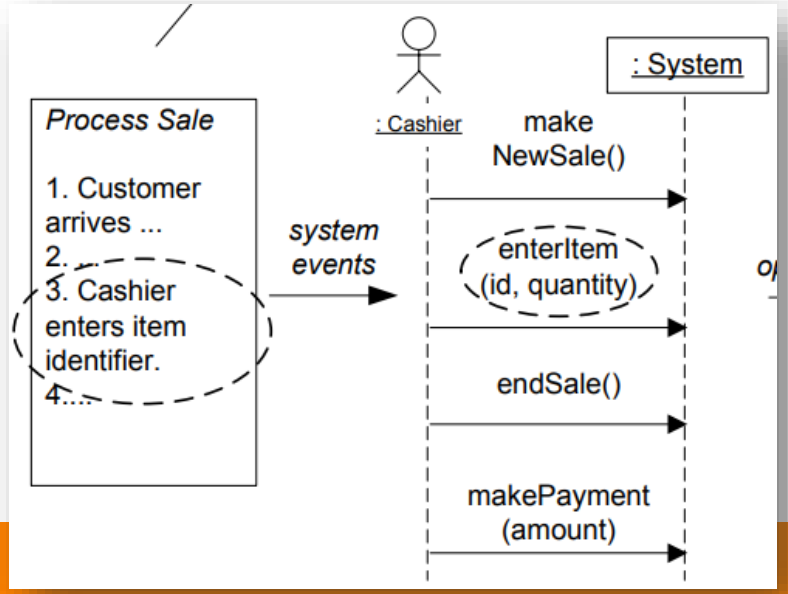


Main Success Scenario

1. Cashier starts newsale
2. Cashier enters itemidentifier
3. System records sale line item and presents item description, price and runningtotal
   - Steps 2 and 3 are repeated until all items are processed.
4. System presents total with taxescalculated
5. Cashier tells Customer the total and asks for payment
6. Customer pays and System handlespayment

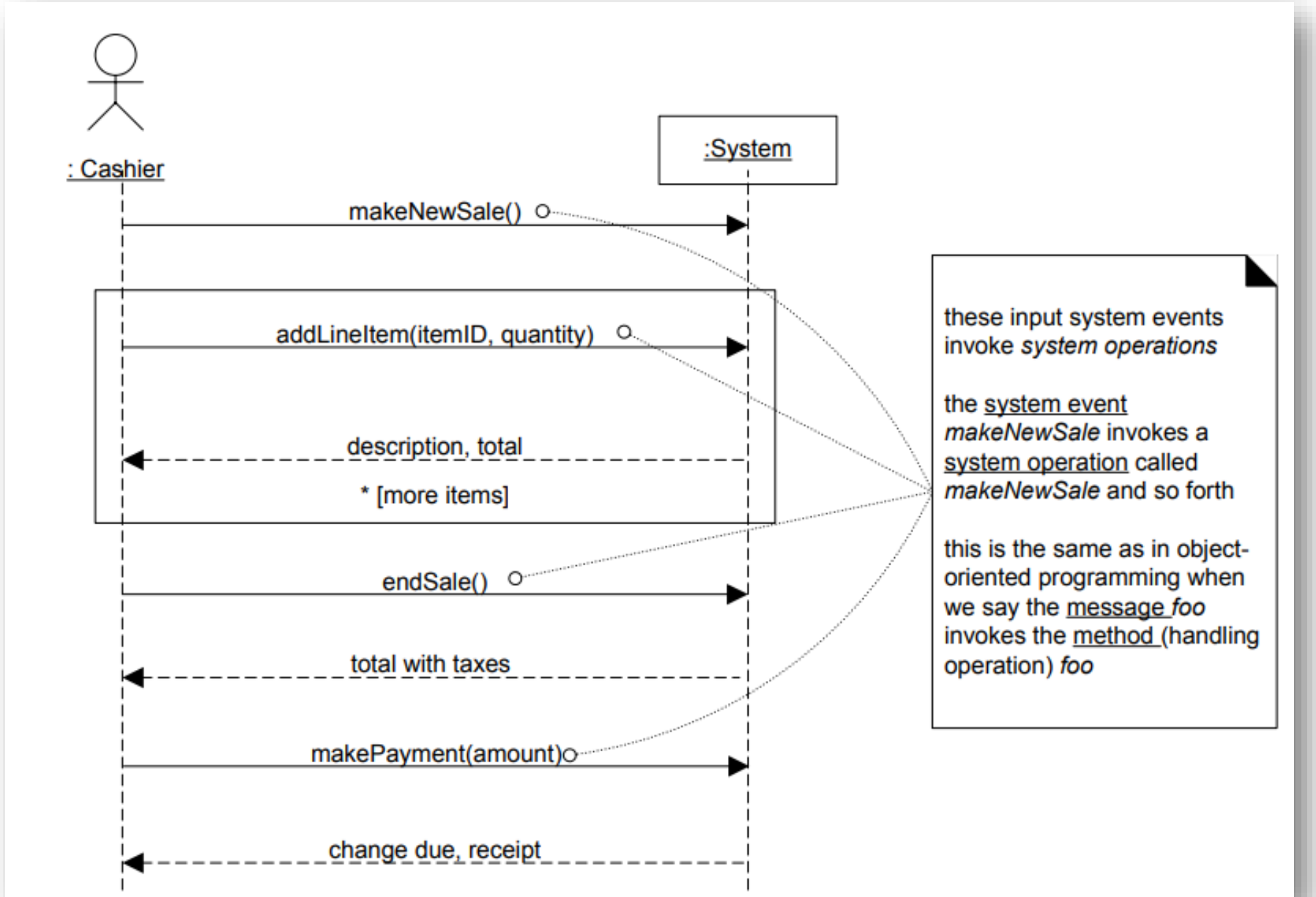
Interaction between cashier and system to perform process Sale





System Sequence Diagram

- An SSD shows – for one particular scenario of a use case –
  - the events that external actors generate,
  - their order, and
  - inter-system events
- The system is treated as a black-box (no implementation details).
- A description of “What” system does with some time aspects.
- SSDs are derived from use cases.
- SSDs are often drawn for the main success scenarios of each use case and frequent or complex alternative scenarios.
- SSDs are used as input for object design


: Cashier
:System
makeNewSale()
addLineItem(itemID, quantity)
description, total
* [more items]
endSale()
total with taxes
makePayment(amount)
change due, receipt
these input system events invoke *system operations*
the system event *makeNewSale* invokes a system operation called *makeNewSale* and so forth
this is the same as in object-oriented programming when we say the message *foo* invokes the method (handling operation) *foo*

Sequence Diagram

- Sequence diagram simply depicts *interaction between objects* in a sequential order (i.e. the order in which these interactions take place.)

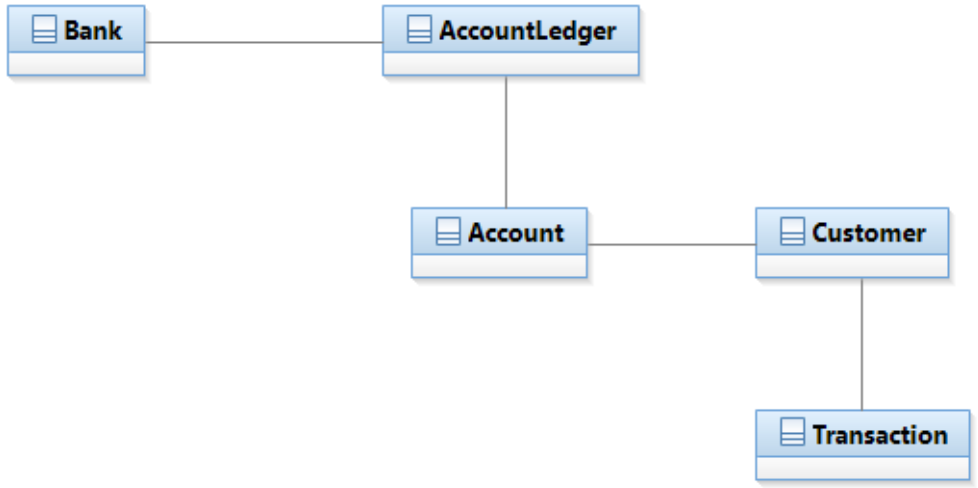
- Sequence diagrams describe *how* and in what order the objects in a system function

Sequence Diagram

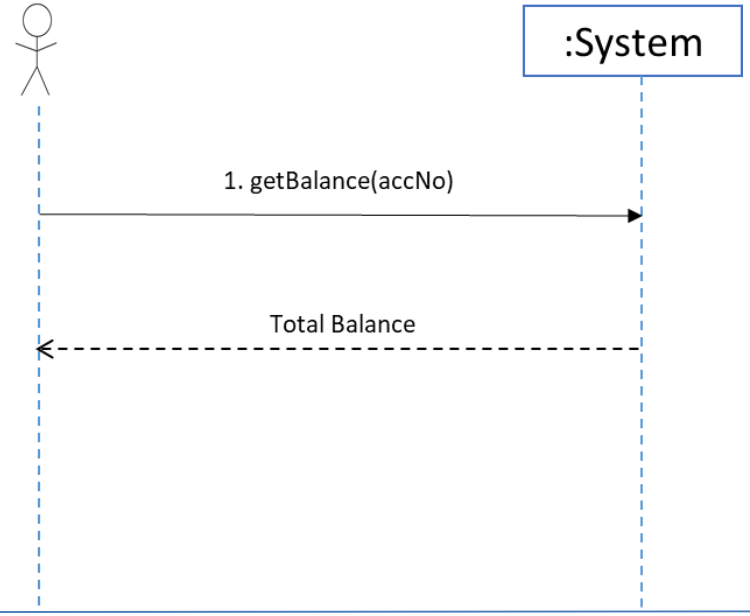
They illustrate how the different parts of a system interact with each other to carry out a function, and the order in which the interactions occur when a particular use case is executed

Interaction diagrams are used to visualize the interaction via messages between objects; they are used for *dynamic object modeling*.

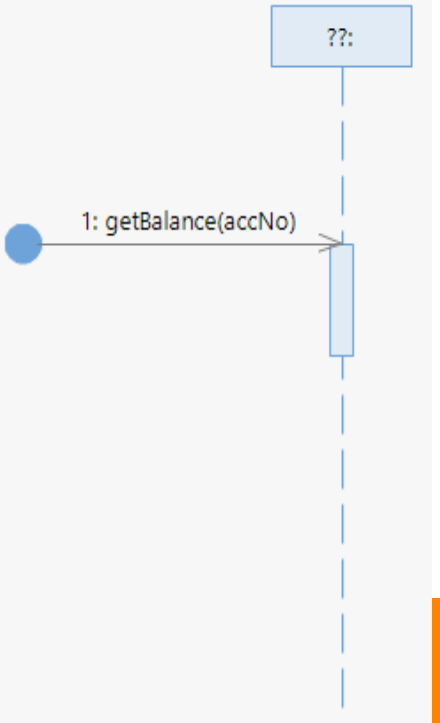
Domain Model

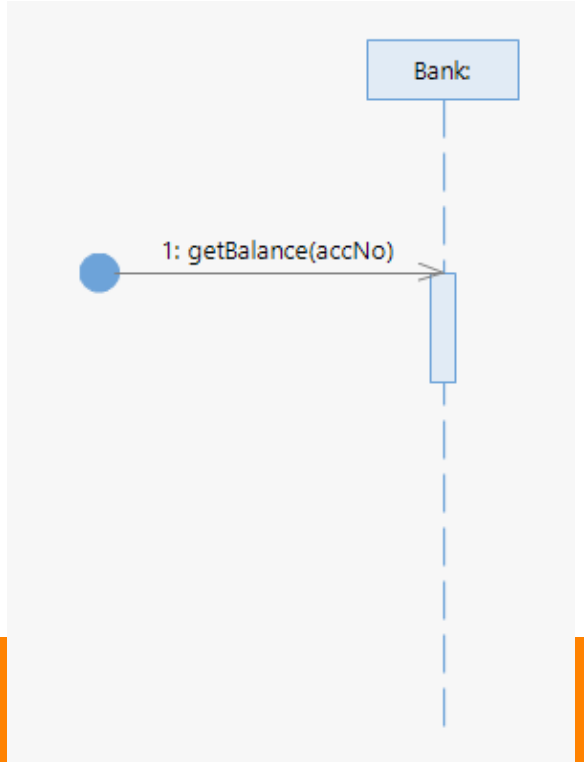


System Sequence Diagram



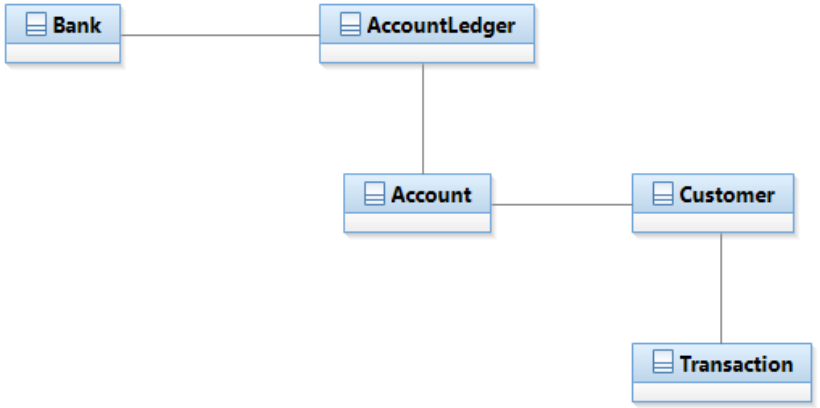
Sequence Diagram



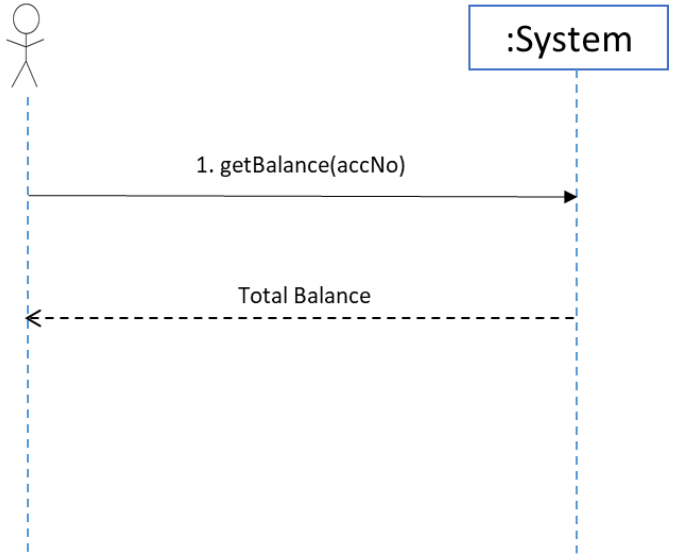


Example

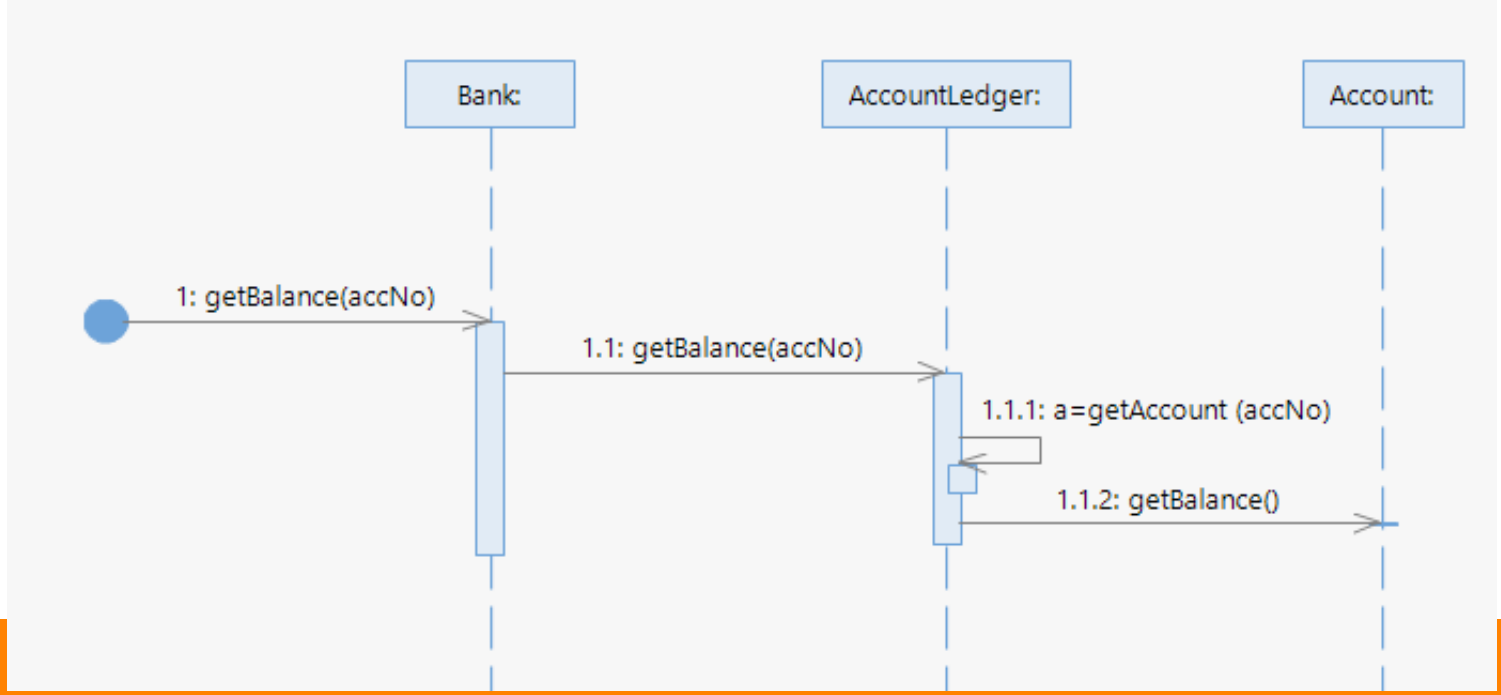
Domain Model



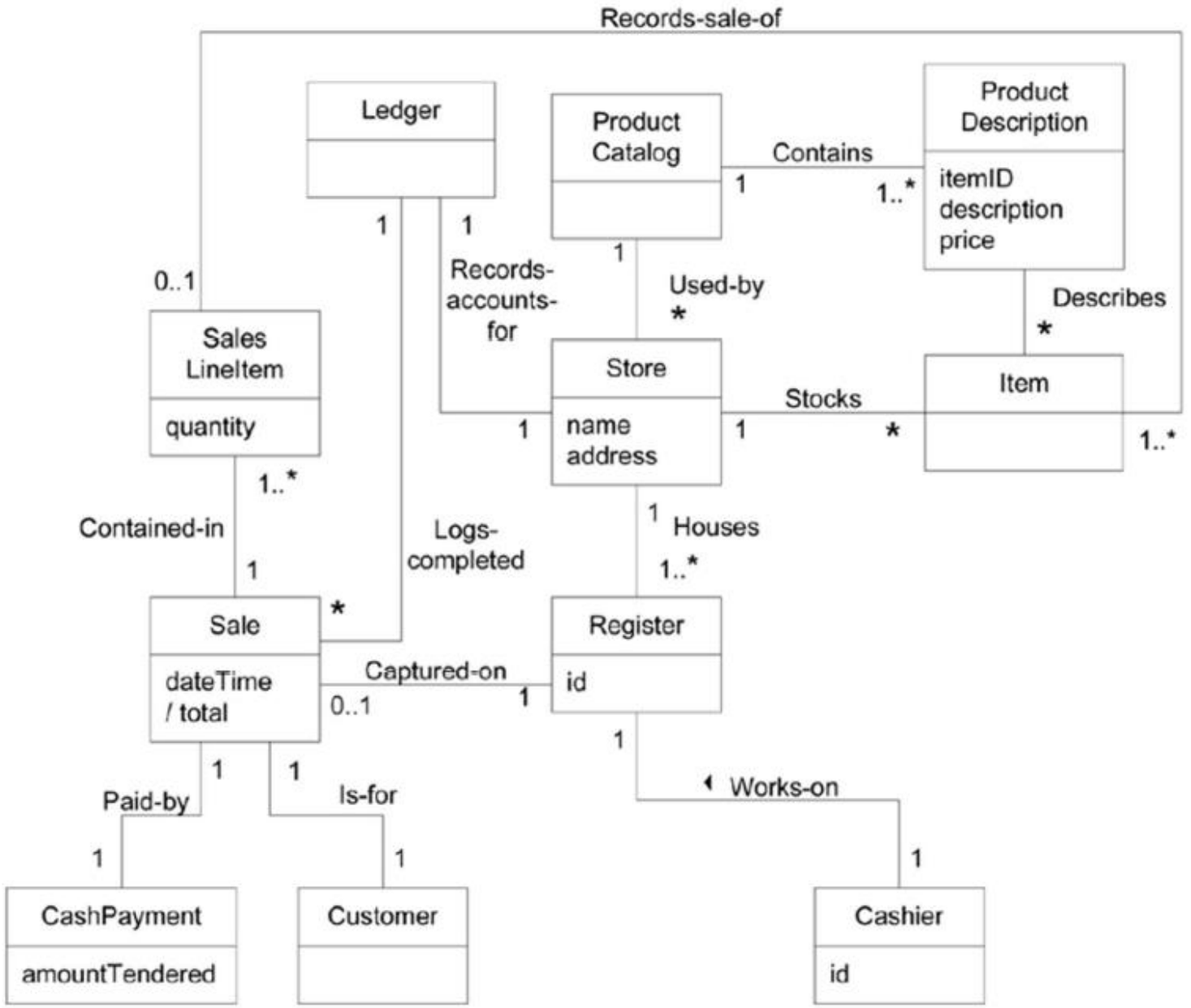
System Sequence Diagram



Sequence Diagram



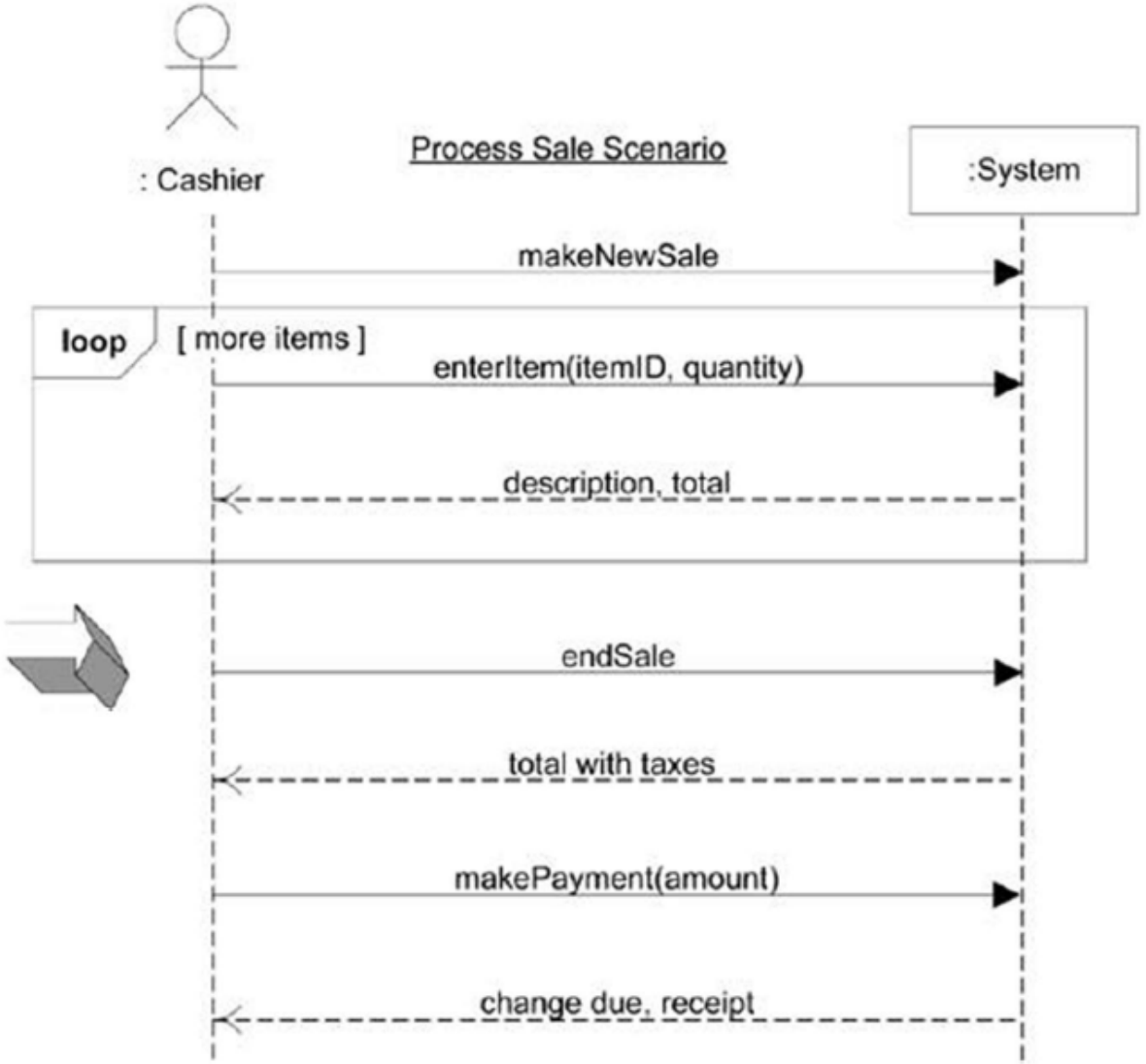
Domain Model



System Sequence Diagram

Simple cash-only Process Sale scenario:

1. Customer arrives at a POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
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.
...

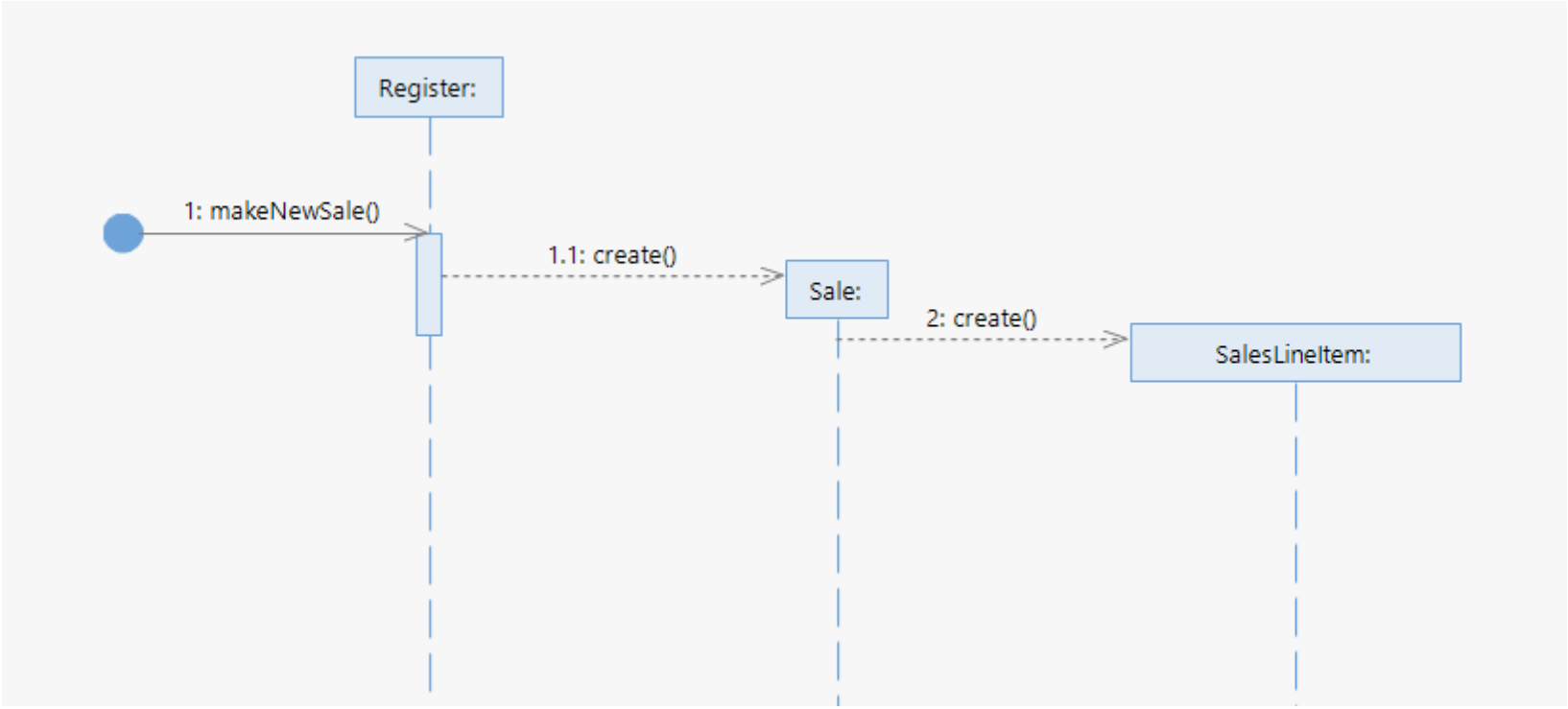


makeNewSale()

Controller ?

Information Expert ??

Creator ??

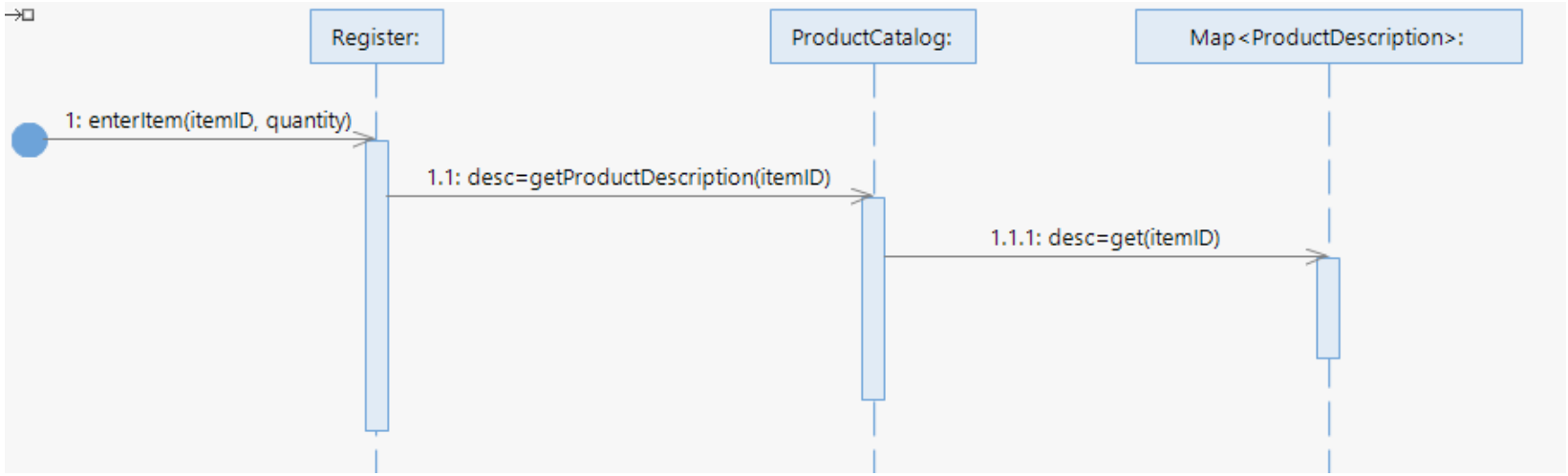


enterNewItem(itemID, quantity)

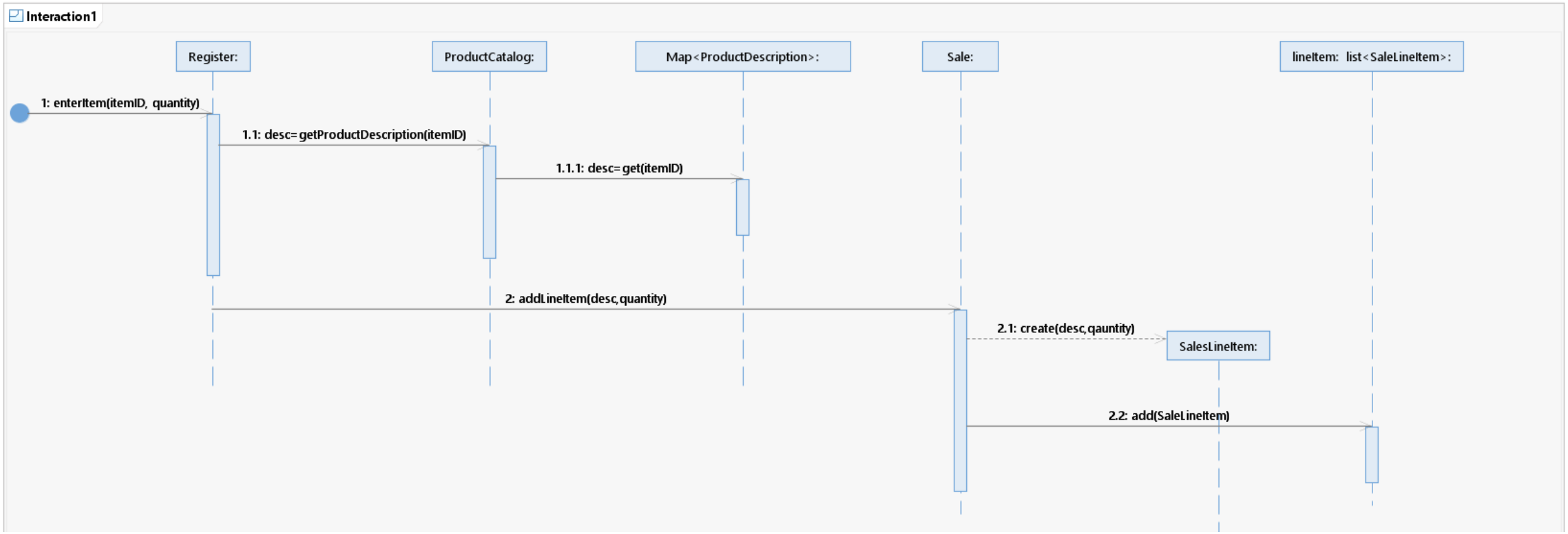
Controller ?

Information Expert ??

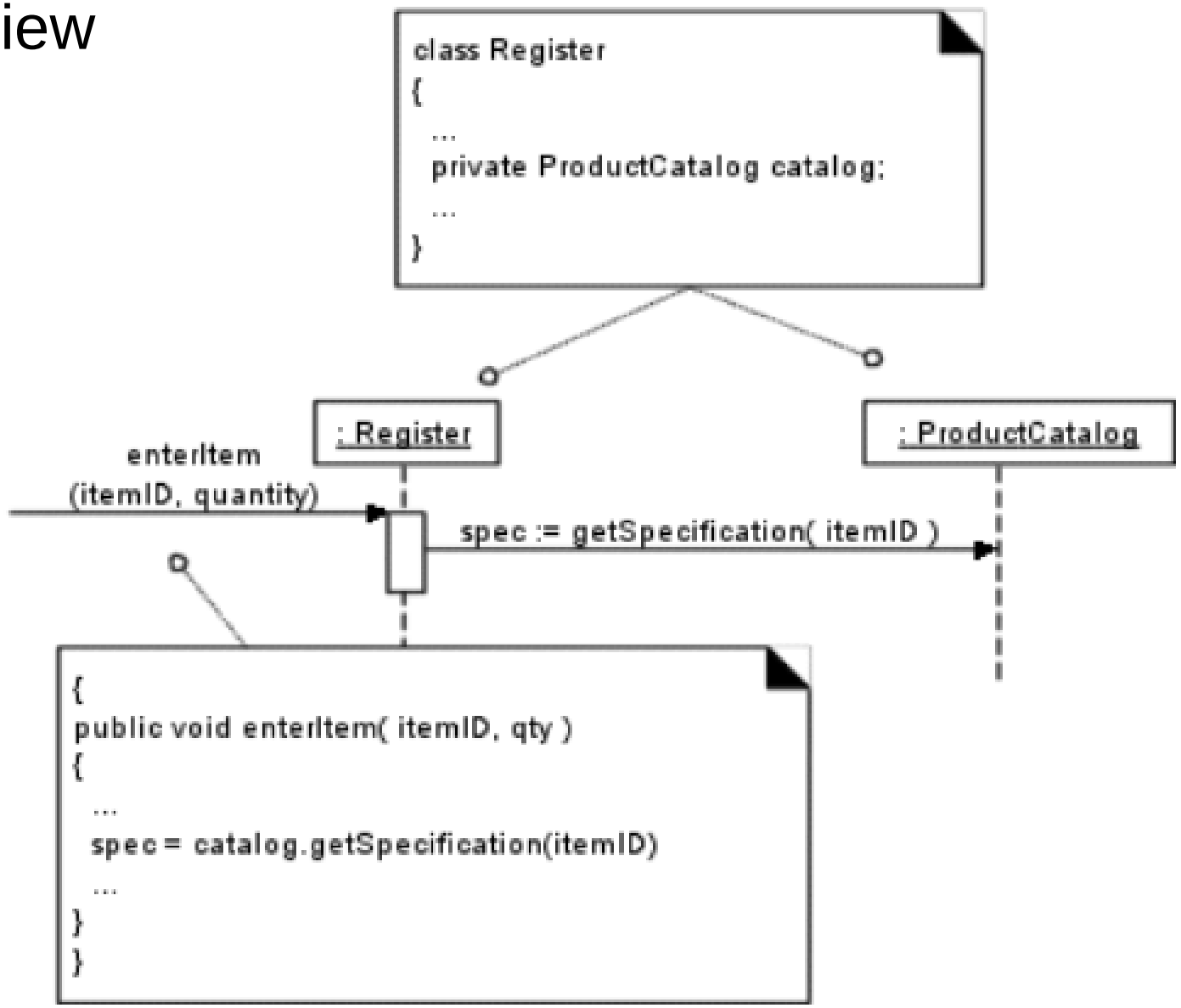
Creator ??

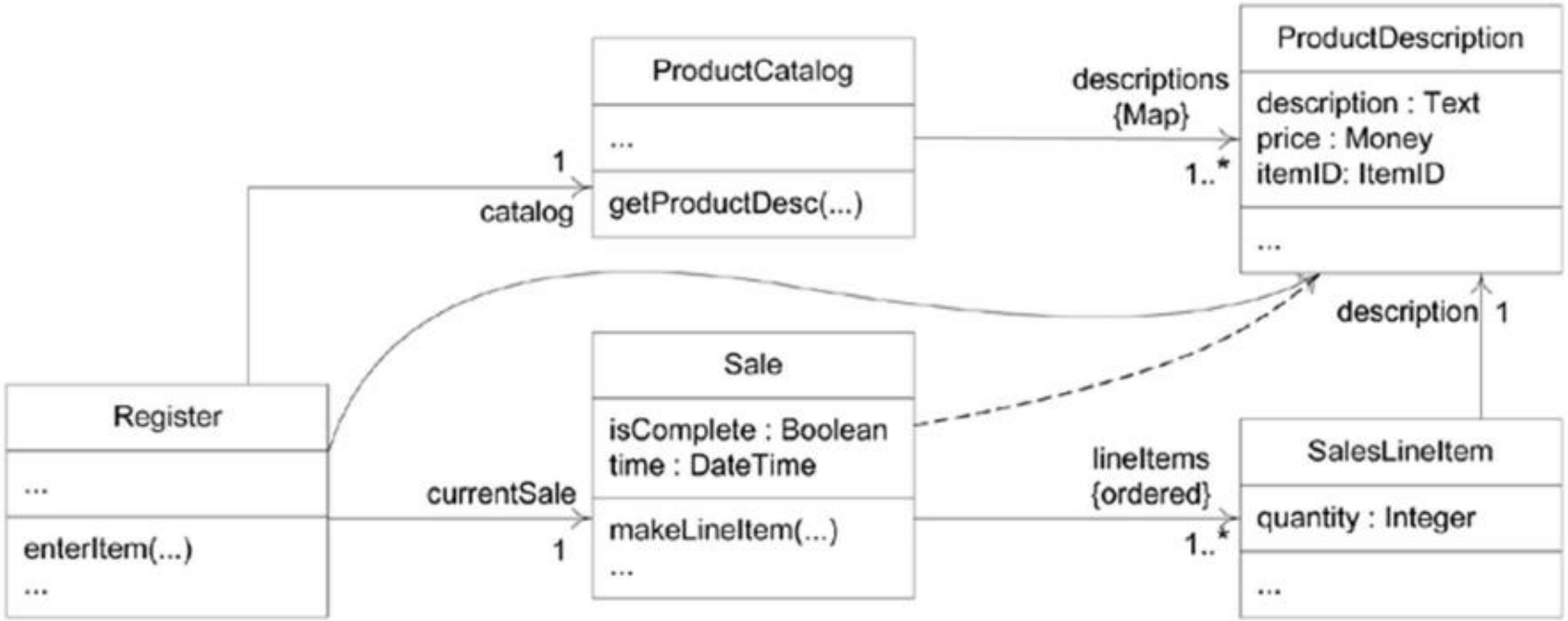


First step: access **ProductDescription** based on the itemID
ProductCatalog is information expert of ProductDescriptions


Interaction1
Register:
ProductCatalog:
Map<ProductDescription>:
Sale:
lineItem: list<SaleLineItem>:
1: enterItem(itemID, quantity)
1.1: desc= getProductDescription(itemID)
1.1.1: desc= get(itemID)
2: addLineItem(desc,quantity)
2.1: create(desc,qauntity)
SalesLineItem:
2.2: add(SaleLineItem)

Code level view

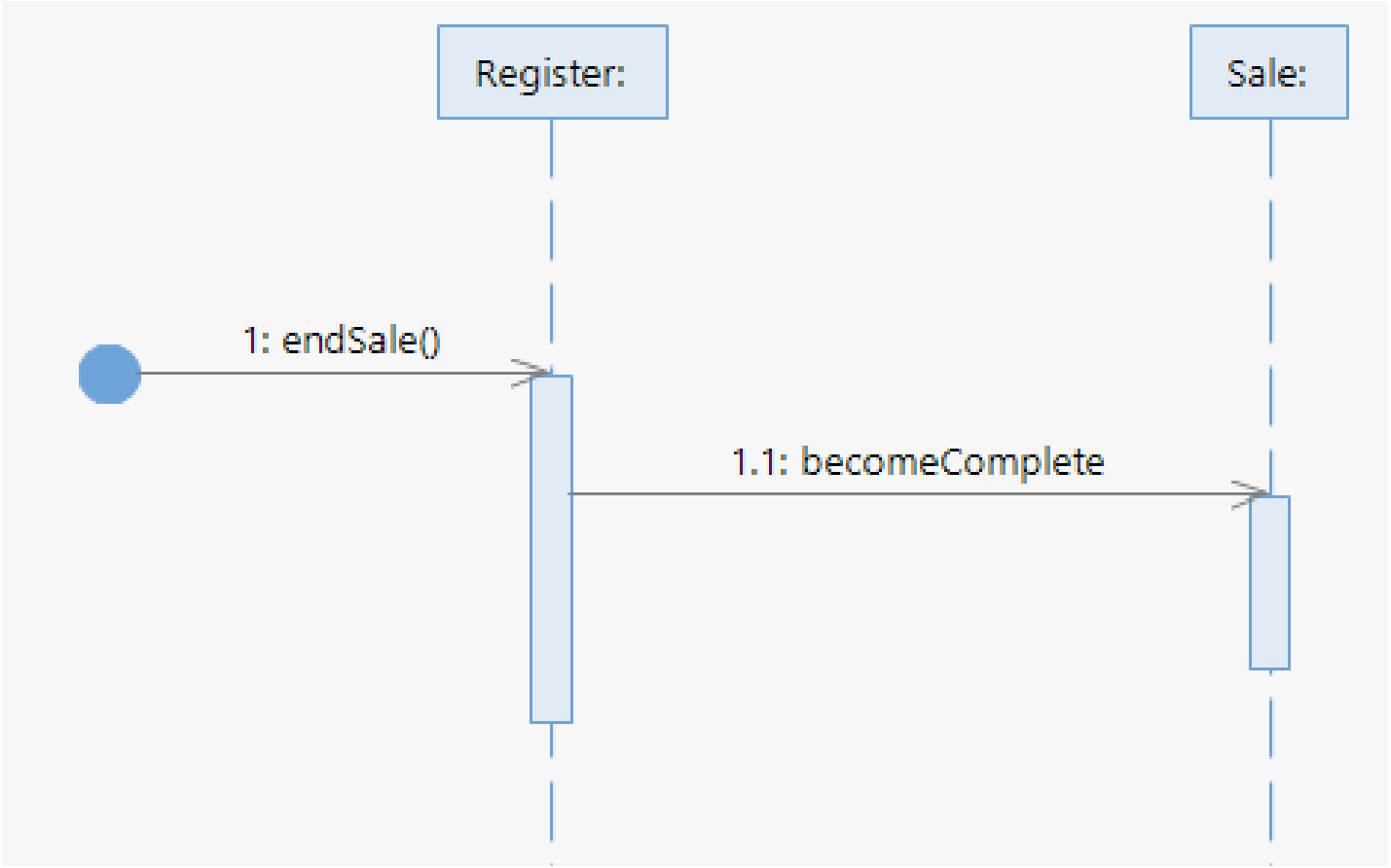


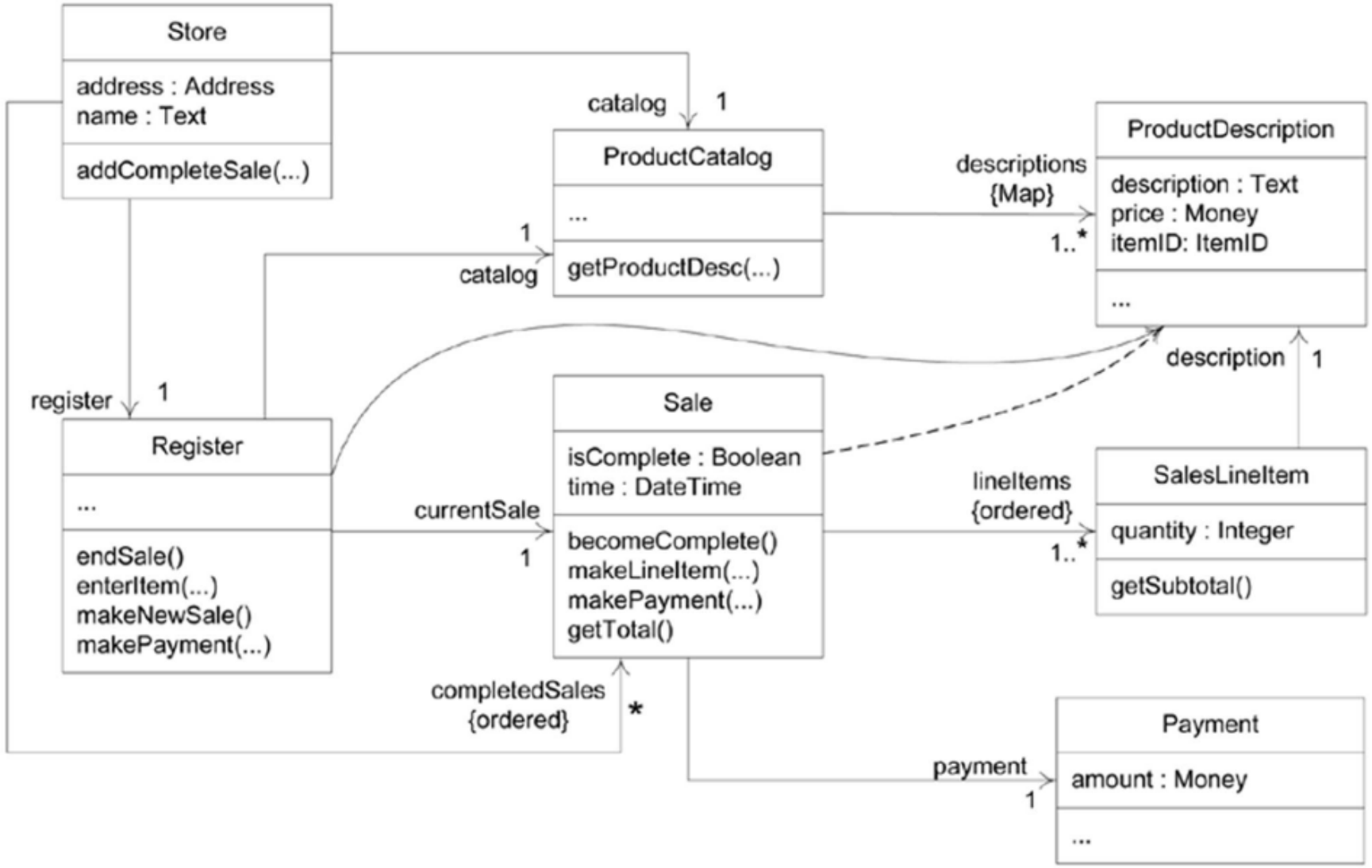

ProductCatalog
...
getProductDesc(...)
1
catalog
descriptions
{Map}
1..*
ProductDescription
description : Text
price : Money
itemID: ItemID
...
description
1
Register
...
enterItem(...)
...
currentSale
1
Sale
isComplete : Boolean
time : DateTime
makeLineItem(...)
...
lineItems
{ordered}
1..*
SalesLineItem
quantity : Integer
...

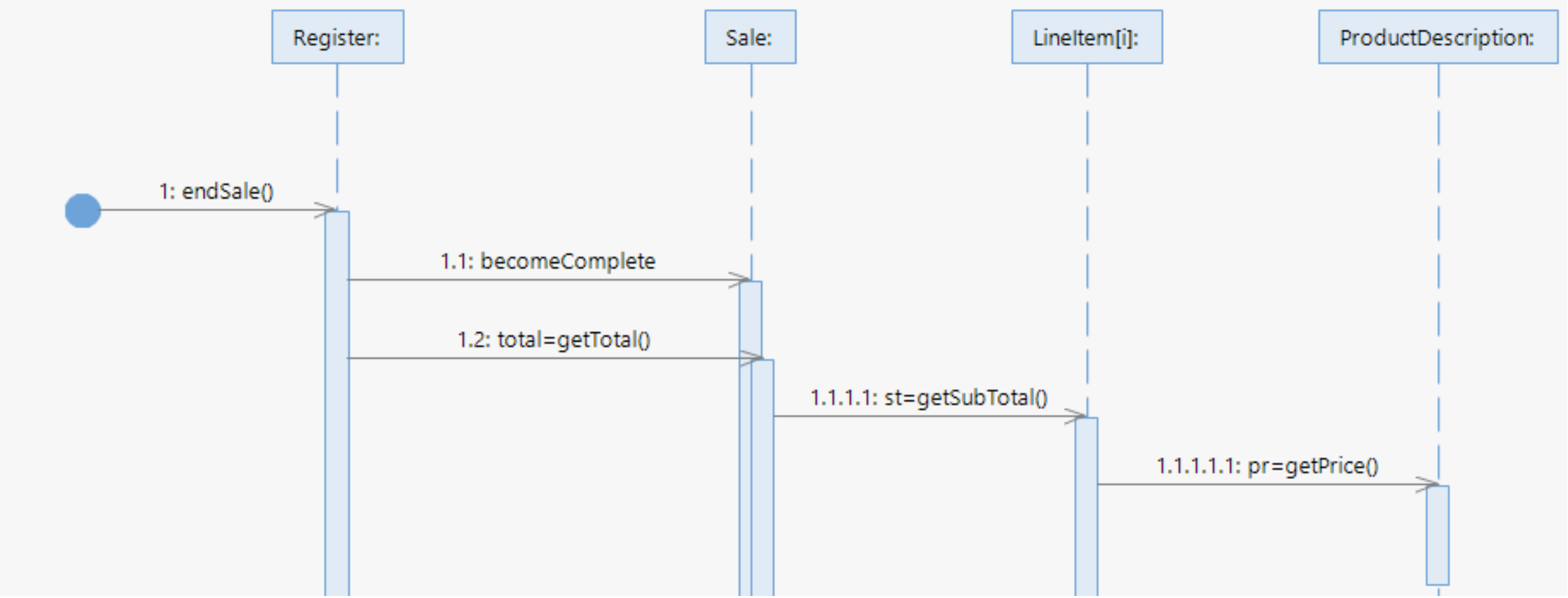
endSale – Design Decisions

- Sale is to be completed
- Total with tax calculated is presented

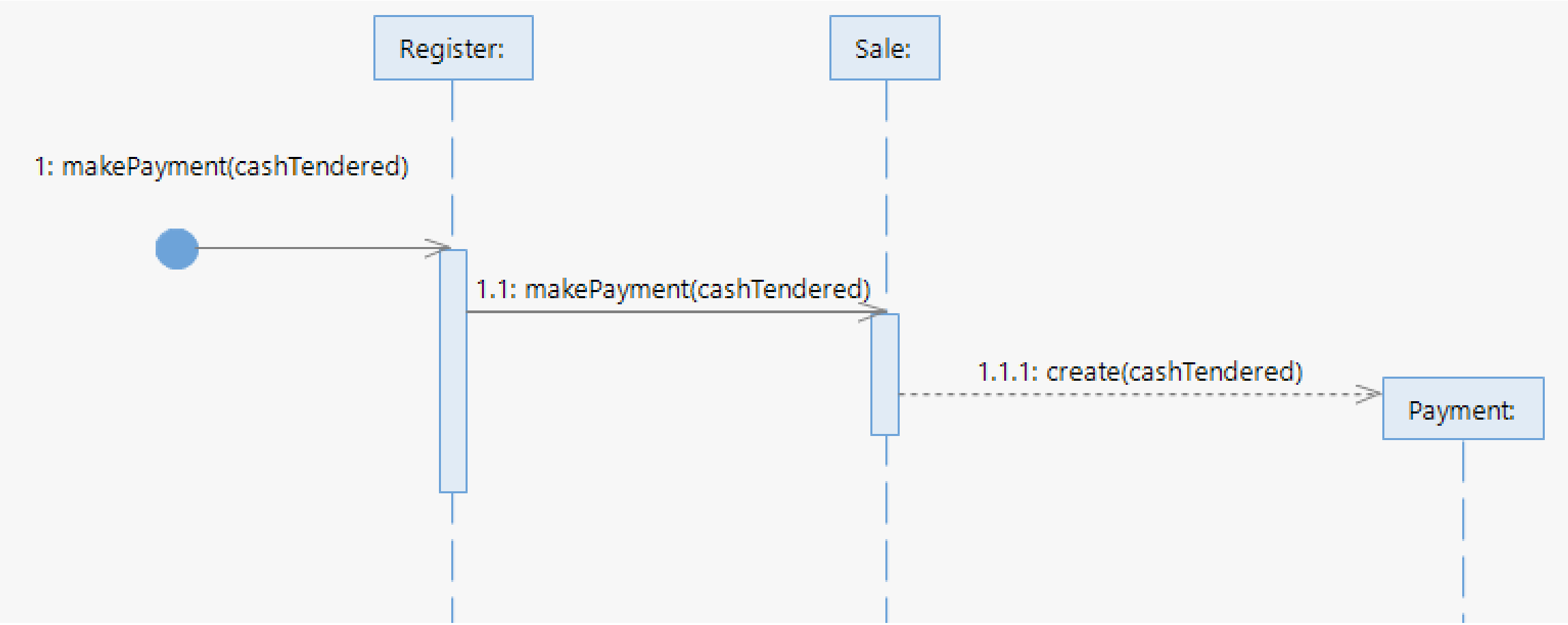
endSale()

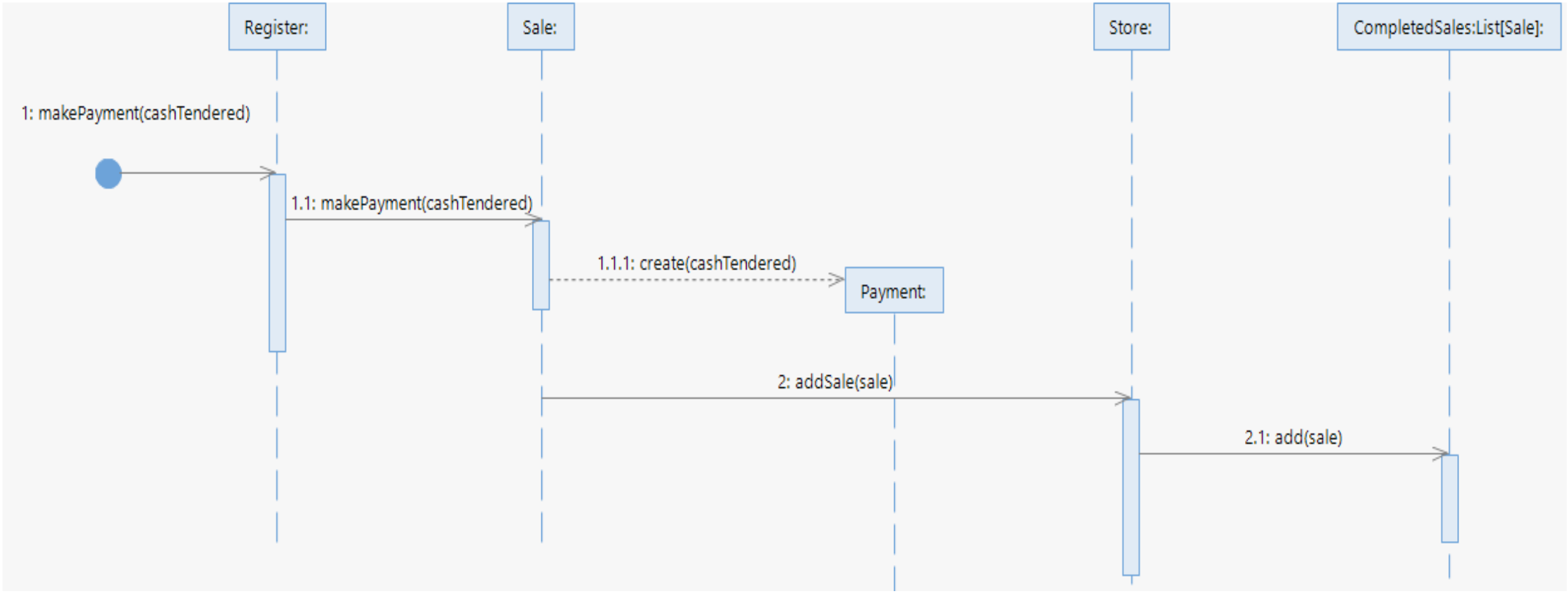



Store
address : Address
name : Text
addCompleteSale(...)
catalog
1
ProductCatalog
...
getProductDesc(...)
descriptions
{Map}
1..*
ProductDescription
description : Text
price : Money
itemID: ItemID
...
1
catalog
register
1
Register
...
endSale()
enterItem(...)
makeNewSale()
makePayment(...)
Sale
isComplete : Boolean
time : DateTime
becomeComplete()
makeLineItem(...)
makePayment(...)
getTotal()
description
1
currentSale
1
lineItems
{ordered}
1..*
SalesLineItem
quantity : Integer
getSubtotal()
completedSales
{ordered}
*
payment
1
Payment
amount : Money
...


Register:
Sale:
LineItem[i]:
ProductDescription:
1: endSale()
1.1: becomeComplete
1.2: total=getTotal()
1.1.1.1: st=getSubTotal()
1.1.1.1.1: pr=getPrice()

makePayment(cashTendered)




Register:
Sale:
Store:
CompletedSales:List[Sale]:
1: makePayment(cashTendered)
1.1: makePayment(cashTendered)
1.1.1: create(cashTendered)
Payment:
2: addSale(sale)
2.1: add(sale)

Class Diagram