




# UML & OBJECT ORIENTED ANALYSIS AND DESIGN

Mehra Borazjany – Summer 2022

[mehra@utdallas.edu](mailto:mehra@utdallas.edu)

- 
- ❑ Part 1: Introduction
  - ❑ Part 2: Analysis
  - ❑ Part 3: Design
  - ❑ Part 4: Example



# OBJECT ORIENTED ANALYSIS AND DESIGN

PART1: Introduction

# SOFTWARE ENGINEERING AND COMPUTER SCIENCE

## Computer Science

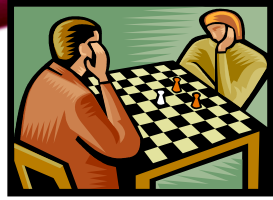
- Pursue optimal solutions
- \$\$\$ is not an important consideration
- Programming in the small
- Technical issues
- Dealing with tame problems
  
- Foundations of software engineering

## Software Engineering

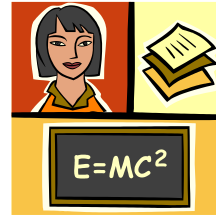
- Good enough is enough
- \$\$\$ is an important factor (PQCT)
- Programming in the large
- All issues and aspects
- Dealing with wicked problems
- Building on top of computer science and other disciplines

Tame vs wicked problems: <http://www.open.ac.uk/cpdtasters/gb052/index.htm>

# EXAMPLES OF TAME PROBLEM



Chess playing



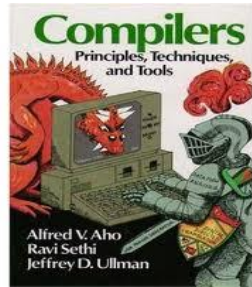
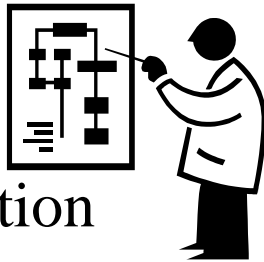
Math problems



Operations research

Many computer science problems

Query optimization



Compiler construction



Operating systems



AI problems

Why are these tame problems?



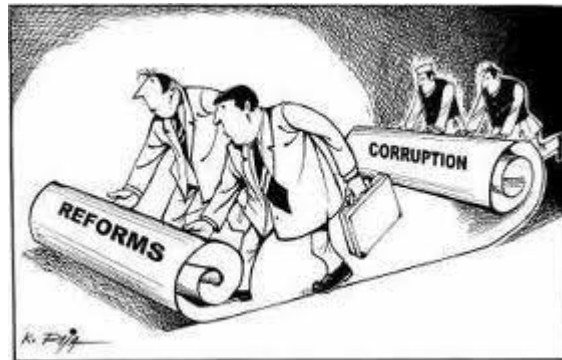
# EXAMPLES OF WICKED PROBLEM



Urban planning



National policy making



Economic reforms

Why are these wicked problems?



Application software development

# CLASS DISCUSSION

- What are the focuses of computer science and software engineering, respectively?
- Some authors say that software engineering is “programming in the large.” What does this mean?
- What is the relationship between software engineering and computer science? Can you have one without the other?

# WHAT IS SOFTWARE ENGINEERING?

*Software engineering* as a **discipline** is focused on

- research, education, and application of engineering processes and methods
- to significantly increase *software productivity (P)* and *software quality (Q)* while reducing *software costs (C)* and *time to market (T)* – software PQCT.



# WHY SOFTWARE ENGINEERING?

To work together, the software engineers must overcome three challenges, among others:



Conceptualization



Communication



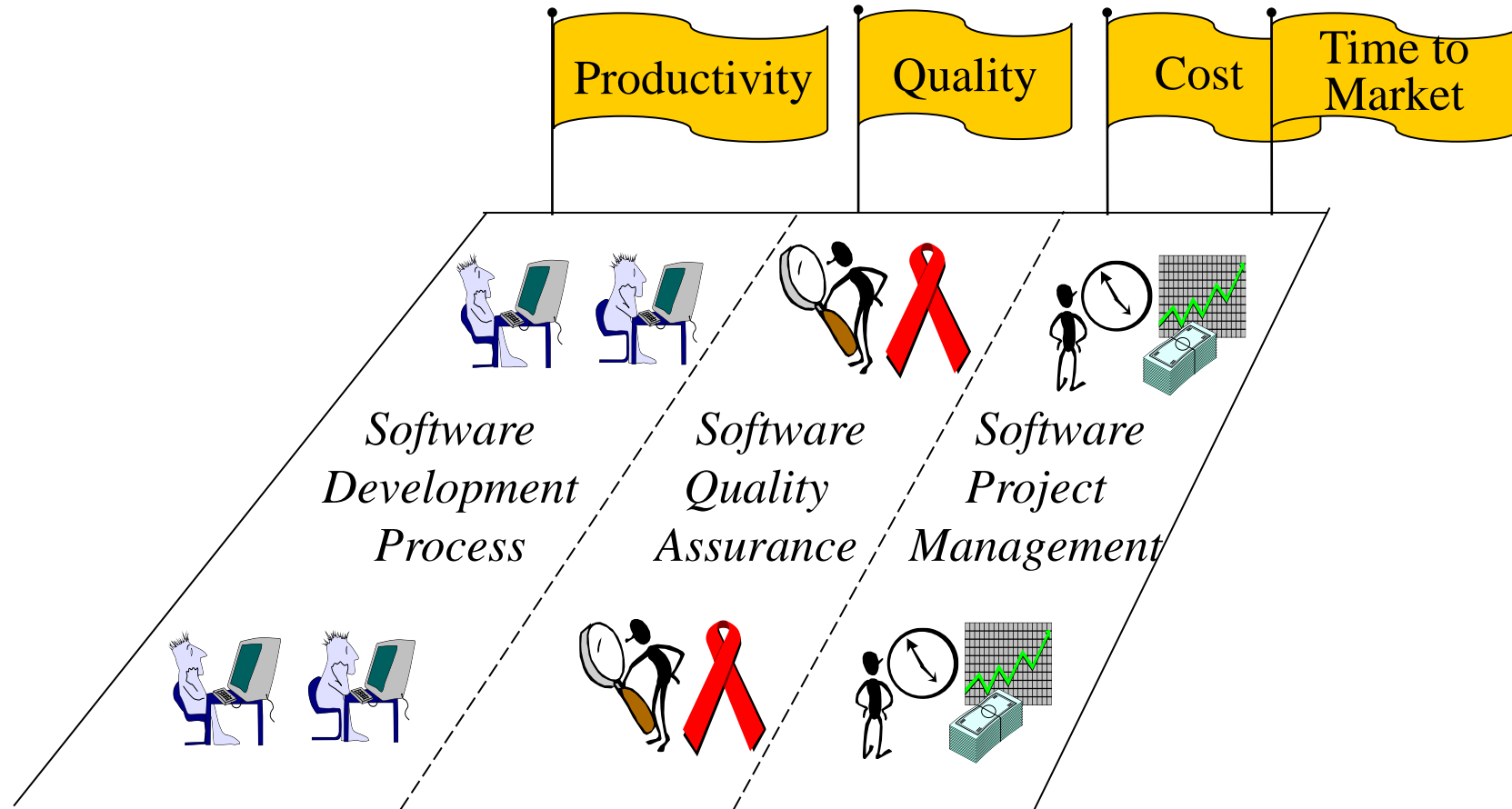
Coordination

Solution:

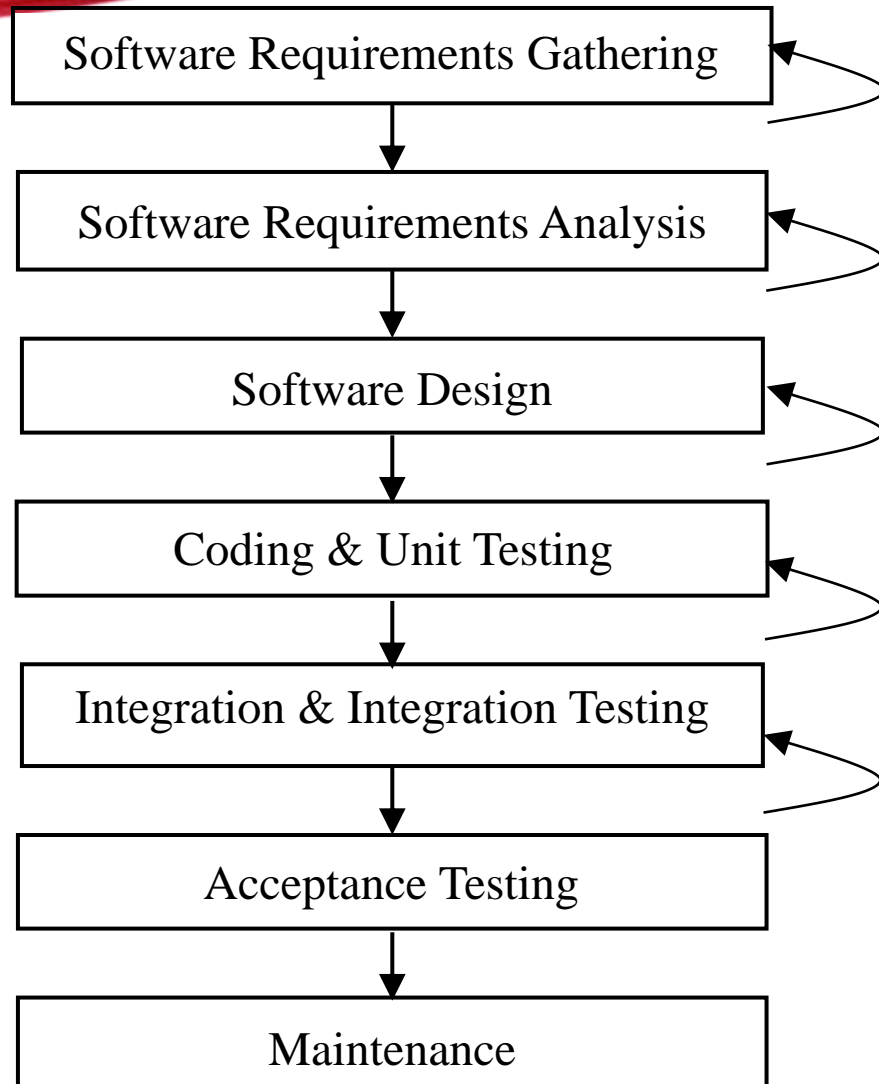
- Processes and methodologies for analysis and design
- **UML** for communication and coordination
- Tools that automate or support methodology steps.

# SOFTWARE LIFE CYCLE ACTIVITIES

Software processes and methodologies consist of life cycle activities:



# THE SOFTWARE PROCESS



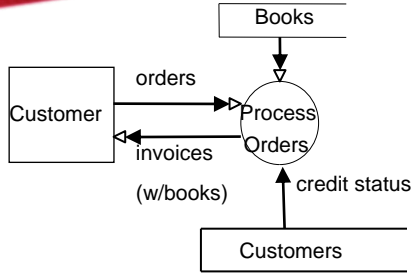
# OBJECT-ORIENTED SOFTWARE ENGINEERING

- Object-oriented software engineering (OOSE) is a specialization of software engineering.
- The object-oriented paradigm views the world and systems as consisting of objects that relate and interact with each other.
- OOSE encompasses:
  - OO processes
  - OO methodologies
  - OO modeling languages
  - OO tools

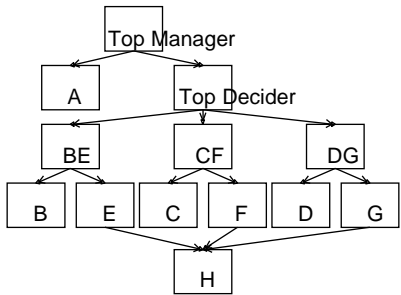
# SOFTWARE PARADIGM

- A software paradigm is a style of software development that constitutes a way of viewing the reality.
- Examples:
  - procedural paradigm
  - **OO paradigm**, and
  - data-oriented paradigm

# PARADIGM AND METHODOLOGY



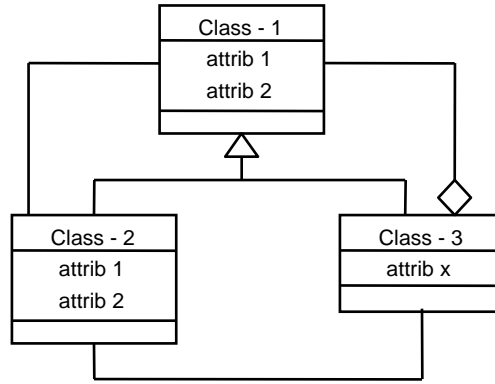
Structured Analysis



Structured Design

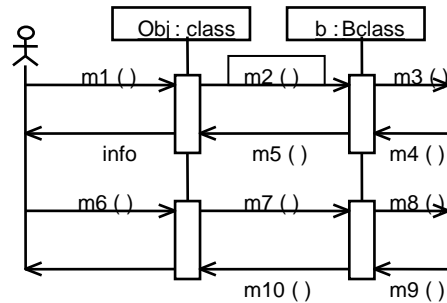
Structured Programming

Procedural Paradigm



Domain model

Object-Oriented Analysis

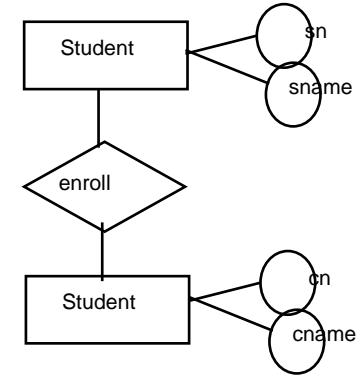


Sequence diagram

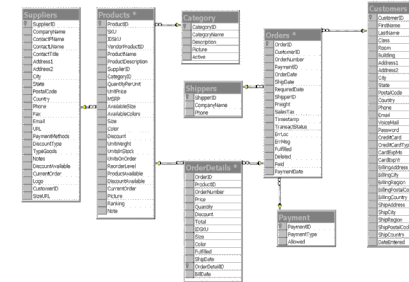
Object-Oriented Design

Object-Oriented Programming

OO Paradigm



Data-Oriented Analysis



Data-Oriented Design

Programming in 4GL (e.g., SQL)

Data-Oriented Paradigm



# CLASS DISCUSSION

- What are the benefits of OOSE?
- Will OOSE replace the conventional approaches, and why?



# OBJECT ORIENTED ANALYSIS AND DESIGN

PART2: Analysis



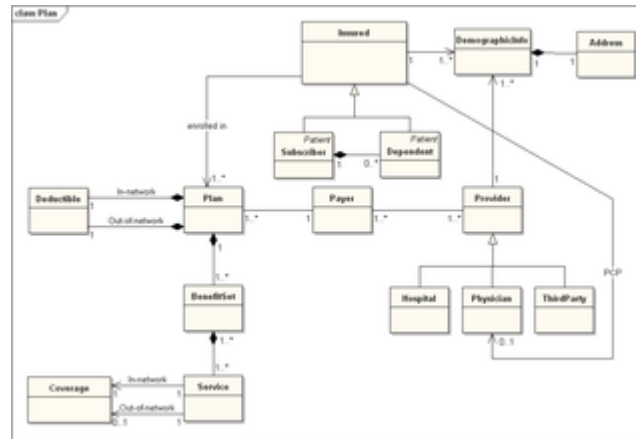
# OBJECTS

- From Merriam-Webster:  
“something material that may be perceived by the senses”
- Look around this room, and imagine having to explain to someone who has never taken a class what happens here ...  
You would explain the activity that occurs, and you would identify specific objects that play a role in that activity (Chairs, tables, projectors, students, professor, white board, etc.) to someone who has never seen these things ...  
Each of these objects is well defined, and plays a separate role in the story. There may be multiple copies of chairs, but a chair is very different from a projector – they have different *responsibilities*  
You would not describe the action by saying “The classroom allows students to sit, and the classroom allows the professor to display slides, ... “ etc. This would make the “classroom” too complex – almost magical  
You would define the various objects in this domain, and use them to tell the story and describe the action

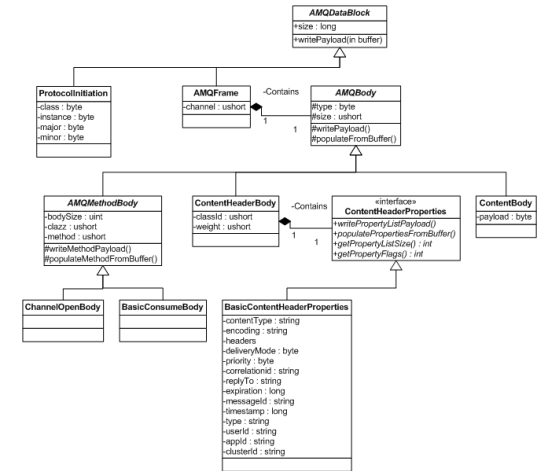
# OOA/OOD



Analyze the system



Model the system



Design the software

# ANALYSIS AND DESIGN:

Analysis is the *investigation* of the problem - **what** are we trying to do?

Here is where use cases are created and requirements analysis are done

Design is a *conceptual solution* that meets the requirements – **how** can we solve the problem

Note: Design is *not* implementation

UML diagrams are not code (although some modeling software does allow code generation)


Object-oriented analysis: Investigate the problem, identify and describe the objects (or concepts) in the problem domain

Also, define the domain!

Object-oriented design: Considering the results of the analysis, define the software classes and how they relate to each other

Not every object in the problem domain corresponds to a class in the design model, and viceversa

Where do we assign responsibilities to the objects? Probably a little in both parts



# DOMAIN MODELS

# DOMAIN MODEL - INTRODUCTION

Very important model in OOA ...

Illustrates the important concepts in the Domain, and will inspire the design of some software objects

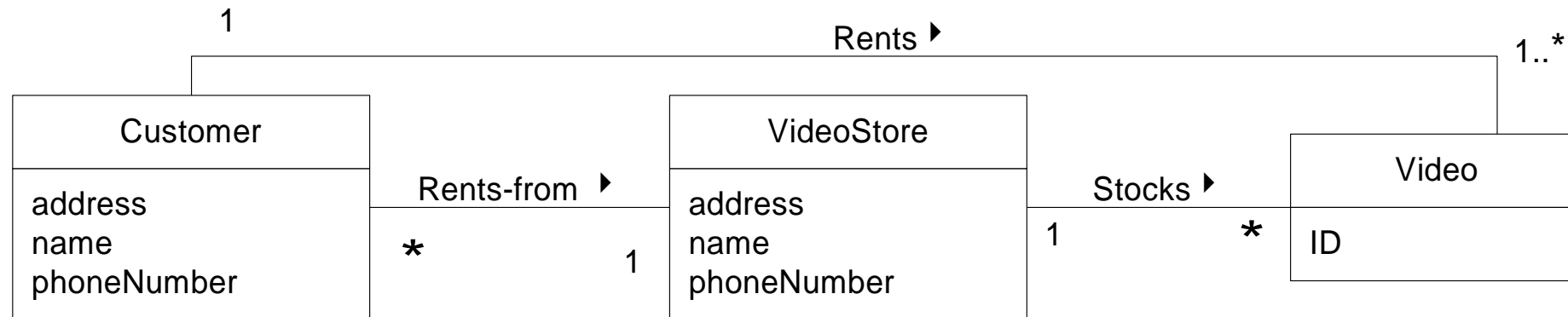
Also provides input to other artifacts

- Glossary
- Design Model (Sequence Diagrams)

- “The Unified Modeling Language is a visual language for specifying, constructing, and documenting the artifacts of systems.” - *OMG, 2003*
- The current version of the Unified Modeling Language™ is UML 2.5, released in June 2015 [[UML 2.5 Specification](https://www.uml-diagrams.org/)]. (<https://www.uml-diagrams.org/>)
  - UML® specification (standard) is updated and managed by the Object Management Group (OMG™) OMG UML.
  - The first versions of UML were created by "Three Amigos".
    - Grady Booch (creator of Booch method)
    - Ivar Jacobson (Object-Oriented Software Engineering, OOSE)
    - Jim Rumbaugh (Object-Modeling Technique, OMT).
- UML is not a technique, it is a combination of several object-oriented notations:
  - + Object-Oriented Design
  - + Object Modeling Technique
  - + Object-Oriented Software Engineering.
  - UML uses the strengths of these three approaches to present a more consistent methodology that's easier to use.
- Standard for diagramming notation.
- We will use UML to sketch out our systems
- UML can be used (by modeling packages) to auto-generate code directly from the model diagrams

- Different perspectives:
  - Conceptual Perspective – defining the problem domain: Raw class diagrams, maybe mention some attributes (Domain Model)
  - Specification Perspective – defining the software classes: Design Class diagram, which shows the actual software classes and their methods, attributes
- We will explore the details of UML diagramming
- For now, understand that UML is a language – it is used to communicate information
- We will use UML to describe the problem domain, describe the activities that occur, and eventually describe the software classes
- Since it is a language, UML has specific rules, and we will see these later in the course
- You need to be able to read UML diagrams, as well as create them
- Here are some examples (we will learn more about how to create these diagrams later ...)

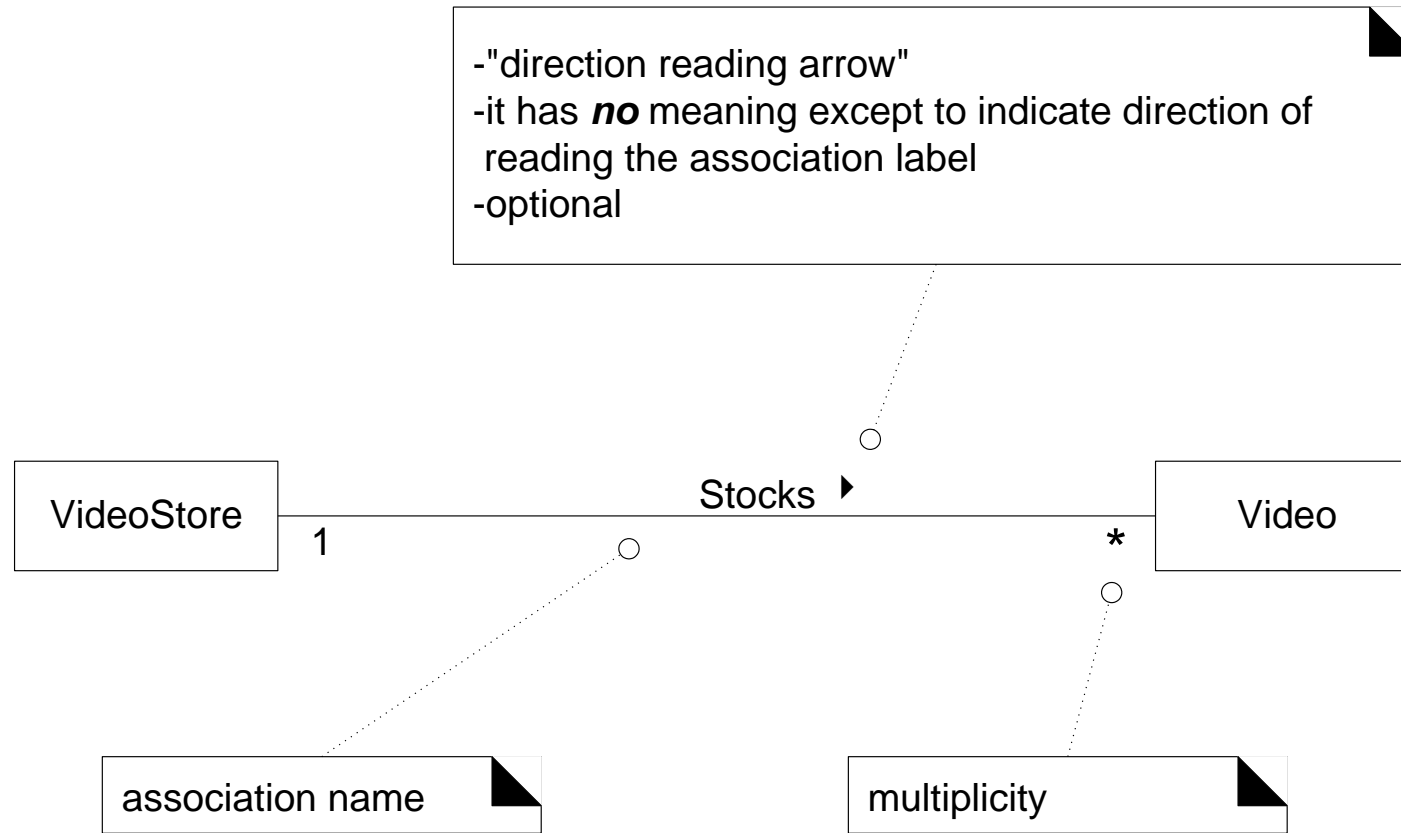
# UML- DOMAIN MODEL



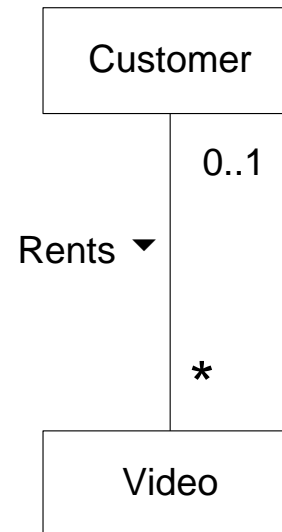
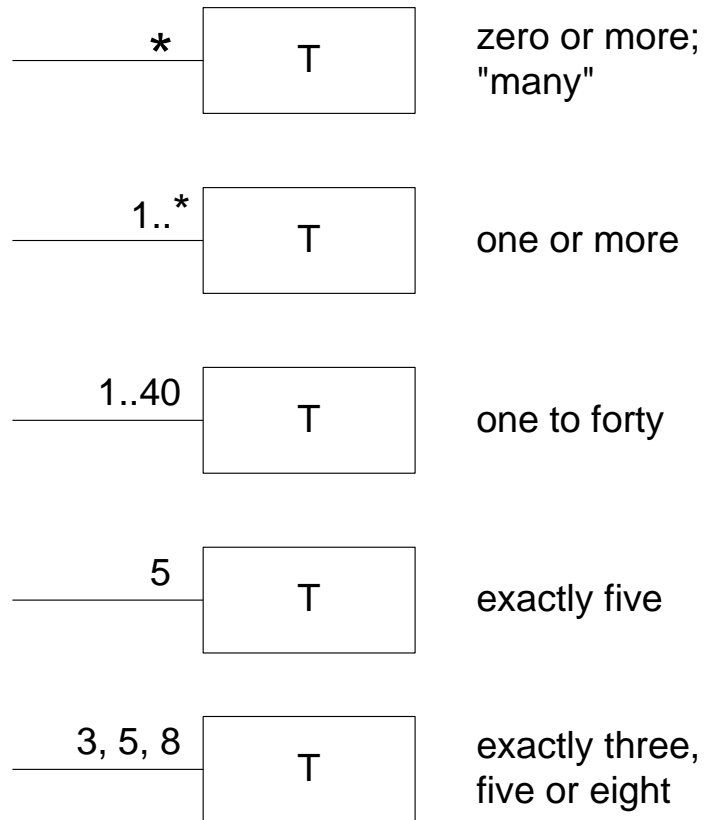
<https://www.uml-diagrams.org/class-diagrams-overview.html#domain-model-diagram>



# UML ASSOCIATION



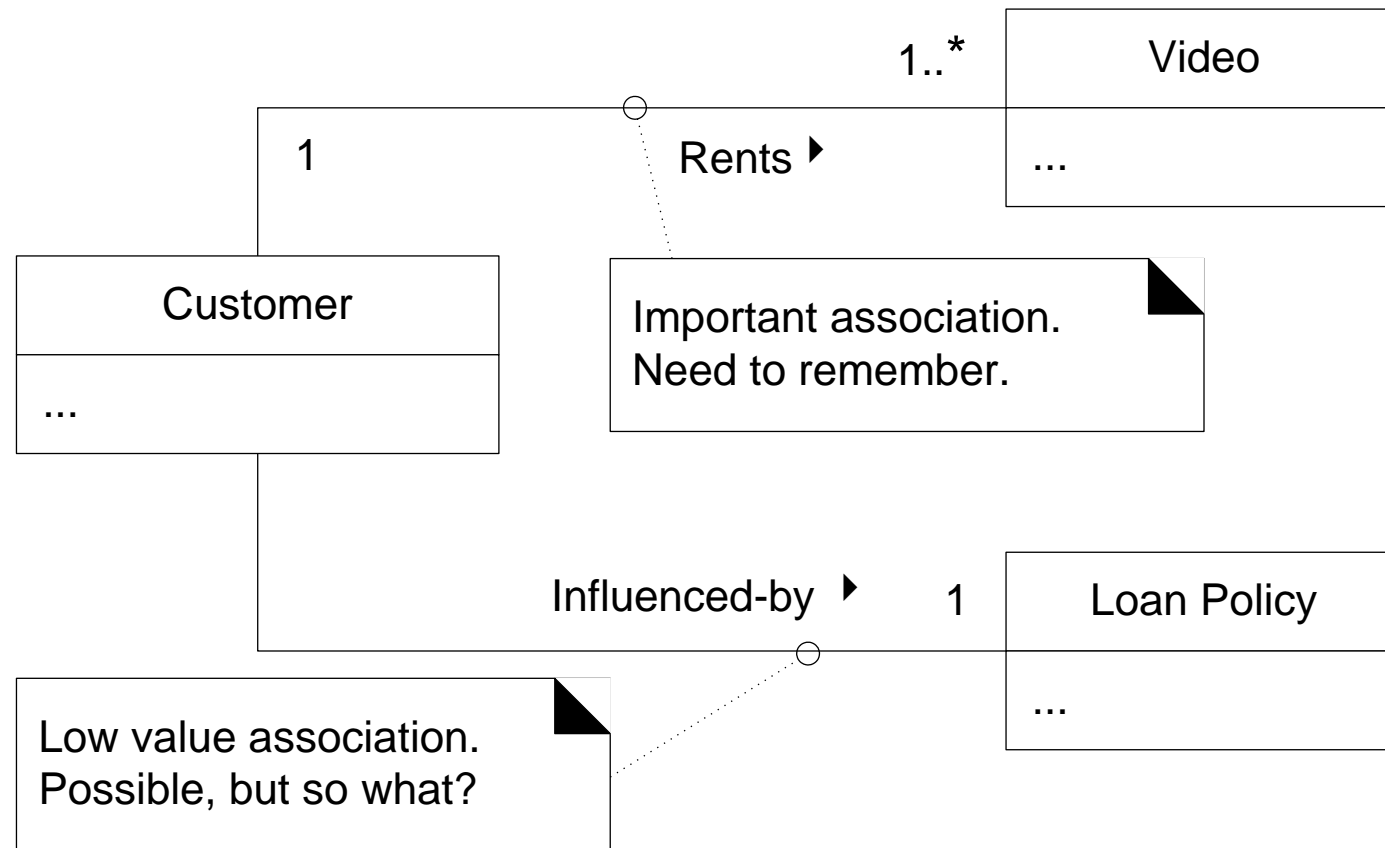
# UML MULTIPLICITY



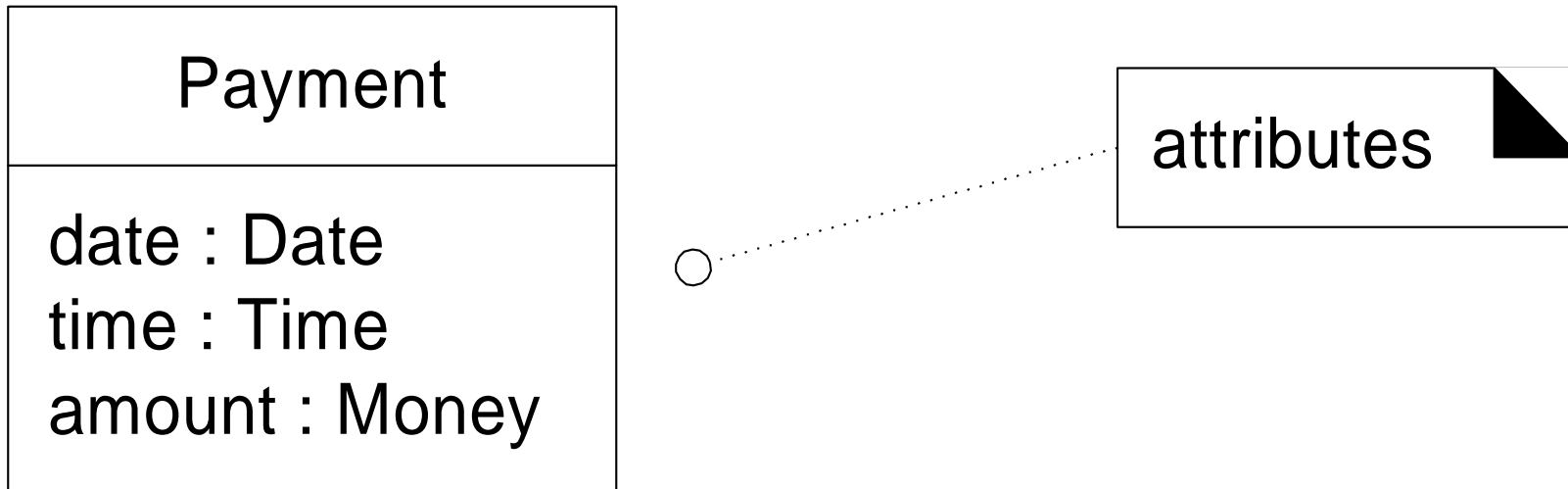
One instance of a Customer may be renting zero or more Videos.

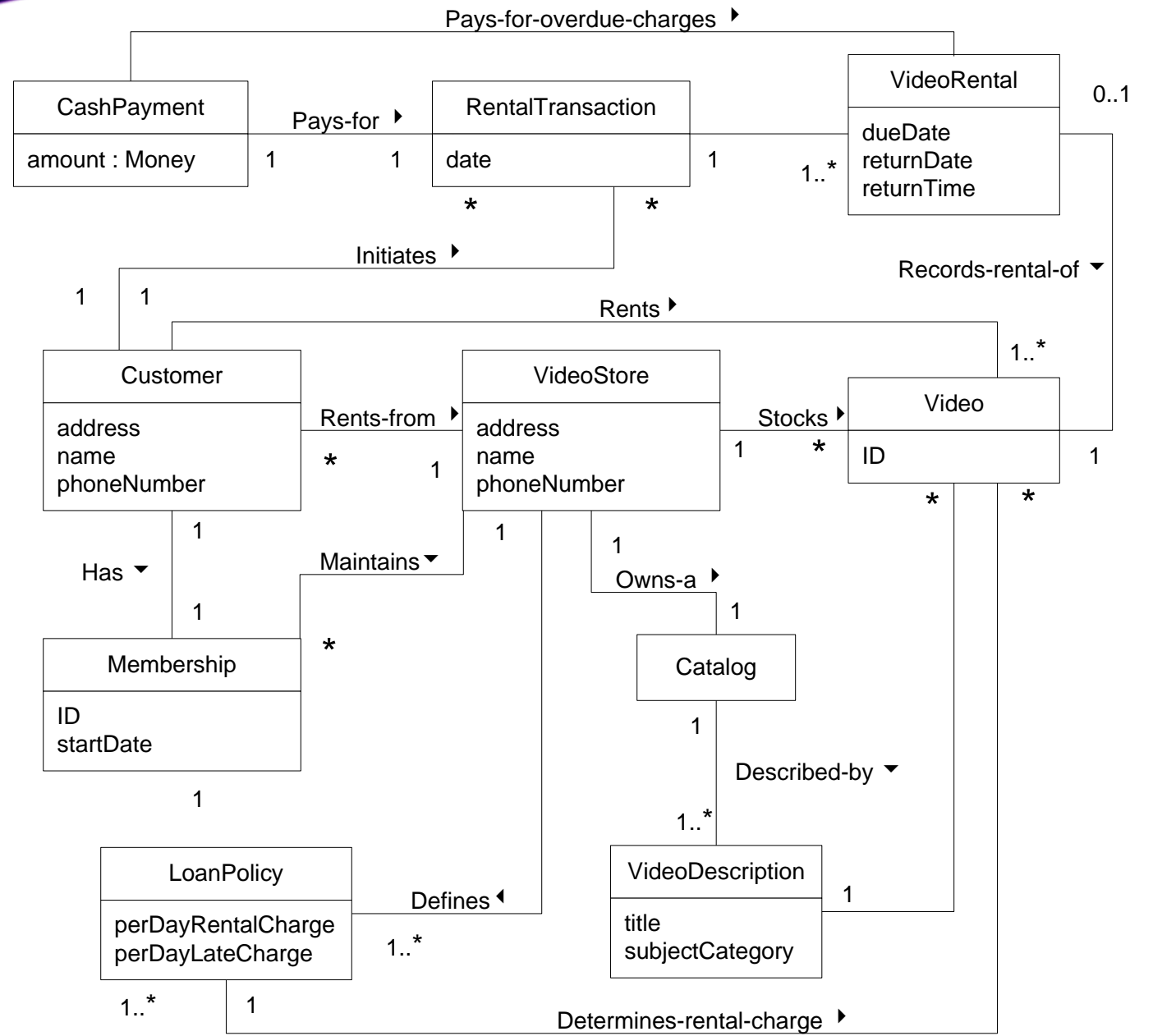
One instance of a Video may be being rented by zero or one Customers.

# UML DOMAIN MODEL



# UML – CLASS AND FEATURES OF A CLASS

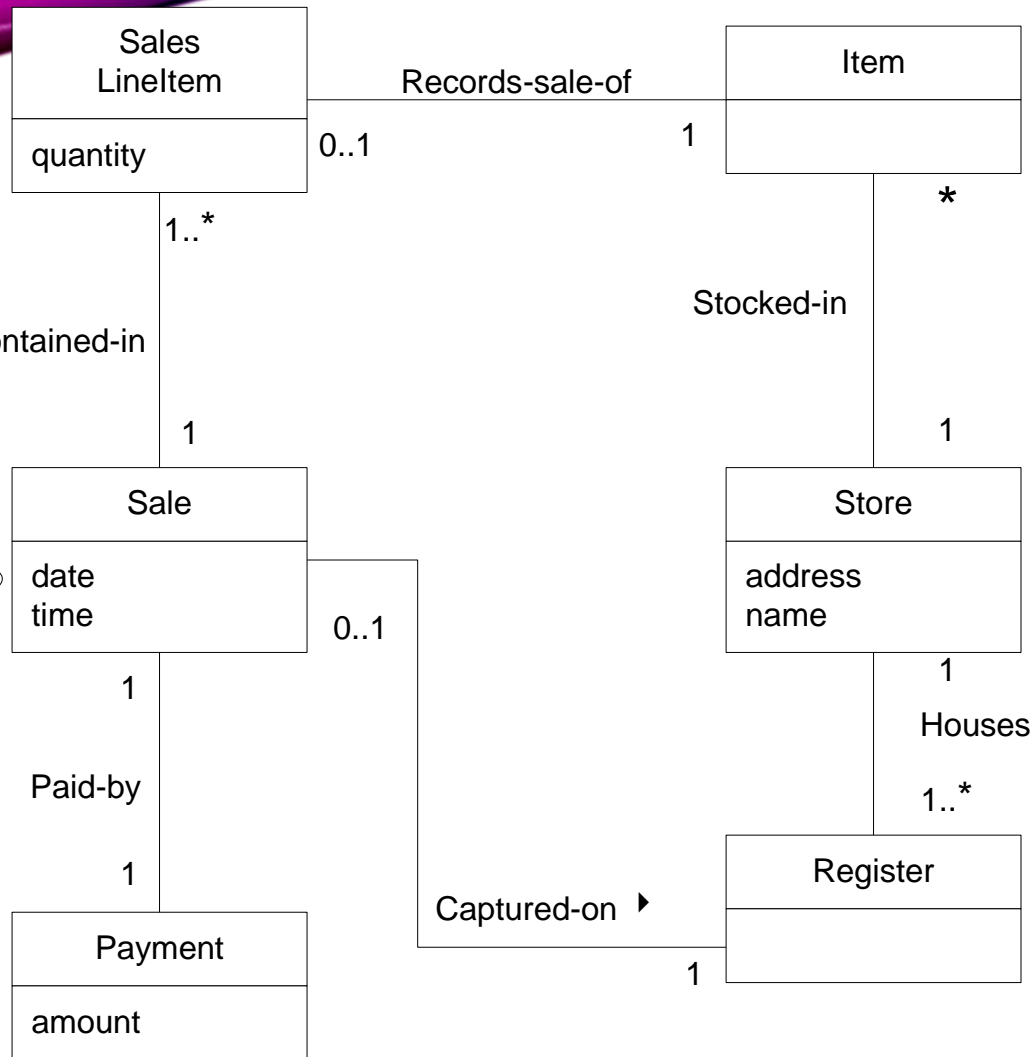




concept  
or domain  
object

association

attributes



This diagram shows an example of an early Domain Model for the Point Of Sale system.

# DOMAIN MODEL: DEFINITION

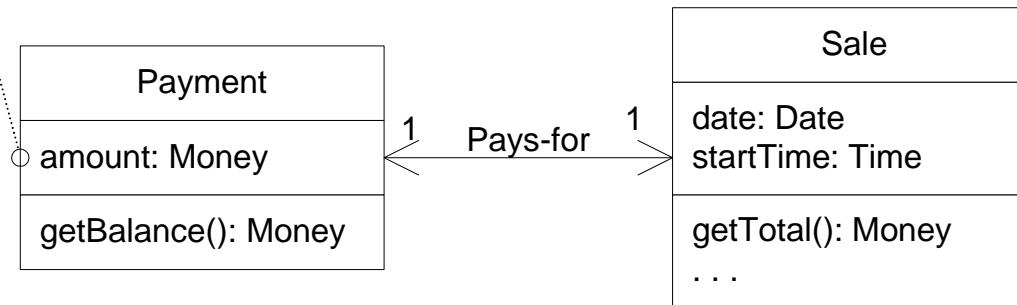
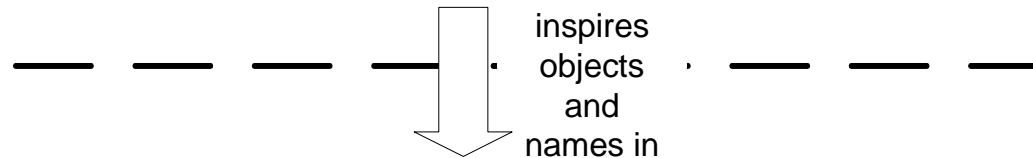
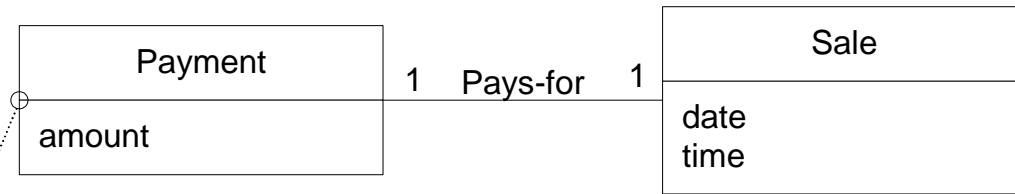
The Domain Model can be thought of as a *visual* representation of conceptual classes or real-situation objects in the domain (i.e. the real world).

*In UP, the term Domain Model means a representation of real-situation conceptual classes, not software objects. The term **does not** mean a set of diagrams describing software classes, the domain layer of the software architecture, or software objects with responsibilities*

Think of as a visual dictionary describing the domain: important abstractions, domain vocabulary, and information content

## UP Domain Model

Stakeholder's view of the noteworthy concepts in the domain.



## UP Design Model

The object-oriented developer has taken inspiration from the real world domain in creating software classes.

Therefore, the representational gap between how stakeholders conceive the domain, and its representation in software, has been lowered.

A Payment in the Domain Model is a concept, but a Payment in the Design Model is a software class. They are not the same thing, but the former *inspired* the naming and definition of the latter.

This reduces the representational gap.

This is one of the big ideas in object technology.

The difference between domain model and design model – UML used in two different ways.



# CREATING DOMAIN MODELS

This is dependent upon which iteration cycle you are in, but in general there are three steps:

1. Find the conceptual classes
2. Draw the classes as UML diagrams (conceptual level)
3. Add associations and attributes

## Finding Conceptual Classes

Use or modify existing models – we will see some of these later

Use a category list

Identify noun phrases in the use cases

# CATEGORY LISTS

- This is a list of common conceptual class categories, generalized to apply to many situations
- Can be used as a starting point; look for these conceptual classes in your domain
  - Book has good list ...
  - Business transactions, transaction line items, where is the transaction recorded, physical objects, catalogs, other collaborating systems, ..
- You can make a list of categories (or use a pre-existing list), and after reviewing use cases and requirements, list all conceptual classes you find that relate to a particular category

# NOUN PHRASE IDENTIFICATION

- Look at a textual description of the domain, and identify all the nouns and noun phrases
  - Try not to do this mechanically – not all nouns are conceptual classes!
- Good place to start is the fully dressed use case
  - Go through the main success scenario, identify all important nouns, use these to name conceptual classes

# EXAMPLE: POS USE CASE

## **Main Success Scenario (cash only):**

1. Customer arrives at POS checkout with goods and/or services to purchases
  2. Cashier starts new sale and enters item identifier
  3. System records sale line item and presents item description, price, and running total
- (repeat 2-3 until no more items)

# EXAMPLE: POS USE CASE (IDENTIFY KEY NOUNS)

## **Main Success Scenario (cash only):**

- 1. Customer** arrives at **POS checkout** with **goods** and/or **services** to purchases
  - 2. Cashier** starts new **sale**
  - 3. Cashier** enters **item identifier**
  - 4. System** records **sale line item** and presents **item description**, **price**, and running **total**
- (repeat 2-3 until no more items)

# EXAMPLE – INITIAL DRAFT OF DOMAIN MODEL FOR POS



# OBSERVATIONS

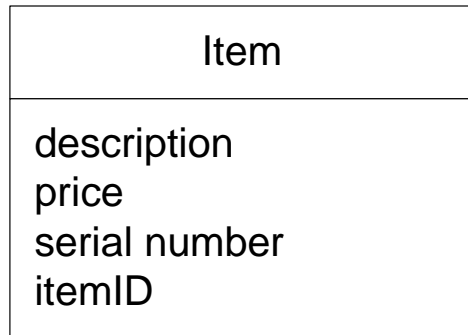
- This model will evolve as the project goes through iterations
- But aside from that, why save this model? Once it has served its purpose, it can be discarded
  - Once the more detailed class diagrams are created, there may not be a need for this model
- It can be maintained in a UML CASE tool (there are many available) such as starUML, draw.io,...

# ATTRIBUTES AND CONCEPTUAL CLASSES

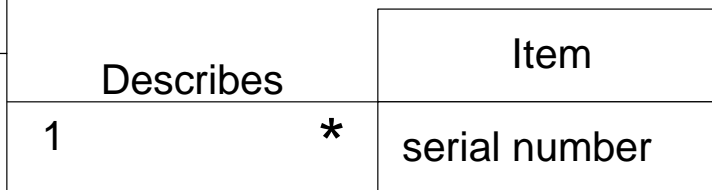
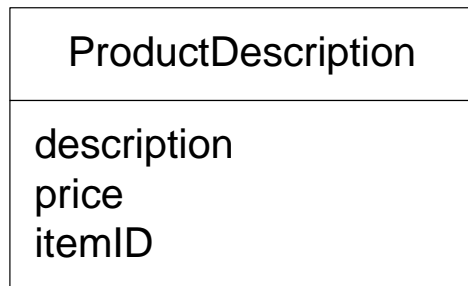
- Be careful not to turn conceptual classes into attributes
  - If X cannot be thought of as a number or text, it is probably a conceptual class
- For example, in the POS case study, the *Store* is not a number or some text, so it should be modeled as a conceptual class (and not an attribute of *Sale*, for example)



# DESCRIPTOR CLASS – STORE ITEM

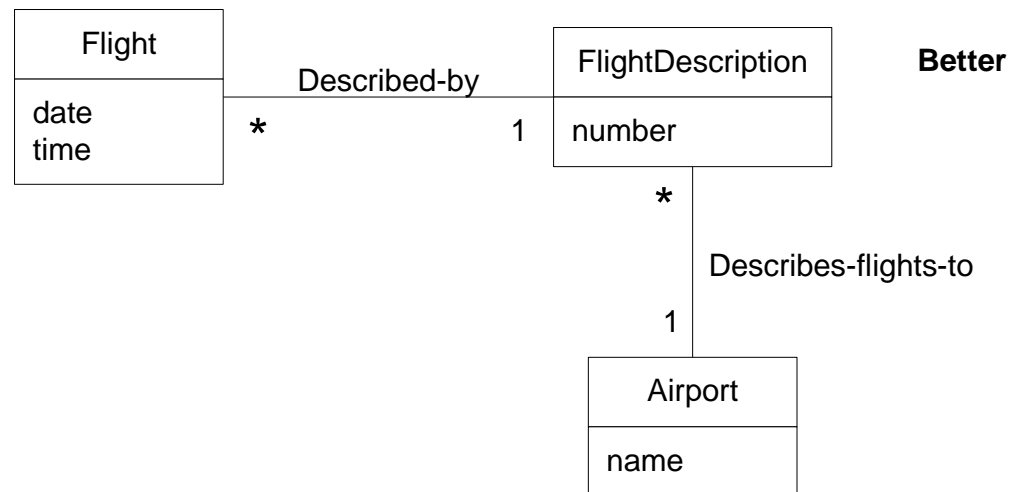
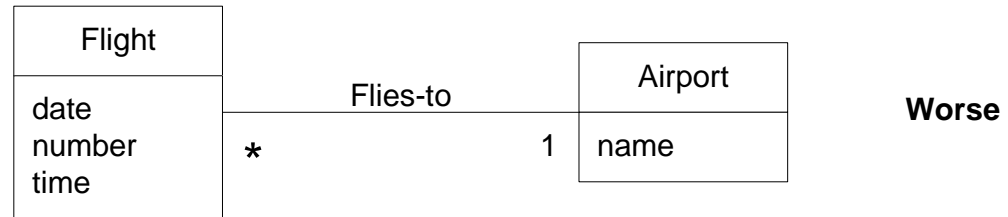


**Worse**



**Better**

# DESCRIPTOR CLASS – AIRLINE FLIGHT



# ASSOCIATIONS

- An association is a relationship between classes that indicates a meaningful and interesting connection.
- When to add an association between conceptual classes to the domain model?

Ask “do we require some *memory* of the relationship between these classes?”

The knowledge of the relationship needs to be preserved for some duration

For example, we need to know that a *SalesLineItem* is associated with a *Sale*, because otherwise we would not be able to do much with the *Sale* (like compute the total amount, print receipt, etc.)

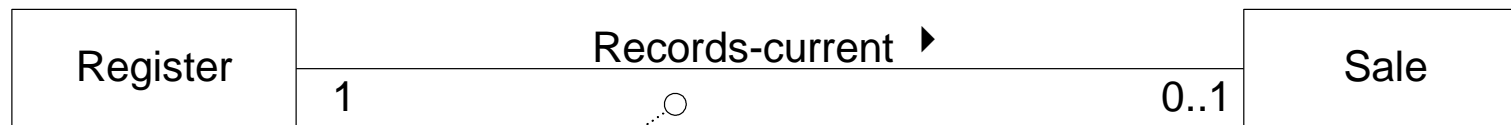
For the Monopoly example, the *Square* would not need to know the value of the *Dice* roll that landed a piece on that square – these classes are probably not associated

# ASSOCIATIONS

- Avoid adding too many associations
  - A graph with  $n$  nodes can have  $(n \times (n - 1)/2)$  associations, so 20 classes can generate 190 associations!
- Realize that there may not be a direct association between software classes in the class definition model just because there is an association between conceptual classes in the domain model
  - Associations in the domain model show that the relationship is meaningful in a conceptual way
  - But many of these relationships do become paths of navigation in the software
- Naming: Use *ClassName – VerbPhrase – ClassName* format
  - Can add a small arrow to help explain the diagram to the reader

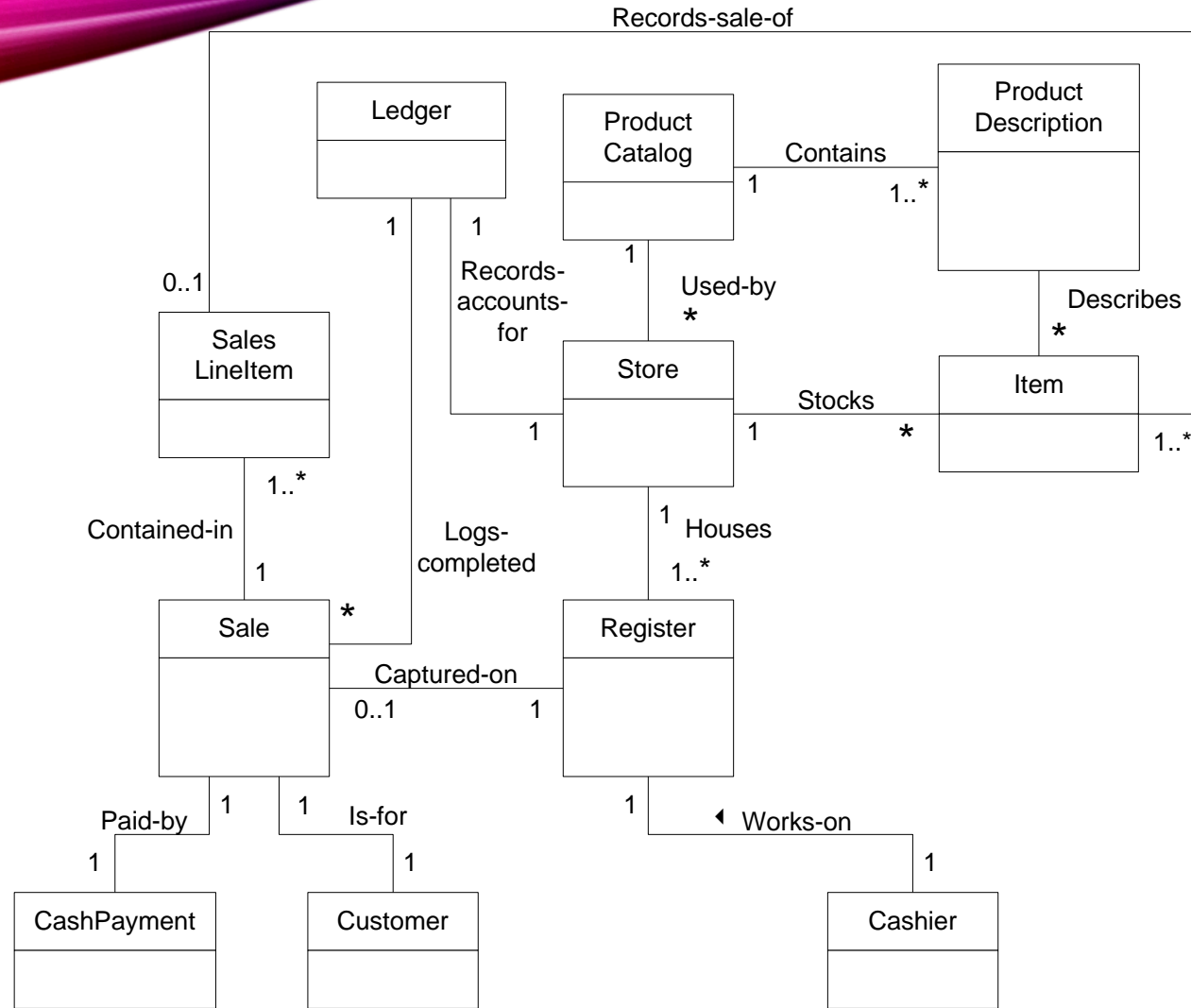
# ASSOCIATIONS

- "reading direction arrow"
- it has *no* meaning except to indicate direction of reading the association label
- often excluded



association name

multiplicity



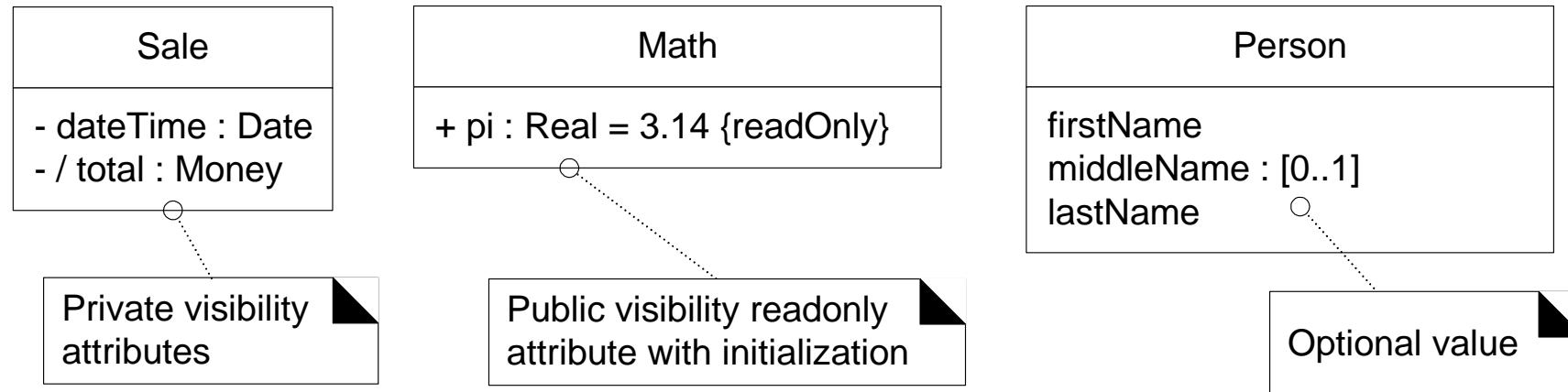
NextGen POS – Domain Model with associations

# DOMAIN MODELS: ADDING ATTRIBUTES

Useful to add attributes to conceptual classes to satisfy an information requirement in a scenario. Note that in this case the *attribute* is a logical data value of an object

Attributes are added to the bottom of the conceptual class box

Notation: **visibility name : type multiplicity = default value {property-string}**



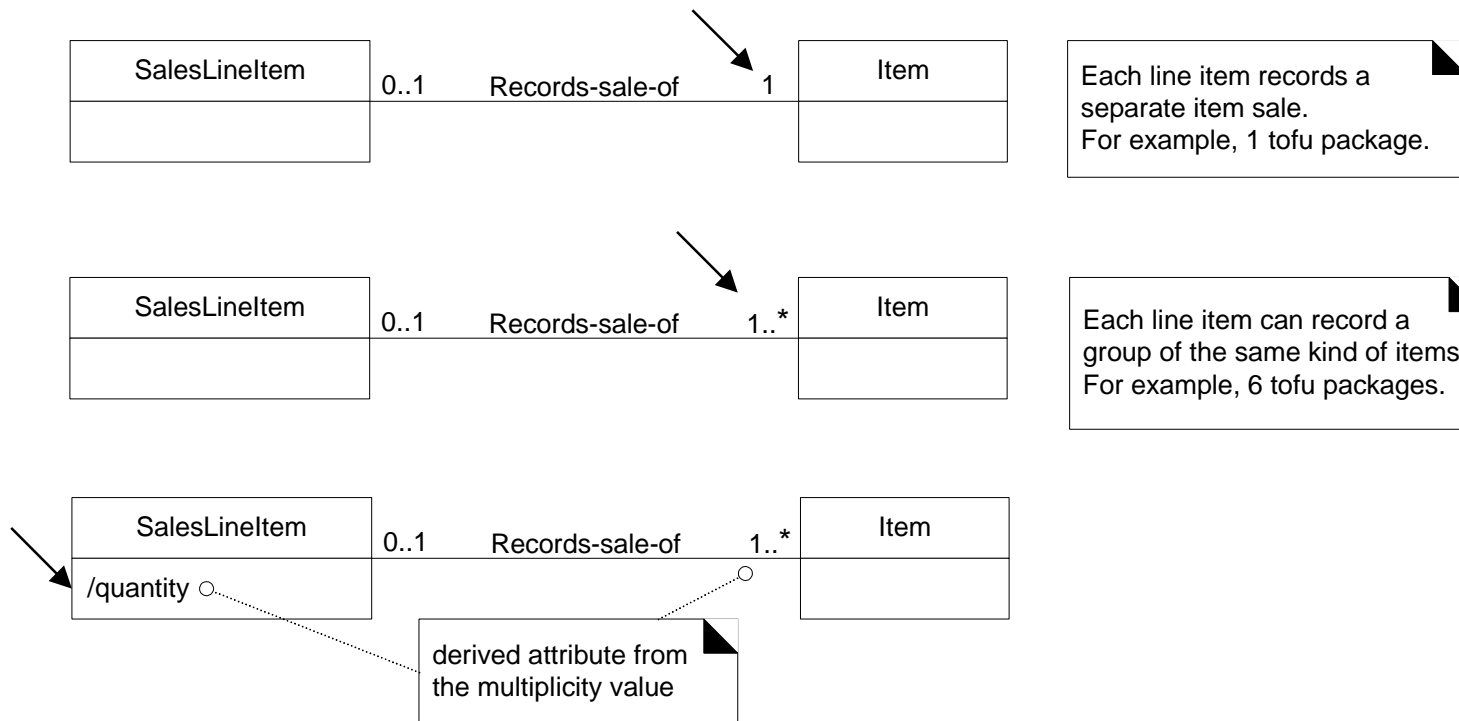
# DOMAIN MODELS: ADDING ATTRIBUTES

- Usually assume that the attribute is private, unless otherwise noted
- Be careful about placing attribute requirements in the Domain Model
  - The Domain Model is generally used as a tool to understand the system under development, and often any requirements that are captured there may be overlooked
  - Best to capture attribute requirements in a Glossary
  - Can also use UML tools that can integrate a data dictionary
- Note in the previous example we use the symbol “/” to indicate that an attribute is *derived*, i.e. computed.



# DERIVED ATTRIBUTE: EXAMPLE

In this case, multiple instances of an item can be added to a SalesLineItem one at a time or as a group. The *quantity* attribute can be computed directly from the multiplicity of the Items:



# ATTRIBUTES VERSUS CLASSES

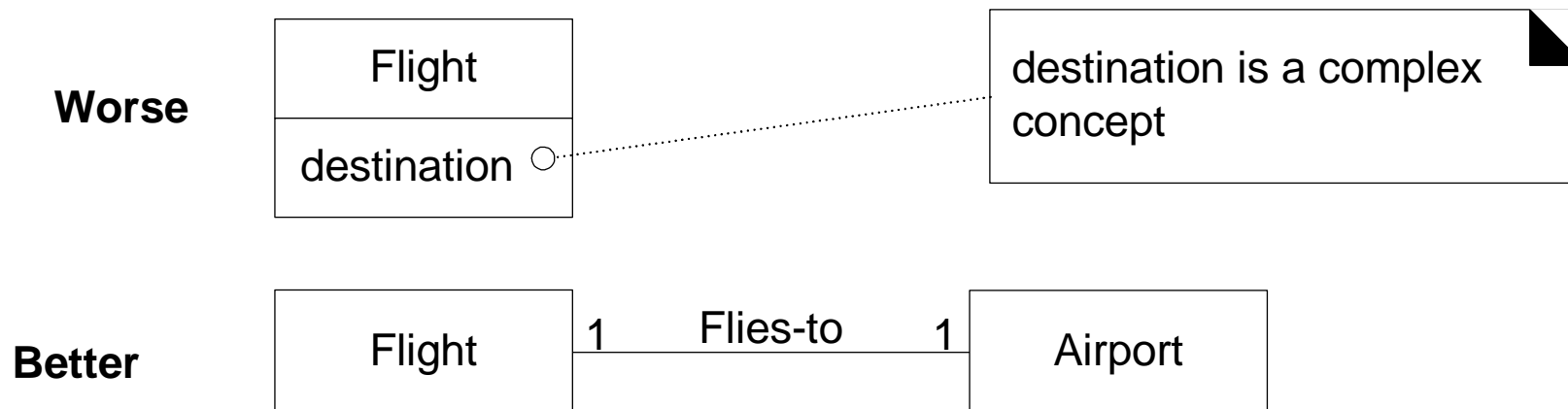
Often attributes are primitive data types

Boolean, Date, Number, Char, String, Time, ...

Do not make a complex domain concept an attribute – this should be a separate class.

Data types are things which can be compared by value; conceptual classes are usually compared by identity

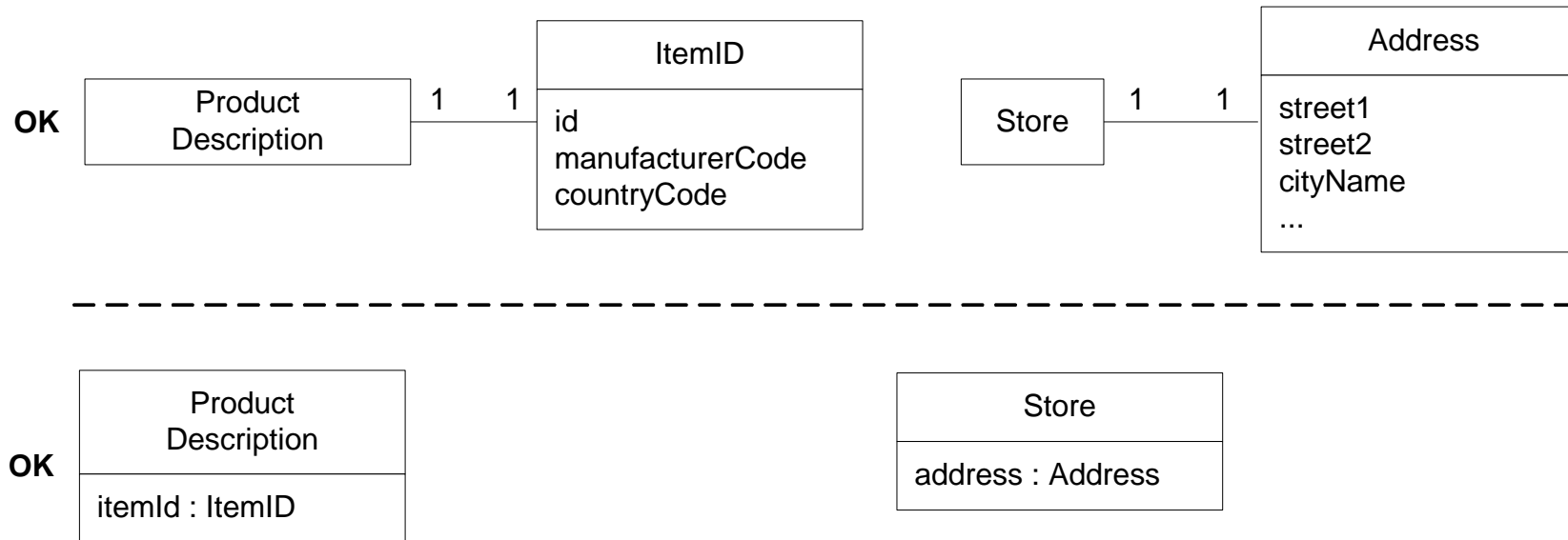
Person class versus name string



# DATA TYPE CLASSES

- It is also possible to have more complex data types as attributes in the Domain Model, and these are often modeled as classes
- For example, in the NextGen POS example, we may have an *itemID* for each item; it is probably contained in the *Item* or *ProductDescription* classes. It could be a number or a string, but it may have more parts too
- In general, your analysis of the system will tell if the attributes are simple or need more complexity
  - For example, upon examining the detail of the *itemID*, we may discover that it is made up of multiple parts, including a unique UPC, a manufacturer ID, a country code, etc. This would be a good candidate for a separate class.

# DATA TYPE CLASS: EXAMPLE



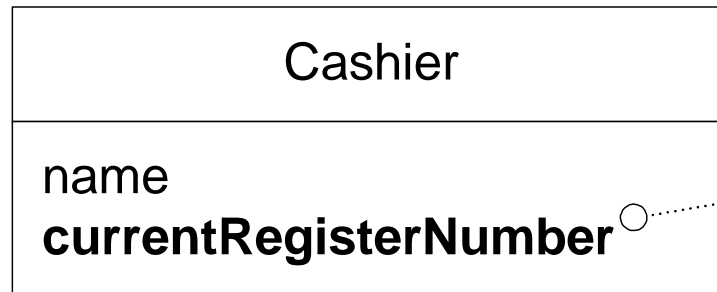
In the bottom example, *itemID* and *address* would need to be described in the Glossary or someplace else in the Domain Model

# GUIDELINES FOR CREATING DATA TYPE CLASS

- If the data type is composed of multiple sections, like *name*, *phone number*, etc.
- There are operations associated with the data type, like parsing
- There are other attributes that are associated with it
  - A *promotionalPrice* may have a *startDate* and an *endDate*.
- It represents a quantity with a unit, e.g. currency
- It is an abstraction with one or more types of the above
- Do not use a “foreign key” to associate two classes – use UML associations, not attributes
  - A simple attribute that is used to relate two classes – see next slide for an example

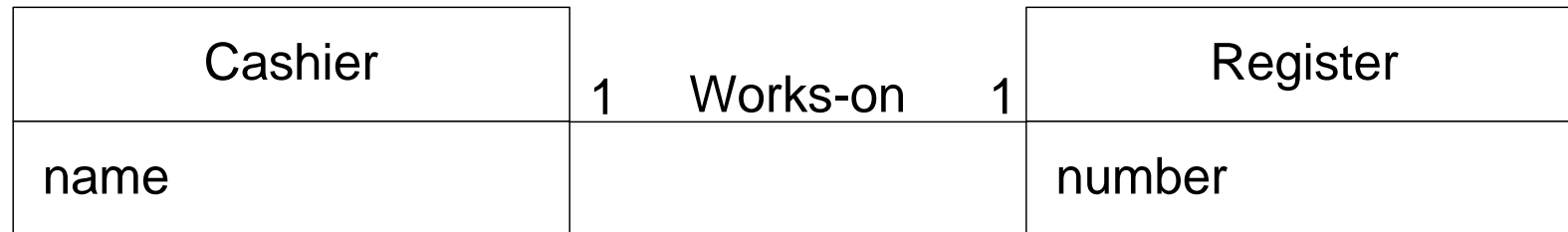
# NO FOREIGN KEYS

**Worse**



a "simple" attribute, but being used as a foreign key to relate to another object

**Better**

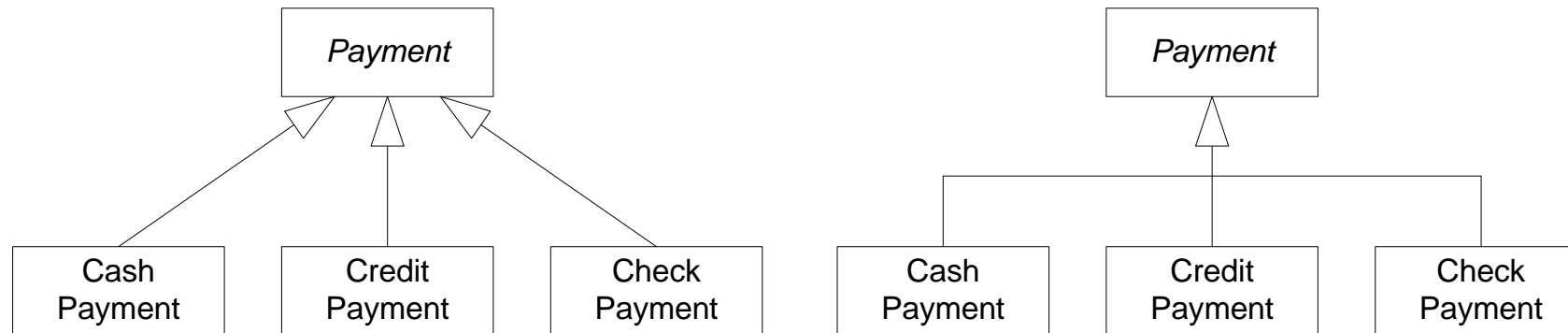


**Note: Classes do not represent tables in a relational database**

# DOMAIN MODEL REFINEMENT: GENERALIZATION

Even though these are not software classes, this type of modeling can lead to better software design later (inheritance, etc.)

UML notation uses an open arrow to denote subclasses of a conceptual class



# REFINING DOMAIN MODELS: SUPER AND SUB CLASSES

Can think of the “sub-class as being *a kind of* super-class”

A Credit Payment is a kind of Payment

Often shortened to: A Credit Payment *is a* Payment

We can identify sub-classes by these two rules

The 100% - all the super-class definition applies to the sub-class

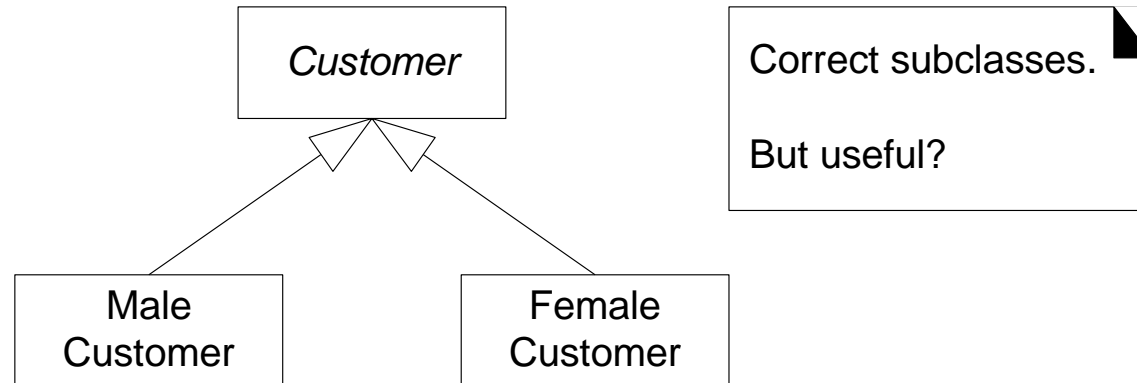
The “is a” rule – the sub-class *is a* kind of super-class

Any sub-class of a super-class must obey these rules



# SUPER- AND SUB-CLASSES: WHEN TO CREATE

Bad example:

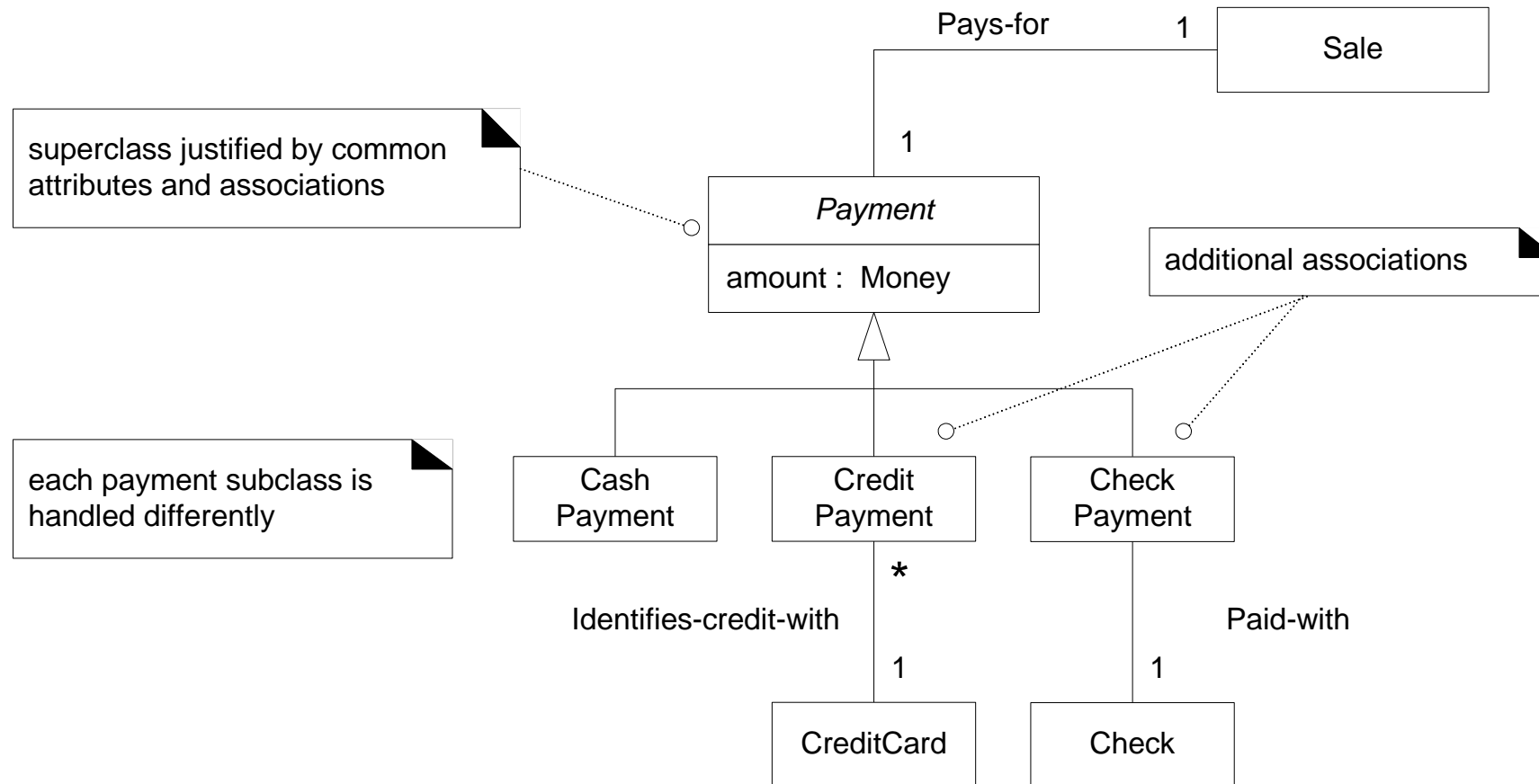


When would this make sense?

Market research model, where there are behaviors of male and female shoppers that are different

Medical research, since men and women are different biologically

# EXAMPLE: PAYMENT SUB-CLASSES



# ASSOCIATION CLASSES

Often, the association between conceptual classes contains information that needs to be captured in the model, but does not belong in either class as an attribute

A *salary* may be an attribute of Employment, but it does not belong as an attribute of the Person or Company classes

General rule: If class *C* can simultaneously have many values of attribute *A*, then *A* should not be placed in *C*.

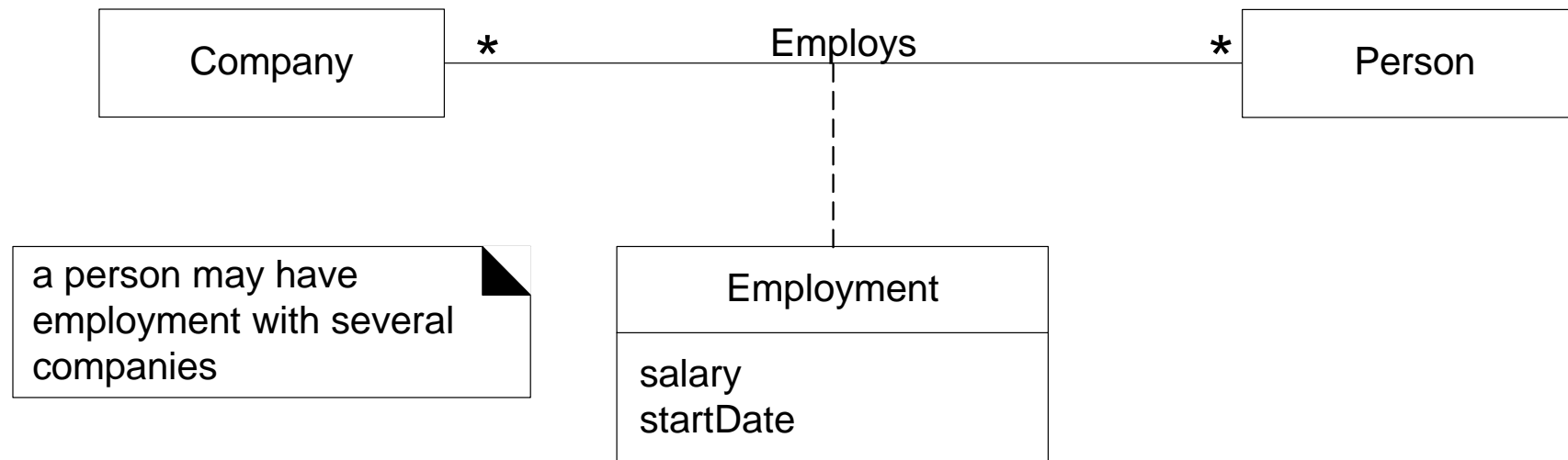
Could create a new conceptual class and associate it with the existing classes, but this can add complexity to the model

Better way: Create a special class that represents the attributes of the association

# ASSOCIATION CLASSES

In UML, an association may be considered a class, with attributes, operations, and other features

Include this when the association itself has attributes associated with it



```
class Company {
    Set<Employment> employments;
}
class Employment{
    Company company ;
    Person person;
    Date startDate;
    Money Salary;
}
class Person{
    Set<Employment> employments;
}
```

providing a method in Company that returns all its personnels

```
public Set<Person> getPersonnels () {
    Set<Person> result = new HashSet<Person>();
    for (Employment e: employments) {
        result.add(e.getPerson());
    }
    return result;
}
```

# AGGREGATION AND COMPOSITION

These are software class concepts that will be important later

Aggregation implies a container or collection of classes

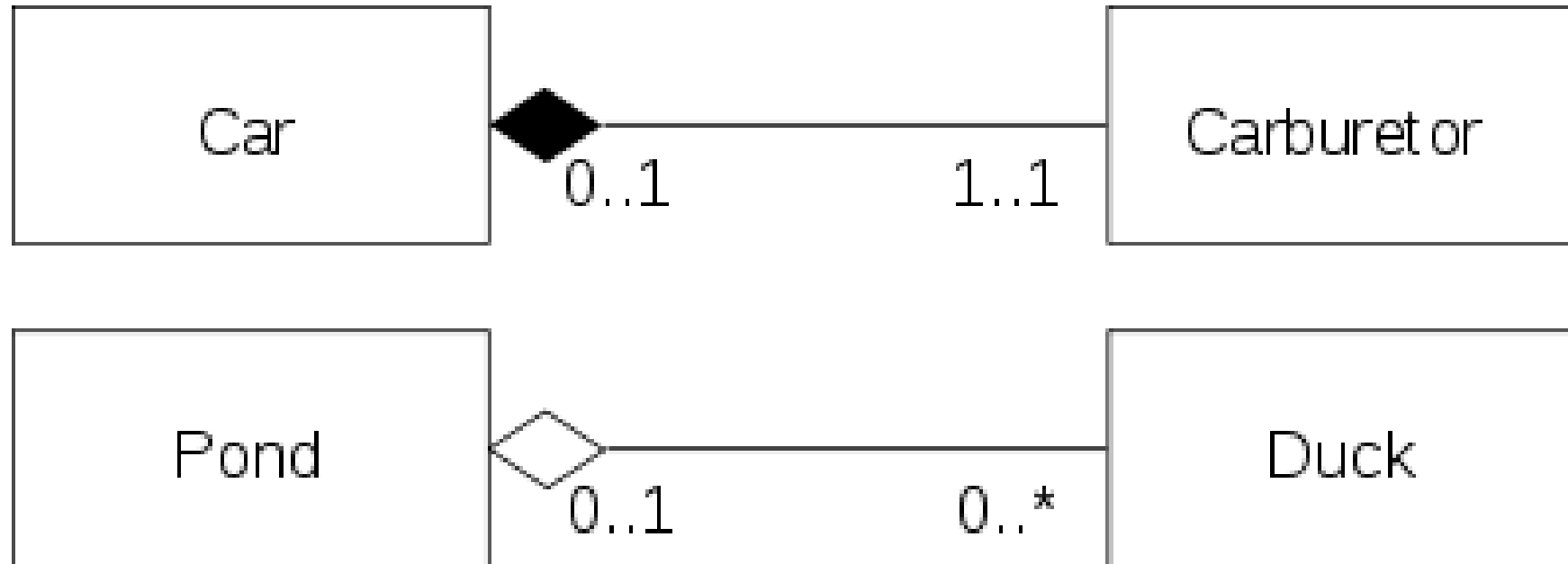
In this case, if the container class is destroyed, the individual parts are not  
Denoted in UML as an open diamond

Composition also implies a collection of classes, but with a stronger life dependency

If the container class is destroyed, the individual component instances are also destroyed

Denoted by a filled in diamond in UML

# EXAMPLES: AGGREGATION AND COMPOSITION



# AGGREGATION AND COMPOSITION

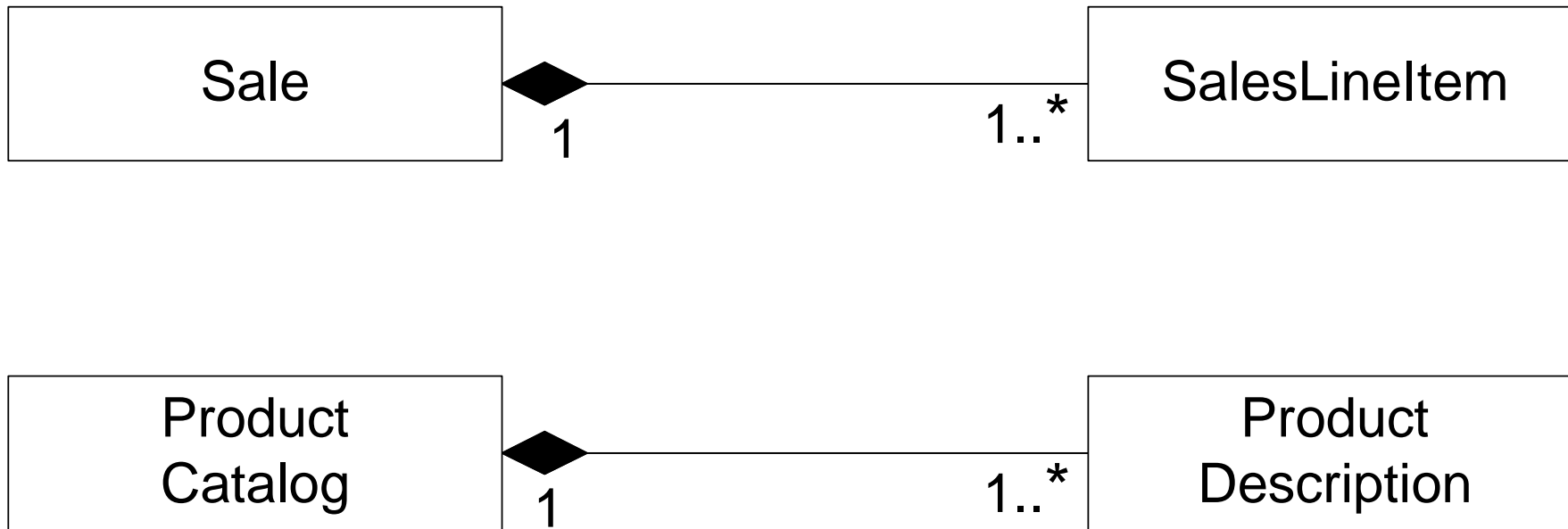
Usually not critical for domain models, but may be used to ...

Clarify constraints in the Domain Model (e.g. existence of a class depends on another class)

Help model situations when create/delete operations apply to many sub-parts



# EXAMPLES: COMPOSITION IN NEXTGEN POS



# ASSOCIATION ROLE NAMES

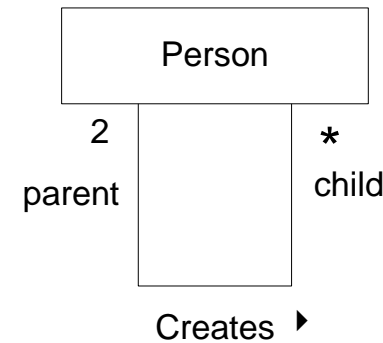
Occasionally a *role name* is added to an association; this name describes the role the object plays in the association

Not required, often included if there role is not clear

Should model the role as a separate class if there are unique attributes, associations, etc. related to the role



role name  
describes the role of a city in the Flies-to association



"Reflexive Association"



# OBJECT ORIENTED ANALYSIS AND DESIGN

PART3: DESIGN



# UML DESIGN INTERACTION DIAGRAMS

DESIGN- DYNAMIC VIEW



# WHAT WILL WE LEARN?

UML Interaction Diagrams – What are they, how to create them

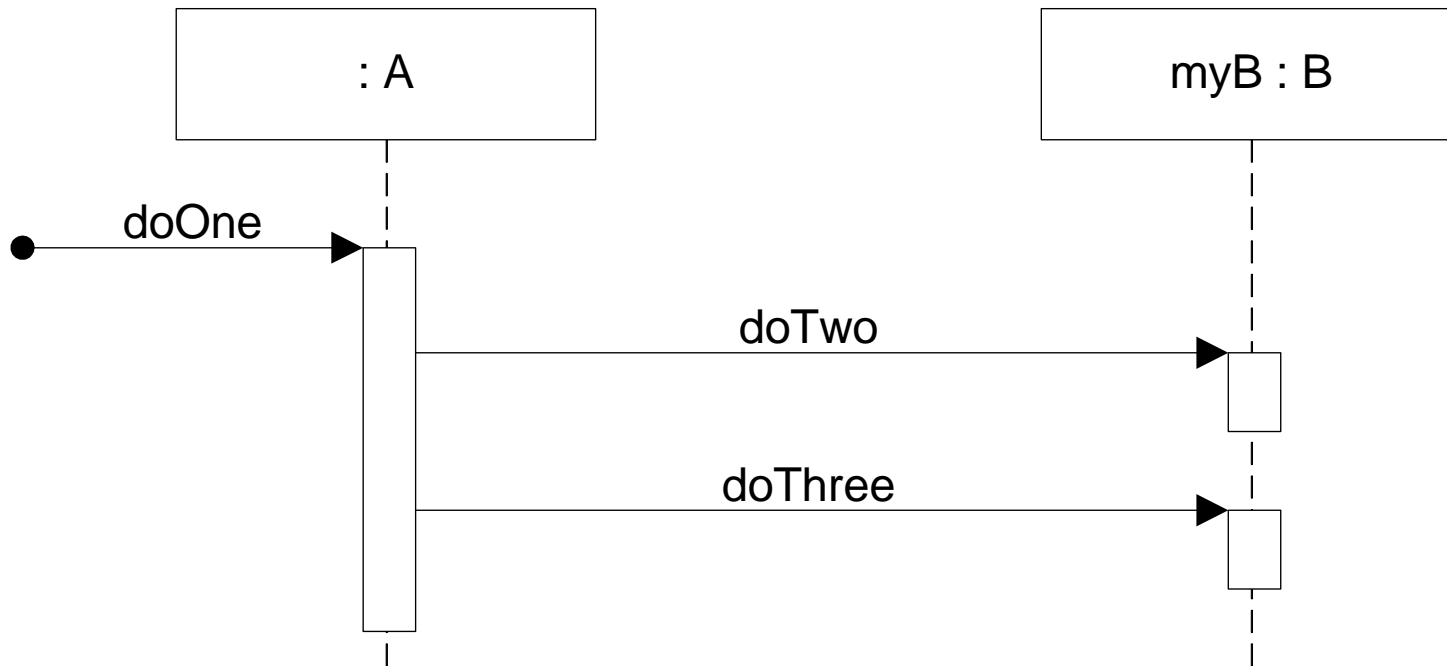
# UML INTERACTION DIAGRAMS

- There are two types: Sequence and Communication diagrams
- We will first look at the notation used to represent these, and then later look at important principles in OO design
- We'll look at various examples here to learn how to create the diagrams

# UML SEQUENCE DIAGRAMS

- They often represent a series of method calls between objects in a system
- The sequence is represented in what is called “fence format”, and each new object in the sequence is added to the right in the diagram
- Interactions between objects are usually method calls, but may also be object creation/deletion
- Especially useful for message flow diagrams, with request-reply pairs
- <https://sequencediagram.org/>

# EXAMPLE: SEQUENCE DIAGRAM

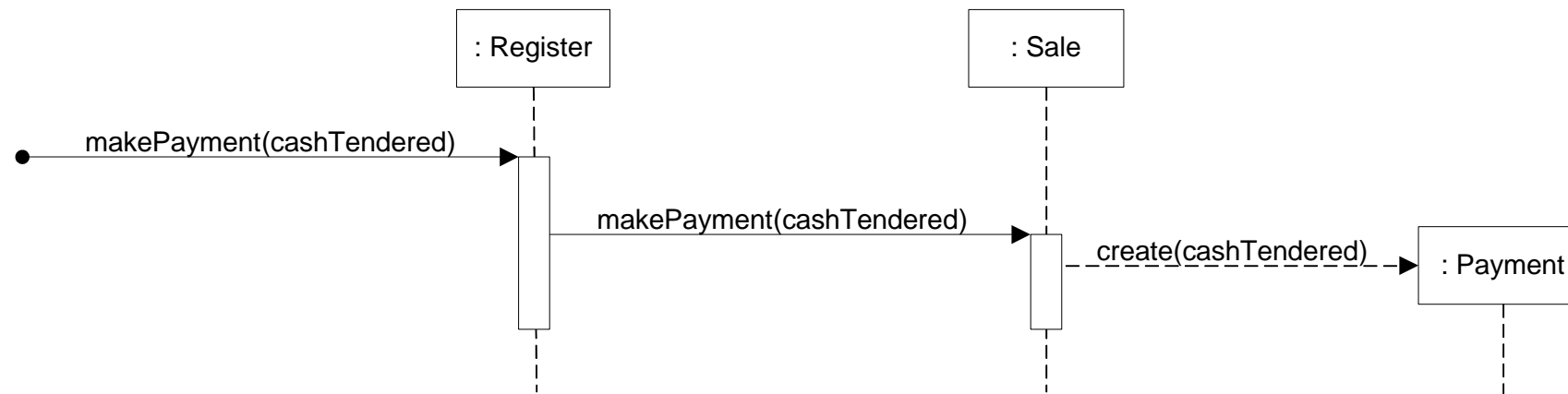


```
public class A
{
    private B myB = new B();

    Public void doOne()
    {
        myB.doTwo();
        myB.doThree();
    }
}
```



# READING A SEQUENCE DIAGRAM



- We would say “The message *makePayment* is sent to an instance of *Register*. The *Register* instance sends the *makePayment* message to the *Sale* instance. The *Sale* instance creates an instance of a *Payment*.” Here, “message” is a method call.

# INTERACTION DIAGRAMS ARE IMPORTANT

- Often left out in favor of class definition diagrams, but these diagrams are important and should be done early
- They describe how the objects interact, and may give clues to the operations and attributes needed in the class diagrams
- These diagrams are part of the *Design Model* artifact, and are started in the Elaboration phase in Agile UP

# SEQUENCE DIAGRAMS: LIFELINE BOX NOTATION

- Basic notation for the entities that make up the sequence diagram – they are called *lifeline* boxes and represent the *participants* in the particular sequence being modeled
- Note that a participant does not need to be a software class, but it usually is for our purposes
- The standard format for messages between participants is:  
**return = message(parameter: paramerType) : returnType**  
Type information is usually omitted, as are parameters

lifeline box representing an unnamed instance of class *Sale*



lifeline box representing a named instance



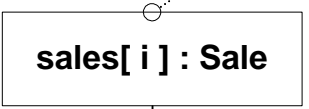
lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass



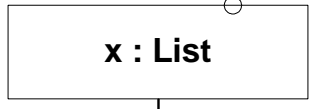
lifeline box representing an instance of an *ArrayList* class, parameterized (templated) to hold *Sale* objects



lifeline box representing one instance of class *Sale*, selected from the *sales* *ArrayList* <*Sale*> collection



*List* is an interface  
in UML 1.x we could not use an interface here, but in UML 2, this (or an abstract class) is legal



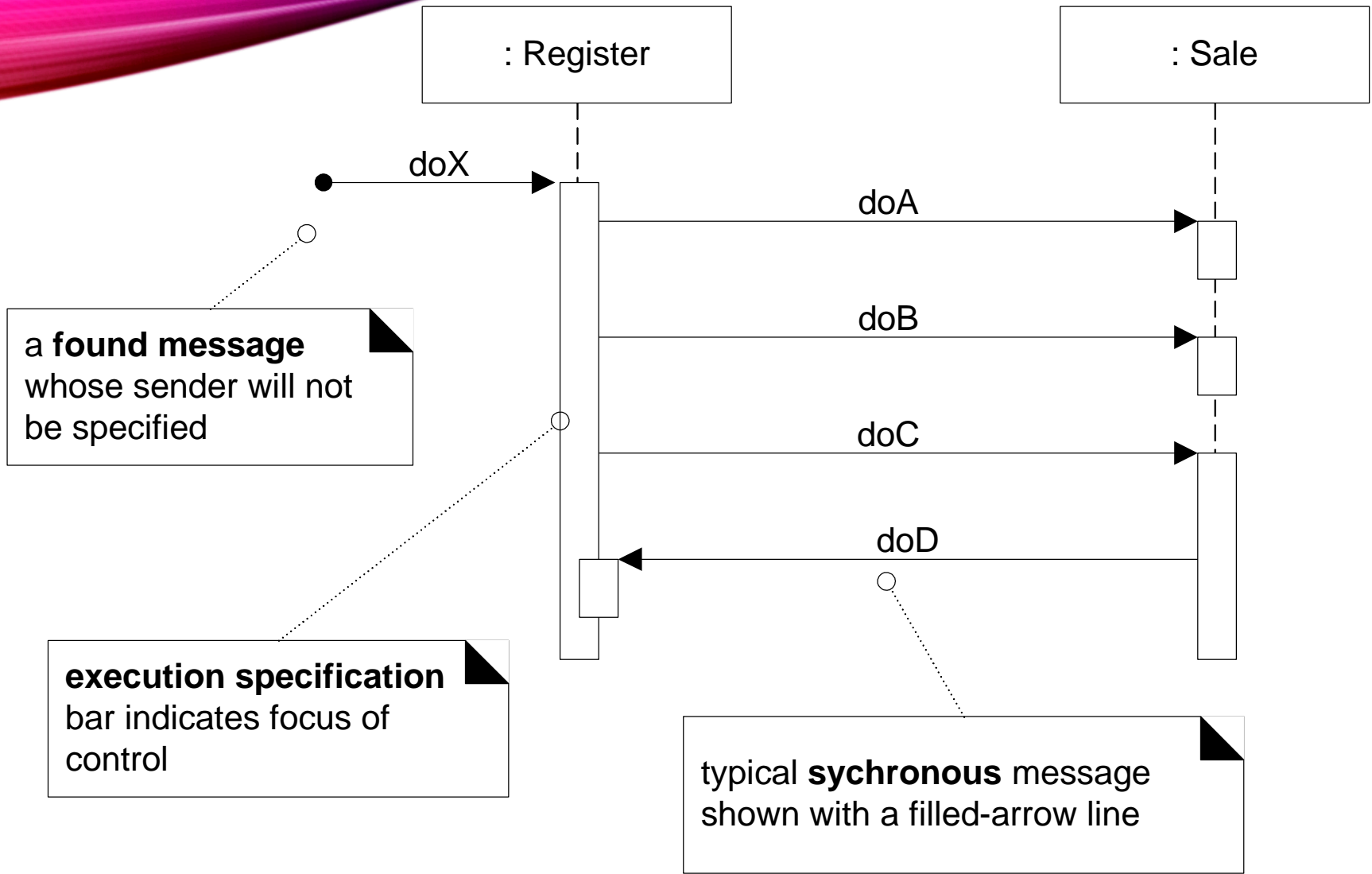
related example

# SEQUENCE DIAGRAMS: MESSAGES

- Messages are notated as solid arrows with filled in arrowheads between lifelines

The lifelines are the dotted lines that extend below each participant box, and literally show the lifespan of the participant

- The first message may come from an unspecified participant, and is called a “found message”. It is indicated with a ball at the source
- Messages can be *synchronous* (sender waits until receiver as finished processing the message, and then continues – blocking call) or *asynchronous* (sender does not wait, more rare in OO designs)
- Dashed arrow is used to indicate return of control, e.g. after receipt of synchronous message. May contain a value.



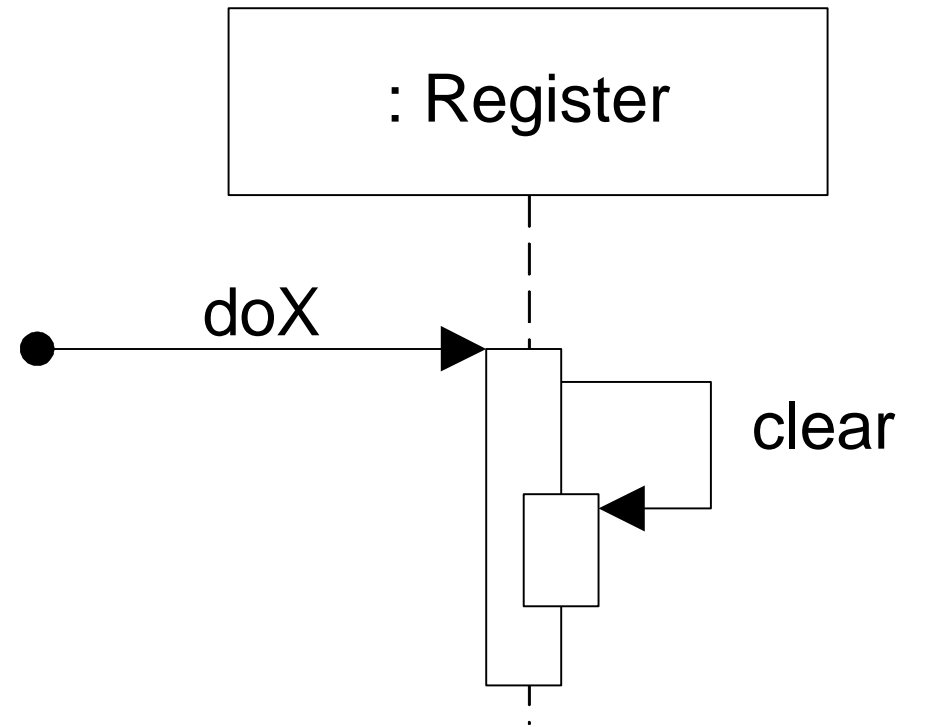
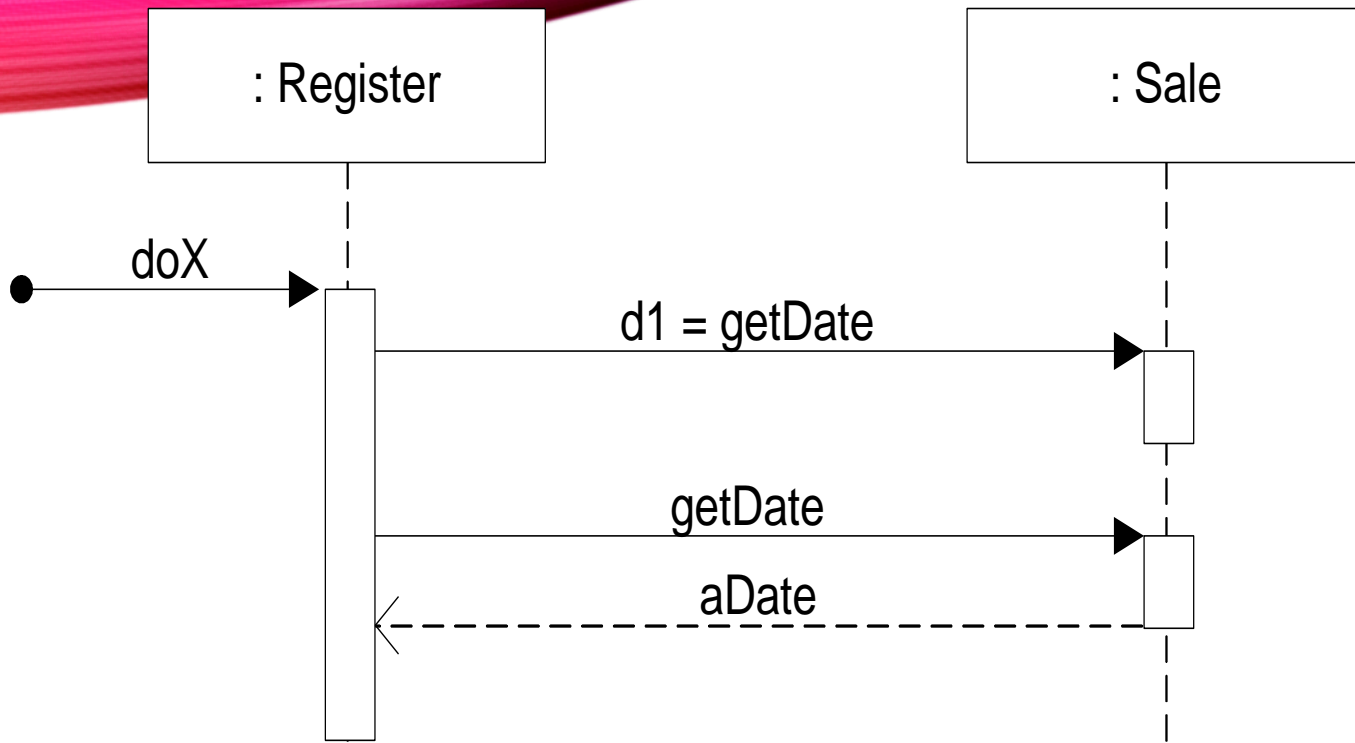
a **found message**  
whose sender will not  
be specified

**execution specification**  
bar indicates focus of  
control

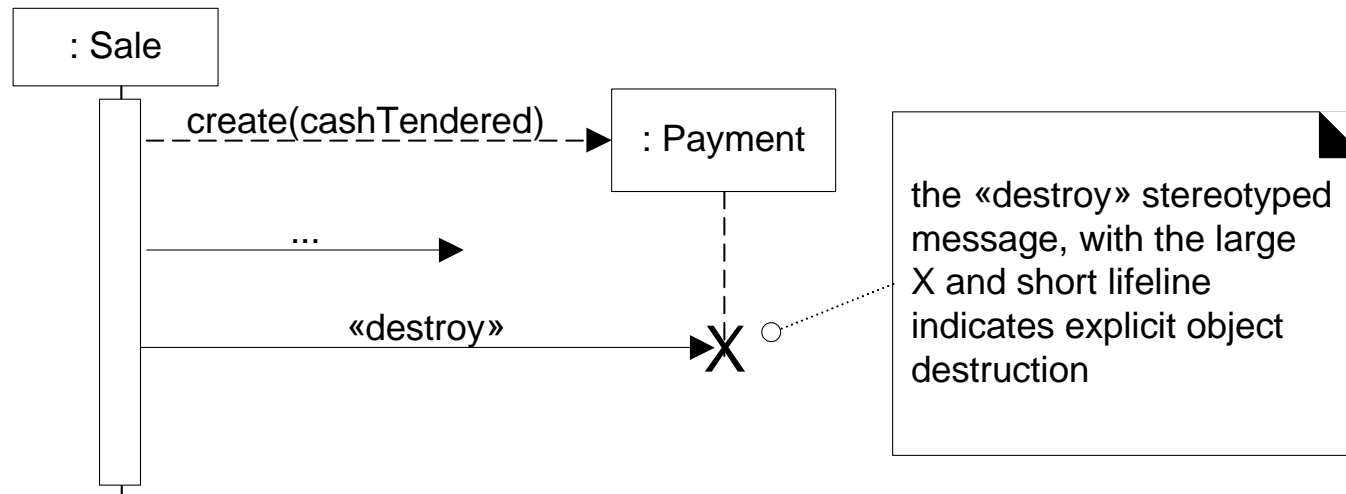
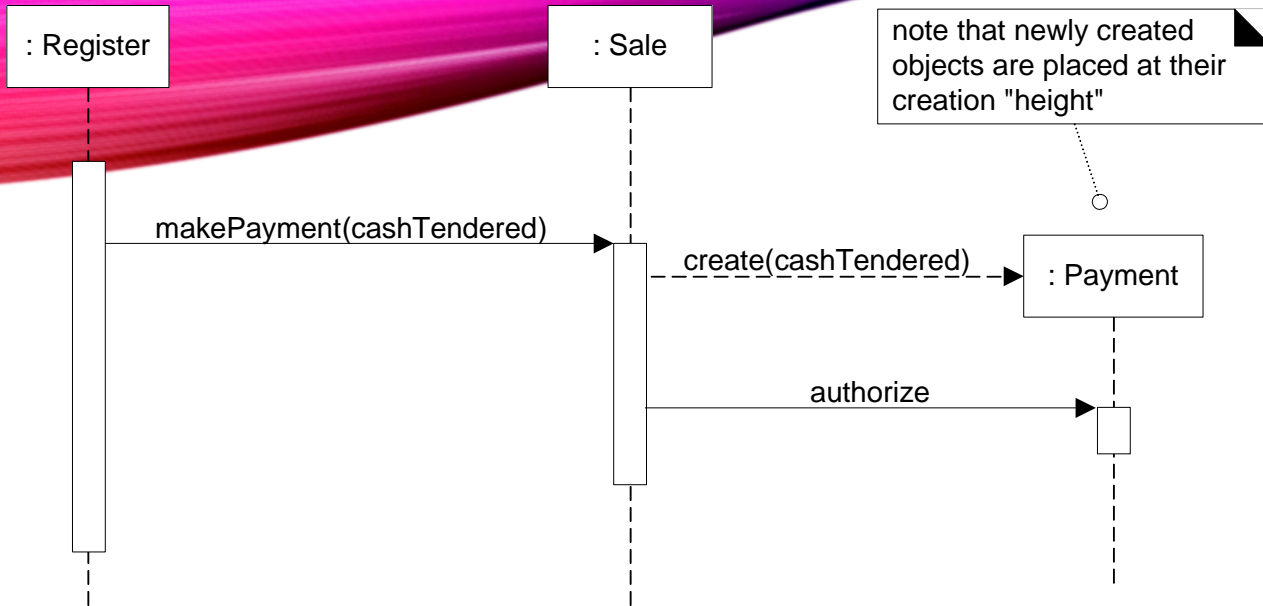
typical **synchronous** message  
shown with a filled-arrow line

# SEQUENCE DIAGRAMS: SPECIFICS

- The *execution specification bar* or *activation bar* indicates that the operation is on the call stack
- Usually replies to messages are indicated with a value or a dotted line (see next slide)
- It is possible to have a message to “self” (or “this”)
- Sequence diagrams can also indicate instance creation (see later slide)
- Likewise, instances can be destroyed (indicated by “X” at the end of lifeline)







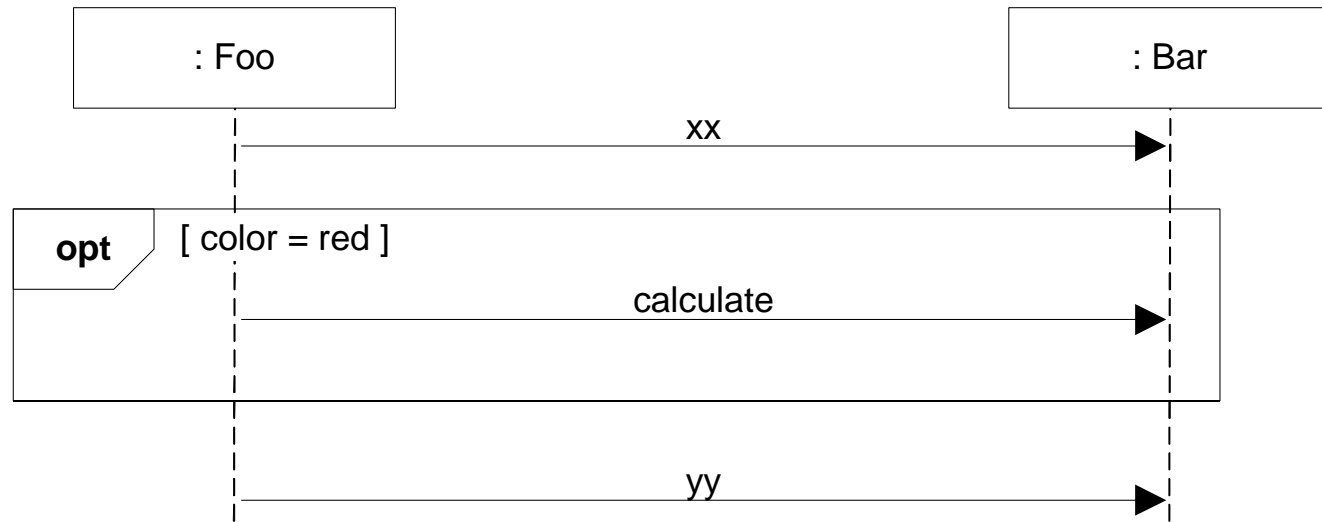
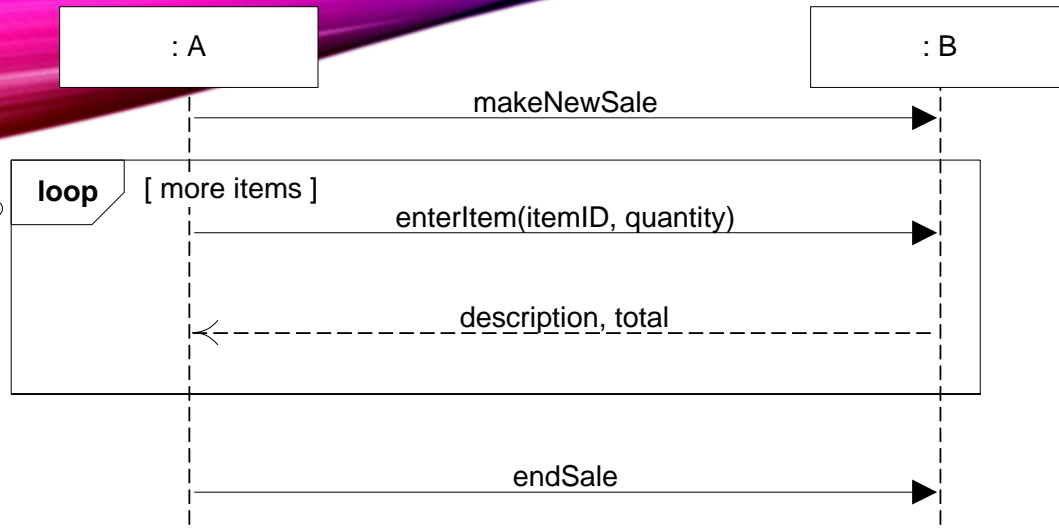
# SEQUENCE DIAGRAMS: SPECIFICS

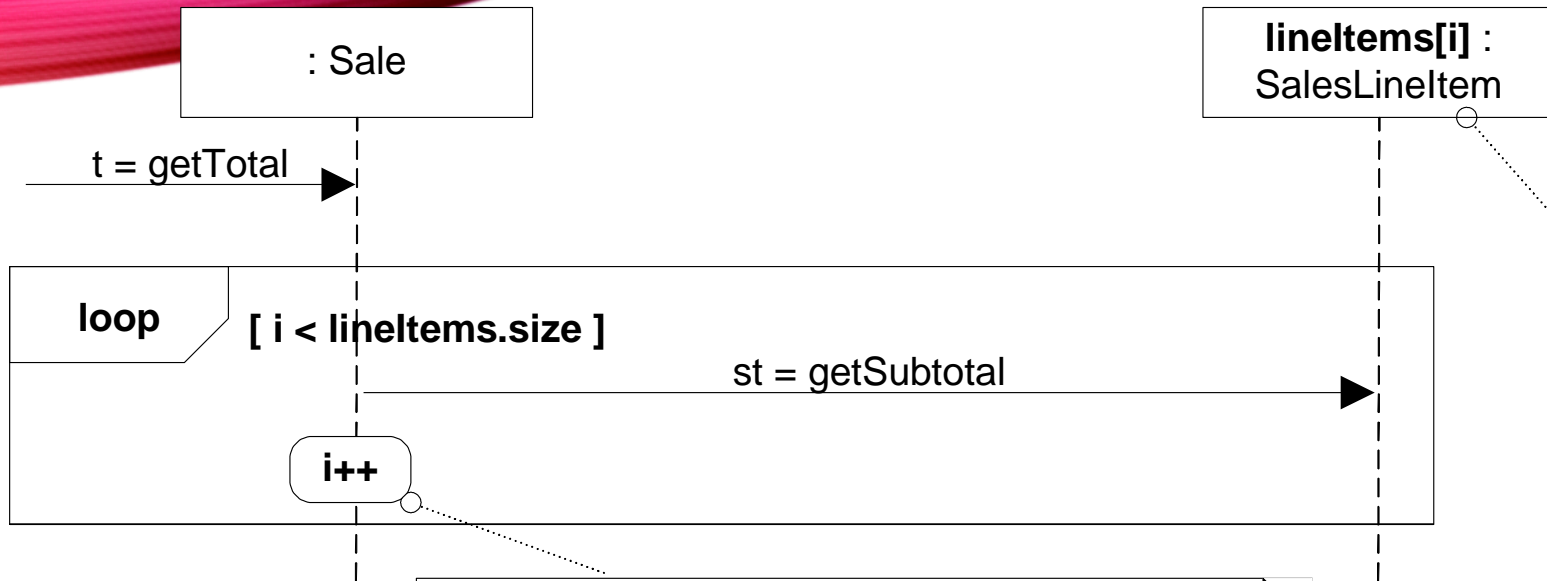
Diagram frames may be used in sequence diagrams to show:

- Loops
- Conditional (optional) messages
- Nesting (a conditional loop)
- Relationships between diagrams

**See next slides for examples**

a UML loop frame, with a boolean guard expression



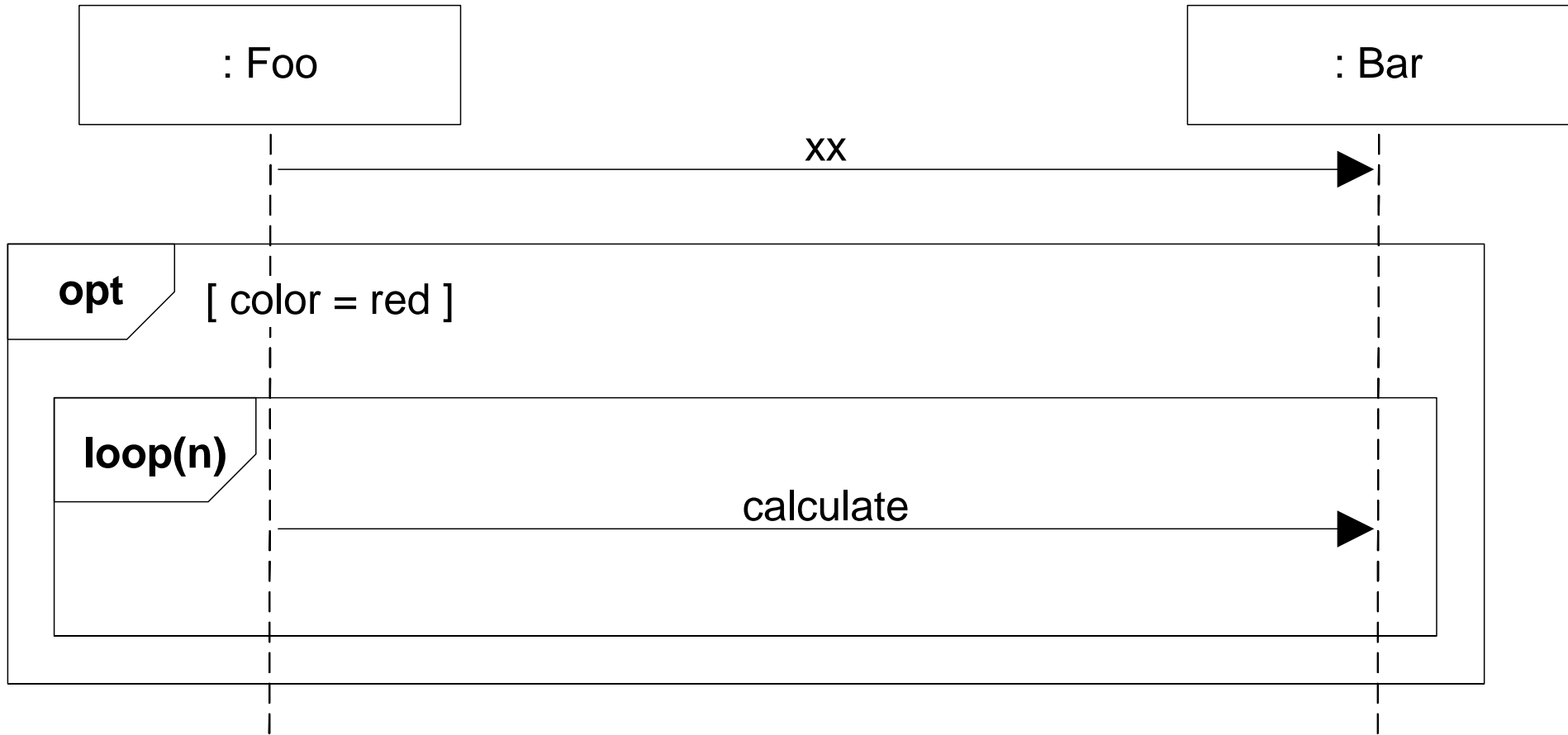


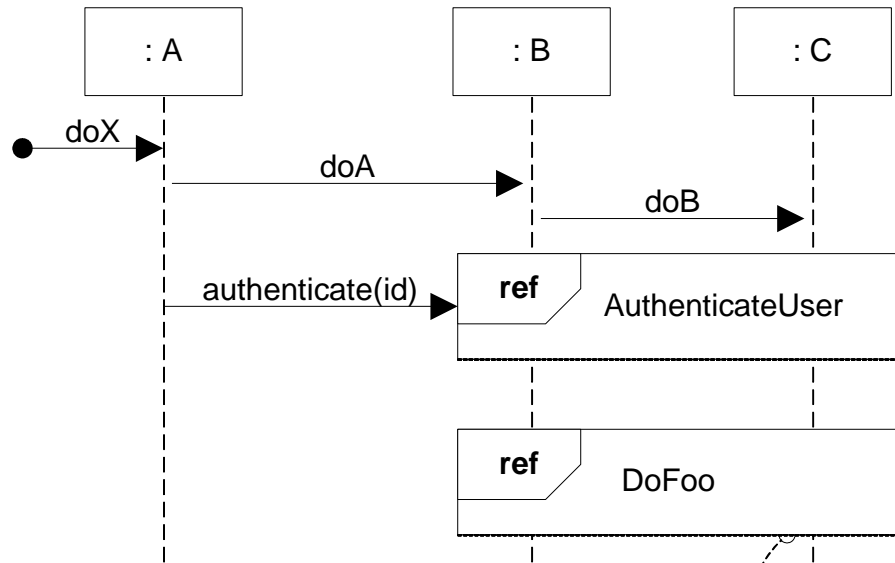
This lifeline box represents one instance from a collection of many *SalesLineItem* objects.

*lineItems[i]* is the expression to select one element from the collection of many *SalesLineItems*; the "i" value refers to the same "i" in the guard in the LOOP frame

an **action box** may contain arbitrary language statements (in this case, incrementing 'i')

it is placed over the lifeline to which it applies

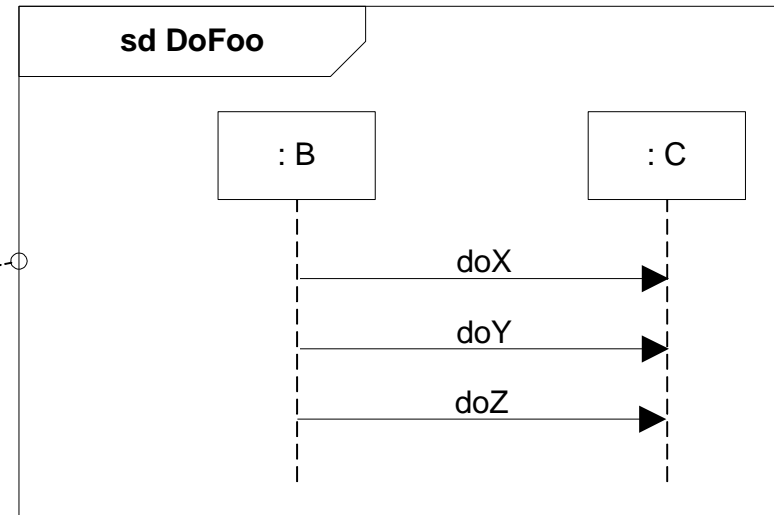
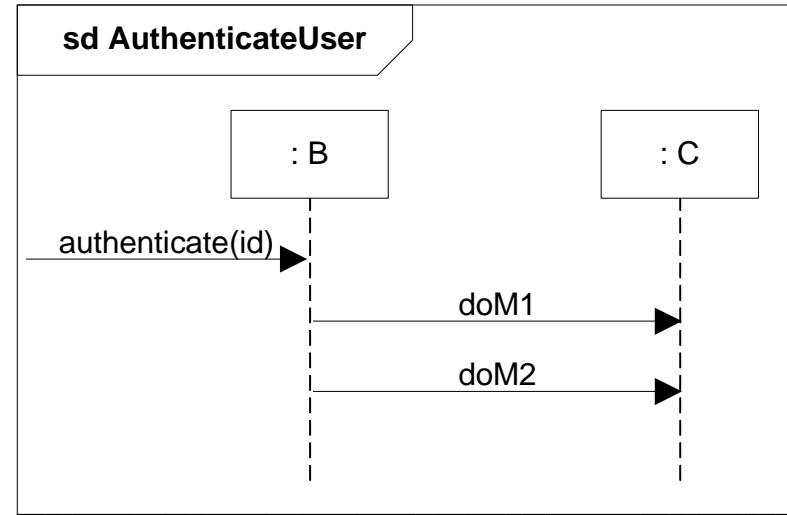


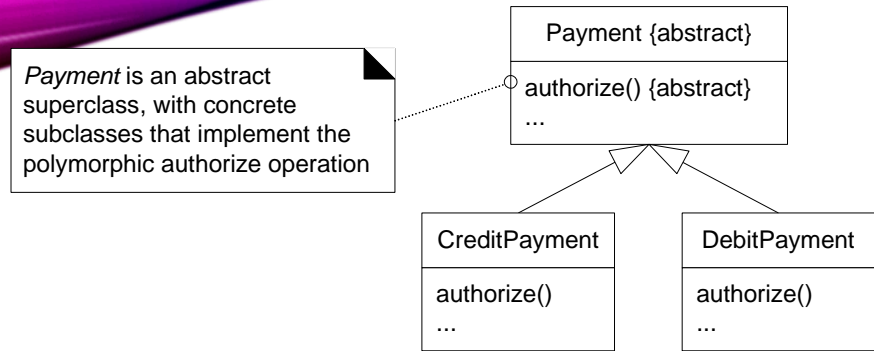


interaction occurrence

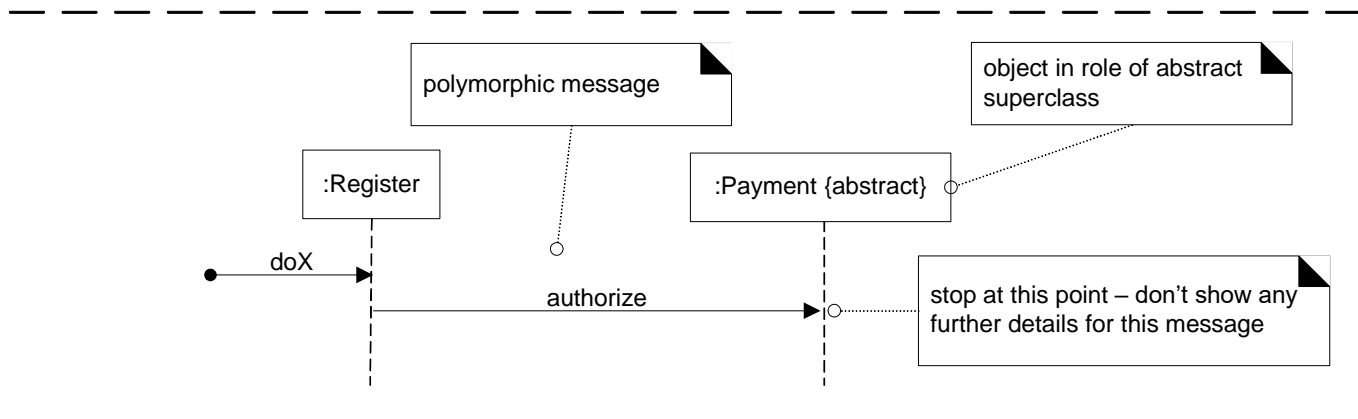
note it covers a set of lifelines

note that the sd frame it relates to has the same lifelines: B and C





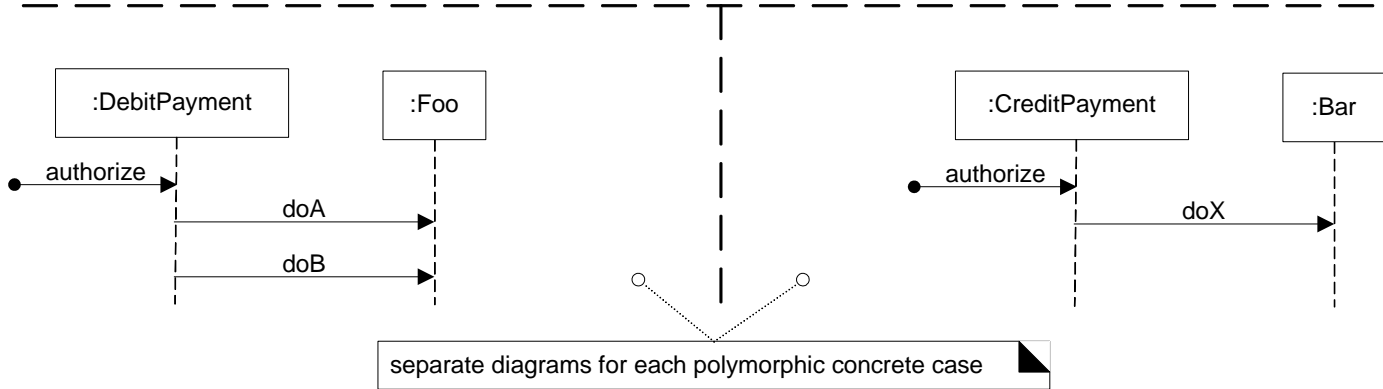
Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation



polymorphic message

object in role of abstract superclass

stop at this point – don't show any further details for this message



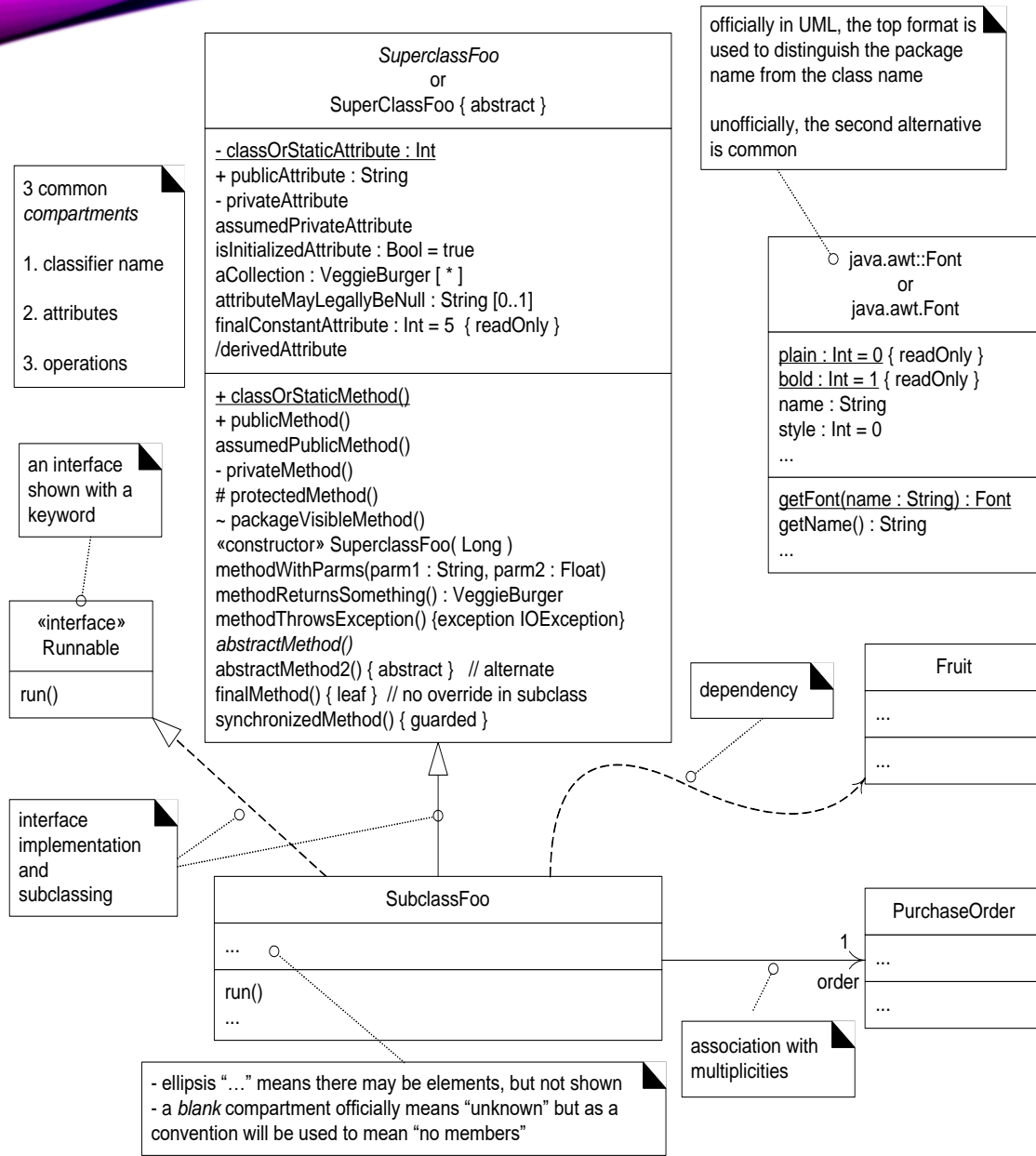
separate diagrams for each polymorphic concrete case

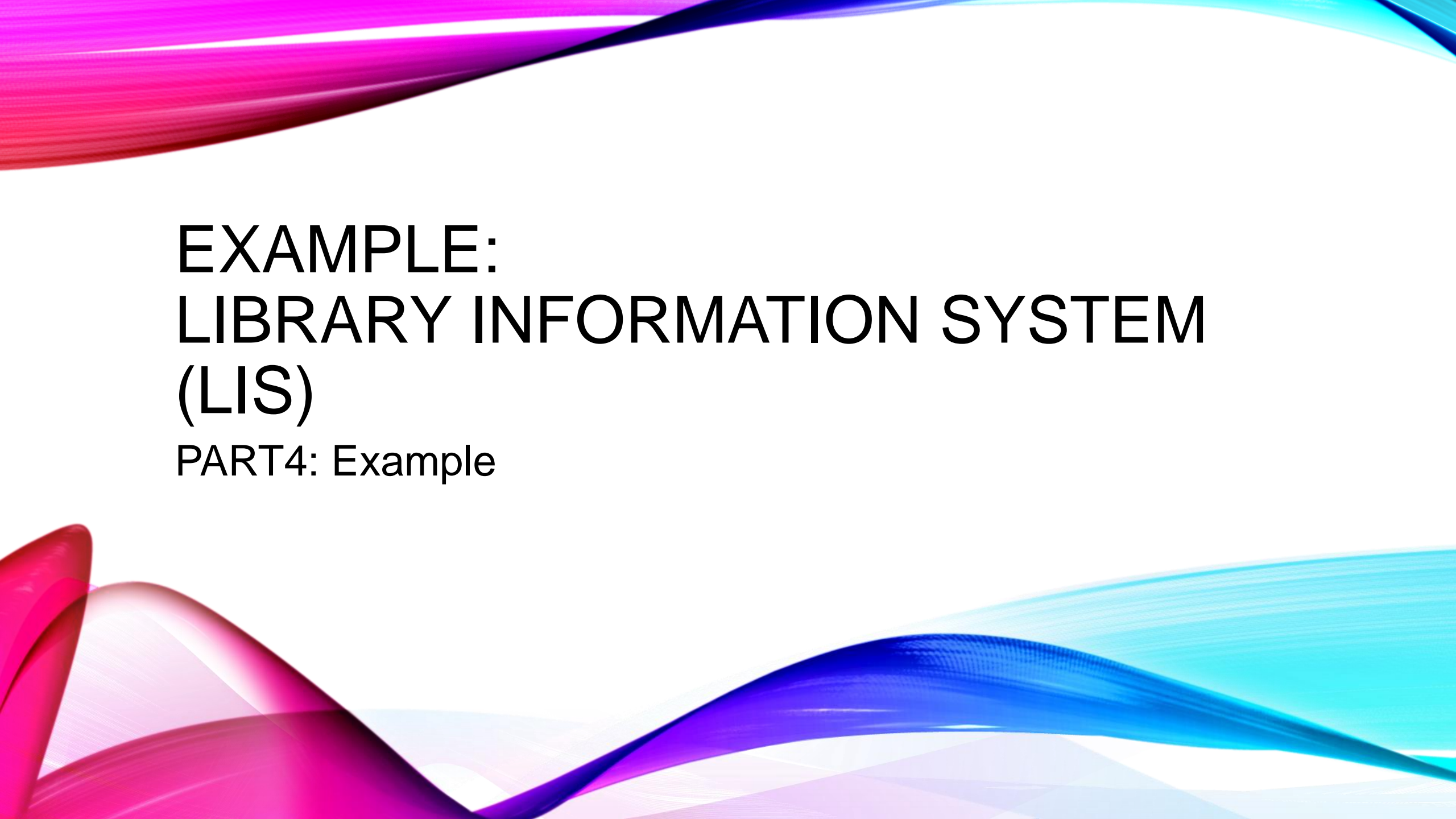


# UML DESIGN CLASS DIAGRAMS

DESIGN- STATIC VIEW







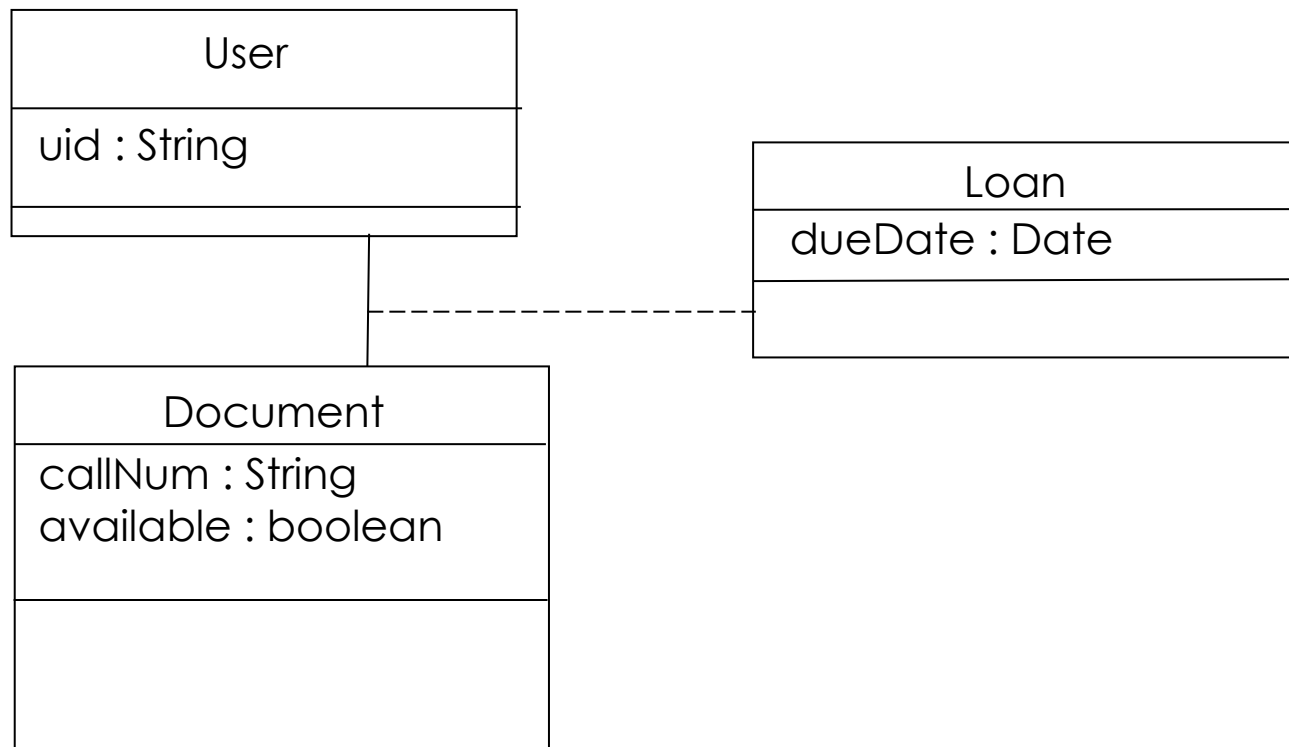
# EXAMPLE: LIBRARY INFORMATION SYSTEM (LIS)

PART4: Example

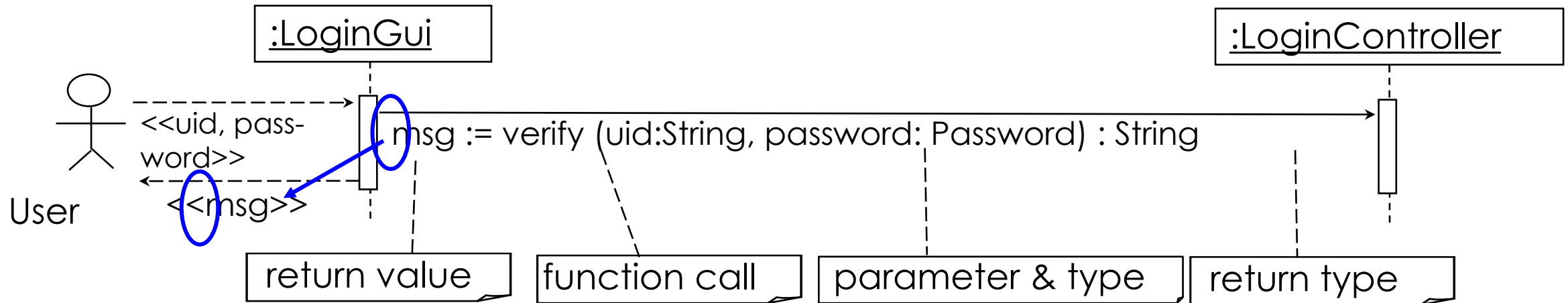
# LIS REQUIREMENTS AND USE CASES

- R1. The LIS must allow a patron to check out documents.
- R2. The LIS must allow a patron to return documents.
  
- UC1. Checkout Document (Actor: Patron, System: LIS)
- UC2. Return Document (Actor : Patron, System: LIS)
- **How about Allow a Patron? Is it a use case? Who is the actor? What is the goal or business task for the actor? Does it start and end with an actor?**

# DOMAIN MODEL



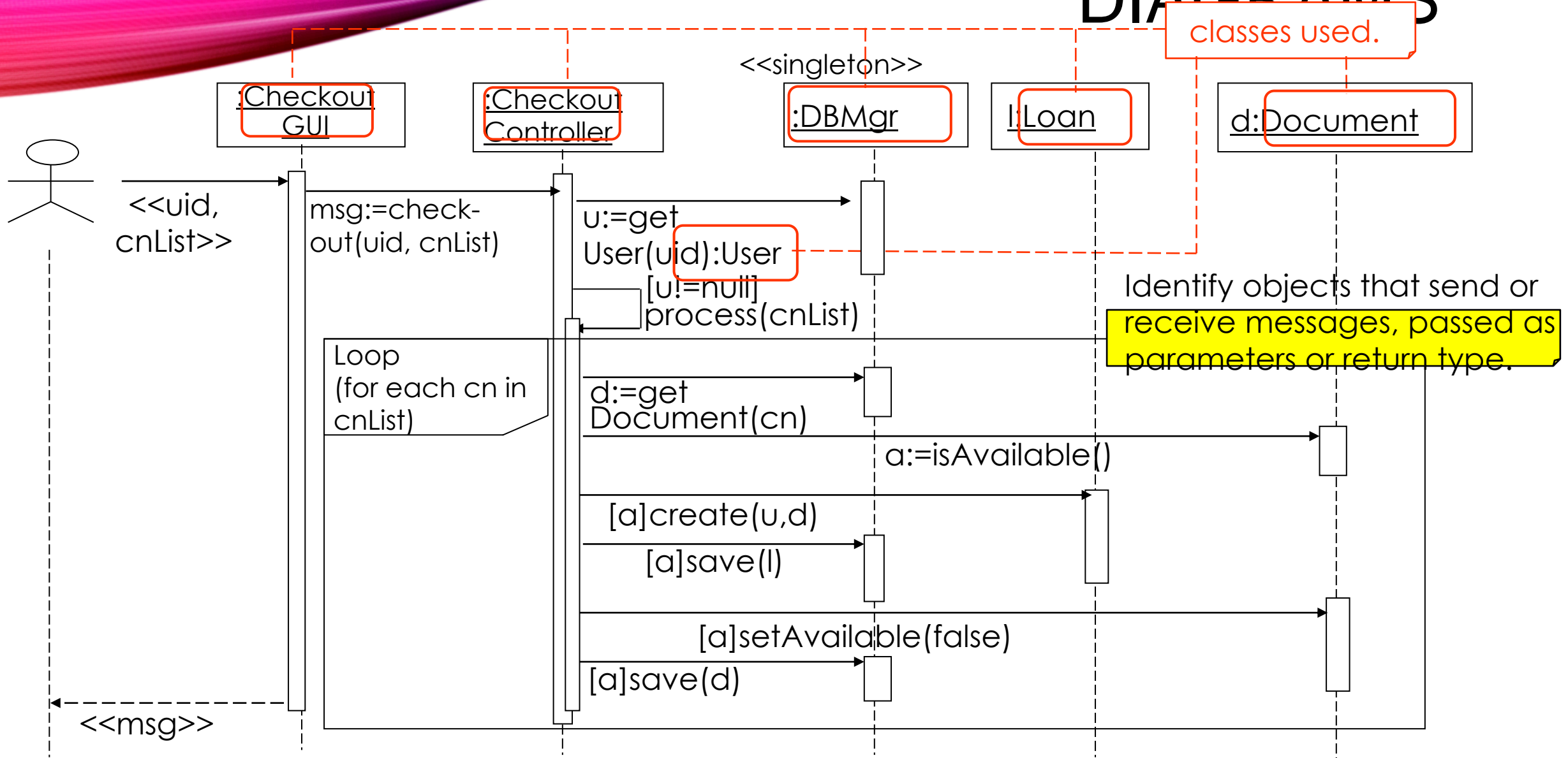
# LOGIN USE CASE



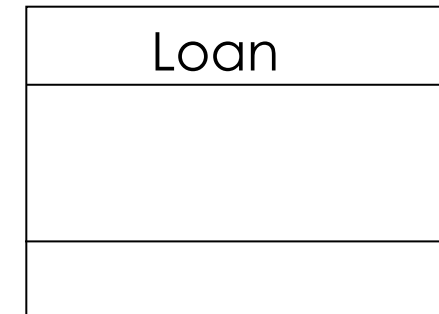
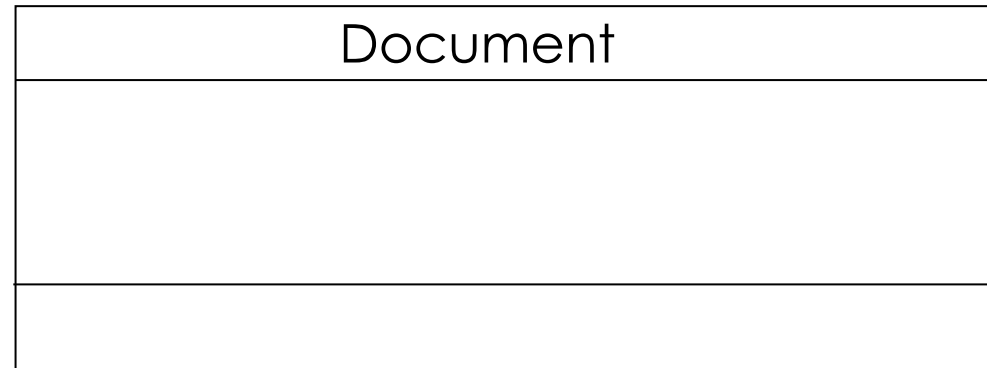
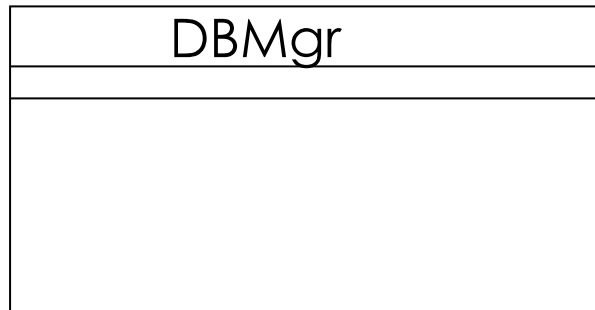
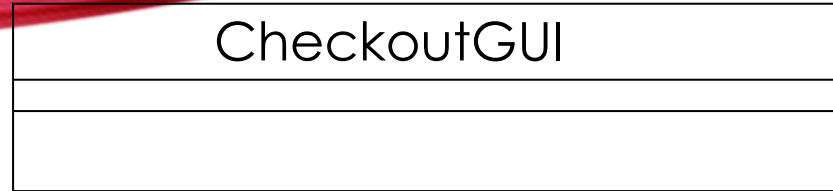
# LIS UC.1 CHECKOUT DOCUMENT

UC1 : Checkout Document	
Precondition: Patron is already logged in	
Actor: Patron	System: LIS
	0. The LIS displays the main menu.
1. Patron clicks the checkout Document button on the main menu.	2.The system displays the checkout menu.
3. The Patron enters the call numbers of documents to be checked out and clicks the Submit button.	4. The system displays the document details for confirmation.
5. The patron click the OK button to confirm the checkout.	6. The system displays a confirmation message to patron.
7. The patron clicks OK button on the confirmation dialog.	

# IDENTIFY CLASSES USED IN SEQUENCE DIAGRAMS

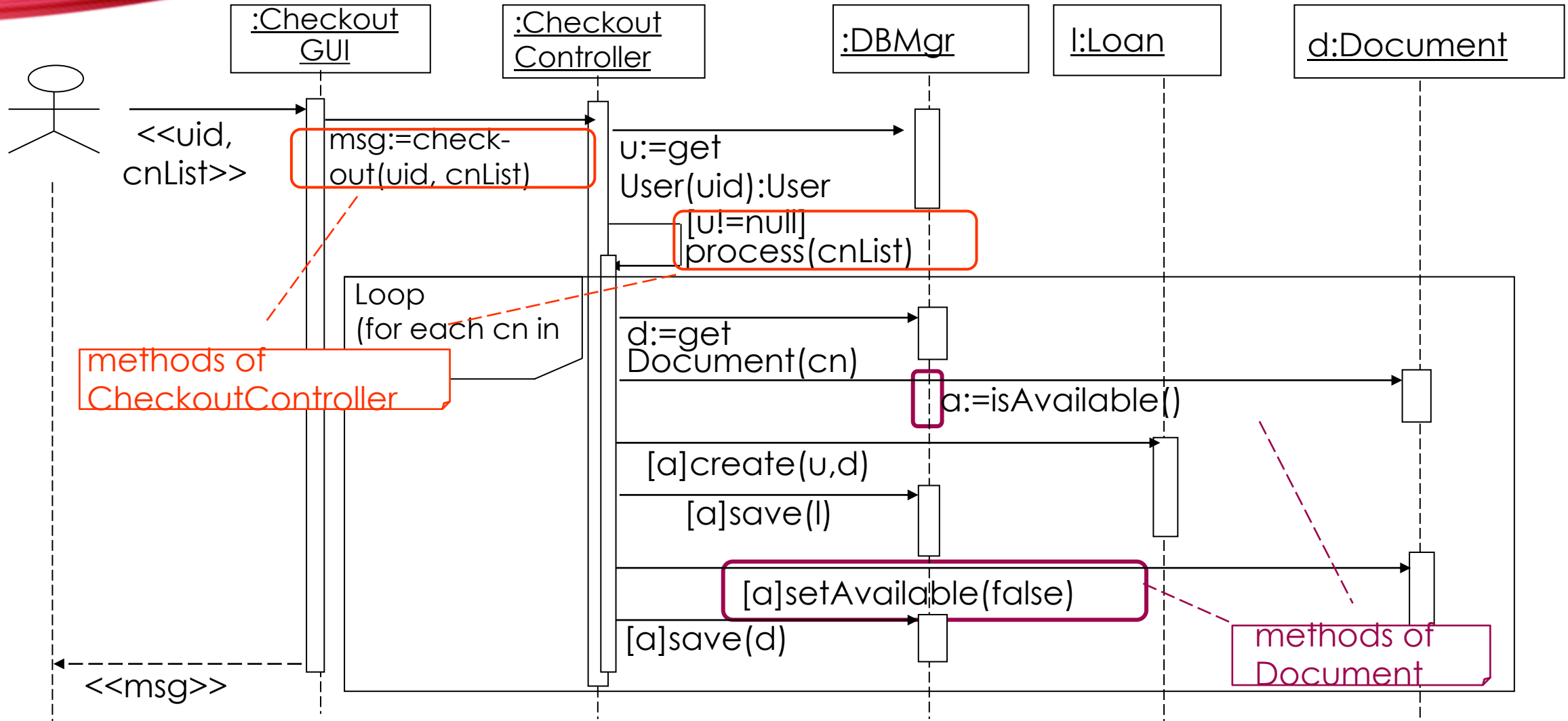


# CLASSES IDENTIFIED

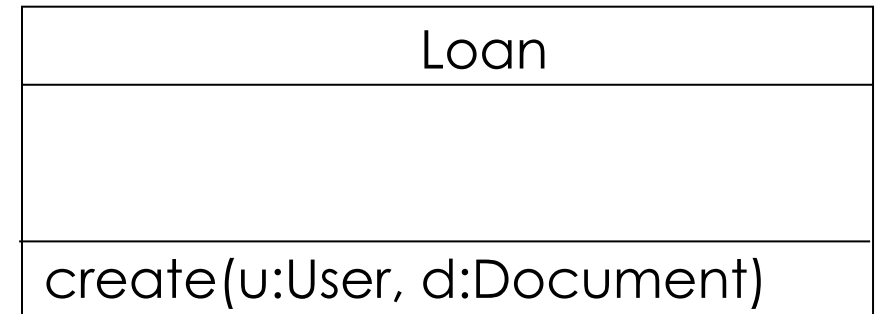
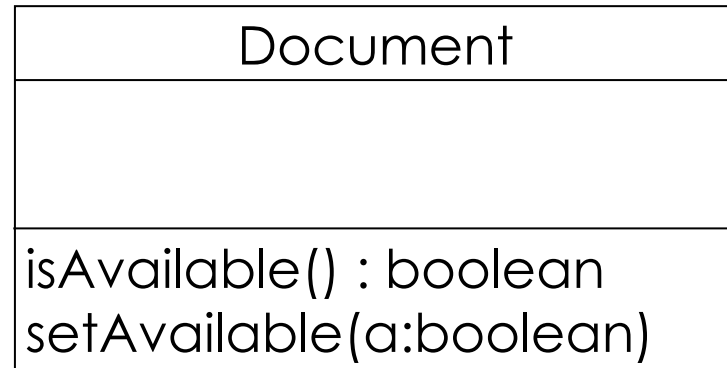
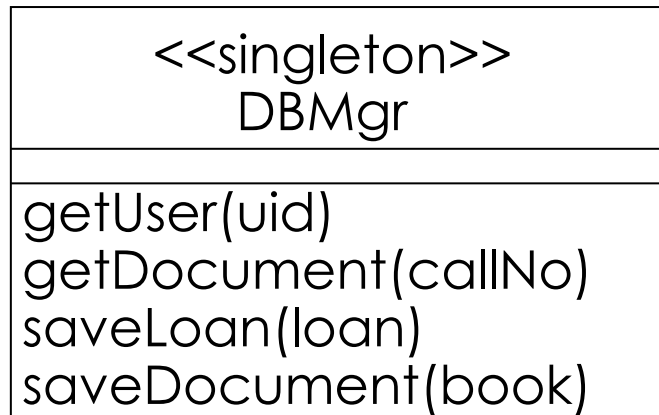
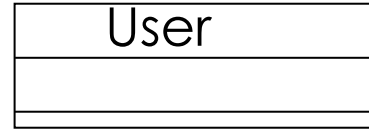
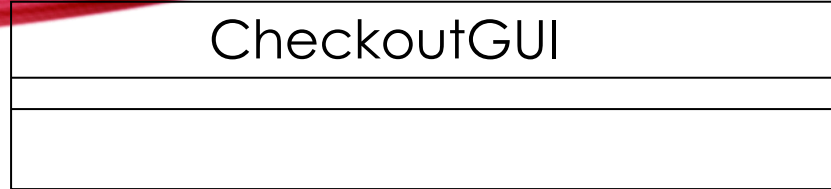




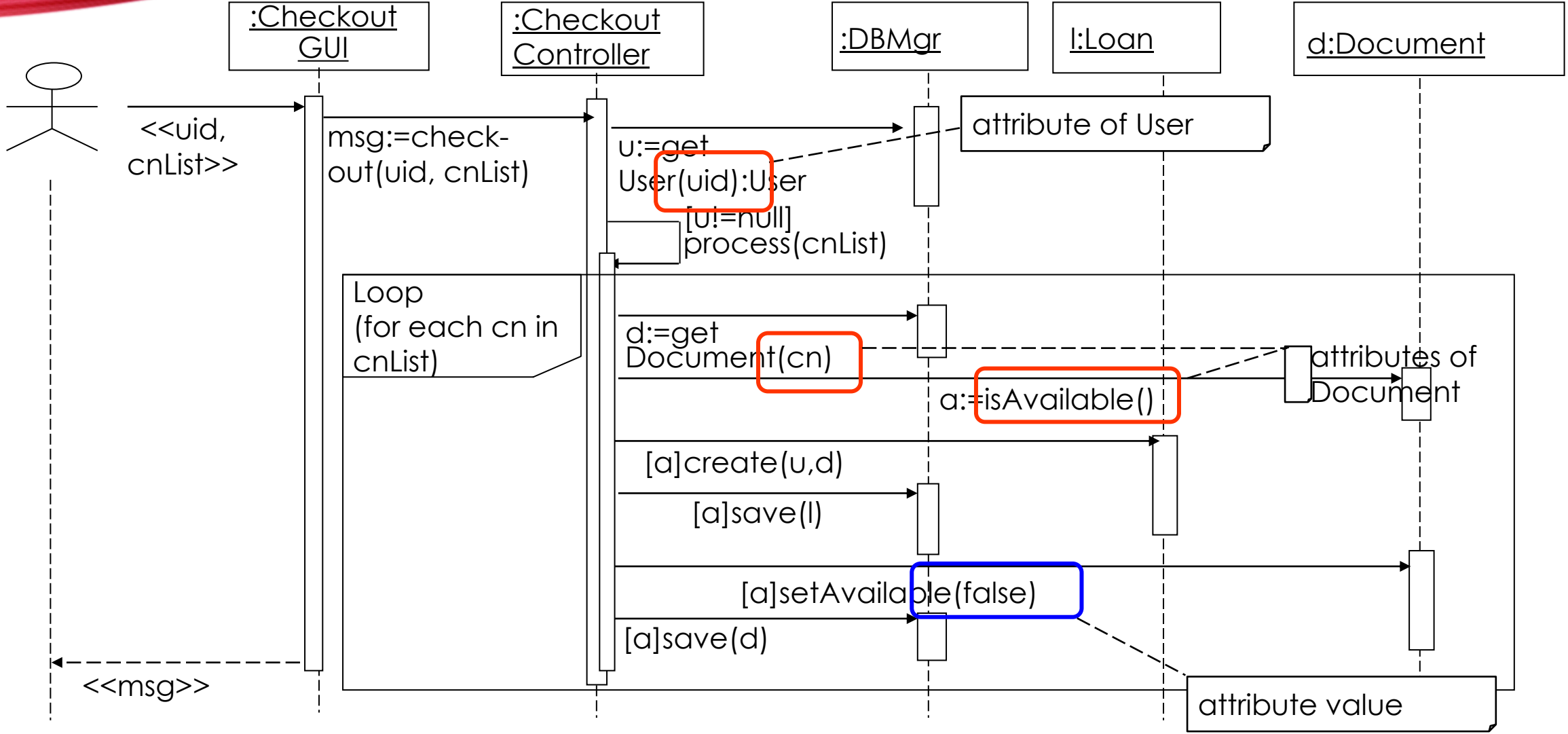
# IDENTIFY METHODS



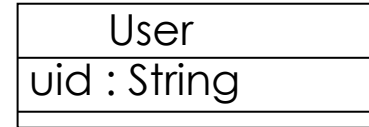
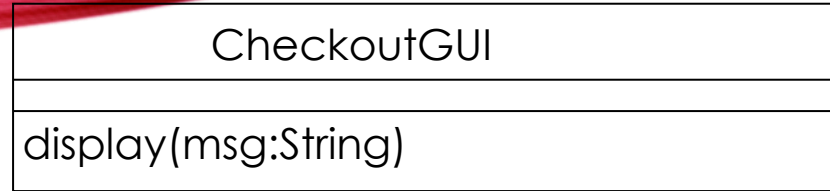
# FILL IN IDENTIFIED METHODS



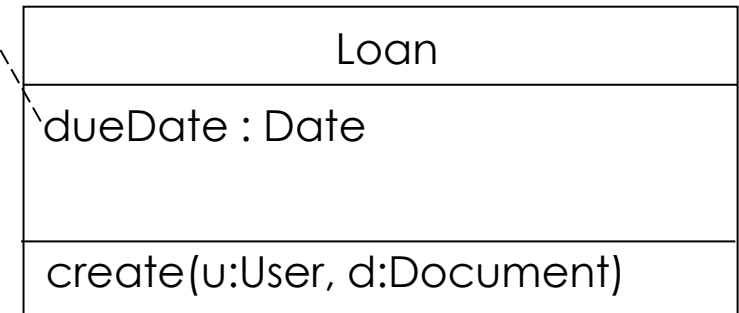
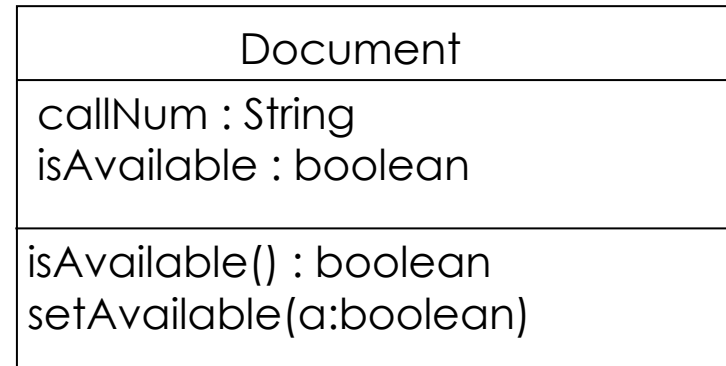
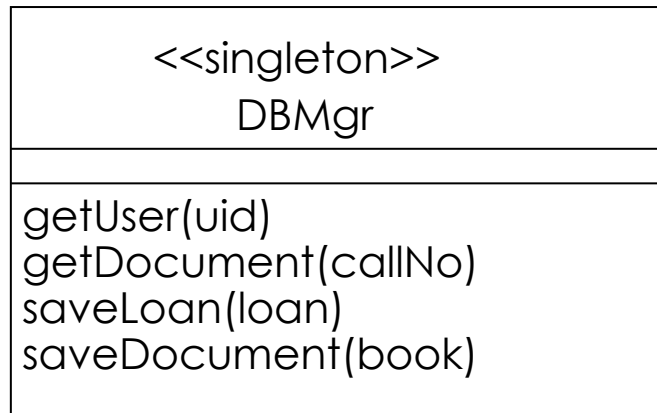
# IDENTIFY ATTRIBUTES



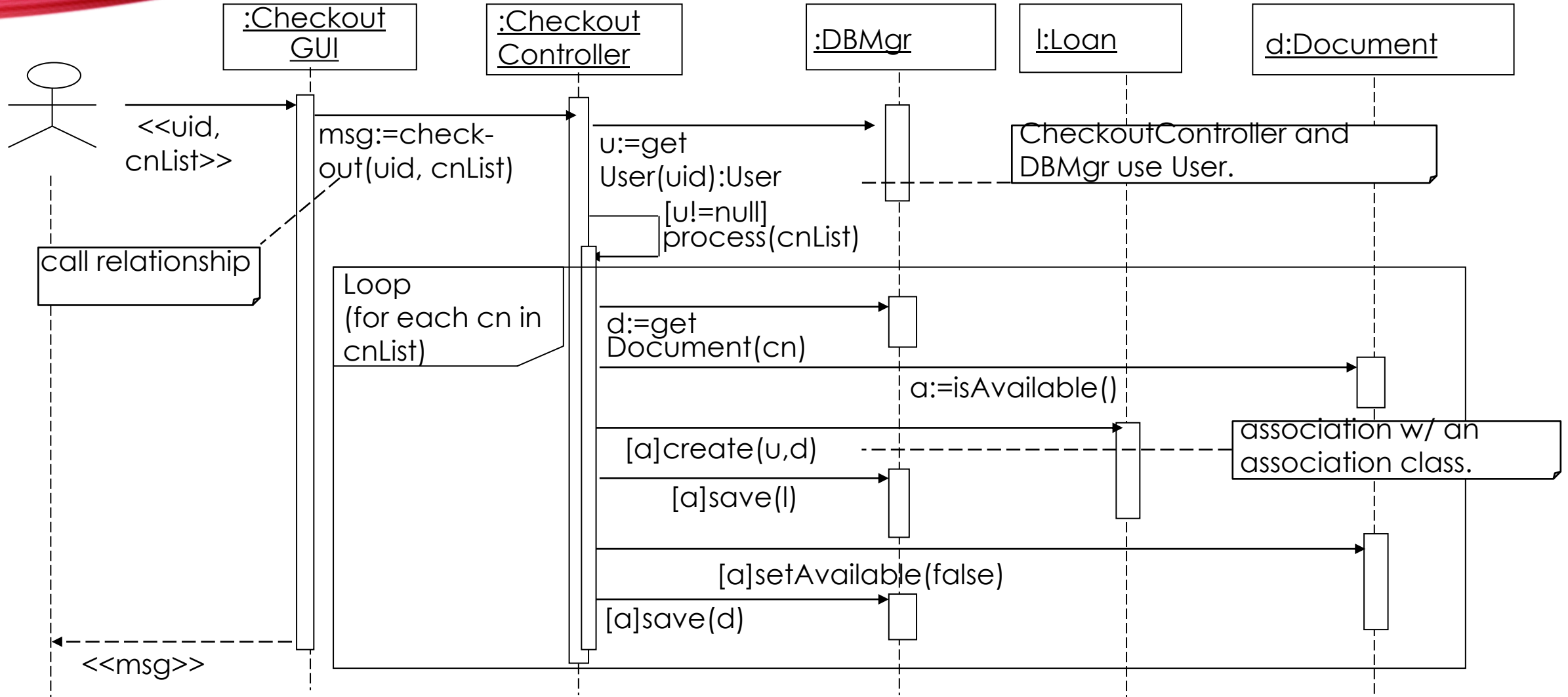
# FILL IN ATTRIBUTES



from domain  
model

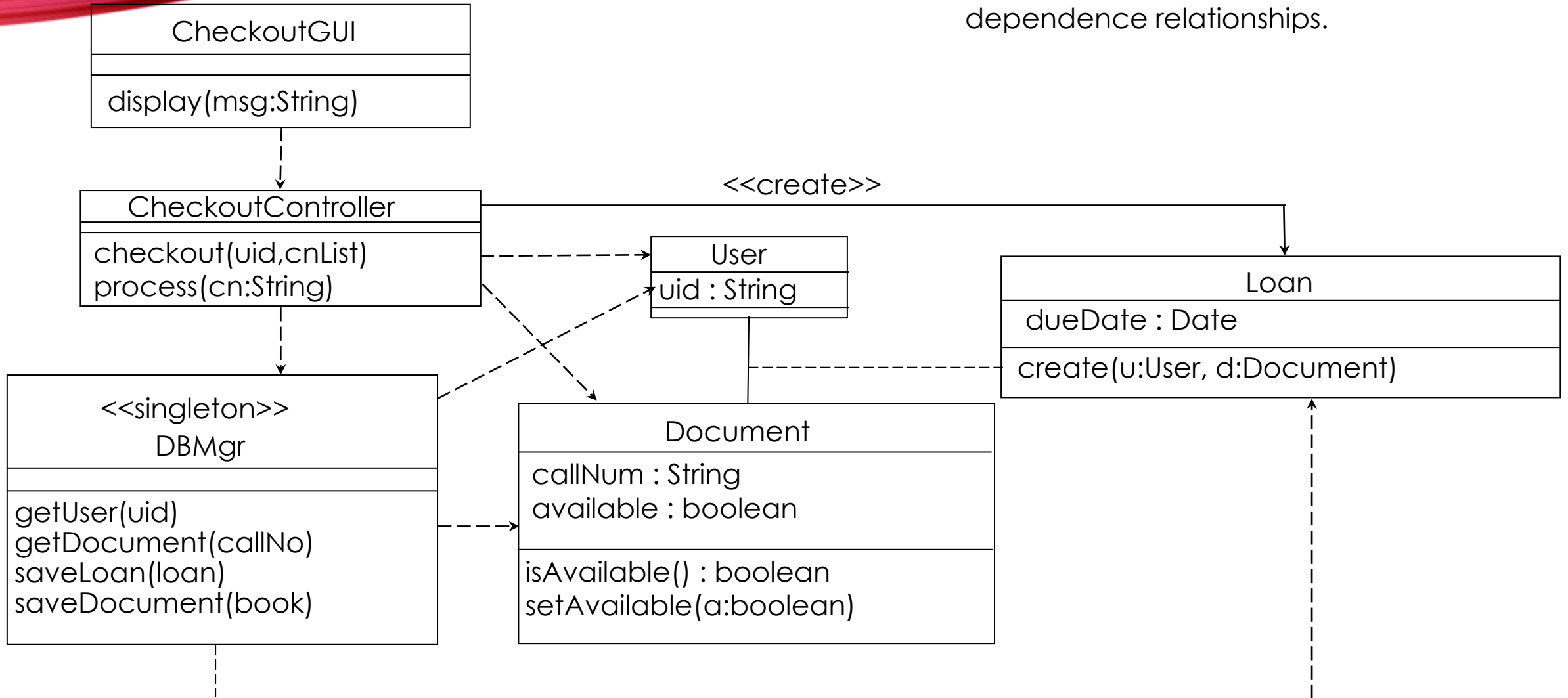


# IDENTIFY RELATIONSHIPS

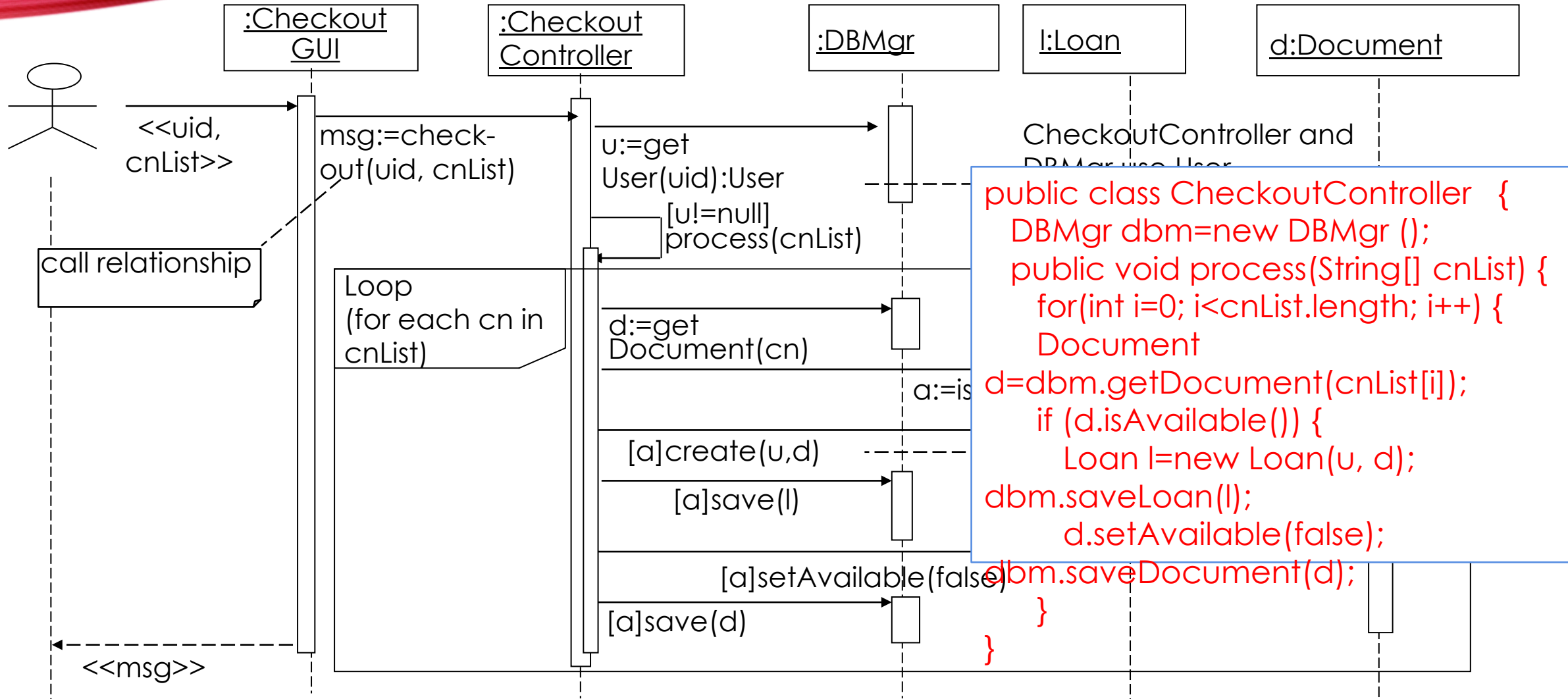


# FILL IN RELATIONSHIPS

The dashed arrow lines denote uses or dependence relationships.



# FROM SEQUENCE DIAGRAM TO IMPLEMENTATION



# TEXTBOOK

- *“Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development”*, Craig Larman, ISBN: 013 148 9062, Prentice-Hall, 2005
- *“Object-Oriented Software Engineering: An Agile Unified Methodology”*, Kung D., ISBN: 978-0073376257, McGraw Hill, 2013