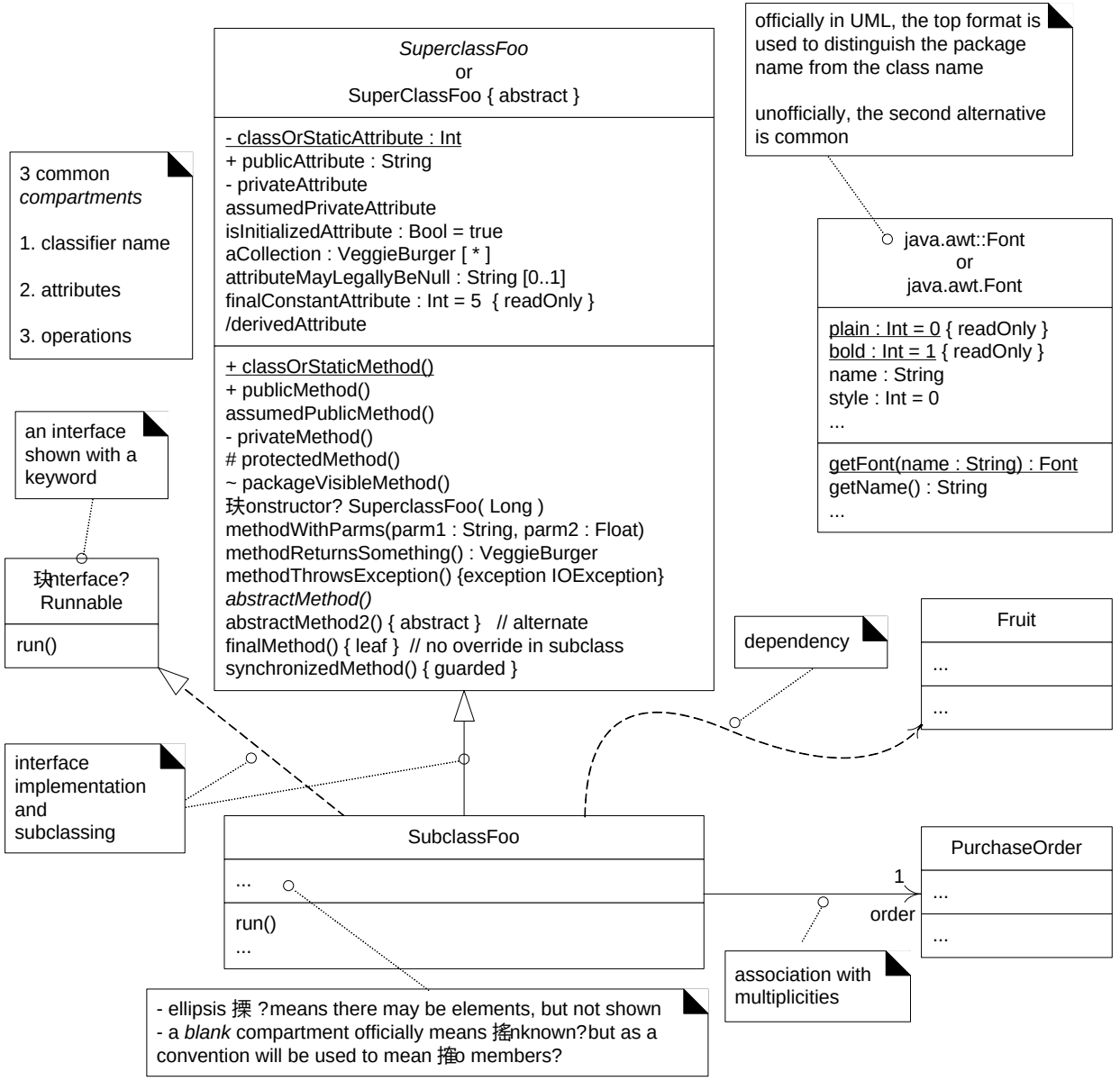
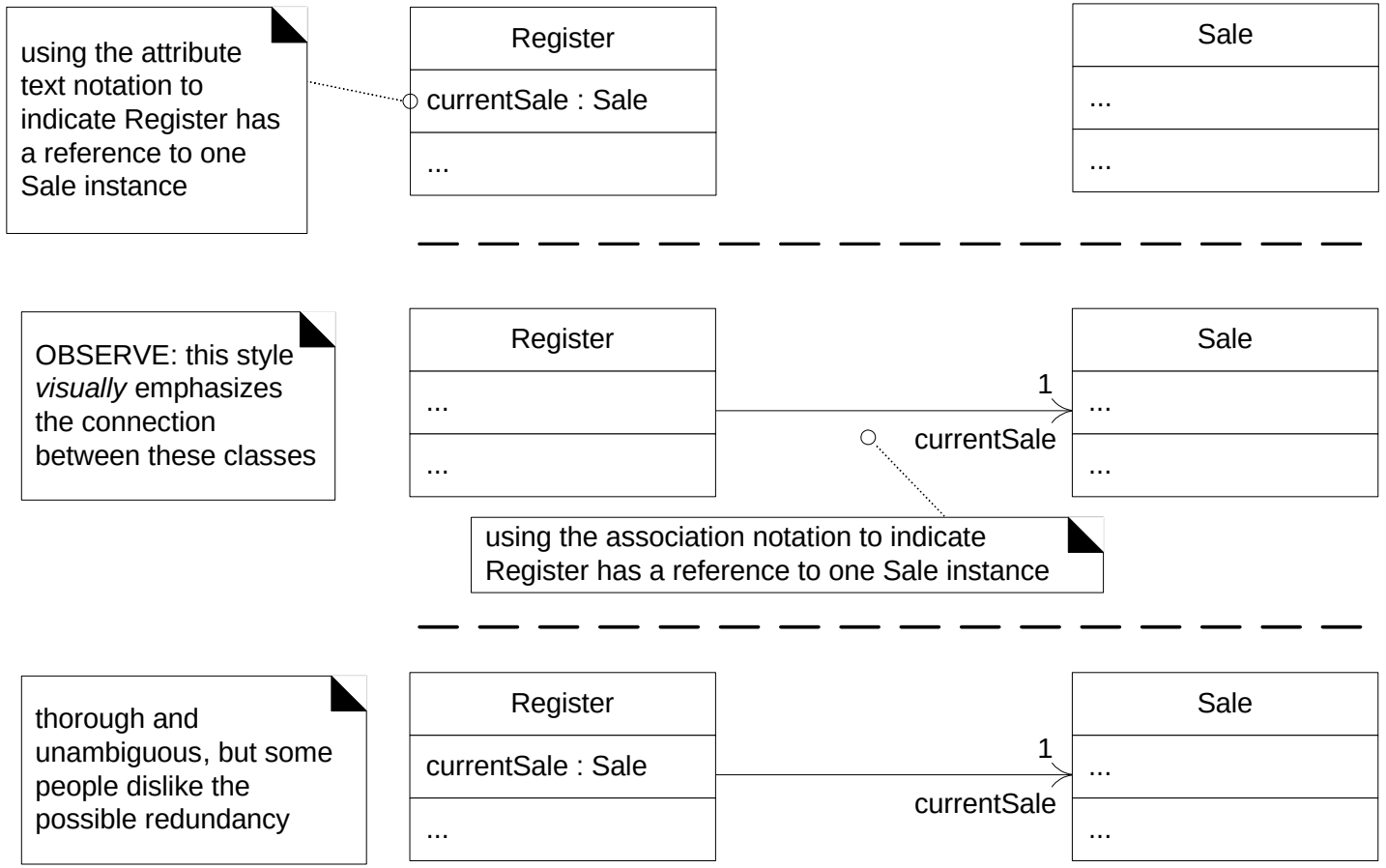
Object-Oriented Analysis and Design

PART2: DESIGN

UML class diagrams


SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
珐onstructor? SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
3 common compartments
1. classifier name
2. attributes
3. operations
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
珐nterface?
Runnable
run()
dependency
Fruit
...
...
interface implementation and subclassing
SubclassFoo
...
run()
...
1
order
PurchaseOrder
...
...
association with multiplicities
- ellipsis 捰 ?means there may be elements, but not shown
- a blank compartment officially means 捲nknown?but as a convention will be used to mean 捵o members?

UML: Notating Attributes



Attributes As Associations

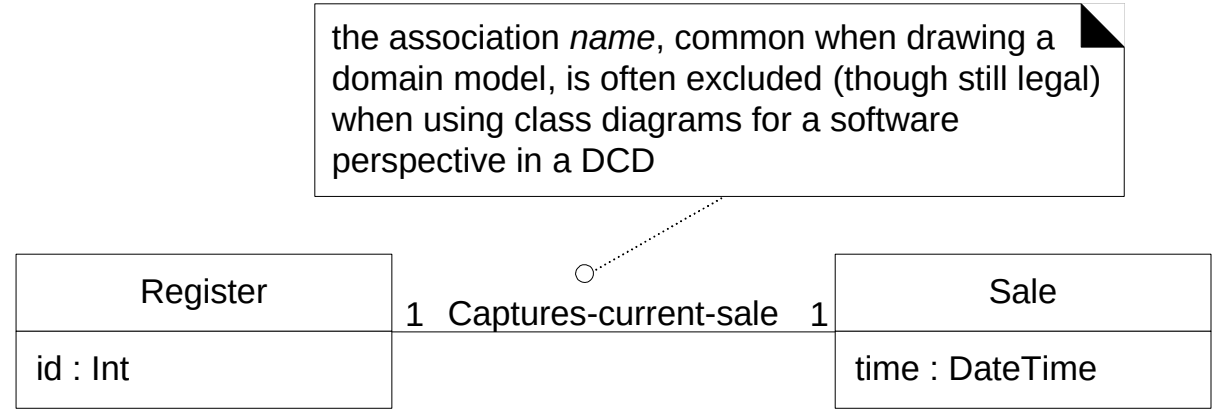
Notice that there are subtle differences between the conceptual perspective (Domain Model) and software perspective (Design Model) for attributes that are defined as associations

For DCDs, there is usually

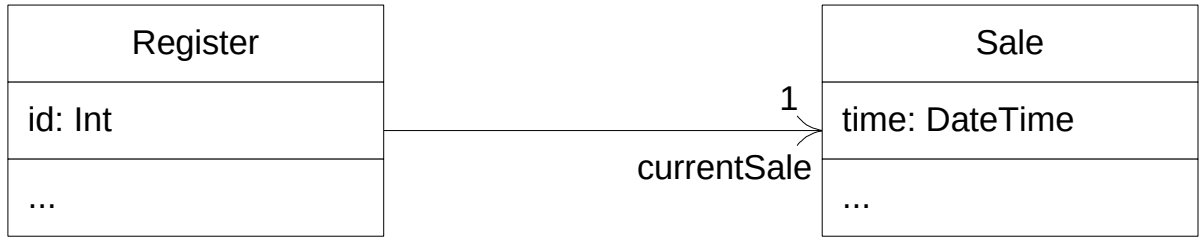
- A navigability arrow
- A multiplicity at the target end, but not the source
- A role name
- No association name

UML: Attributes as Associations

UP Domain Model
conceptual perspective



UP Design Model
DCD
software perspective



Attributes: Text versus Associations

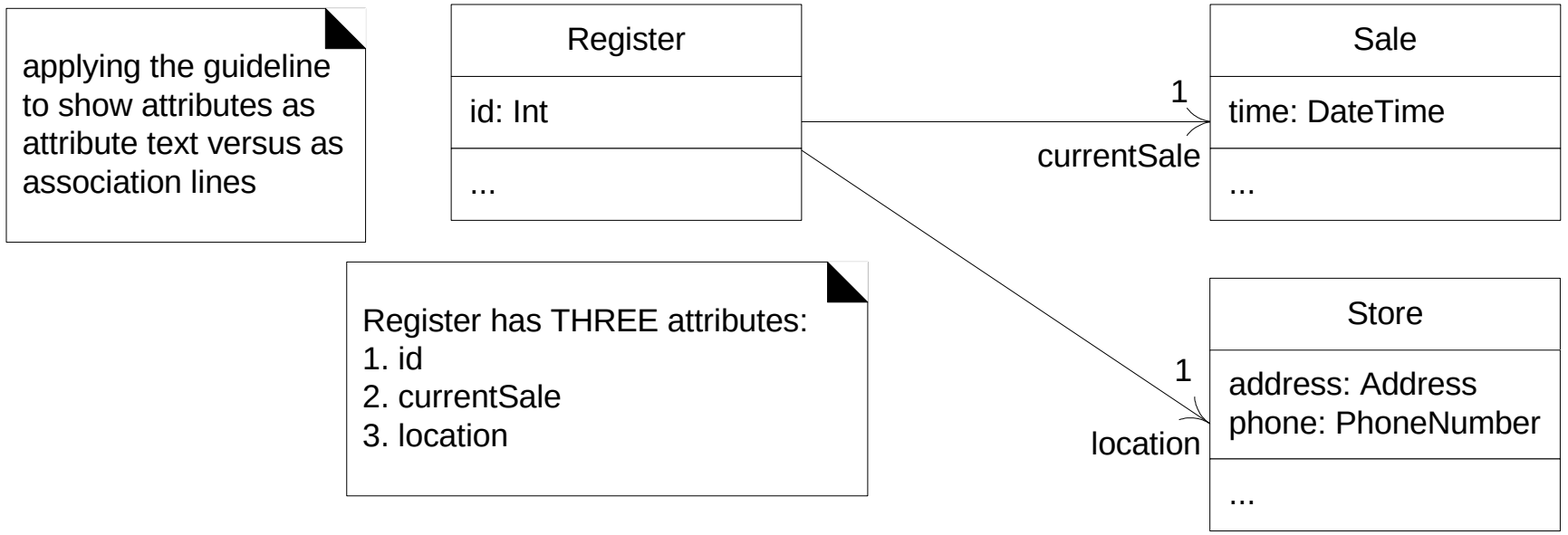
We explored the idea of data type objects earlier (Domain Models)

- Data types referred to objects for which individual identity is not important
- Recall a *Person* object versus a *Name* data type

One guideline is to use the text for data types (basically primitive types) and associations for more complicated classes

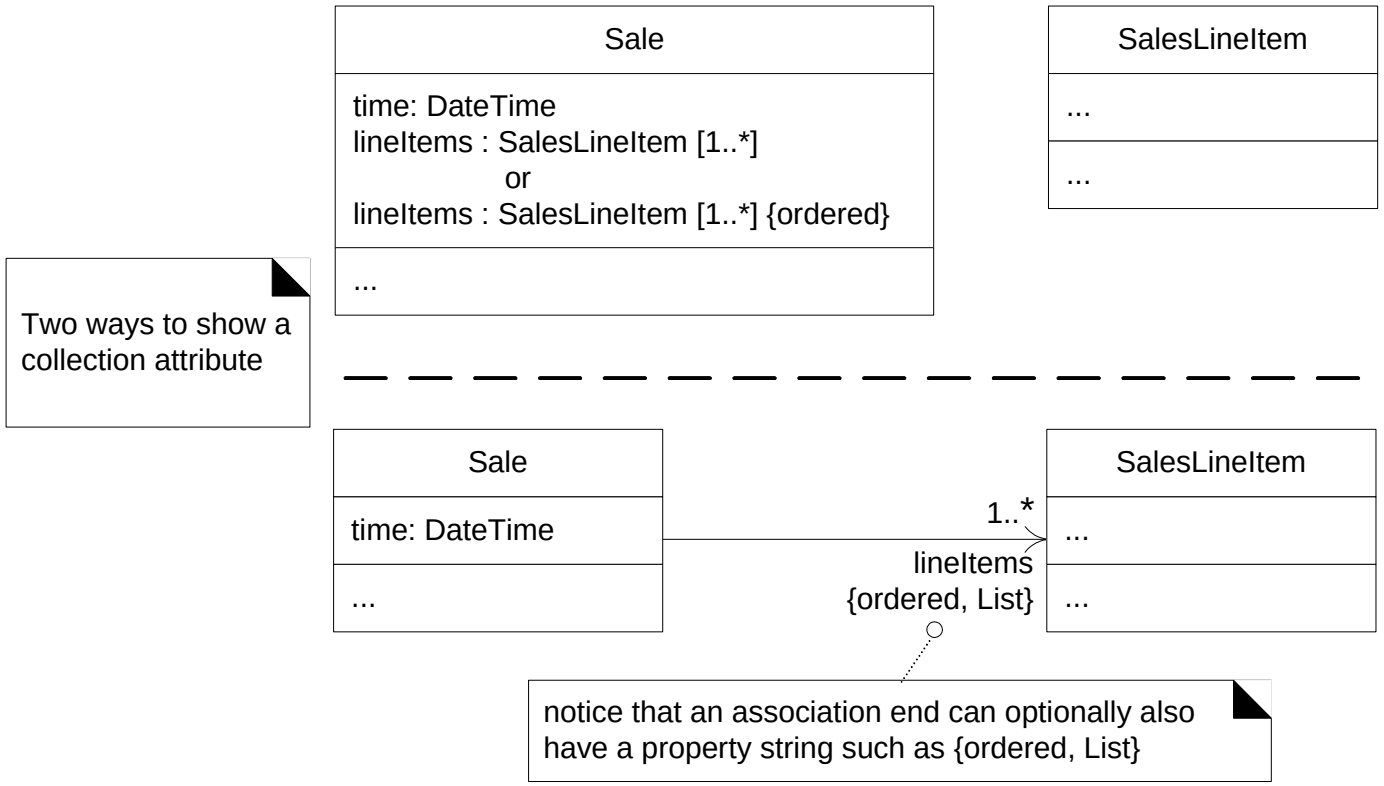
Note that this is a diagram preference – does not matter in the final code

UML: Attributes as Associations



Attributes: Lists

How do we notate a list of attributes, e.g. an ArrayList in Java?



Operations and Methods

Operations are usually displayed in the class box with the notation:

visibility name (parameter-list) {property-string}

Sometime a *return-type* is value is added

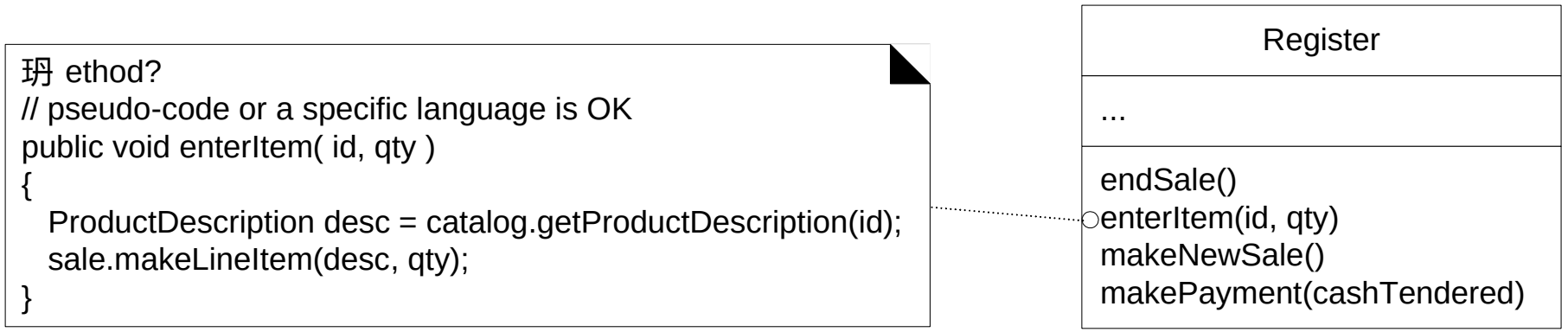
Assume public if no visibility is shown

An *operation* is a declaration (name, parameters, return type, exceptions list)

A *method* is an implementation of an operation

- in sequence diagrams, may show the details and sequence of messages
- In class diagram, usually include some pseudo-code in a note with the <<method>> tag

Method Notation in UML



Often times constructors (if included) are notated with the <<constructor>> tag

Usually, getters and setters are ignored in class diagrams

They are assumed to exist, or are added to the code on an as-needed basis

Keywords

Keywords are textual adornments used to categorize a model element – they provide some additional information about the element.

Usually notated <<*keyword*>>, and sometimes {*keyword*}

Some examples:

<<actor>> - this entity is an actor

<<interface>> - this entity is an interface

{abstract} – this is an abstract element, it can't be instantiated

{ordered} – this set of objects is ordered, e.g. the ArrayList example shown earlier

Abstract Classes

As we saw earlier in Domain Models, UML has the ability to denote *generalization*

- Solid line with open, fat arrow; can also notate {*abstract*} in super-class
- It represents a relationship between more general classifier and more specific classifier. The specific classifier indirectly has the features of the more general classifier

Dependency in UML

In UML, dependency lines can be used in any diagram, but they are especially common in class and package diagrams.

In UML, a general *dependency relationship* indicates that a *client* element (class, package, use case, etc.) has knowledge of a *supplier* element and that a change in the supplier could affect the client

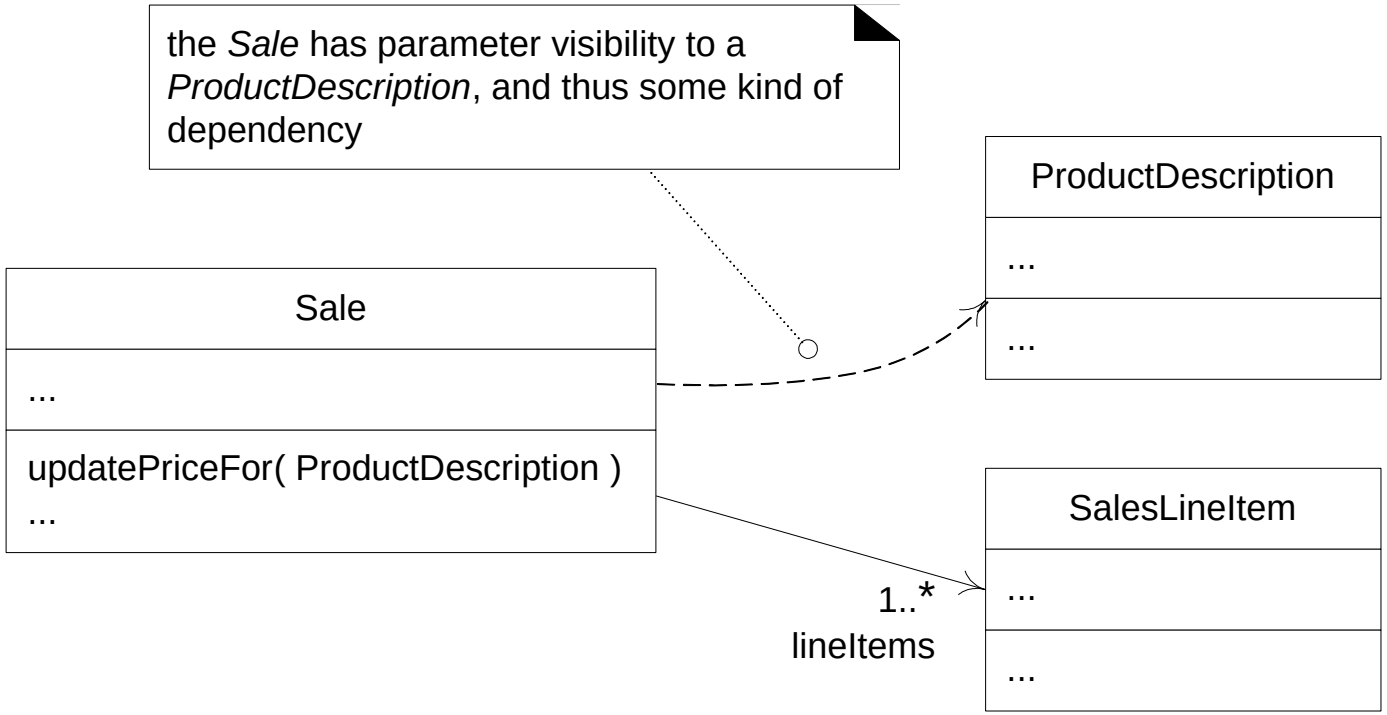
Indicated by a dashed arrow from the *client to the supplier*

Note that we often associate elements with associations (e.g. super- and sub-classes as we just saw), so we do not need to add dependency arrows if an association already exists

Often used when a class has an attribute of another class type, or if one class sends a message to another class

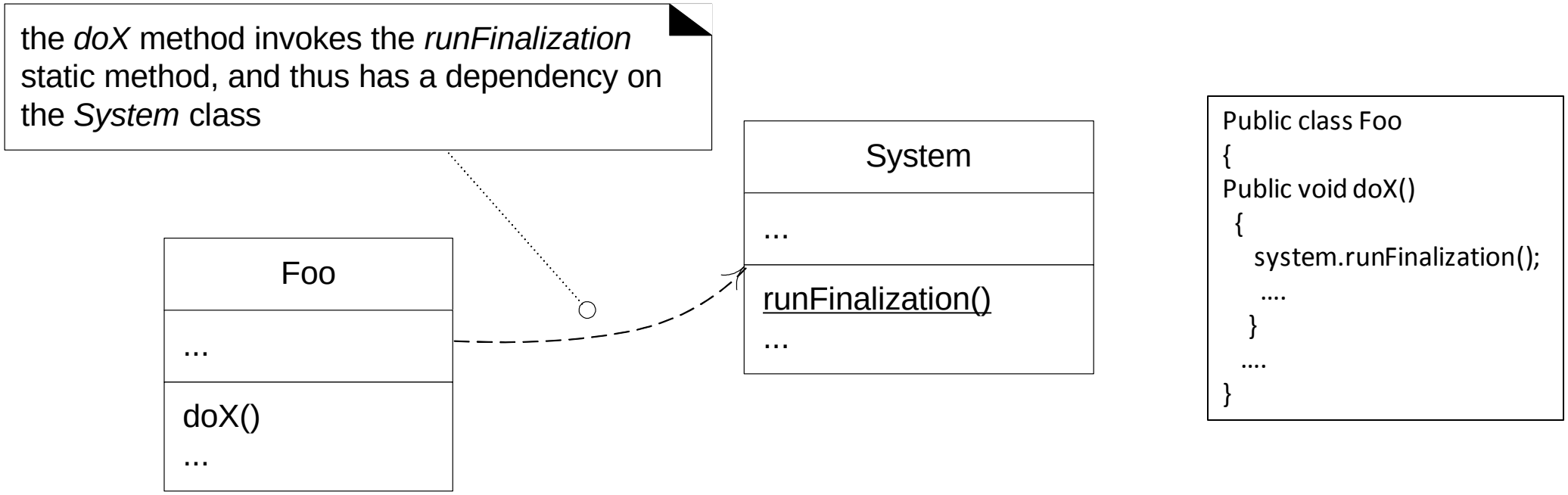
Dependency in UML

Guideline: Use dependency in UML to depict global parameter variable, local variable, and static-method call to another class.

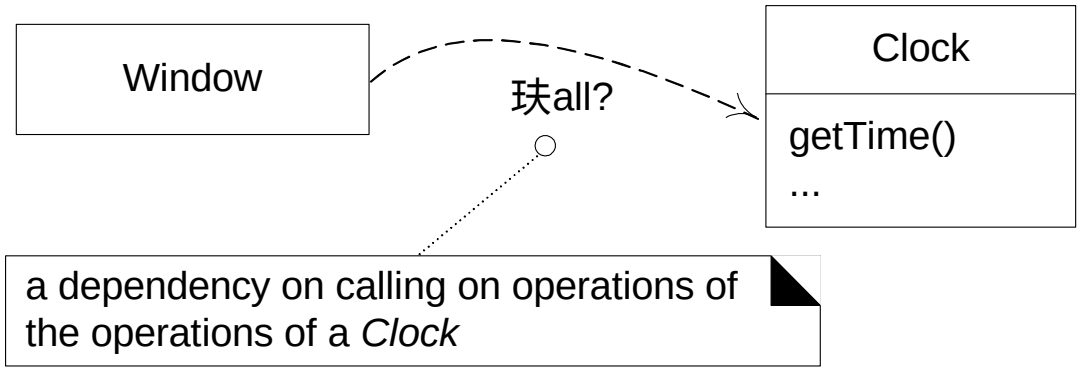


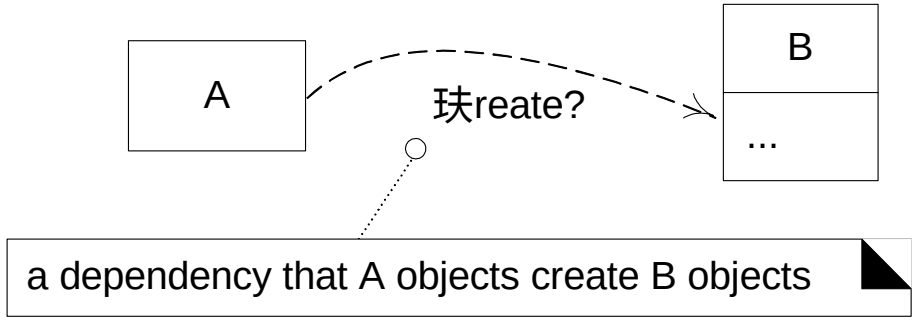

```
Public class Sale
{
Public void updatePrice(ProductDescription description)
 {
    Money basePrice = description.getPrice();
     ....
  }
 ....
}
```

Dependency in UML



Dependency Labels





Composition and Aggregation

We saw this earlier in Domain Models …

Composition is a whole-part relationship between model entities, such that

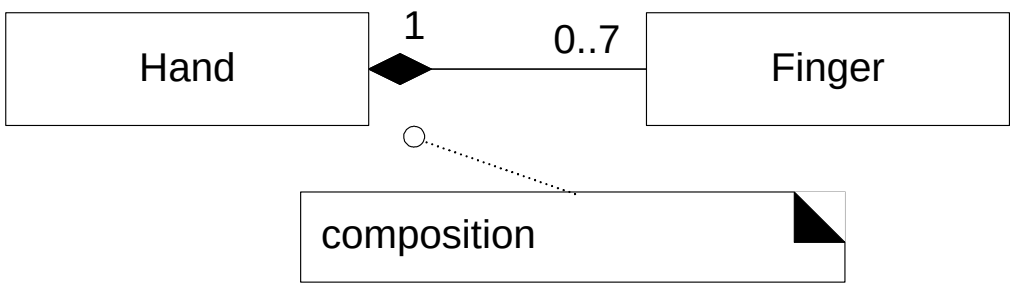
- an instance of the part belongs to only one instance of the composite
- a part must belong to a composite
- the composite is responsible for creating/deleting the parts. (So if the composite is destroyed, the parts are destroyed or become attached to another composite.)

Aggregation is a weaker form of composition, where the above requirements are not necessarily true

- Aggregation does not imply ownership of the parts

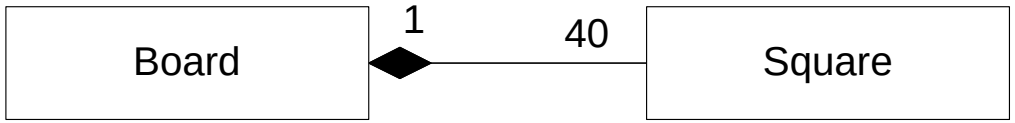
Composition involves instantiating objects, aggregation involves pointers to other objects

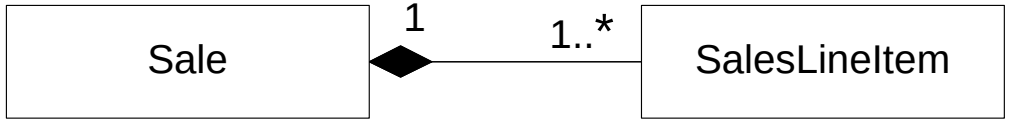
Composition: Example



composition means
-a part instance (*Square*) can only be part of one composite (*Board*) at a time

-the composite has sole responsibility for management of its parts, especially creation and deletion



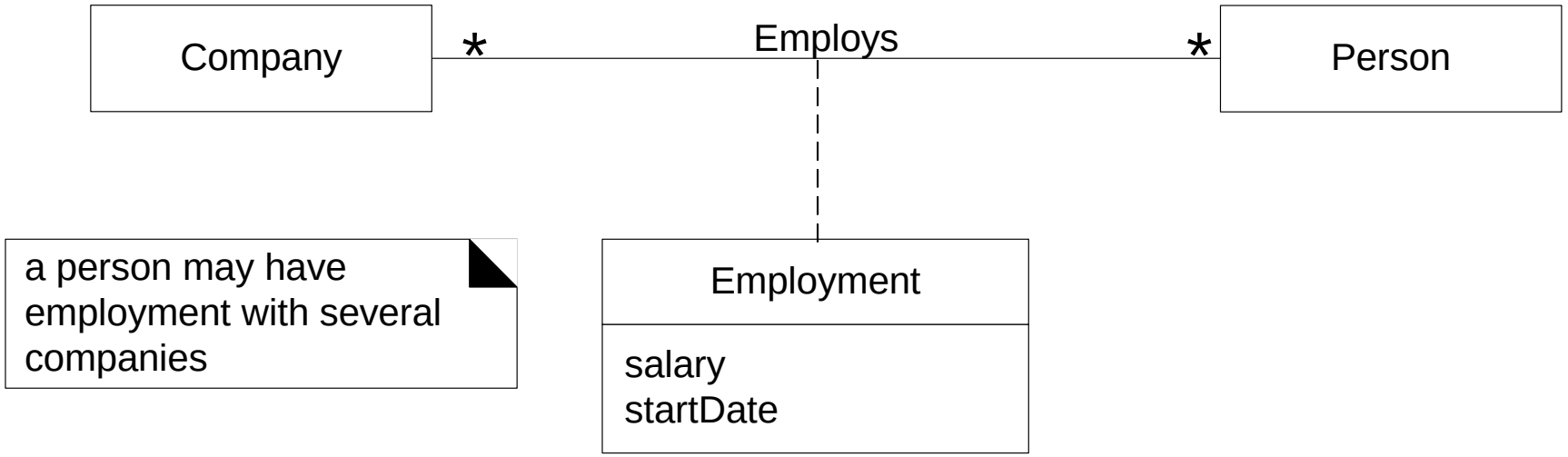


Generally look for “has a” associations

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

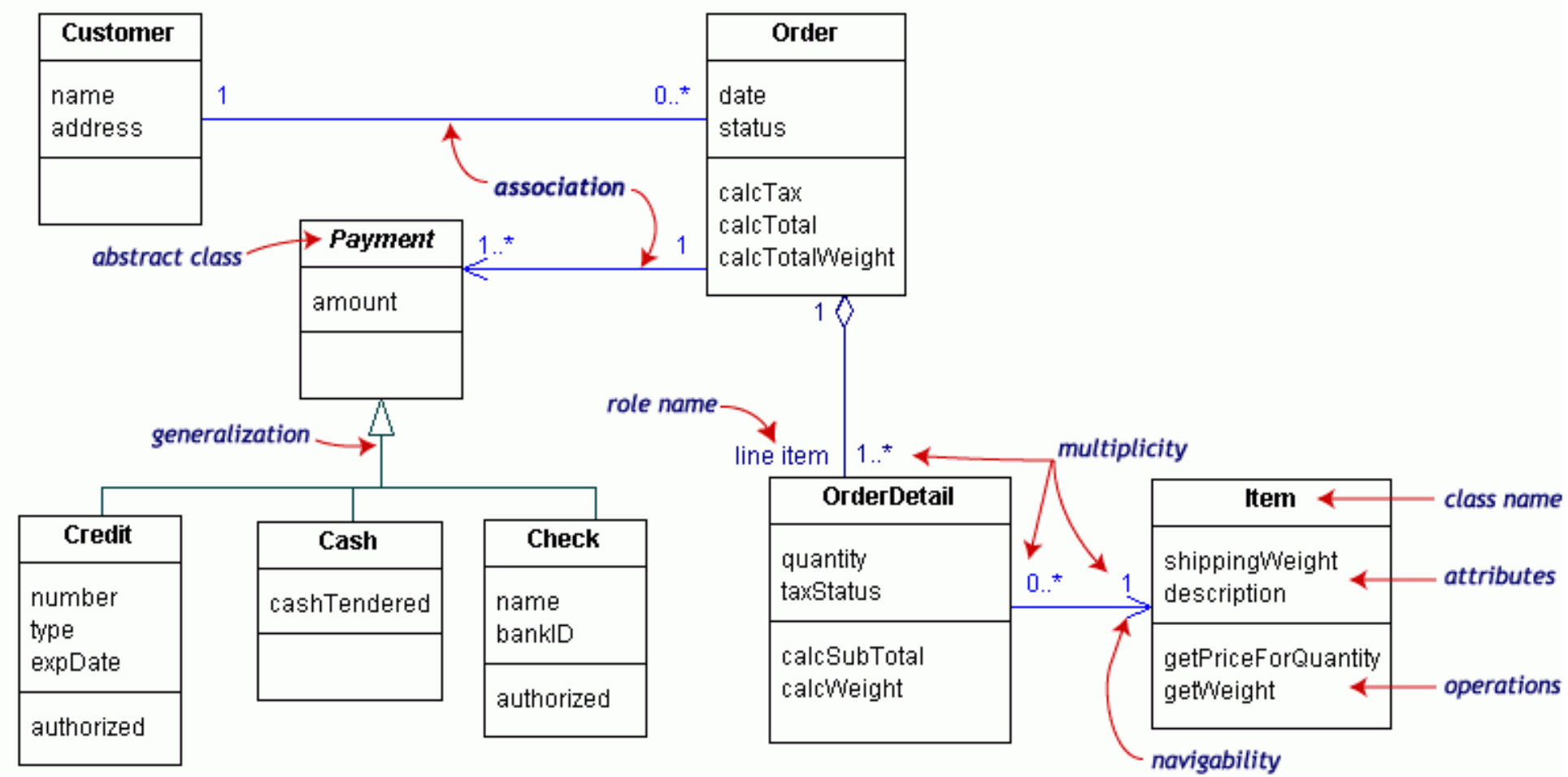


```
class Company {
    Set<Employment> employments;
}
```

providing a method in Company that returns all its personnels

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

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


Customer
name
address
Order
date
status
calcTax
calcTotal
calcTotalWeight
1
0..*
association
abstract class
Payment
amount
1..*
1
1
role name
line item
1..*
multiplicity
generalization
Credit
number
type
expDate
authorized
Cash
cashTendered
Check
name
bankID
authorized
OrderDetail
quantity
taxStatus
calcSubTotal
calcWeight
0..*
1
Item
shippingWeight
description
getPriceForQuantity
getWeight
class name
attributes
operations
navigability

UML interaction diagrams

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

Sequence diagrams are more detailed than communication diagrams

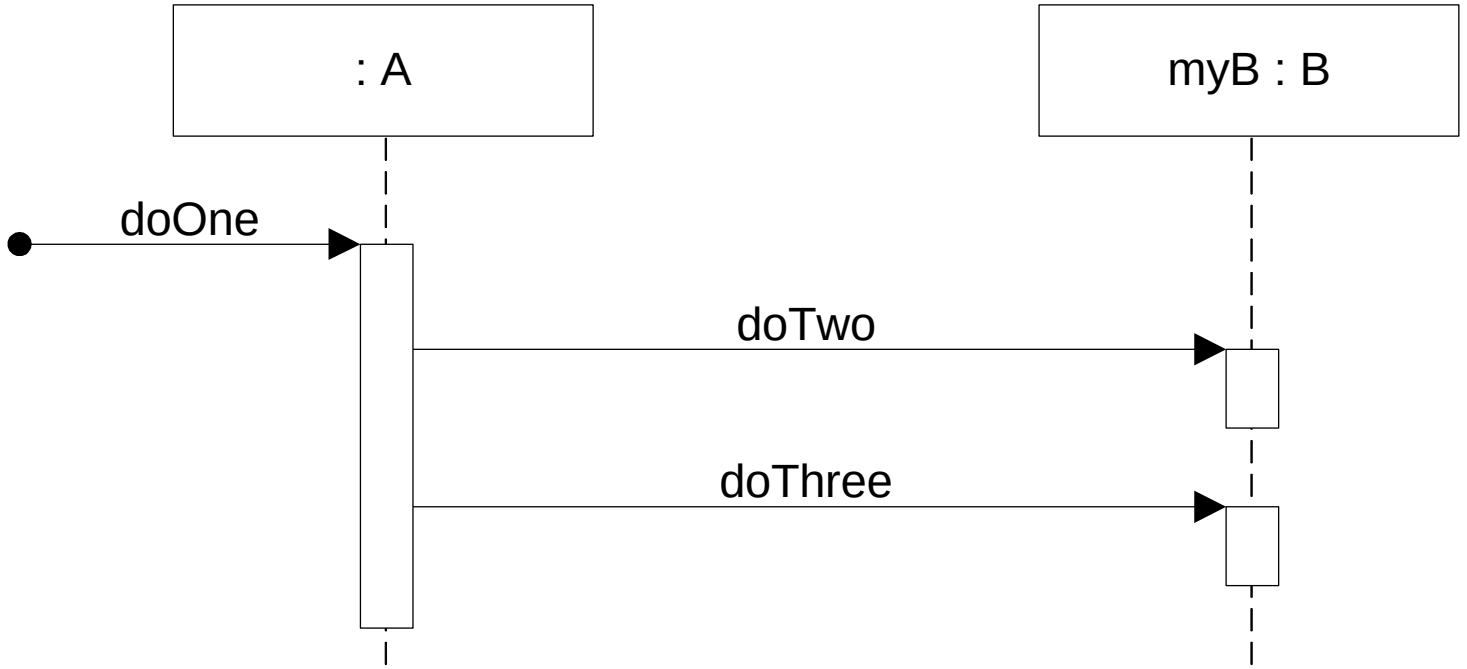
They often represent a series of method calls between objects in a system

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

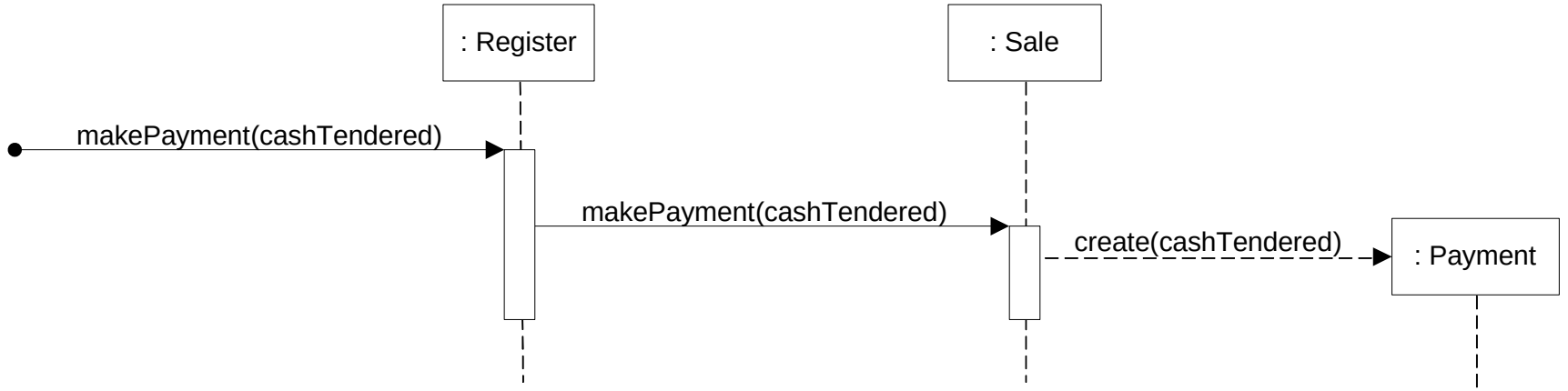
Example: Sequence Diagram




```
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

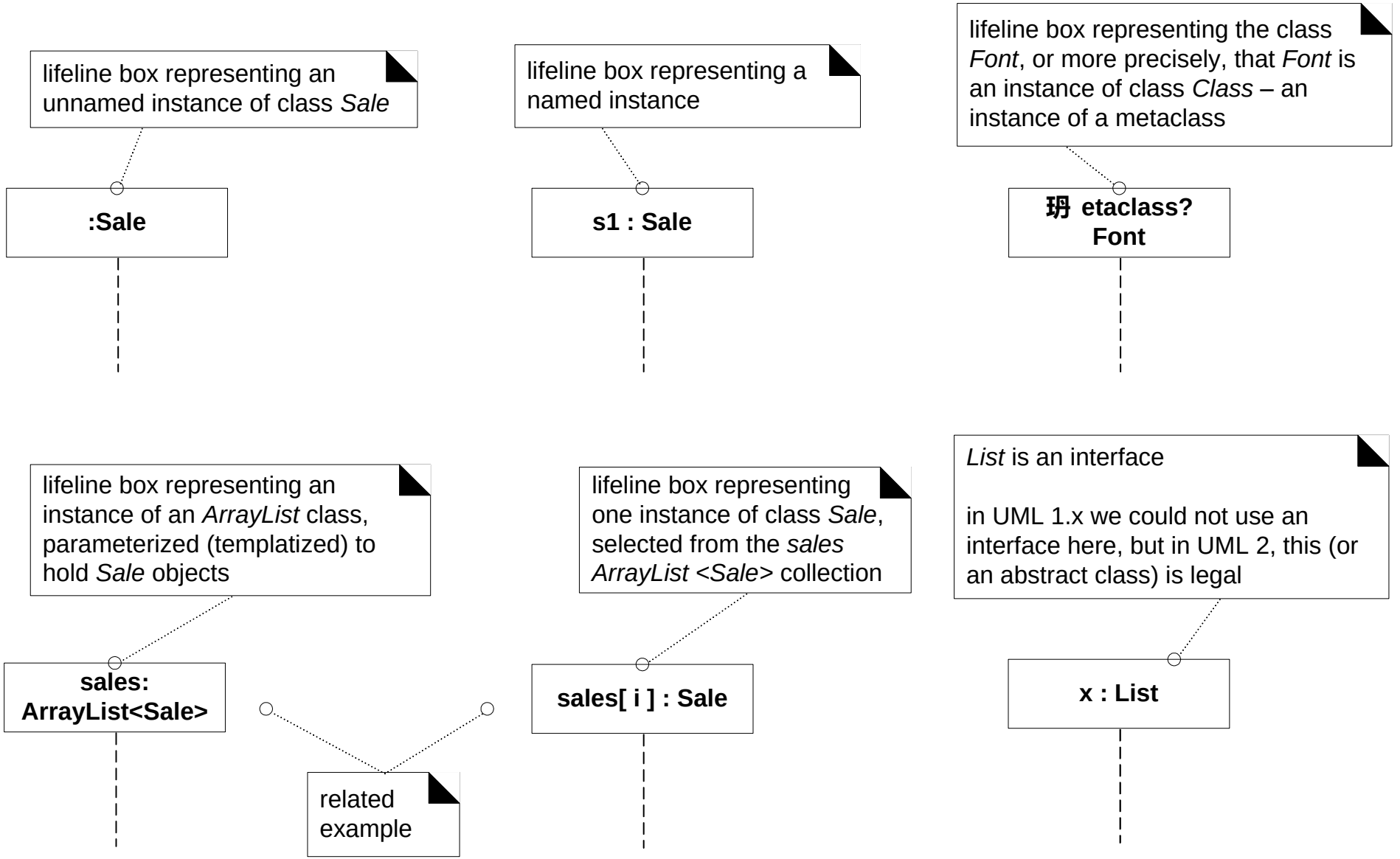
Basic notation for the entities that make up the sequence diagram – they are called *lifeline* boxes and represent the *participants* in the particular sequence being modeled

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*
:Sale
lifeline box representing a named instance
s1 : Sale
lifeline box representing the class *Font*, or more precisely, that *Font* is an instance of class *Class* – an instance of a metaclass
珼 etaclass?
Font
lifeline box representing an instance of an *ArrayList* class, parameterized (templatized) to hold *Sale* objects
sales:
ArrayList<Sale>
related example
lifeline box representing one instance of class *Sale*, selected from the *sales ArrayList* <*Sale*> collection
sales[i] : Sale
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
x : List

Sequence Diagrams: Messages

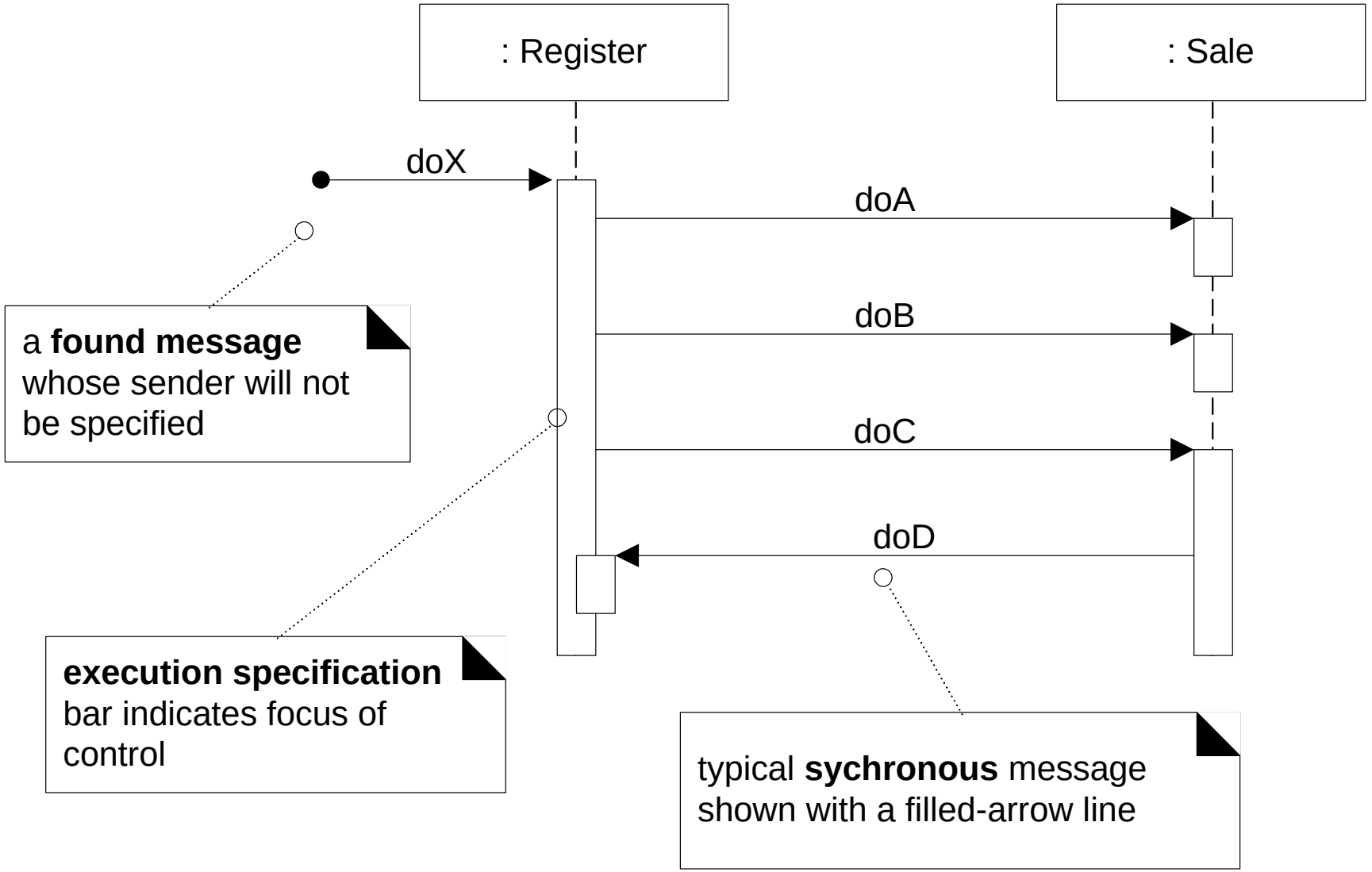
Messages are notated as solid arrows with filled in arrowheads between lifelines

- 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.


: Register
: Sale
doX
doA
doB
doC
doD
a **found message** whose sender will not be specified
execution specification bar indicates focus of control
typical **sychronous** message shown with a filled-arrow line

Sequence Diagrams: Specifics

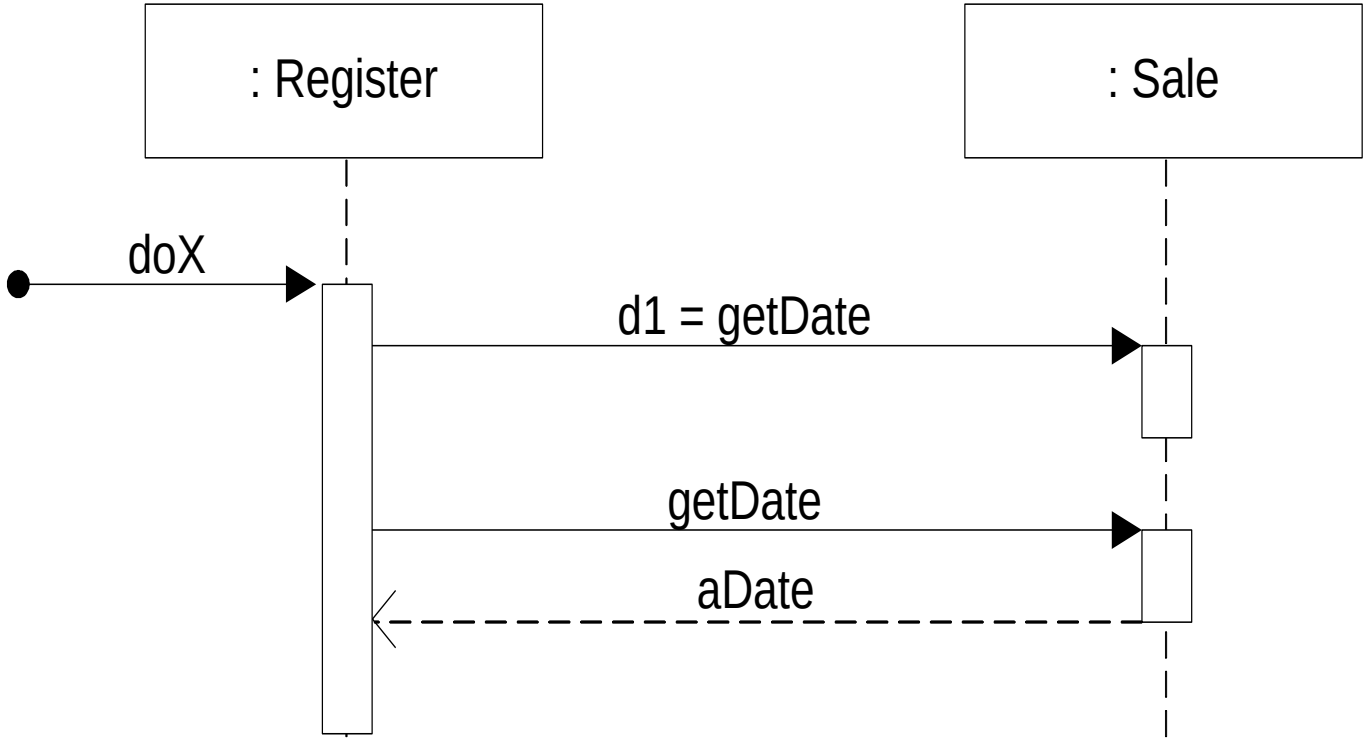
The *execution specification bar* or *activation bar* indicates that the operation is on the call stack

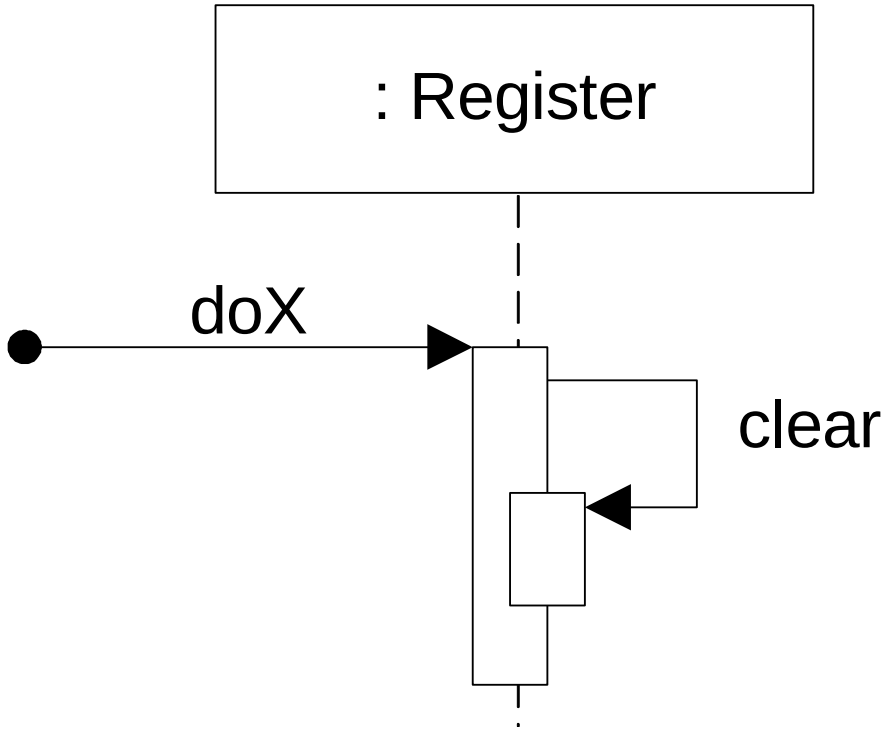
Usually replies to messages are indicated with a value or a dotted line (see next slide)

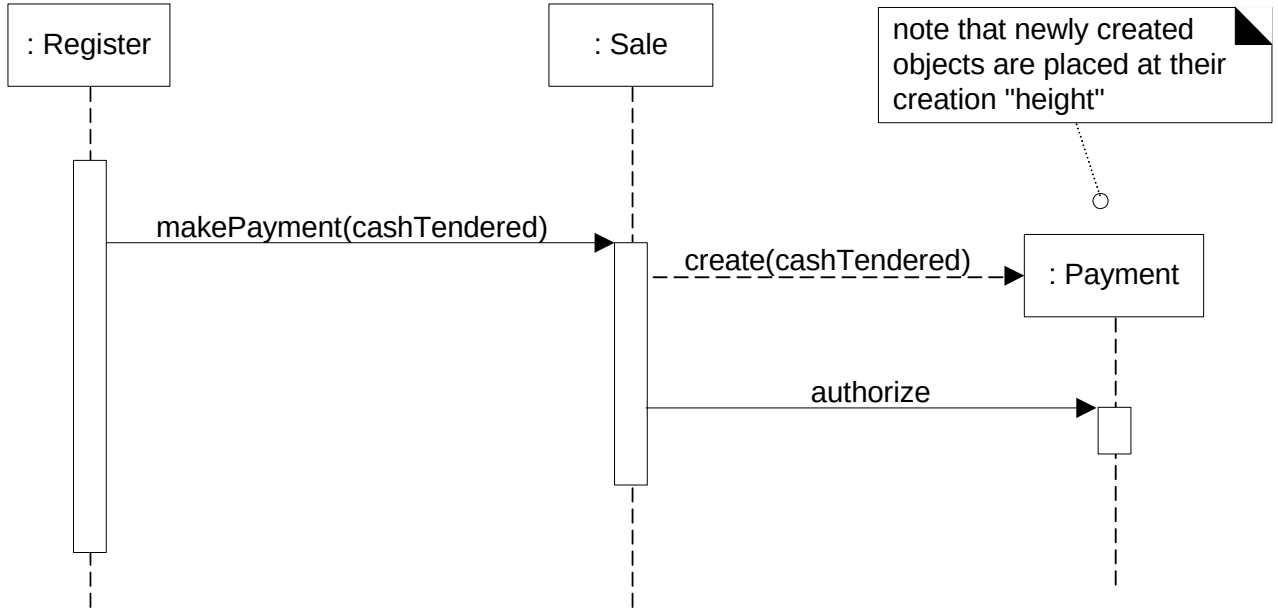
It is possible to have a message to “self” (or “this”)

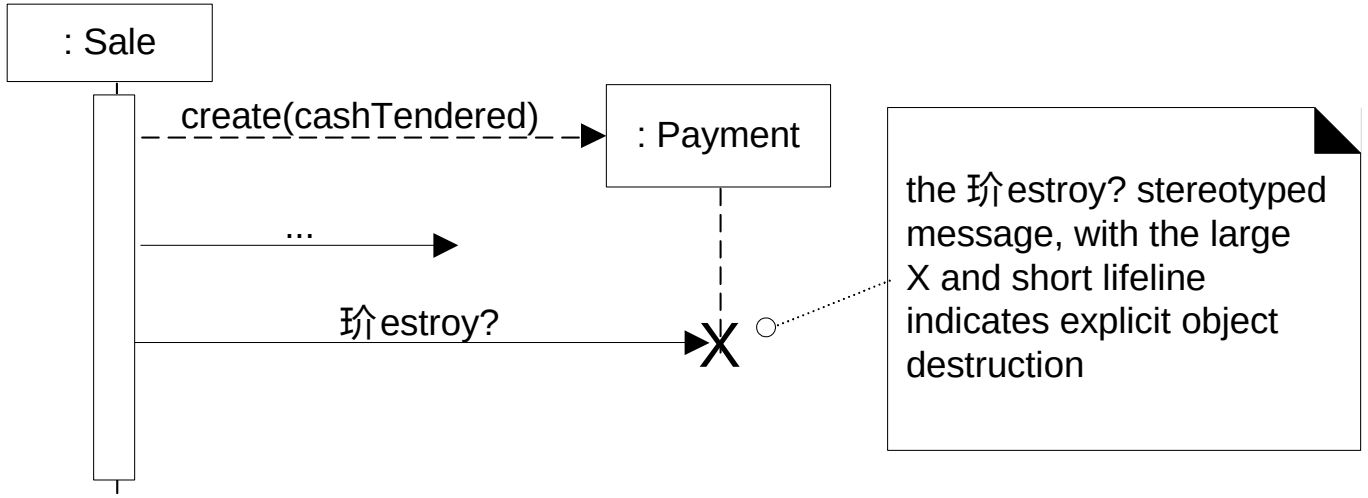
Sequence diagrams can also indicate instance creation (see later slide)

Likewise, instances can be destroyed (indicated by “X” at the end of lifeline)


: Register
: Sale
doX
d1 = getDate
getDate
aDate


: Register
doX
clear


: Register
: Sale
note that newly created objects are placed at their creation "height"
makePayment(cashTendered)
create(cashTendered)
: Payment
authorize

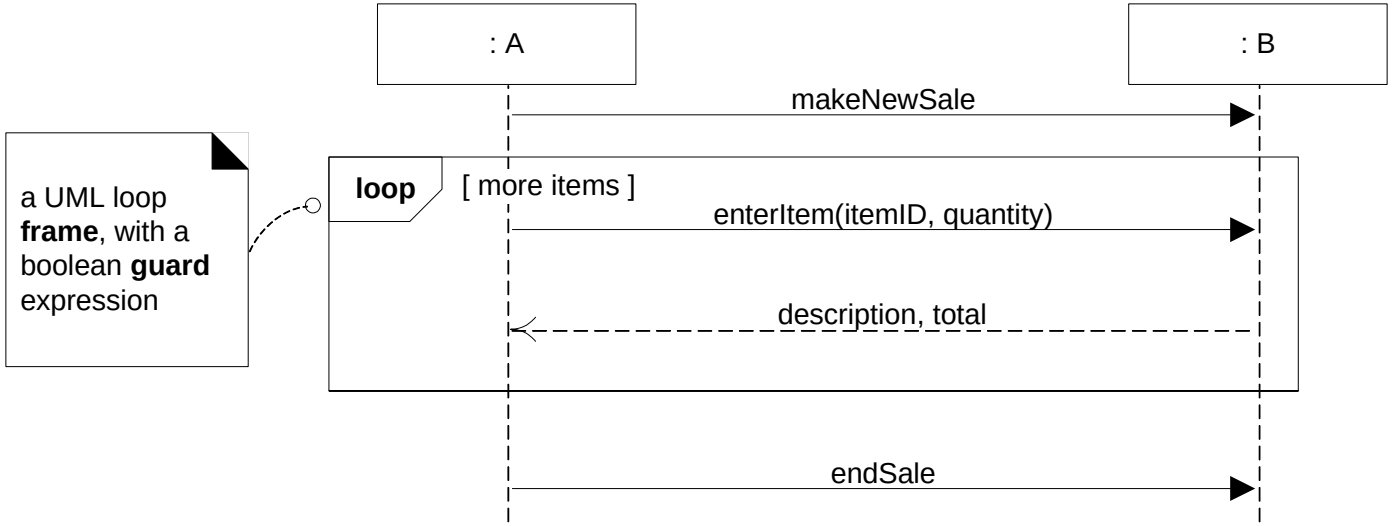

: Sale
create(cashTendered)
: Payment
...
玠estroy?
the 玠estroy? stereotyped message, with the large X and short lifeline indicates explicit object destruction

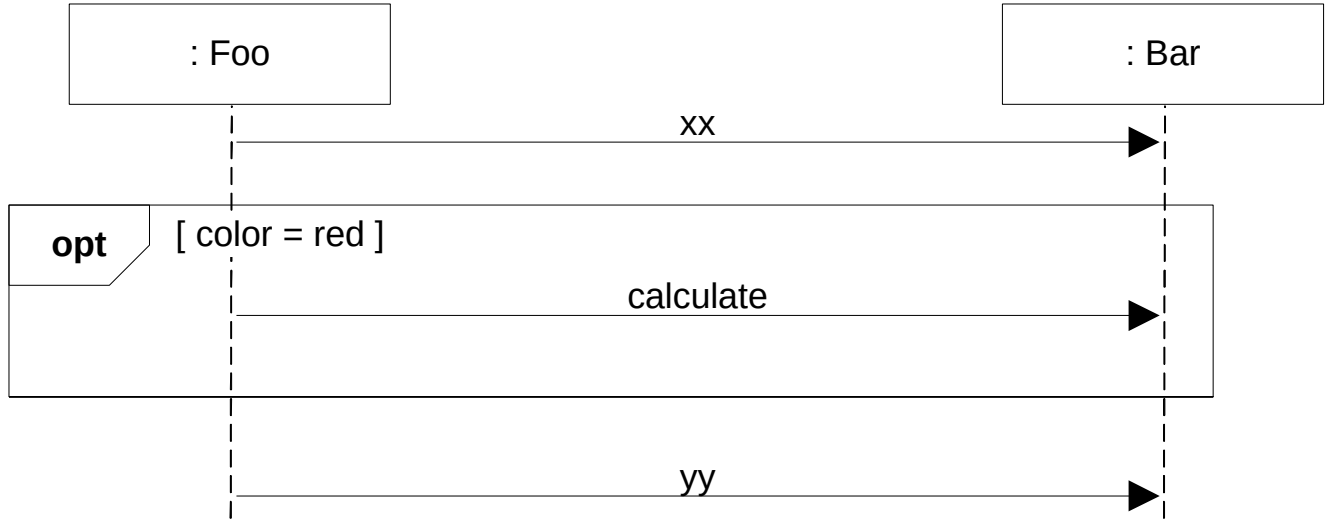
Sequence Diagrams: Specifics

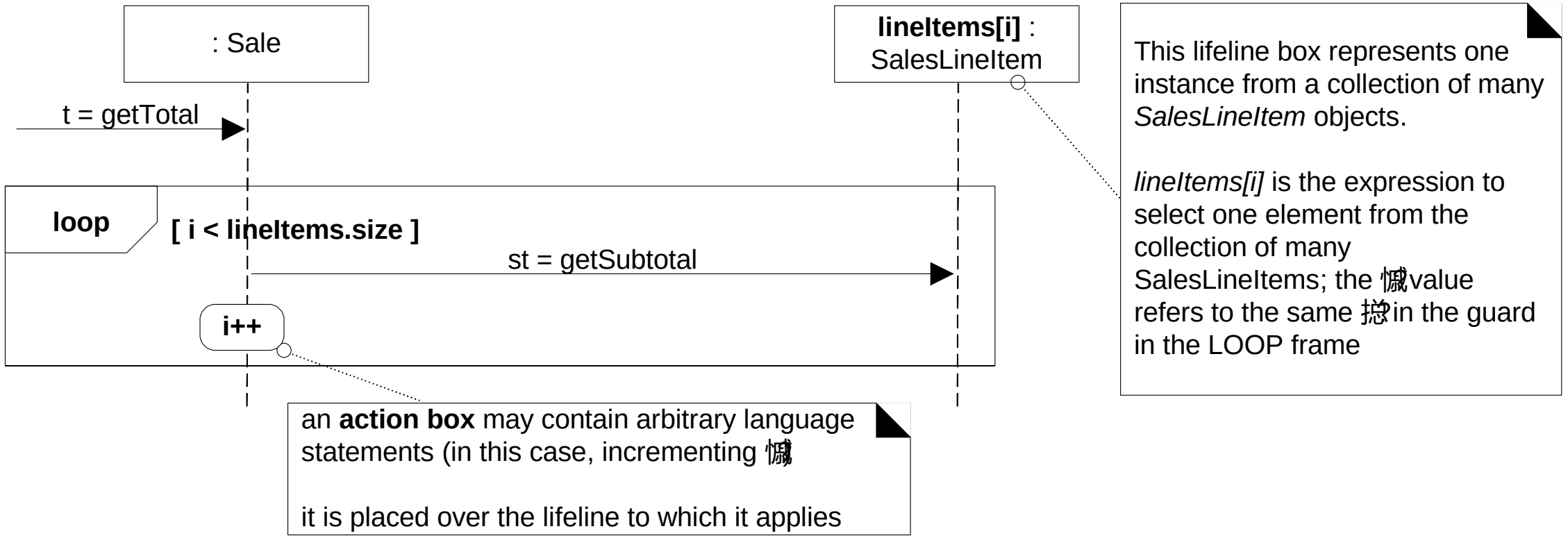
Diagram frames may be used in sequence diagrams to show:

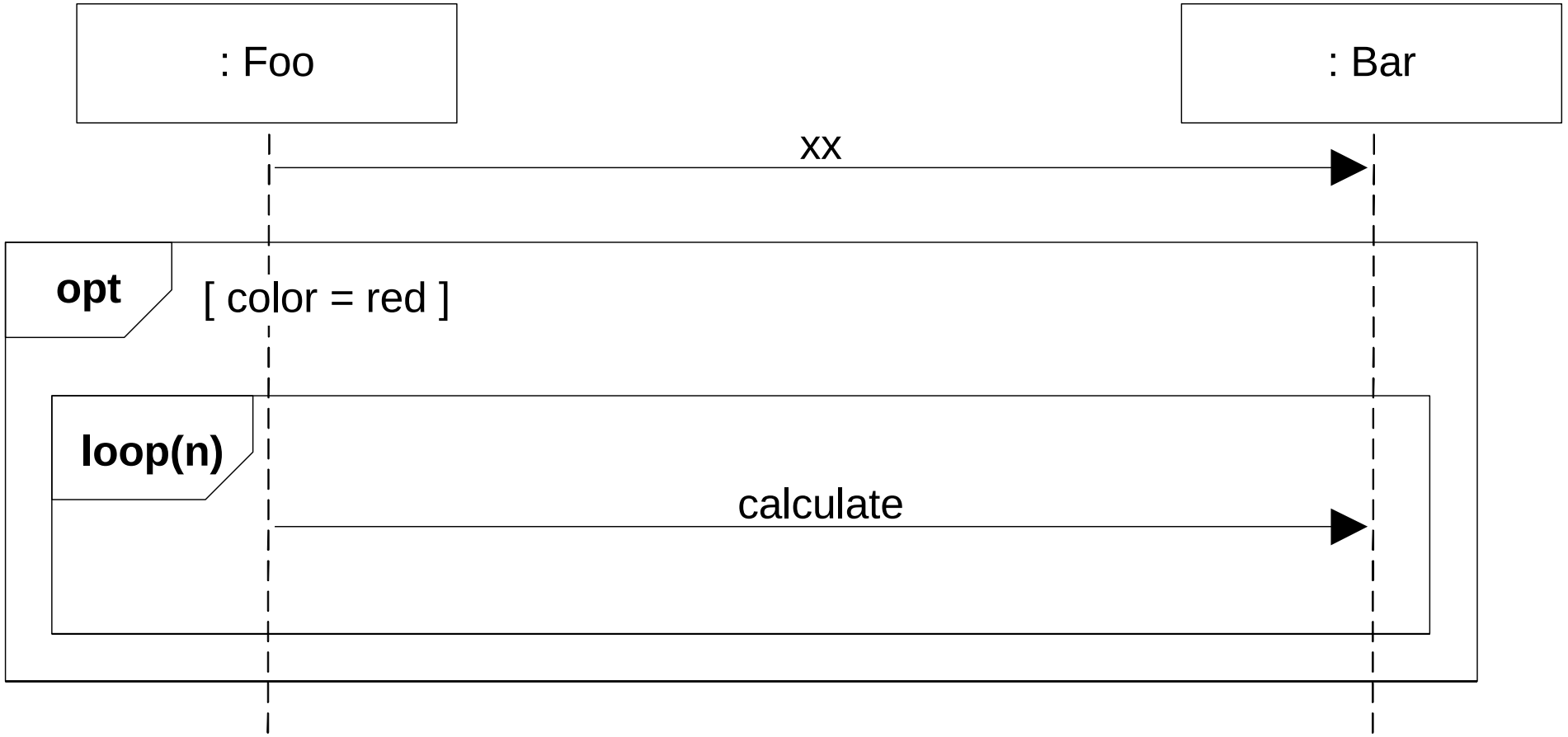
- Loops
- Conditional (optional) messages
- Nesting (a conditional loop)
- Relationships between diagrams

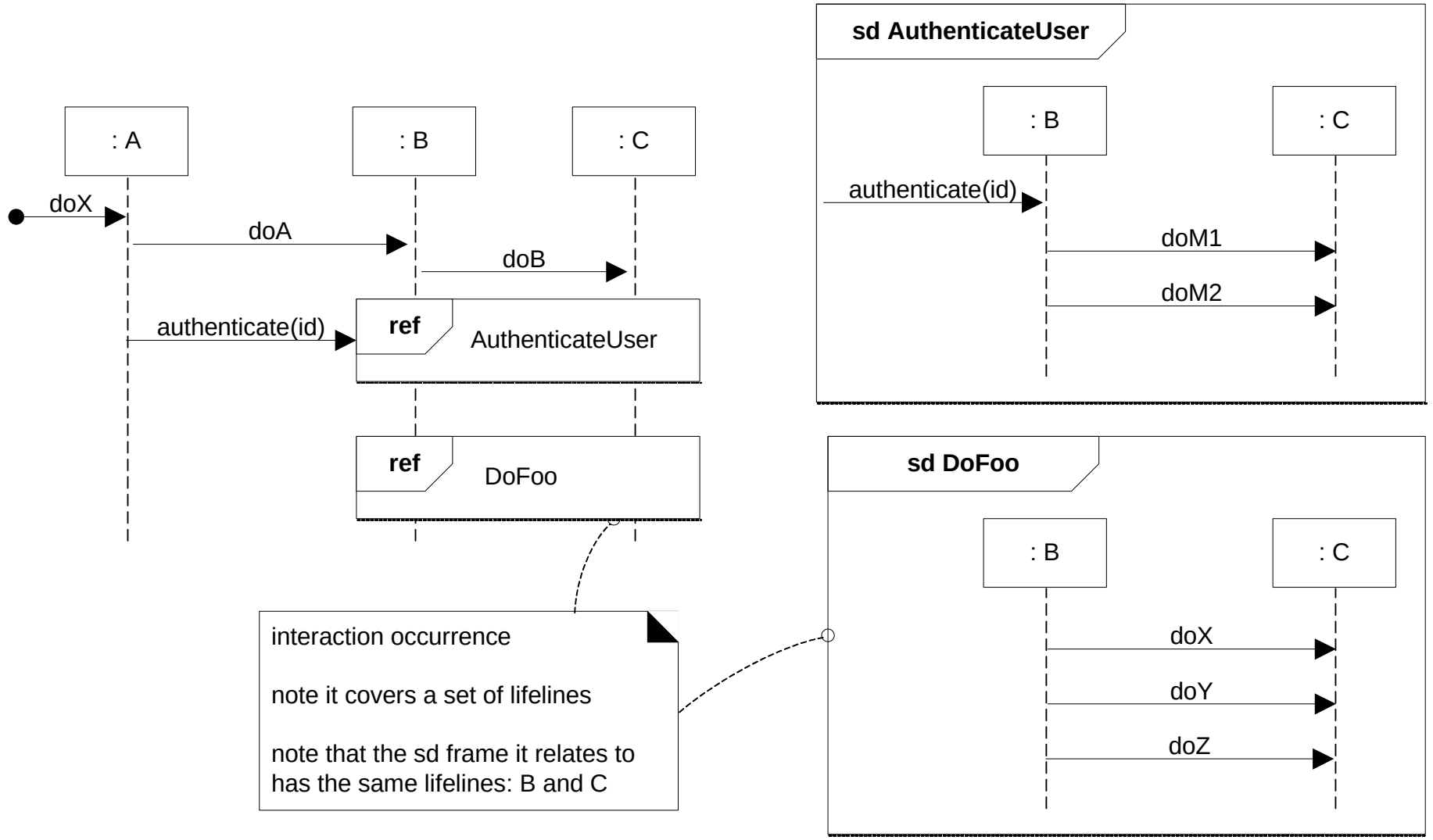
See next slides for examples

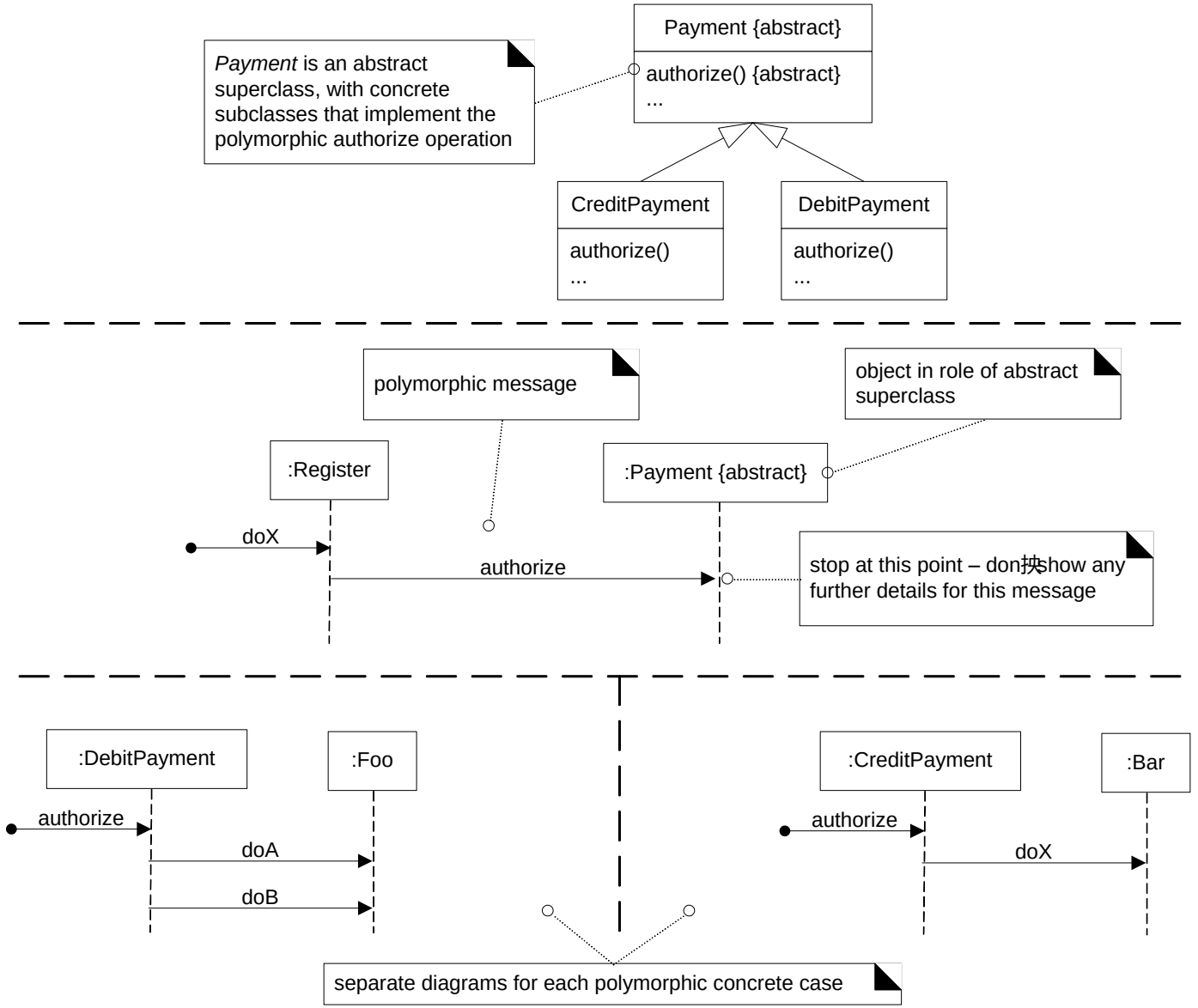

: A
: B
makeNewSale
a UML loop **frame**, with a boolean **guard** expression
loop
[more items]
enterItem(itemID, quantity)
description, total
endSale


: Foo
: Bar
xx
opt
[color = red]
calculate
yy


: Sale
lineItems[i] : SalesLineItem
t = getTotal
loop
[i < lineItems.size]
st = getSubtotal
i++
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 慼value refers to the same 揑in the guard in the LOOP frame
an action box may contain arbitrary language statements (in this case, incrementing 慼
it is placed over the lifeline to which it applies


: Foo
: Bar
xx
opt
[color = red]
loop(n)
calculate


: A
: B
: C
doX
doA
doB
authenticate(id)
ref
AuthenticateUser
ref
DoFoo
interaction occurrence
note it covers a set of lifelines
note that the sd frame it relates to has the same lifelines: B and C
sd AuthenticateUser
: B
: C
authenticate(id)
doM1
doM2
sd DoFoo
: B
: C
doX
doY
doZ


Payment {abstract}
authorize() {abstract}
...
Payment is an abstract superclass, with concrete subclasses that implement the polymorphic authorize operation
CreditPayment
authorize()
...
DebitPayment
authorize()
...
polymorphic message
object in role of abstract superclass
:Register
:Payment {abstract}
doX
authorize
stop at this point – don抉show any further details for this message
:DebitPayment
:Foo
:CreditPayment
:Bar
authorize
doA
doB
authorize
doX
separate diagrams for each polymorphic concrete case

Example:
Library Information System (LIS)

LIS Requirements and Use cases

R1. The LIS must allow a patron to check out documents.

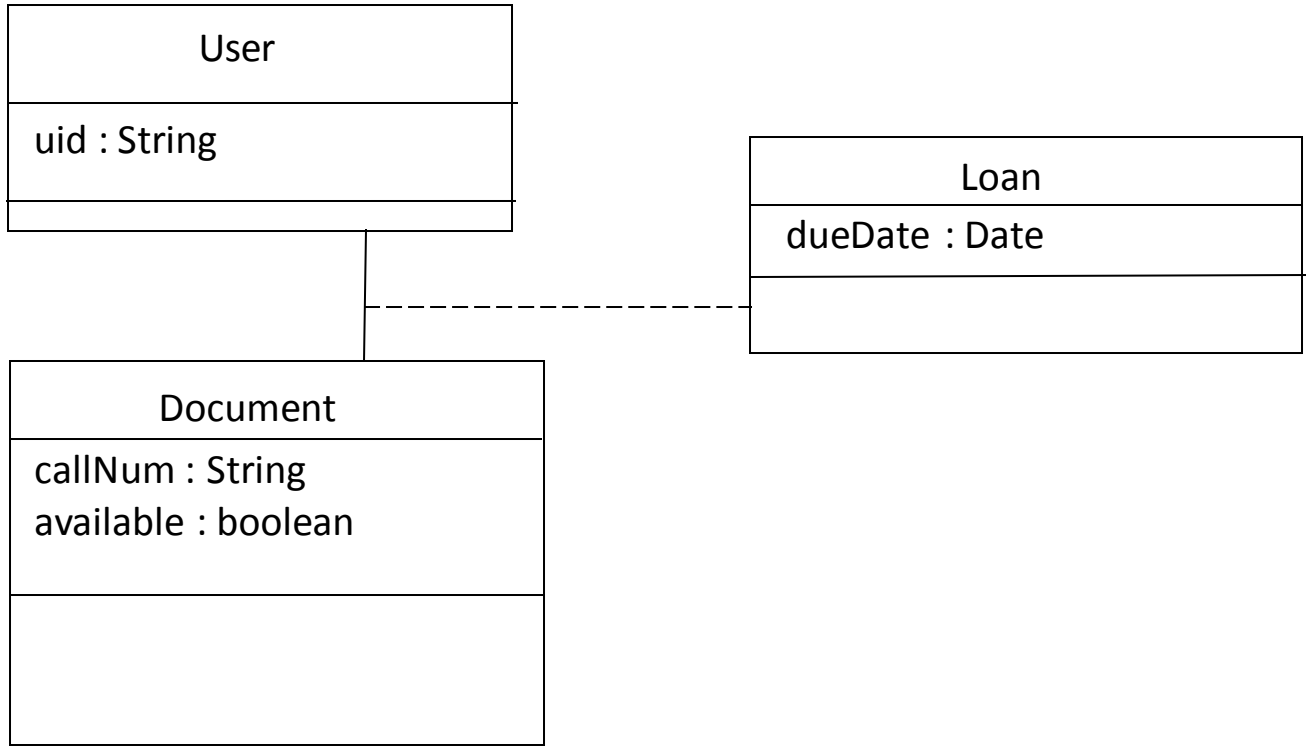
R2. The LIS must allow a patron to return documents.

UC1. Checkout Document (Actor: Patron, System: LIS)

UC2. Return Document (Actor : Patron, System: LIS)

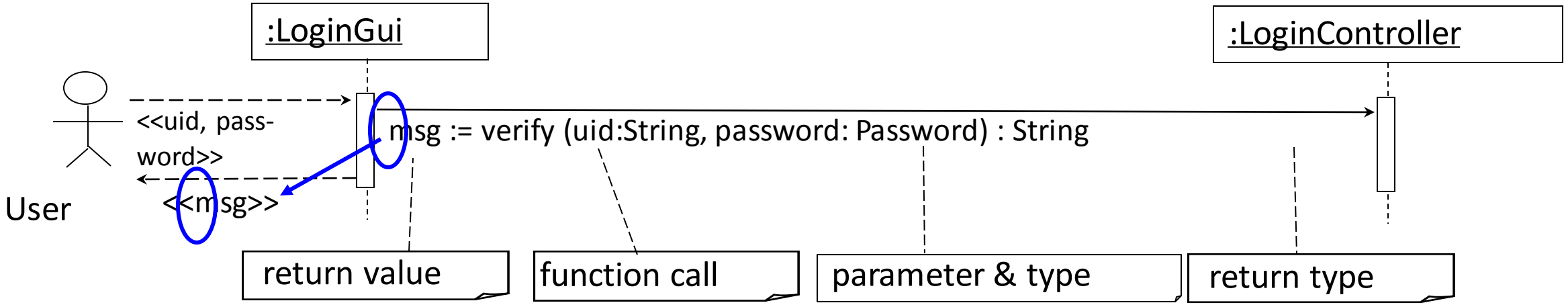
How about Allow a Patron? Is it a use case? Who is the actor? What is the goal or business task for the actor? Does it start and end with an actor?

Domain Model

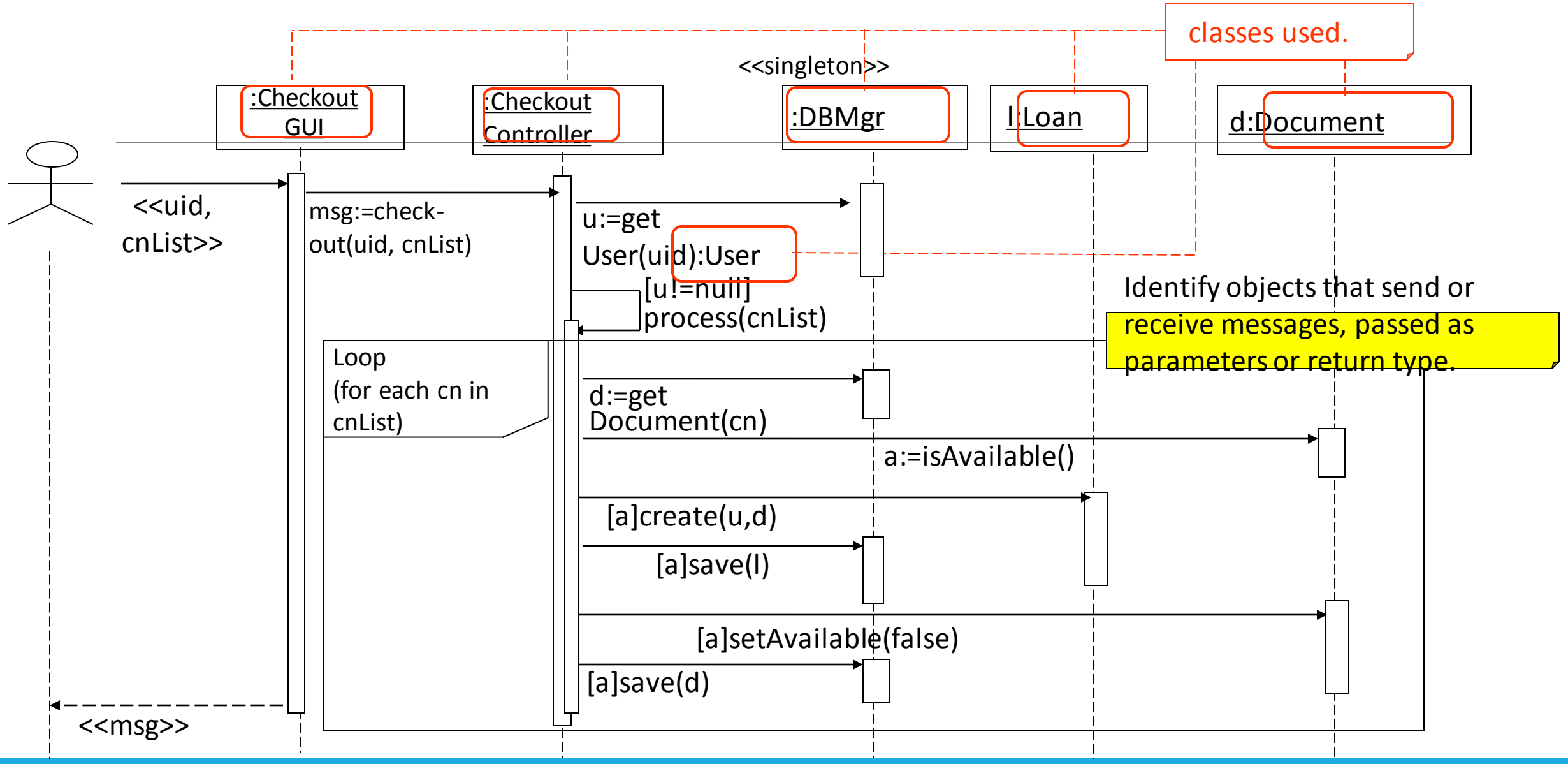


LIS UC.1Text

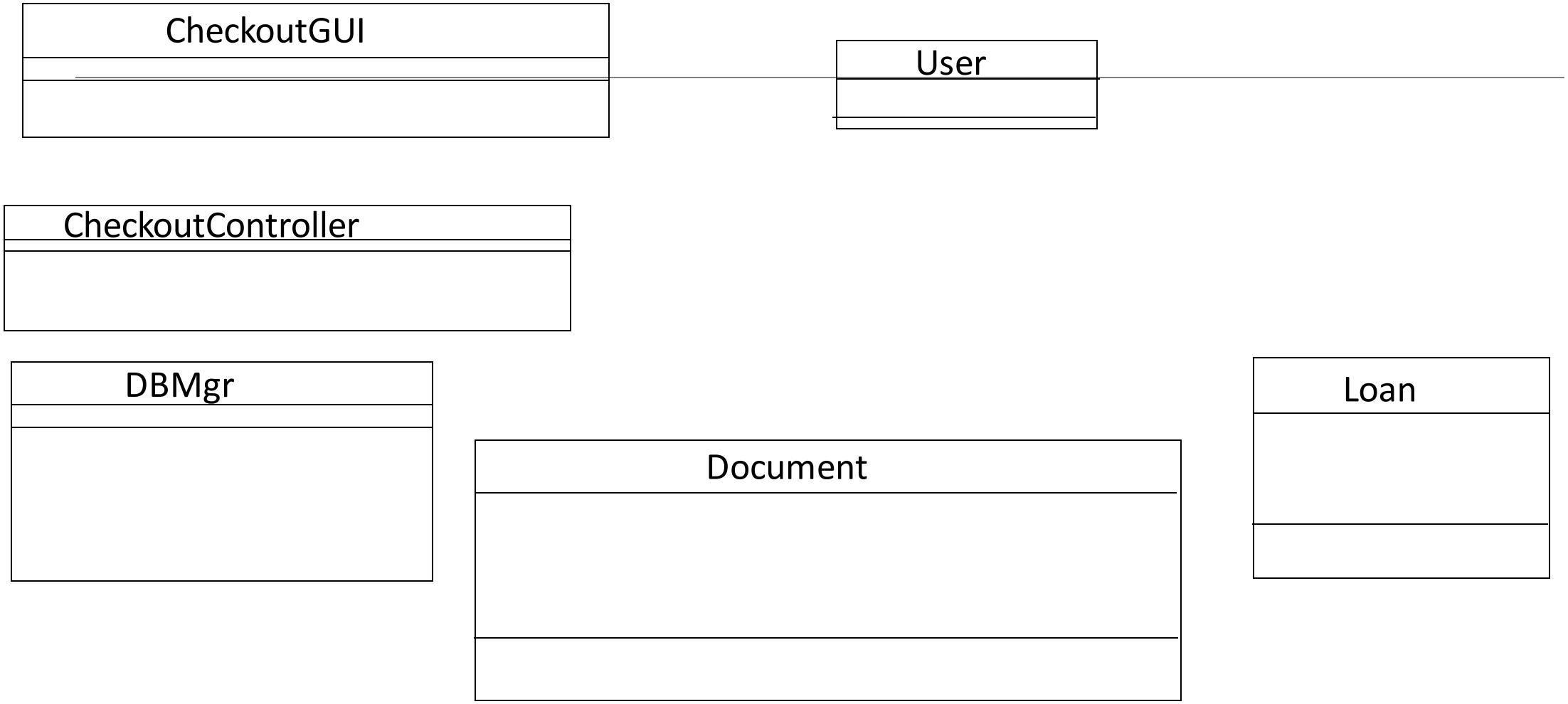
UC1 : Checkout Document	
Actor: Patron	System: LIS
	0. The LIS displays the main menu.
1. Patron clicks the checkout Document button on the main menu.	2.The system displays the checkout menu.
3. The Patron enters the call numbers of documents to be checked out and clicks the Submit button.	4. The system 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.	


:LoginGui
:LoginController
User
<<uid, pass-
word>>
<<msg>>
msg := verify (uid:String, password: Password) : String
return value
function call
parameter & type
return type

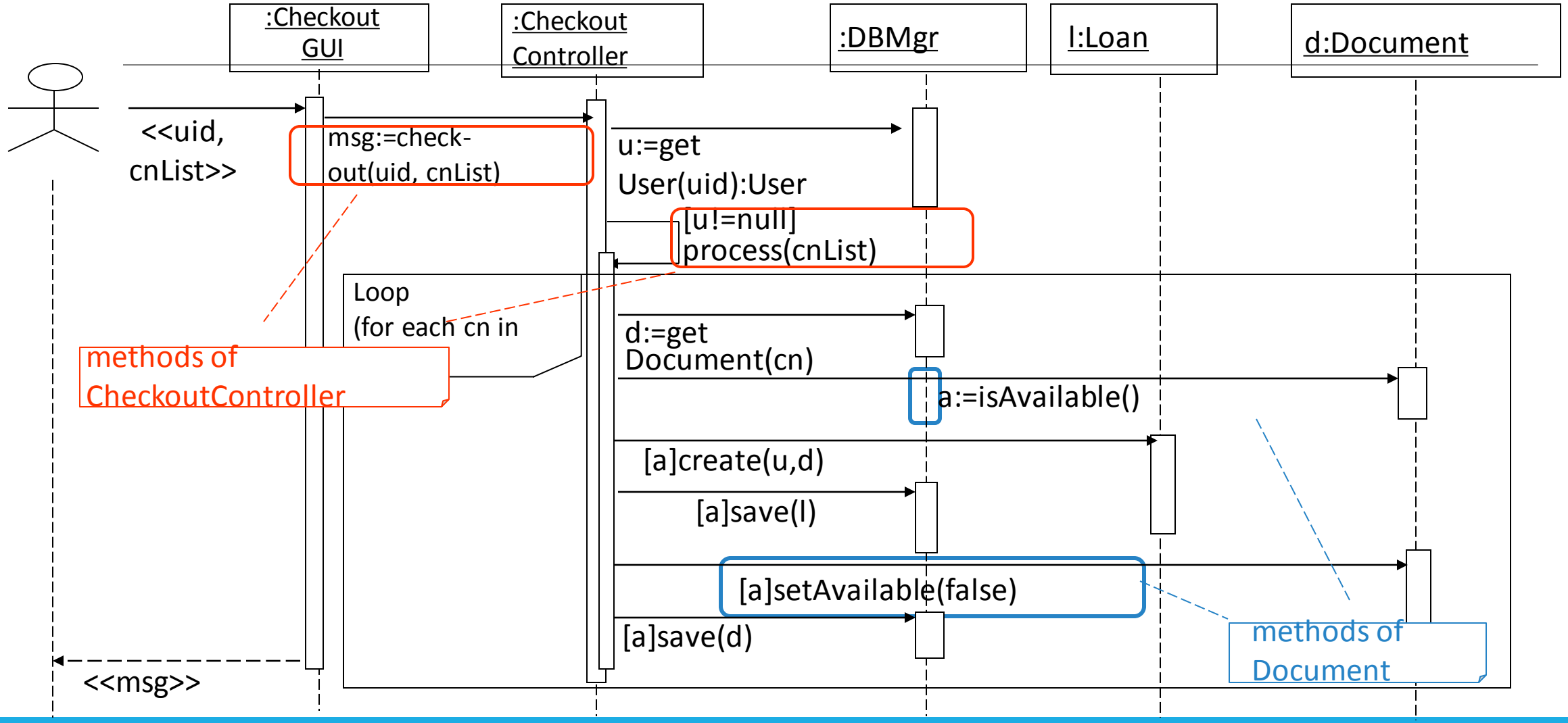
Identify Classes Used in Sequence Diagrams



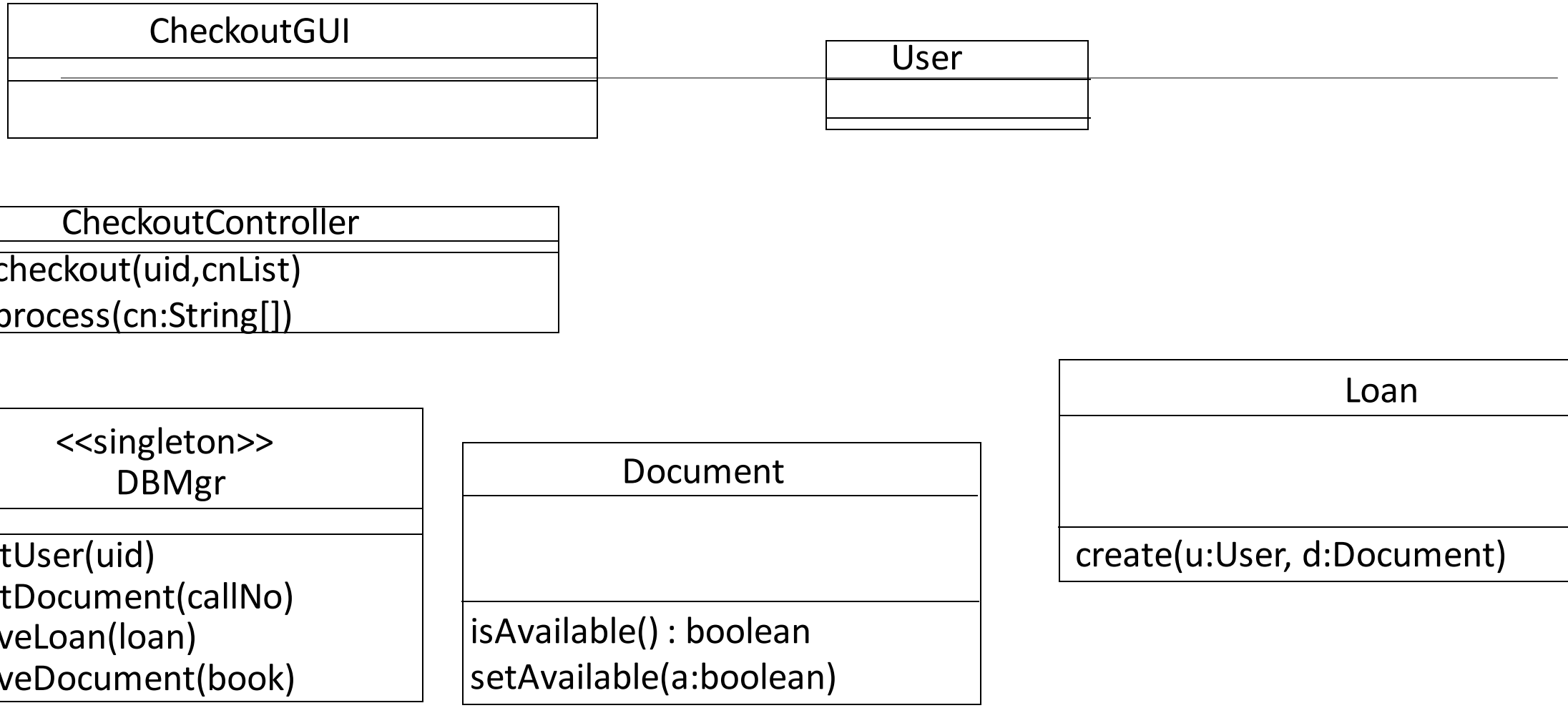
Classes Identified



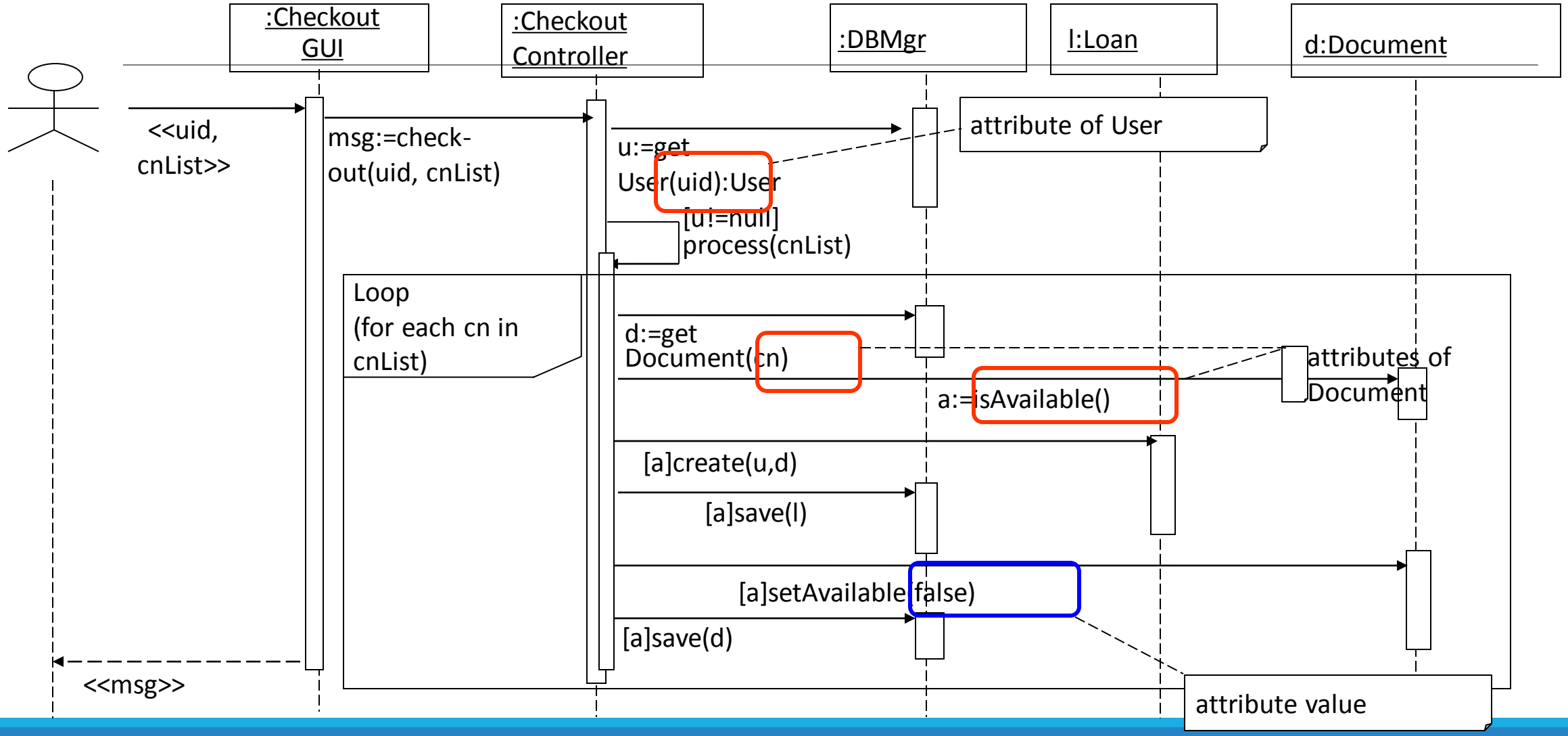
Identify Methods



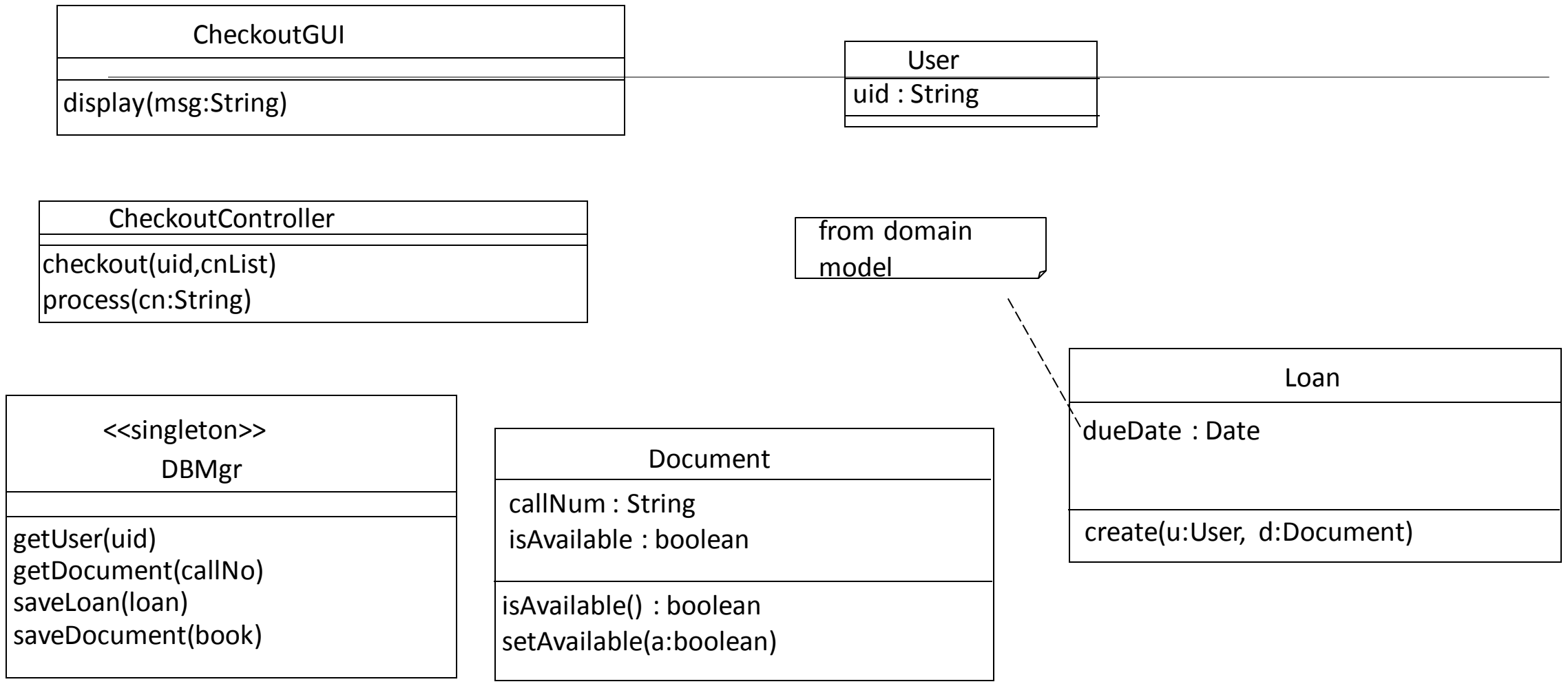
Fill In Identified Methods



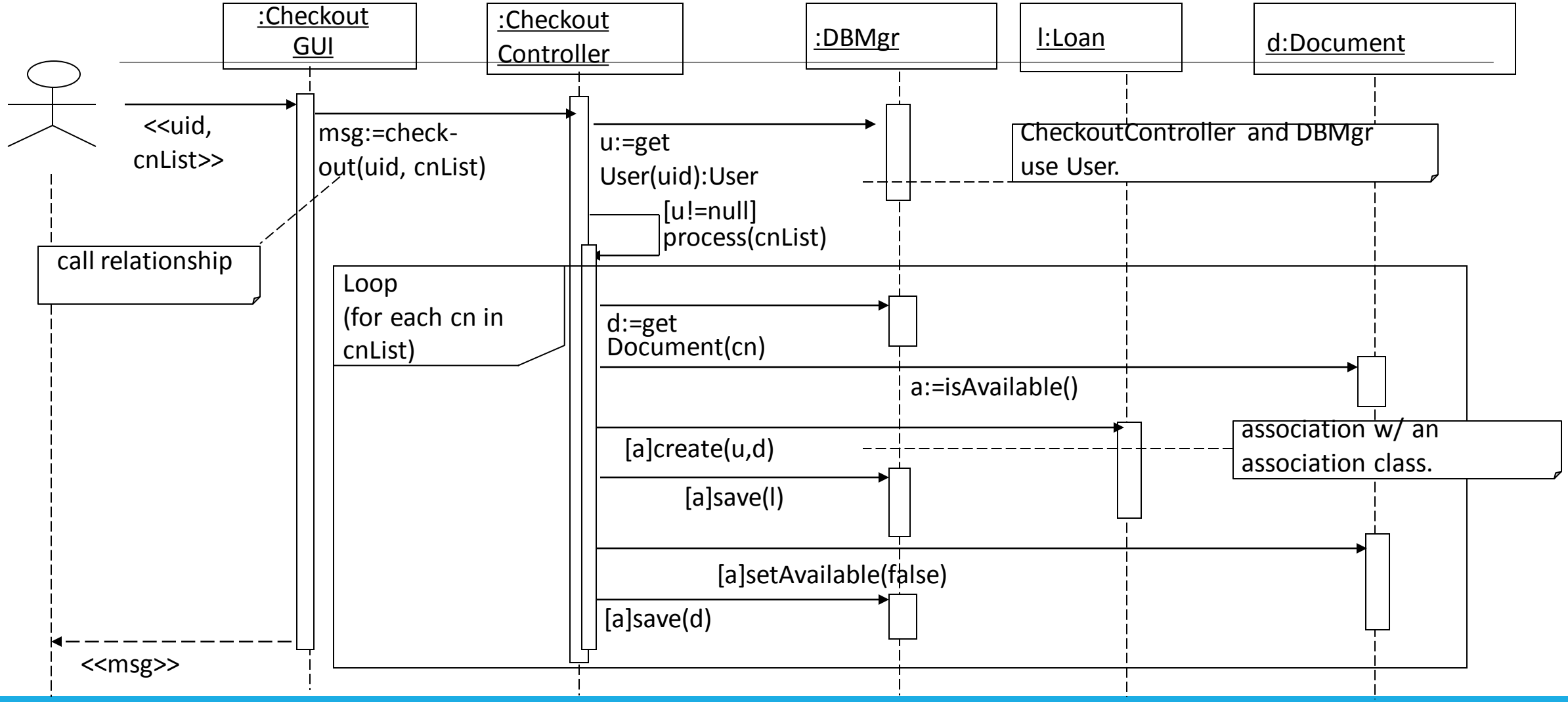
Identify Attributes



Fill In Attributes

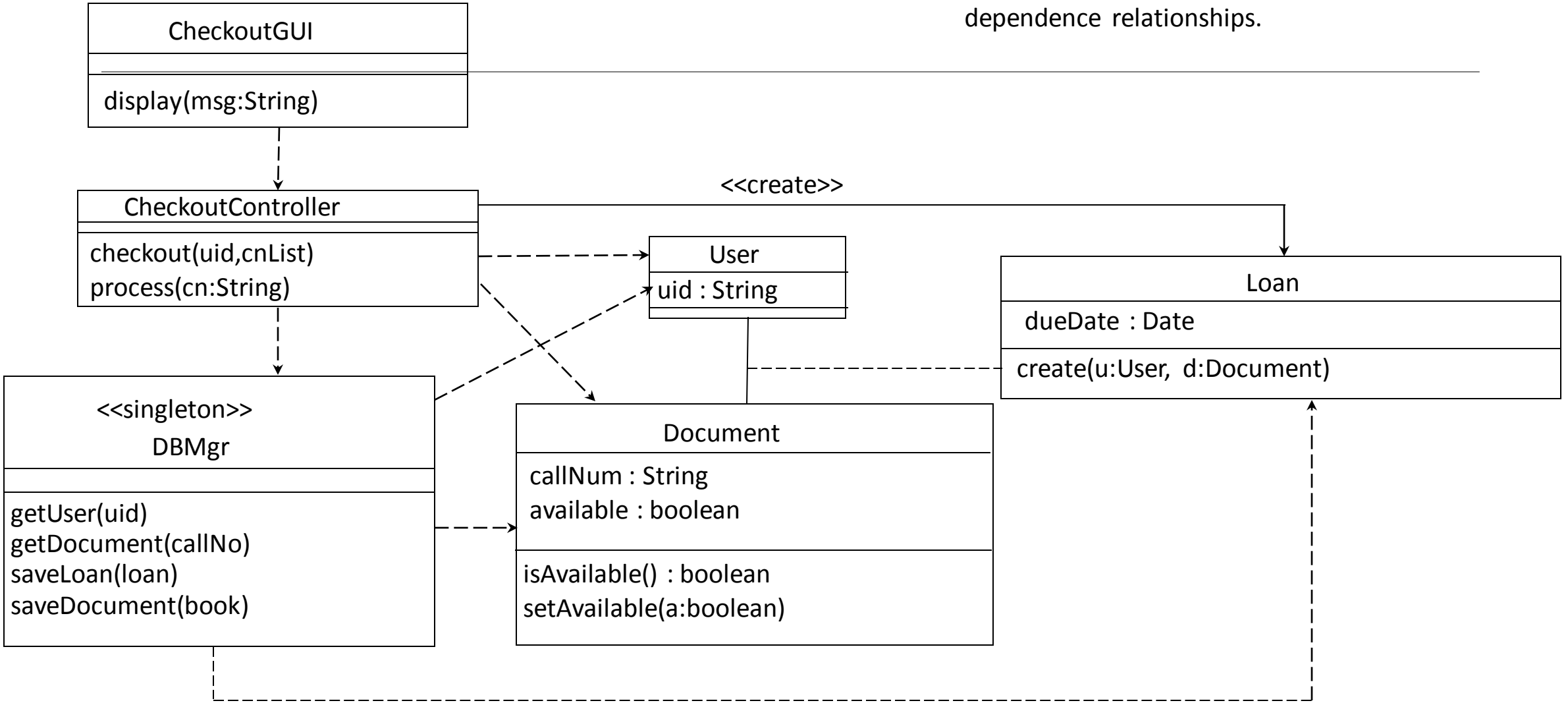


Identify Relationships

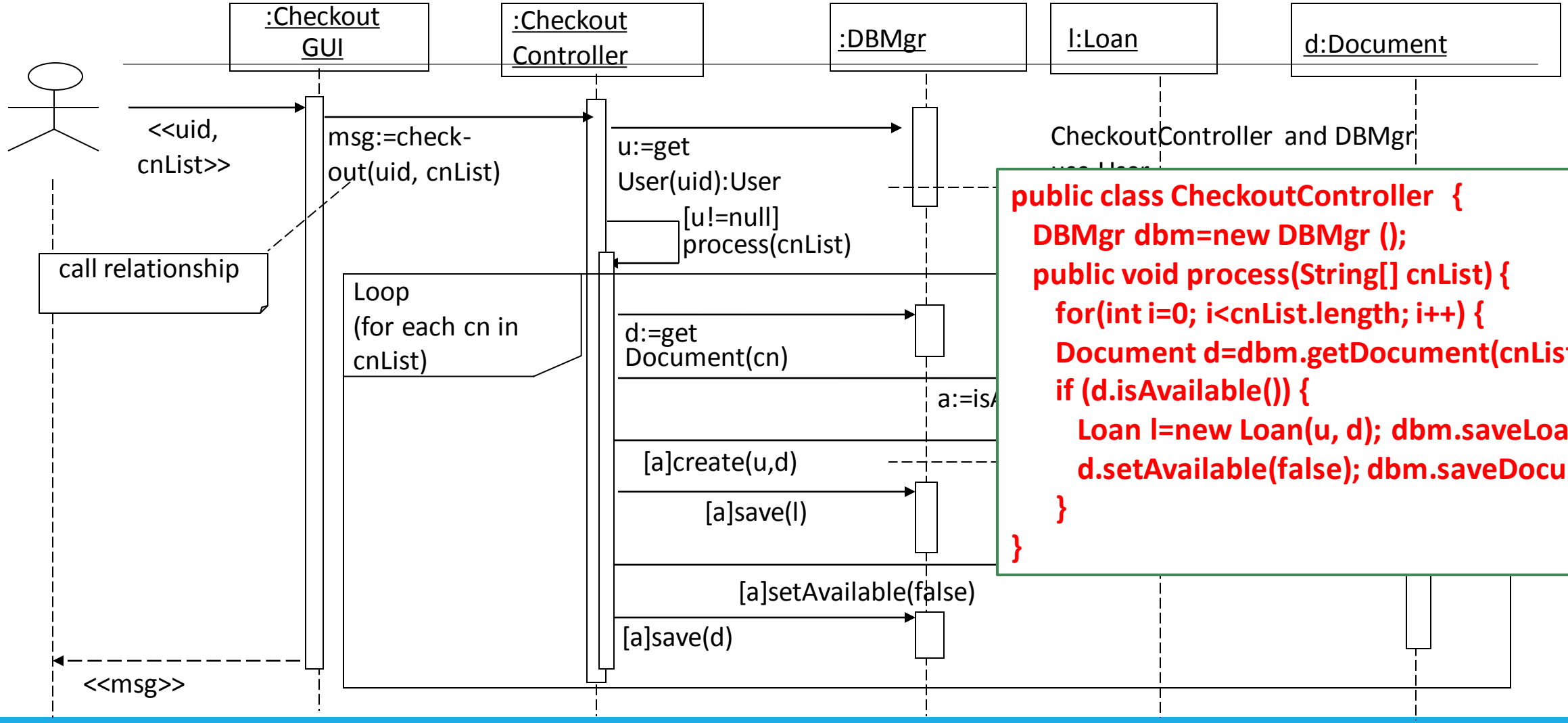


Fill In Relationships

The dashed arrow lines denote uses or dependence relationships.



From Sequence Diagram to Implementation




```
public class CheckoutController  {
  DBMgr dbm=new DBMgr ();
  public void process(String[] cnList) {
    for(int i=0; i<cnList.length; i++) {
    Document d=dbm.getDocument(cnList[i]);
    if (d.isAvailable()) {
      Loan l=new Loan(u, d); dbm.saveLoan(l);
      d.setAvailable(false); dbm.saveDocument(d);
    }
}
```