Interaction Diagrams

- A series of diagrams describing the *dynamic behavior* of an object-oriented system.
  - A set of messages exchanged among a set of objects within a context to accomplish a purpose.
- Often used to model the way a use case is realized through a sequence of messages between objects.

Interaction Diagrams (Cont.)

- The purpose of Interaction diagrams is to:
  - Model interactions between objects
  - Assist in understanding how a system (a use case) actually works
  - Verify that a use case description can be supported by the existing classes
  - Identify responsibilities/operations and assign them to classes

Interaction Diagrams (Cont.)

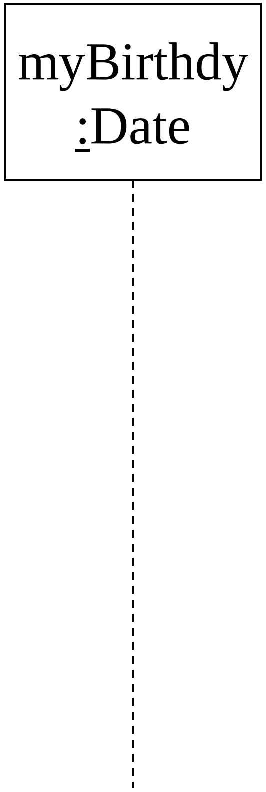
- UML
  - Sequence Diagram
    - Emphasizes time ordering of messages.
  - Collaboration Diagrams
    - Emphasizes structural relations between objects
  - Timing
    - Focuses on timing constraints
  - Interaction overview
    - visualize the cooperation between other Interaction diagrams

Sequence Diagrams

- A **sequence diagram** displays the object interactions arranged in a time sequence.
  - The diagram shows the objects and classes required for the scenario with the sequence of messages exchanged between the objects.
- Sequence diagrams are composed of:
  - Class roles that represent the roles that objects play in the use case.
  - Lifelines that represent the existence of an object over a period of time.
  - Activations that represent the time during which an object is performing an operation.
  - Messages that are the communication between objects.

Sequence Diagrams : Object

- Object naming:
  - syntax: *[instanceName][:className]*
  - Name classes consistently with your class diagram (same classes).
  - Include instance names when objects are referred to in messages or when several objects of the same type exist in the diagram.



myBirthdy
:Date

Sequence Diagrams : Life Line

L→R U→D

- Sequence diagrams are read and developed from left to right.
  - Usually the first item on the left is the actor for the scenario.
  - This is then followed by the objects in the sequence that they will be accessed.
- A **lifeline** shows the object's life during the interaction (scenario).
  - Lifelines are shown as a line displayed vertically from the bottom of each object.

Sequence Diagrams : Messages

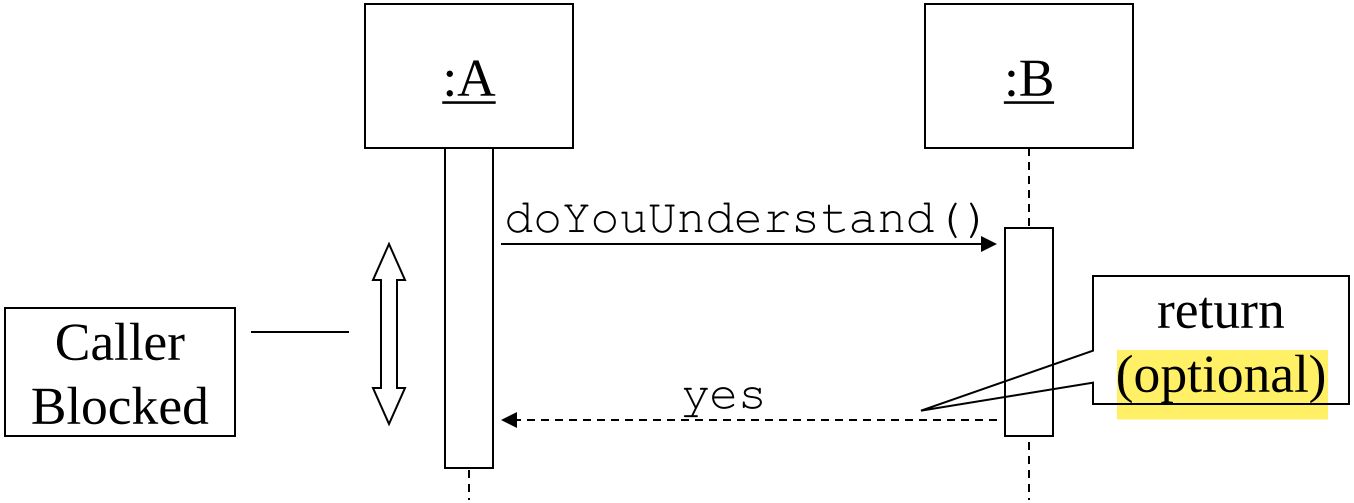
- An interaction between two objects is performed as a message sent from one object to another (simple operation call, Signaling, RPC)
- If object obj_1 sends a message to another object obj_2 some link must exist between those two objects (dependency, same objects)

Messages (Cont.)

- A message is represented by an arrow between the life lines of two objects.
  - Self calls are also allowed
  - The time required by the receiver object to process the message is denoted by an *activation-box*.
- A message is labeled at minimum with the message name.
  - Arguments and control information (conditions, iteration) may be included.
- Two types of messages
  - Synchronous
  - Asynchronous

Synchronous Messages

- Synchronous message between active objects indicates wait semantics
- The sender waits for the message to be handled before it continues.
- This typically shows a method call..

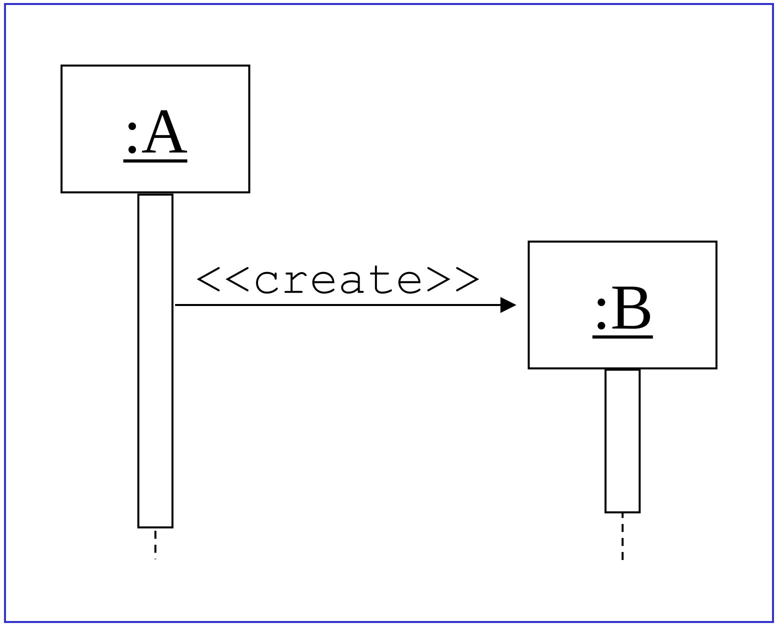


Asynchronous Messages

- There is no explicit return message to the caller.
- Asynchronous message between objects indicates no-wait semantics
- The sender does not wait for the message before it continues.
- This allows objects to execute concurrently.
  - This is used when threads have been implemented.
  - Asynchronous messages are represented with half-arrowheads on the message link.

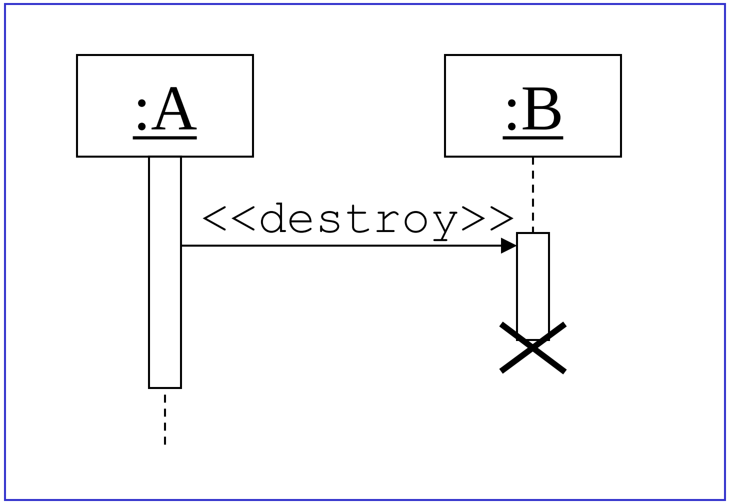
Object Creation

- An object may create another object via a `<<create>>` message.



Object Destruction

- An object may destroy another object via a `<<destroy>>` message.
  - An object may destroy itself.
  - A large X is displayed and indicates that the object will self-destruct.
  - Avoid modeling object destruction unless memory management is critical.



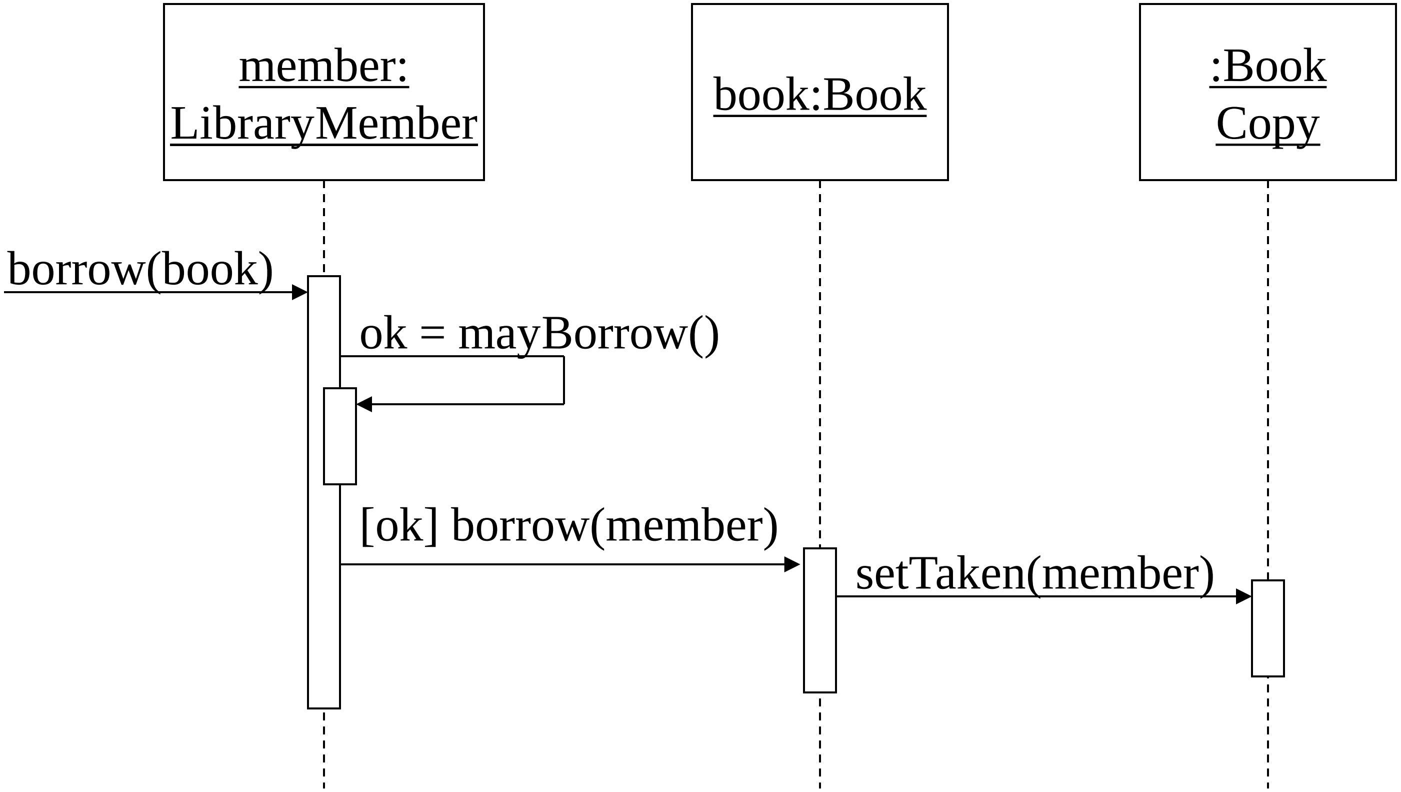
Sequence Diagrams

- There may be operations that require certain information exist before the operation exists, this is referred to as a condition.
  - In this case the operation is only executed if the condition is met.
    - Conditions are shown using [ ].
- A **return** shows that an operation has completed and returns to the calling operation.
  - The return is shown on the diagram as a dashed line.
  - Usually returns are only shown for clarity, not for every message.

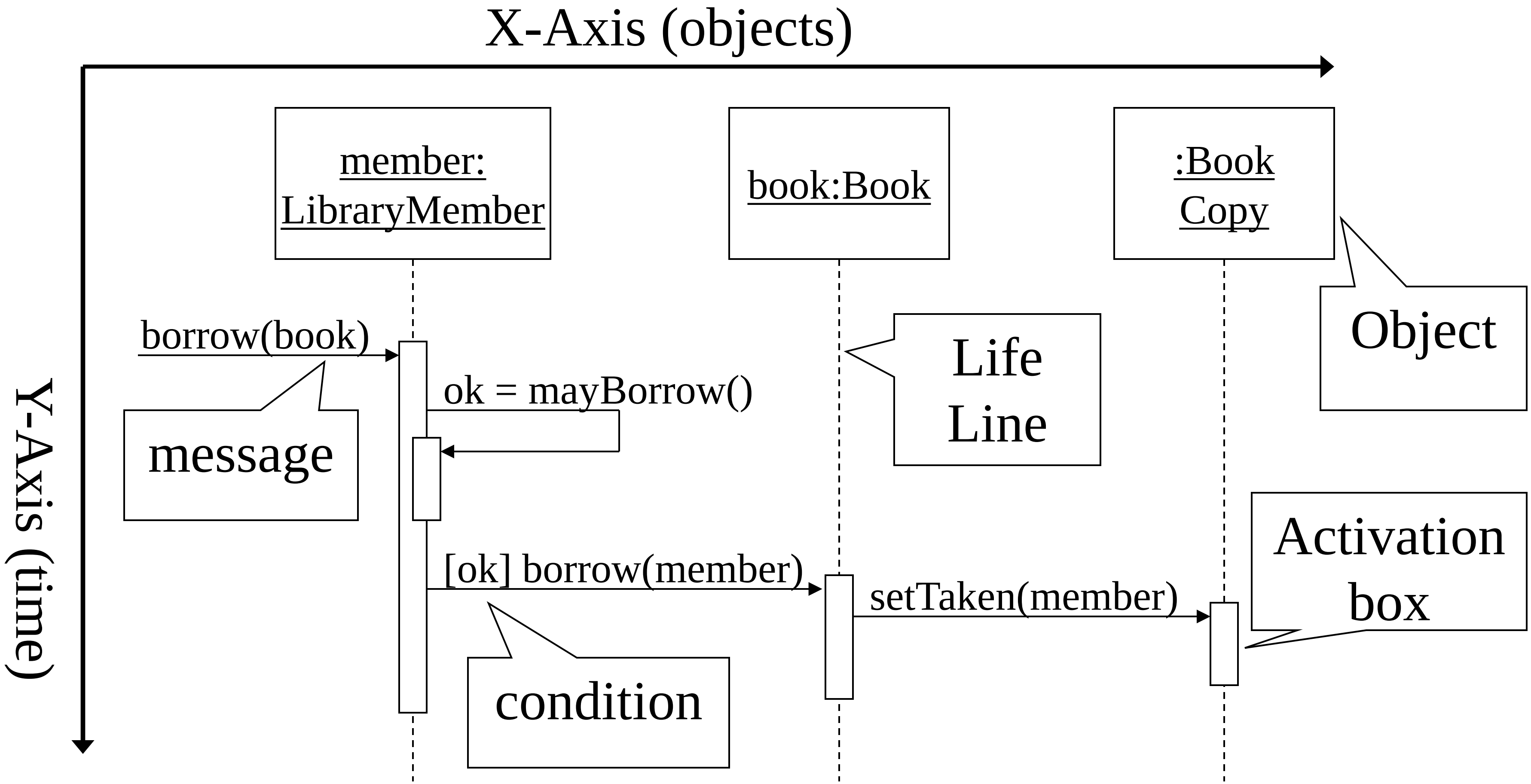
A First Look at Sequence Diagrams

- Illustrates how objects interacts with each other.
- Emphasizes time ordering of messages.
- Can model simple sequential flow, branching, iteration, recursion and concurrency.

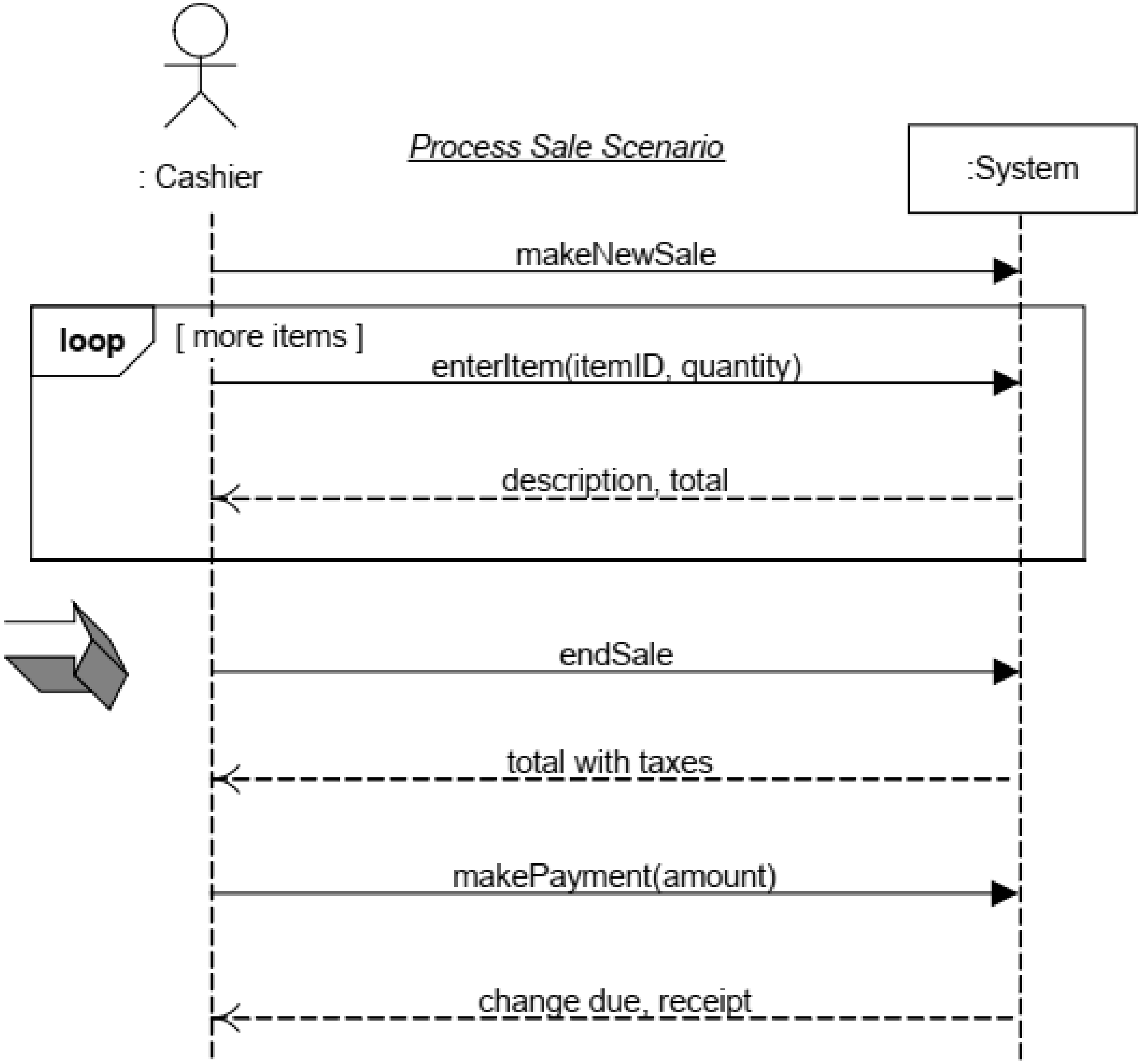
A Sequence Diagram

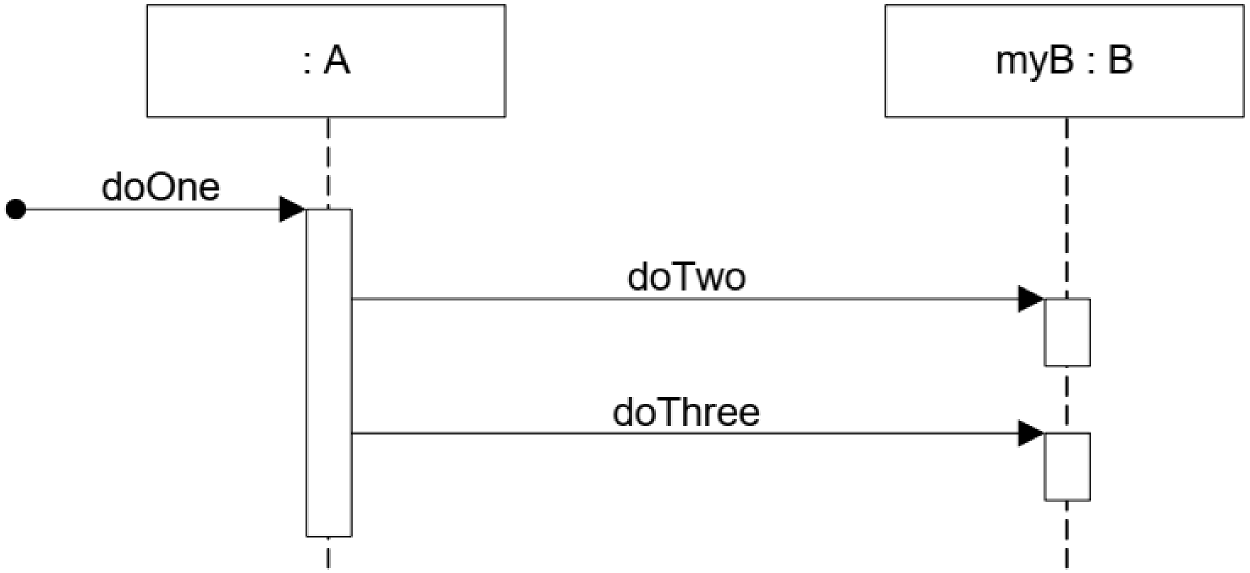


A Sequence Diagram

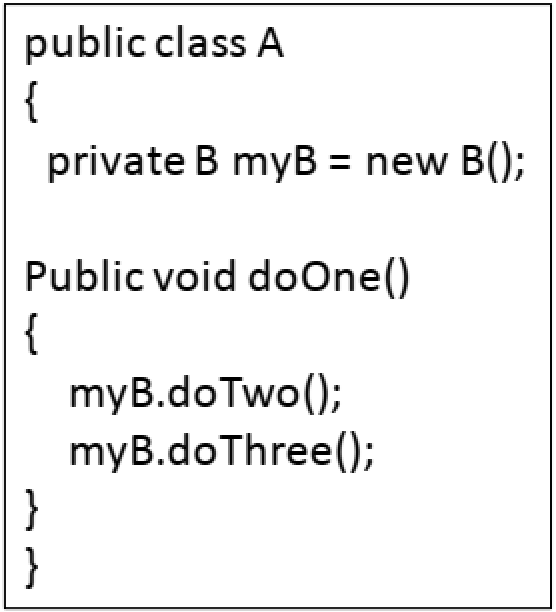


System level Sequence diagram

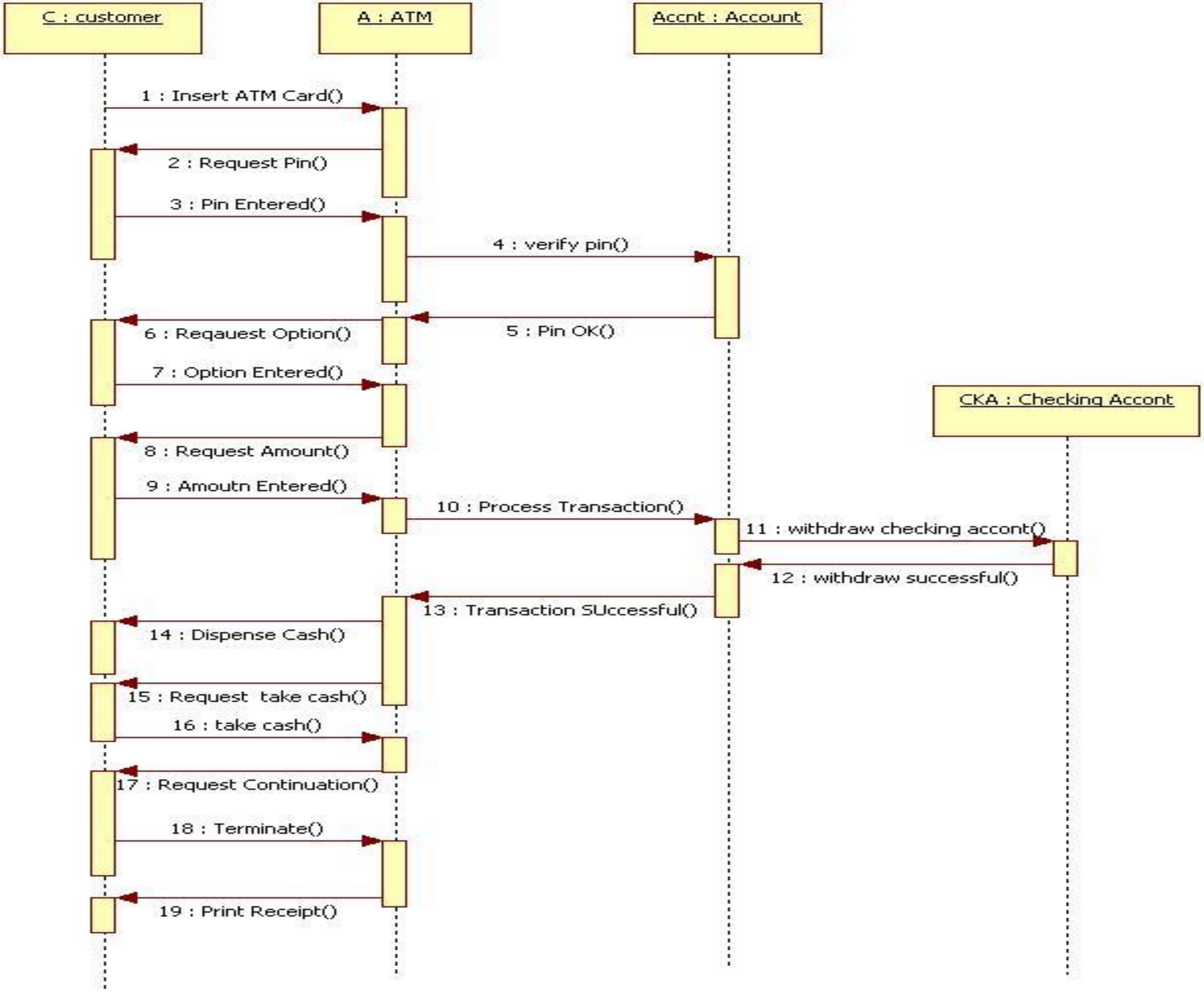


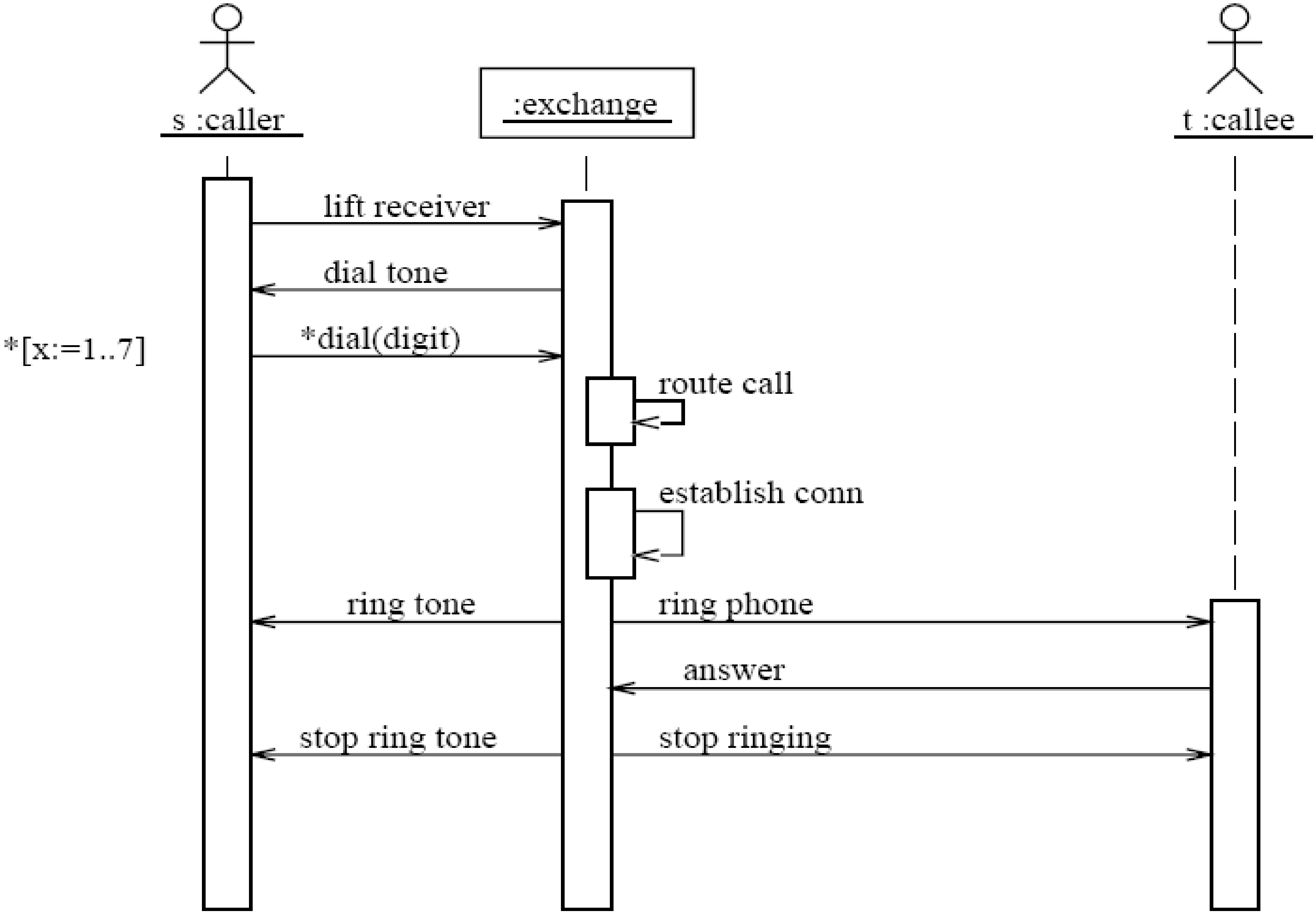


Design level Sequence diagram



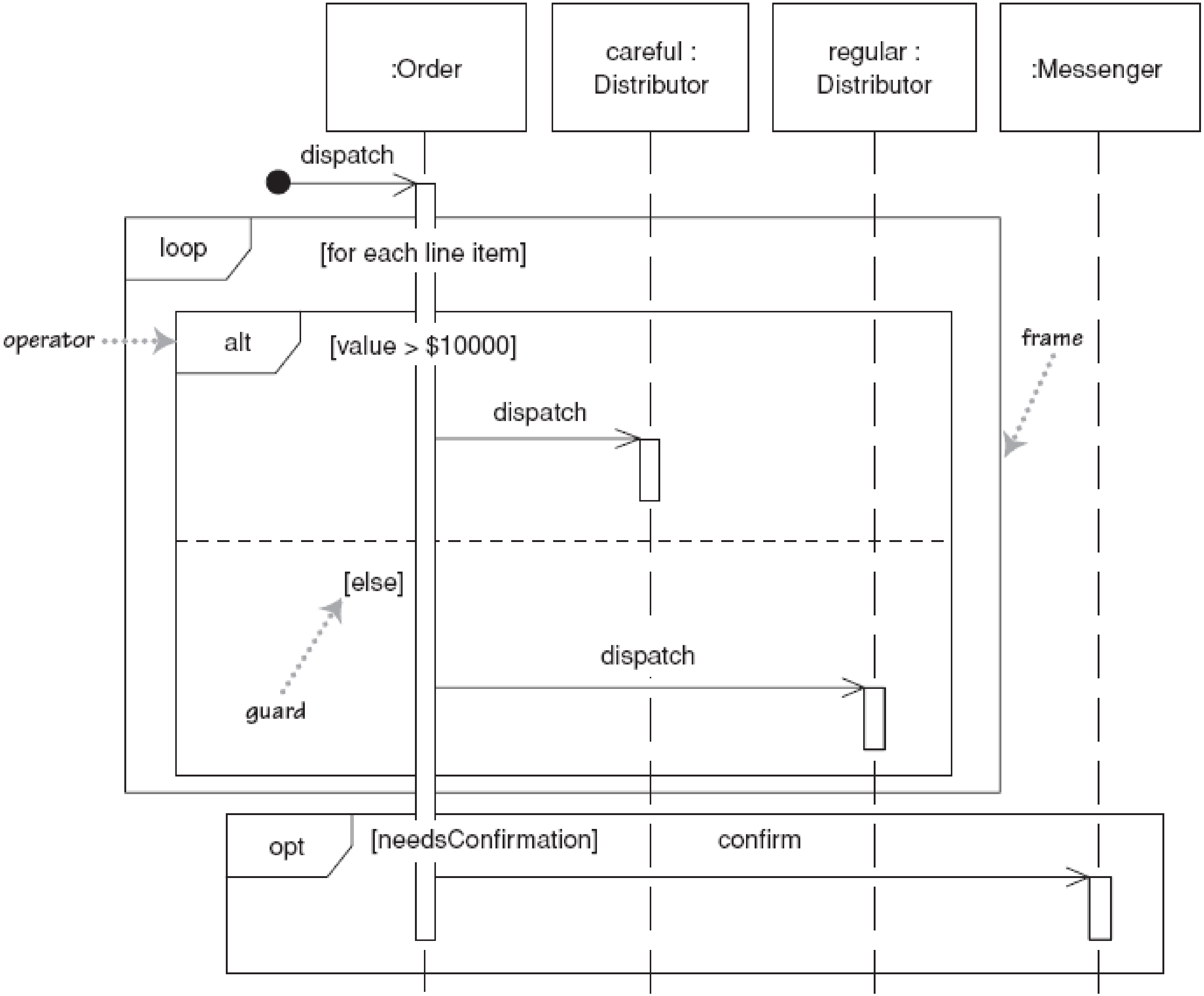
```
public class A
{
  private B myB = new B();

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

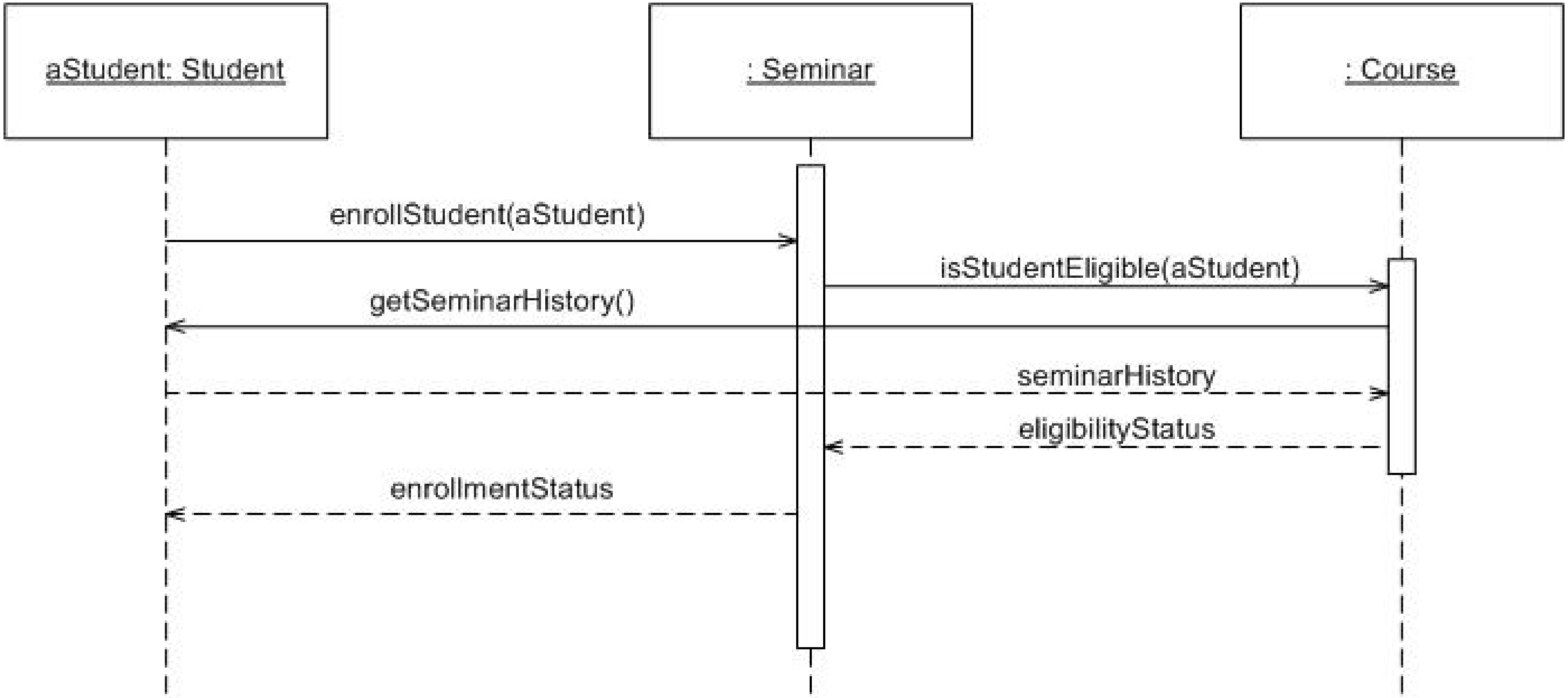

C : customer
A : ATM
Accnt : Account
CKA : Checking Accont
1 : Insert ATM Card()
2 : Request Pin()
3 : Pin Entered()
4 : verify pin()
5 : Pin OK()
6 : Reqauest Option()
7 : Option Entered()
8 : Request Amount()
9 : Amoutn Entered()
10 : Process Transaction()
11 : withdraw checking accont()
12 : withdraw successful()
13 : Transaction SUccessful()
14 : Dispense Cash()
15 : Request take cash()
16 : take cash()
17 : Request Continuation()
18 : Terminate()
19 : Print Receipt()


s :caller
:exchange
t :callee
lift receiver
dial tone
*dial(digit)
*[x:=1..7]
route call
establish conn
ring tone
ring phone
answer
stop ring tone
stop ringing

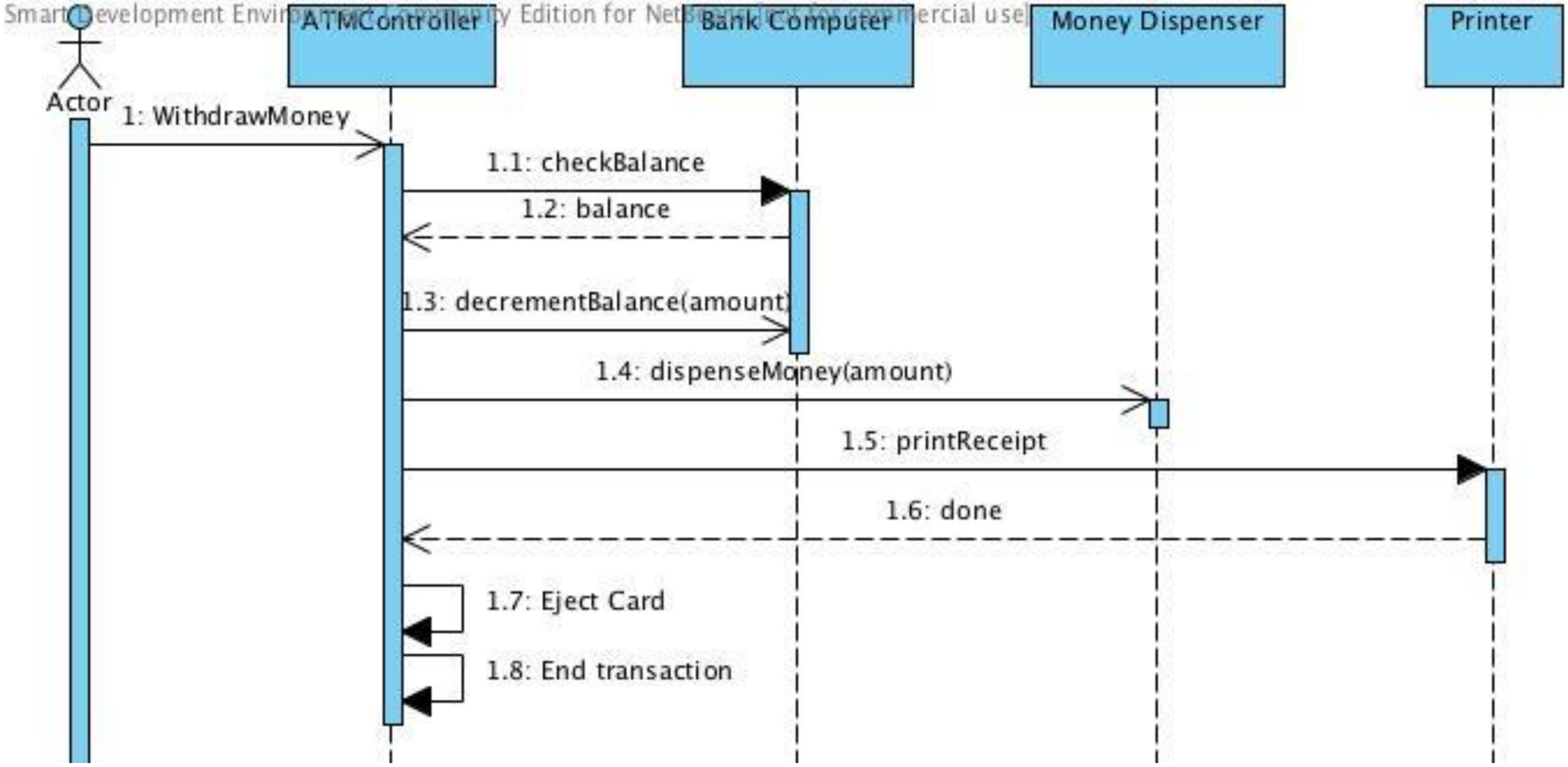
Indicating selection and loops

- frame: box around part of a sequence diagram to indicate selection or loop
  - `if` -> (opt) [condition]
  - `if/else` -> (alt) [condition], separated by horizontal dashed line
  - loop -> (loop) [condition or items to loop over]

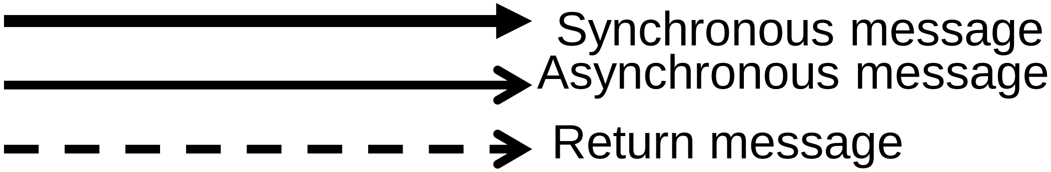

:Order
careful : Distributor
regular : Distributor
:Messenger
dispatch
loop
[for each line item]
operator
alt
[value > $10000]
frame
dispatch
[else]
guard
dispatch
opt
[needsConfirmation]
confirm

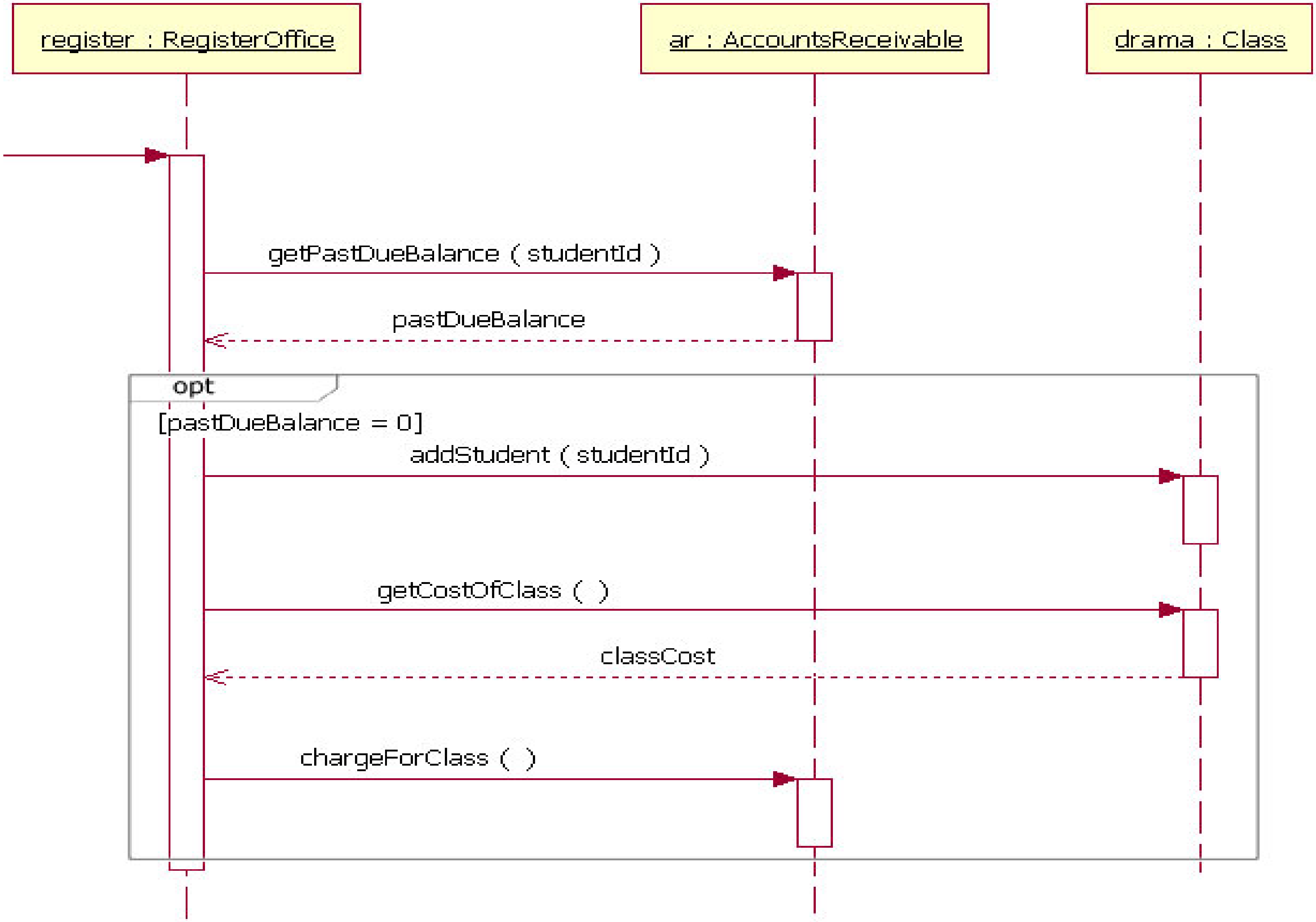

aStudent: Student
: Seminar
: Course
enrollStudent(aStudent)
isStudentEligible(aStudent)
getSeminarHistory()
seminarHistory
eligibilityStatus
enrollmentStatus

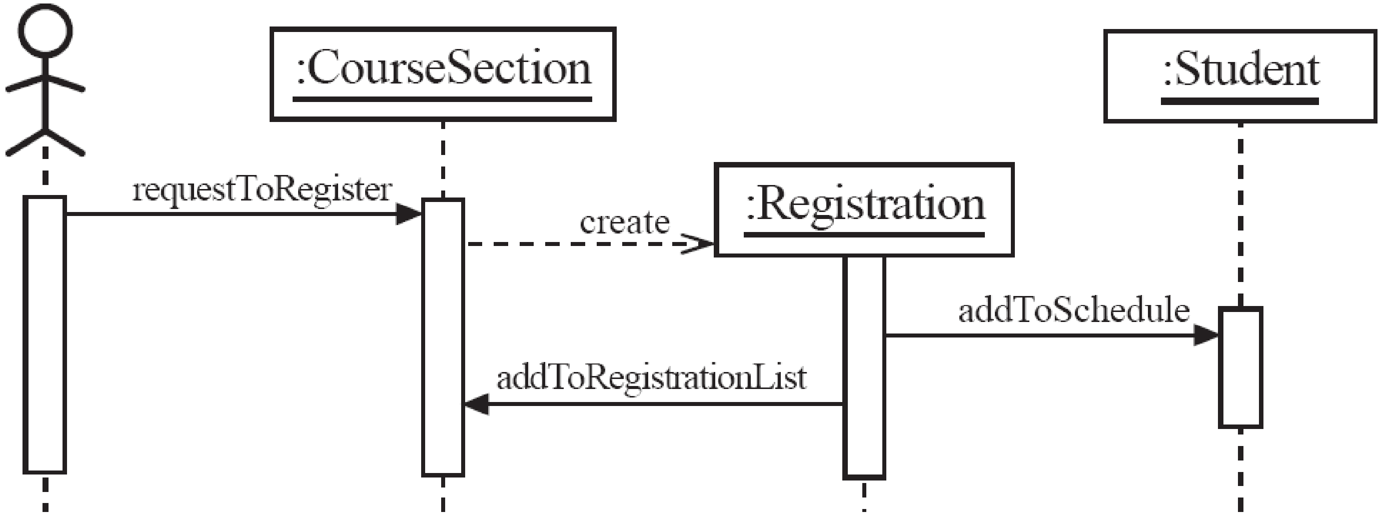
Async Message Example

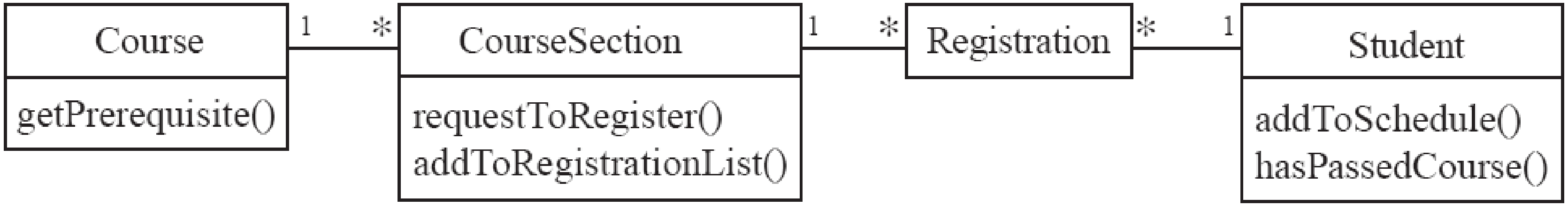


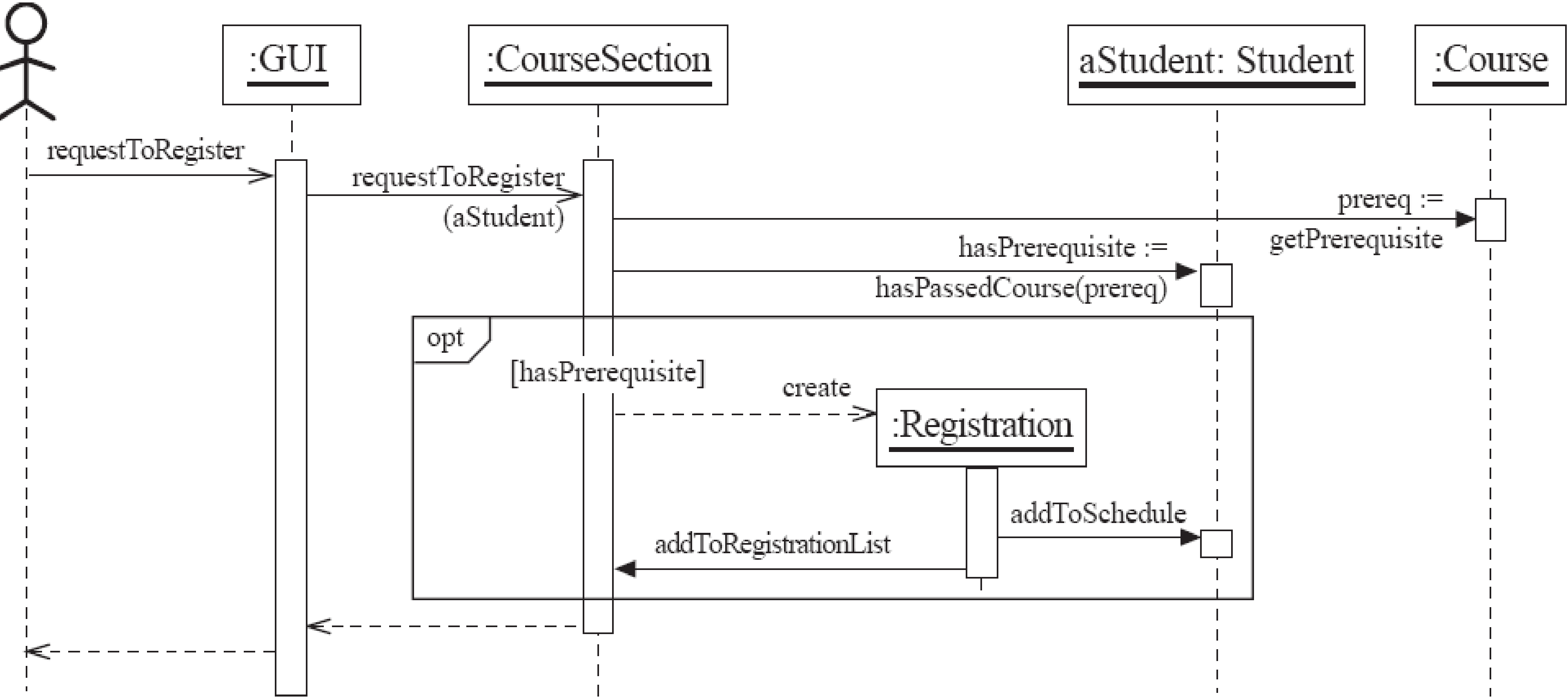
There are problems here… what are they?

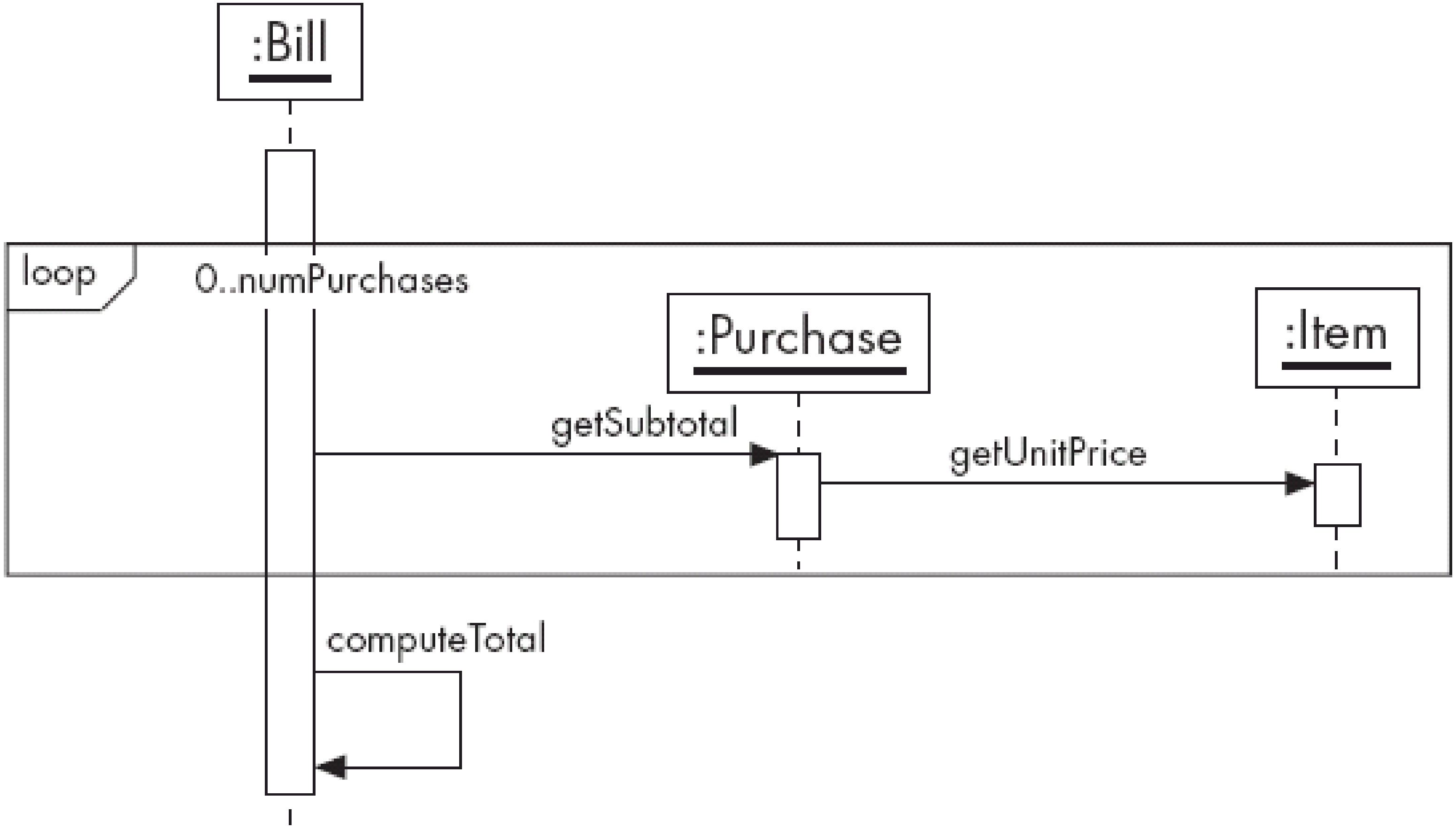


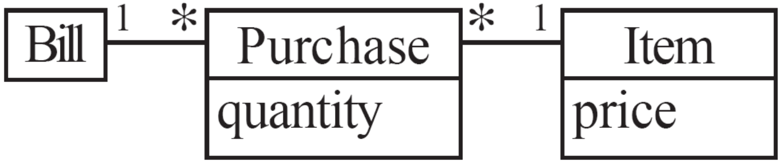

register : RegisterOffice
ar : AccountsReceivable
drama : Class
getPastDueBalance (studentId)
pastDueBalance
opt
[pastDueBalance = 0]
addStudent (studentId)
getCostOfClass ( )
classCost
chargeForClass ( )

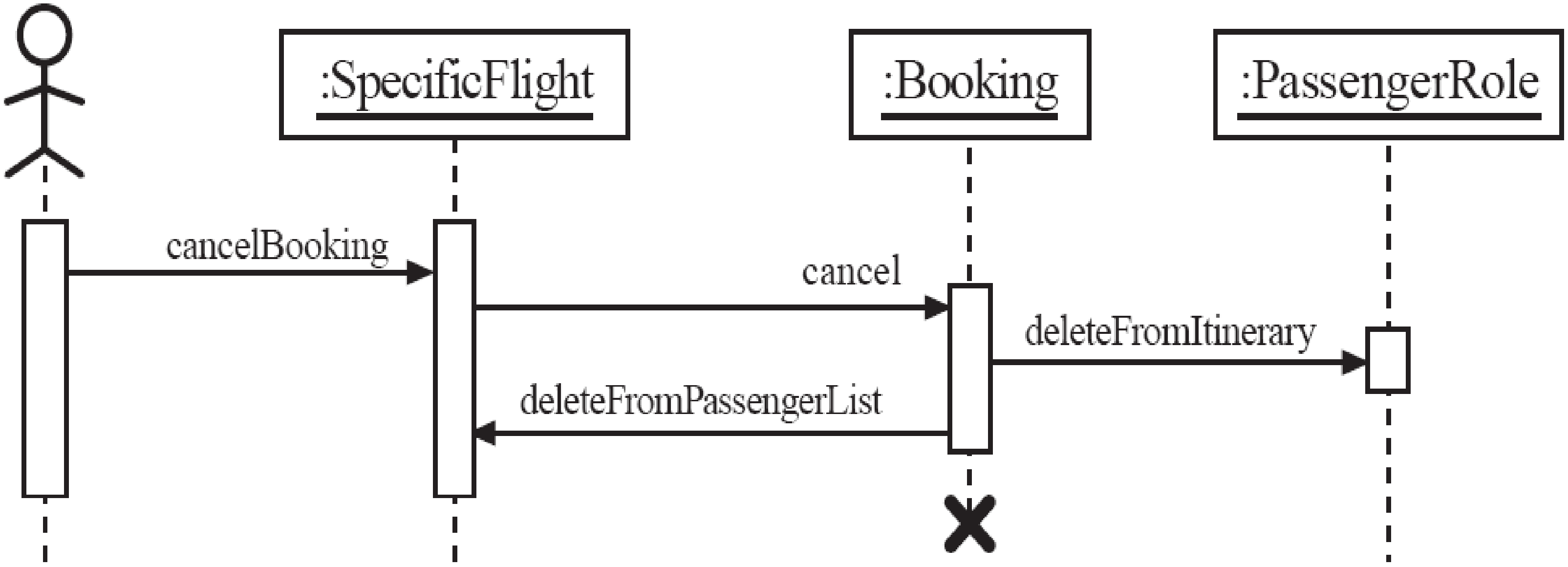

:CourseSection
:Student
requestToRegister
create
:Registration
addToSchedule
addToRegistrationList


Course
getPrerequisite()
1
*
CourseSection
requestToRegister()
addToRegistrationList()
1
*
Registration
*
1
Student
addToSchedule()
hasPassedCourse()


:GUI
:CourseSection
aStudent: Student
:Course
requestToRegister
requestToRegister
(aStudent)
prereq :=
getPrerequisite
hasPrerequisite :=
hasPassedCourse(prereq)
opt
[hasPrerequisite]
create
:Registration
addToSchedule
addToRegistrationList


:Bill
loop
0..numPurchases
:Purchase
:Item
getSubtotal
getUnitPrice
computeTotal


Bill
1
*
Purchase
quantity
*
1
Item
price


:SpecificFlight
:Booking
:PassengerRole
cancelBooking
cancel
deleteFromItinerary
deleteFromPassengerList

Why not just code it?

- Sequence diagrams can be somewhat close to the code level. So why not just code up that algorithm rather than drawing it as a sequence diagram?
  - a good sequence diagram is still a bit above the level of the real code (not all code is drawn on diagram)
  - sequence diagrams are language-agnostic (can be implemented in many different languages
  - non-coders can do sequence diagrams
  - easier to do sequence diagrams as a team
  - can see many objects/classes at a time on same page (visual bandwidth)

Rules of thumb

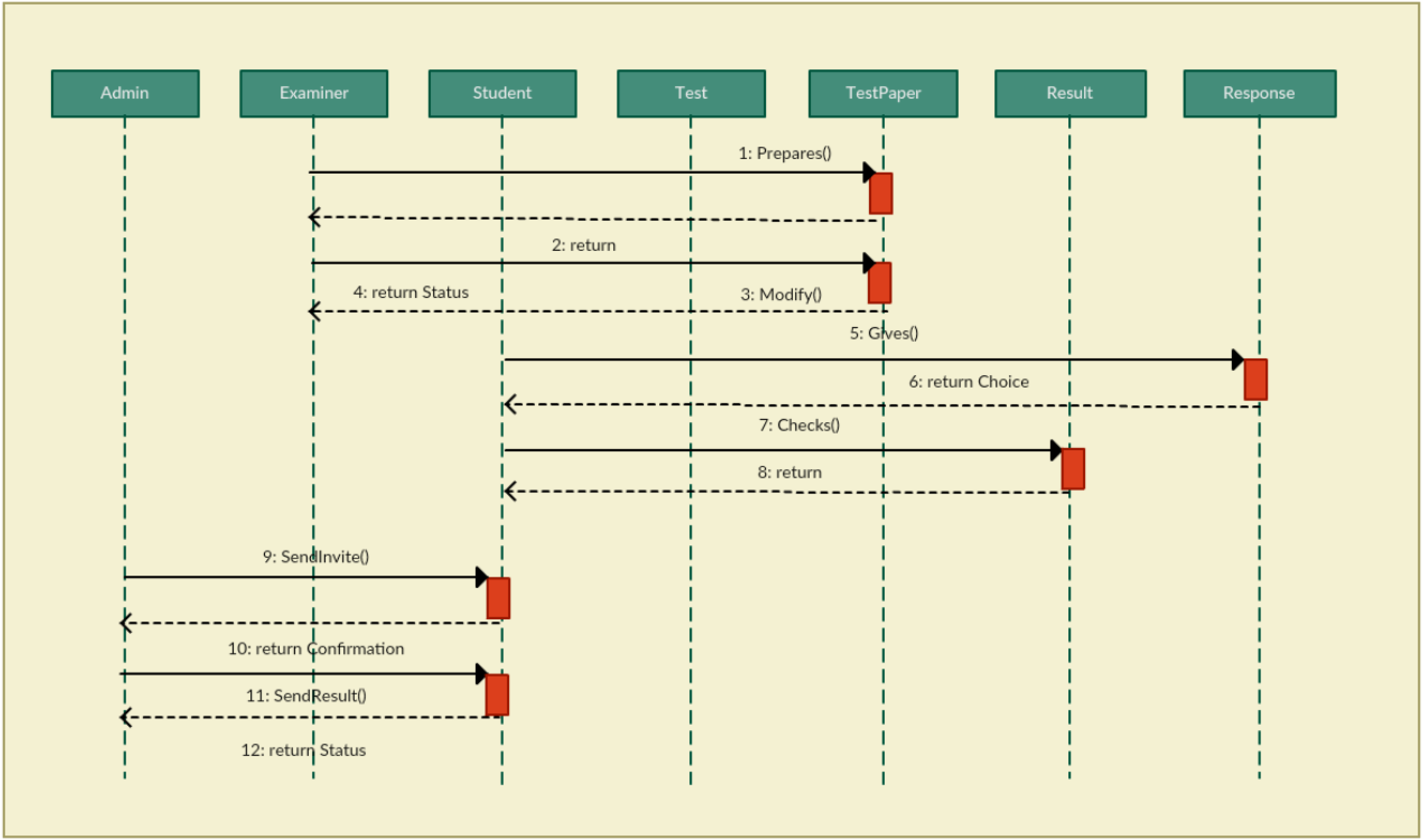
- Rarely use options, loops, alt/else
  - These constructs complicate a diagram and make them hard to read/interpret.
  - Frequently it is better to create multiple simple diagrams

- Create sequence diagrams for use cases when it helps clarify and visualize a complex flow

- Remember: the goal of UML is communication and understanding

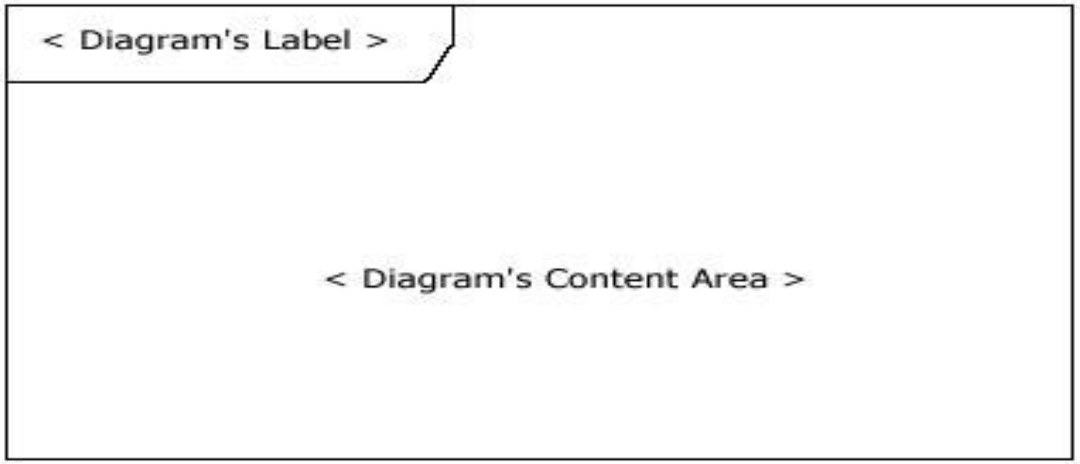
- In Beauty and the Beast kitchen, items came to life.
- Draw a sequence diagram for making a peanut butter and jelly sandwich if the following objects are alive: knife, peanut butter jar (and peanut butter), jelly jar (and jelly), bread, plate.
- I may or may not want the crusts cut off.
- Don’t forget to open and close things like the jars, and put yourself away, cleanup, etc…

Online Exam System

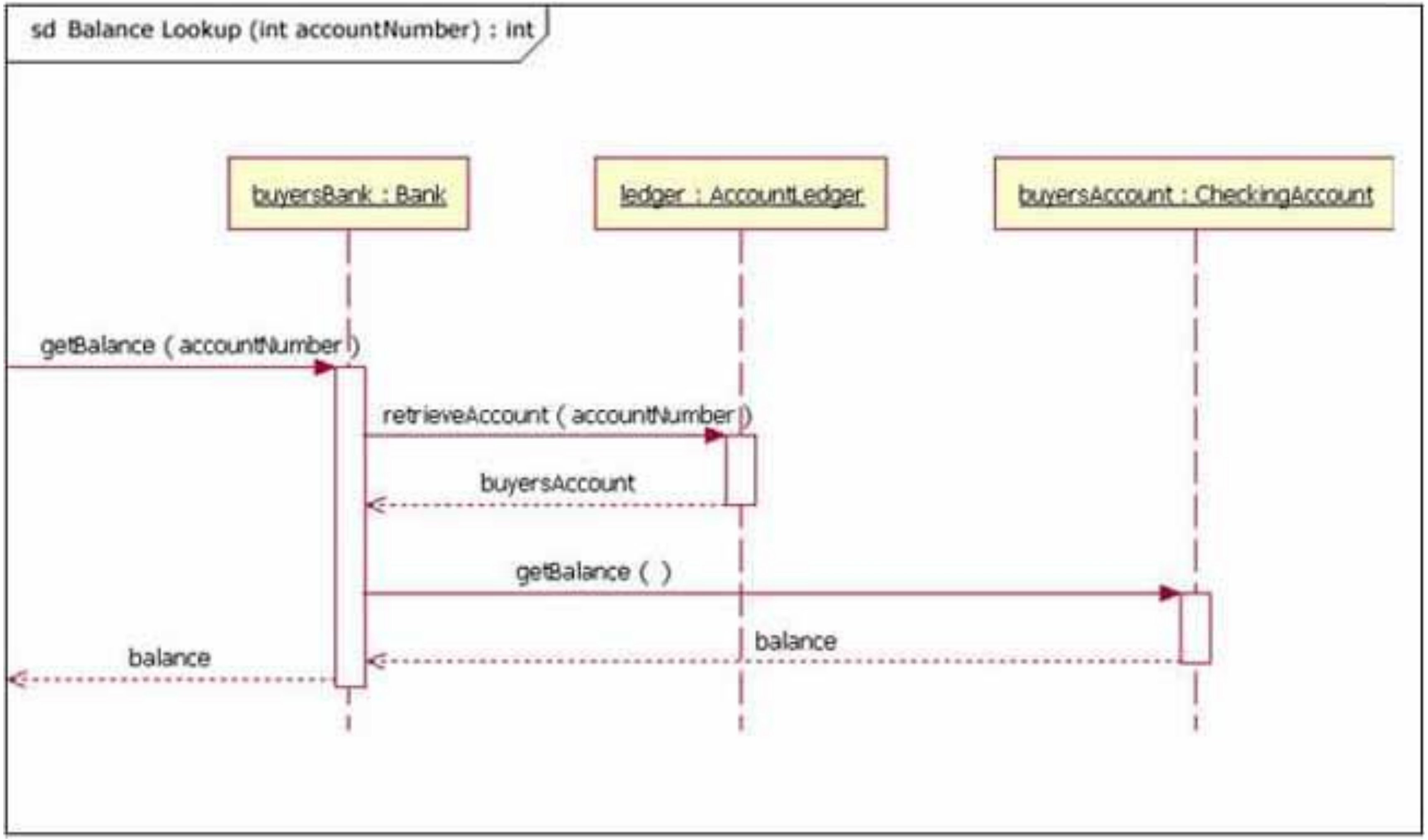


Frame Element

- The graphical boundary of a diagram.
- provides a consistent place for a diagram's label
- The diagram's label needs to follow the format of
  - Diagram Type Diagram Name
- The UML specification provides specific text values for diagram types (e.g., sd = Sequence Diagram, activity = Activity Diagram, and use case = Use Case Diagram).
- Optional in UML diagrams
- Incoming and outgoing messages for a sequence can be modeled by connecting the messages to the border of the frame element



Frame Element

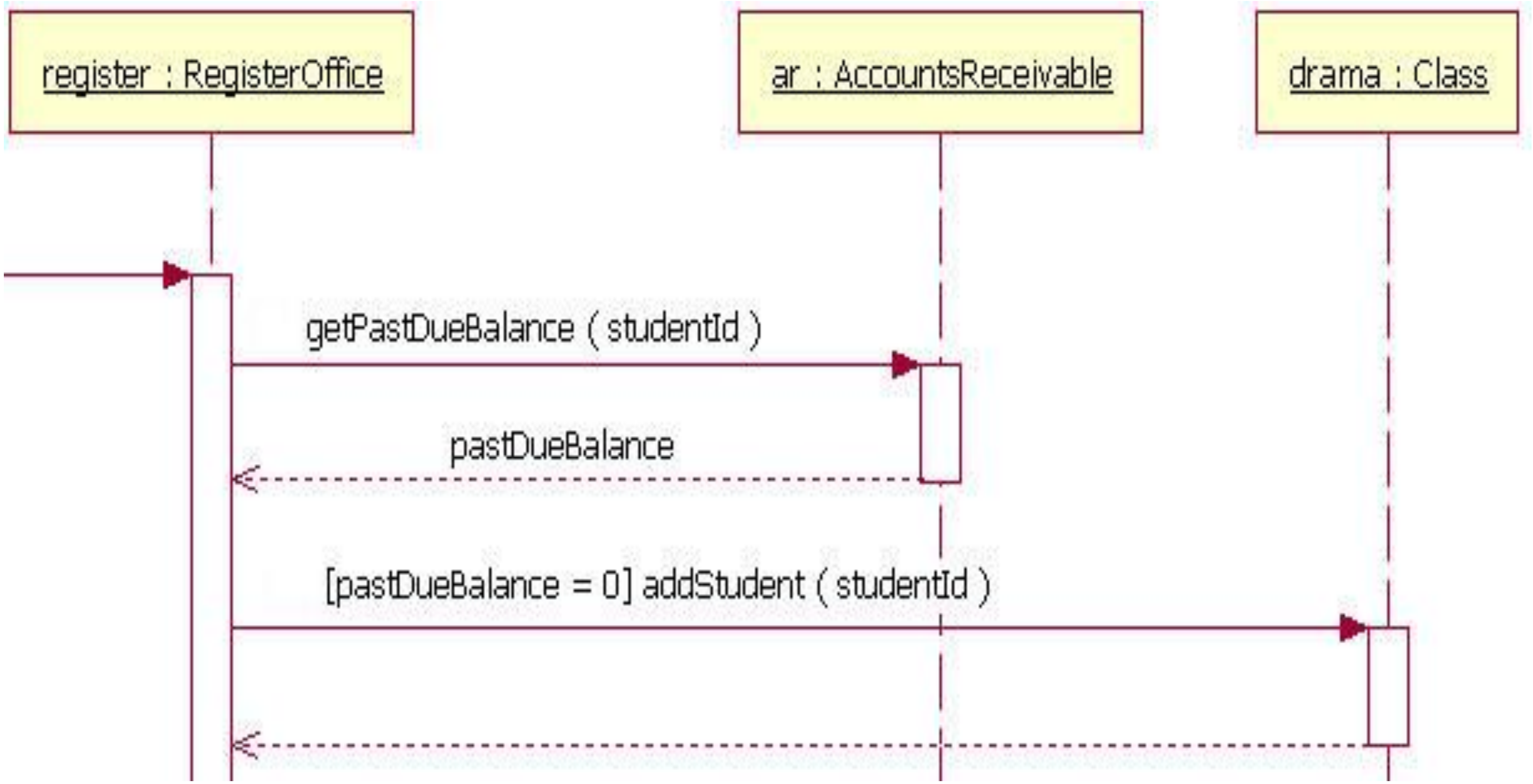

sd Balance Lookup (int accountNumber) : int
buyersBank : Bank
ledger : AccountLedger
buyersAccount : CheckingAccount
getBalance (accountNumber)
retrieveAccount (accountNumber)
buyersAccount
getBalance ( )
balance
balance

Guards

- When modelling object interactions, there will be times when a condition must be met for a message to be sent to the object.
- Guards are used throughout UML diagrams to control flow.
- In UML 1.x, a guard could only be assigned to a single message.
- To draw a guard on a sequence diagram, you place the guard element above the message line being guarded and in front of the message name.
- The format is : [Boolean Test]

Guards

By having the guard on this message, the addStudent message will only be sent if the accounts receivable system returns a past due balance as zero

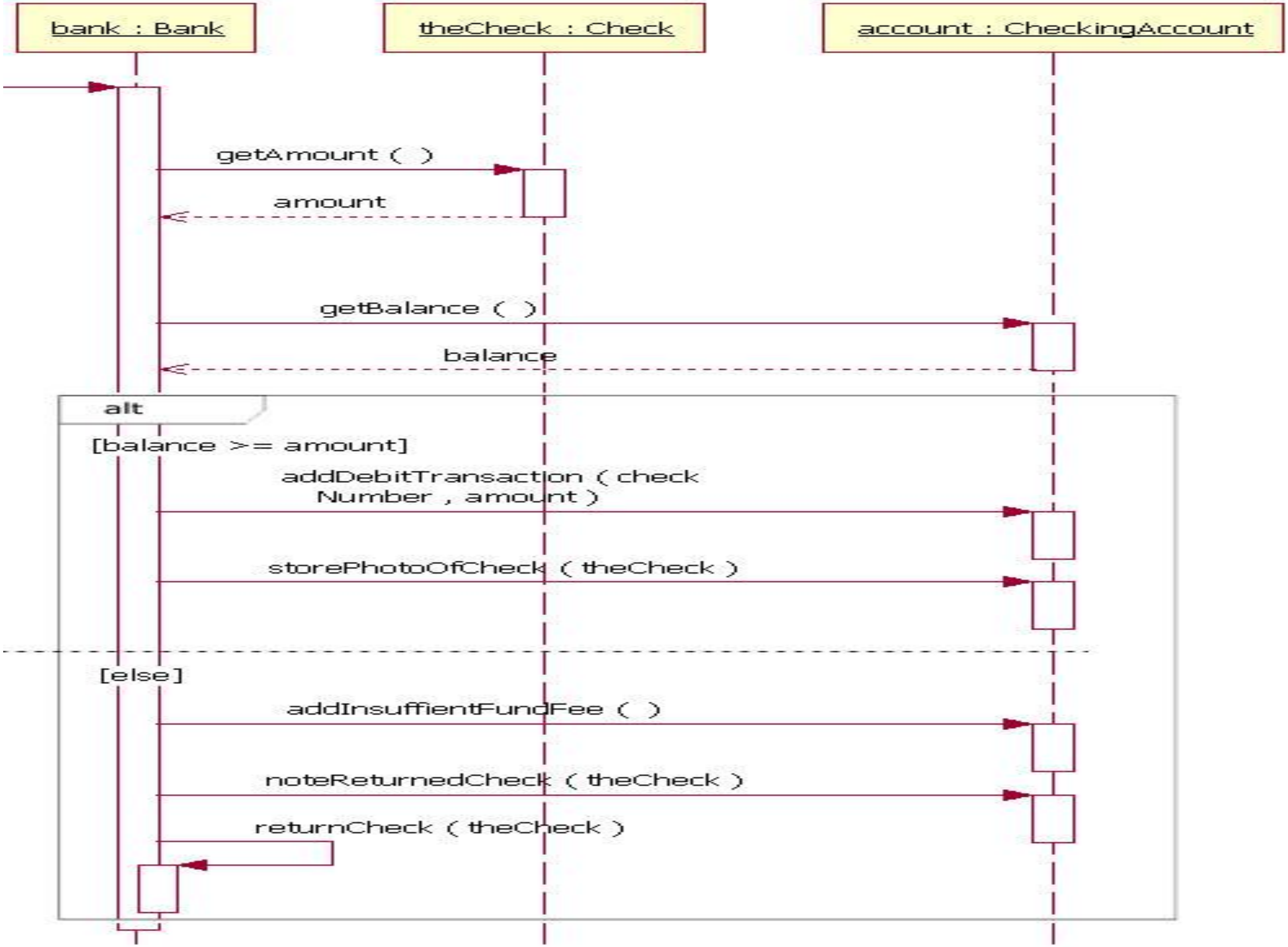

register : RegisterOffice
ar : AccountsReceivable
drama : Class
getPastDueBalance (studentId)
pastDueBalance
[pastDueBalance = 0] addStudent (studentId)

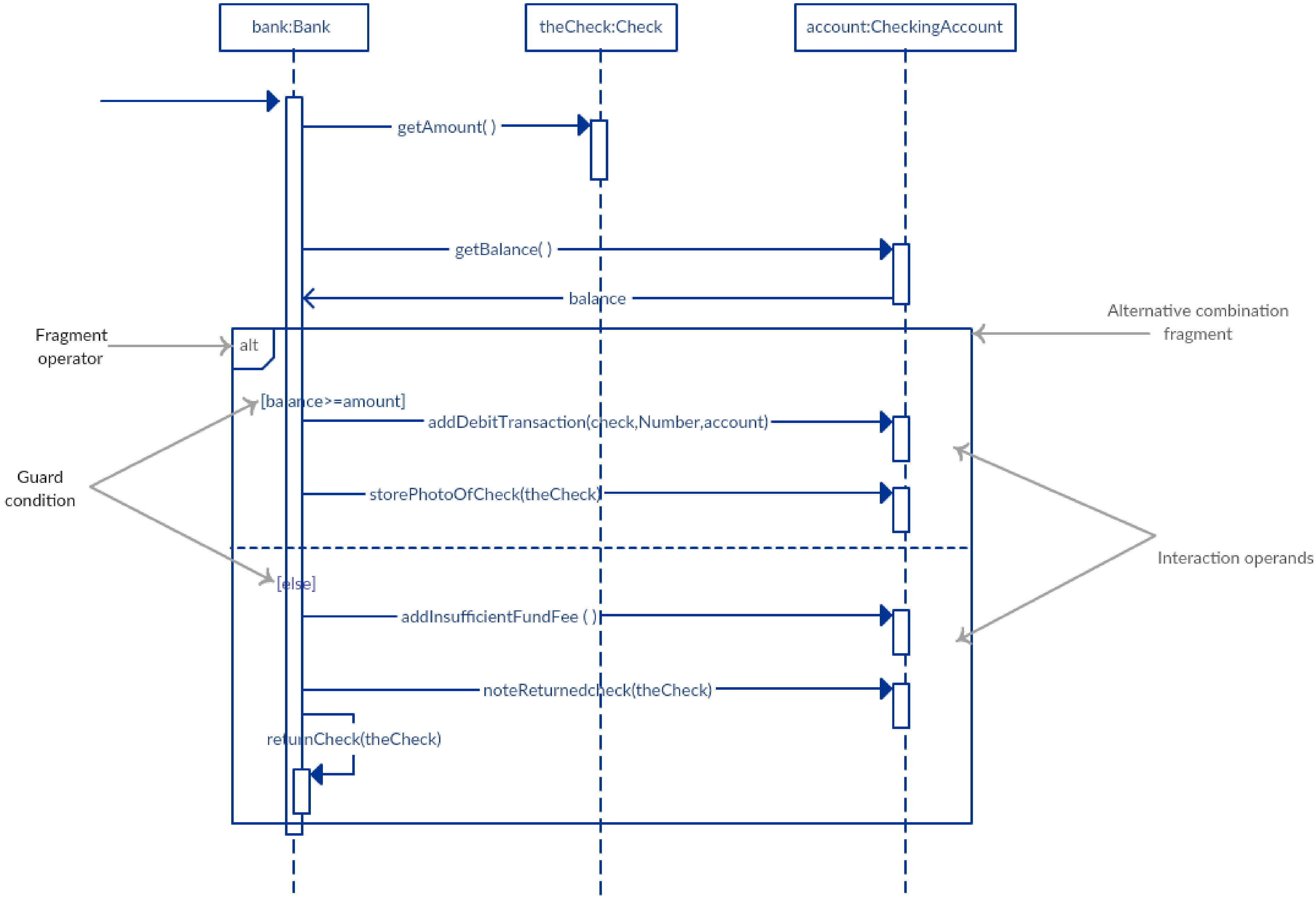
Guards

- The UML 1.x "in-line" guard is not sufficient to handle the logic required for a sequence being modelled.
- A combined fragment is used to group sets of messages together to show conditional flow in a sequence diagram.
- The UML 2 specification identifies 12 interaction types for combined fragments
  - **alt -** alternatives
  - **opt -** option
  - **loop -** iteration
  - **break -** break
  - **par -** parallel
  - **strict** - strict sequencing
  - **seq** - weak sequencing
  - **critical** - critical region
  - **ignore** - ignore
  - **consider** - consider
  - **assert** - assertion
  - **neg** - negative

Alternatives

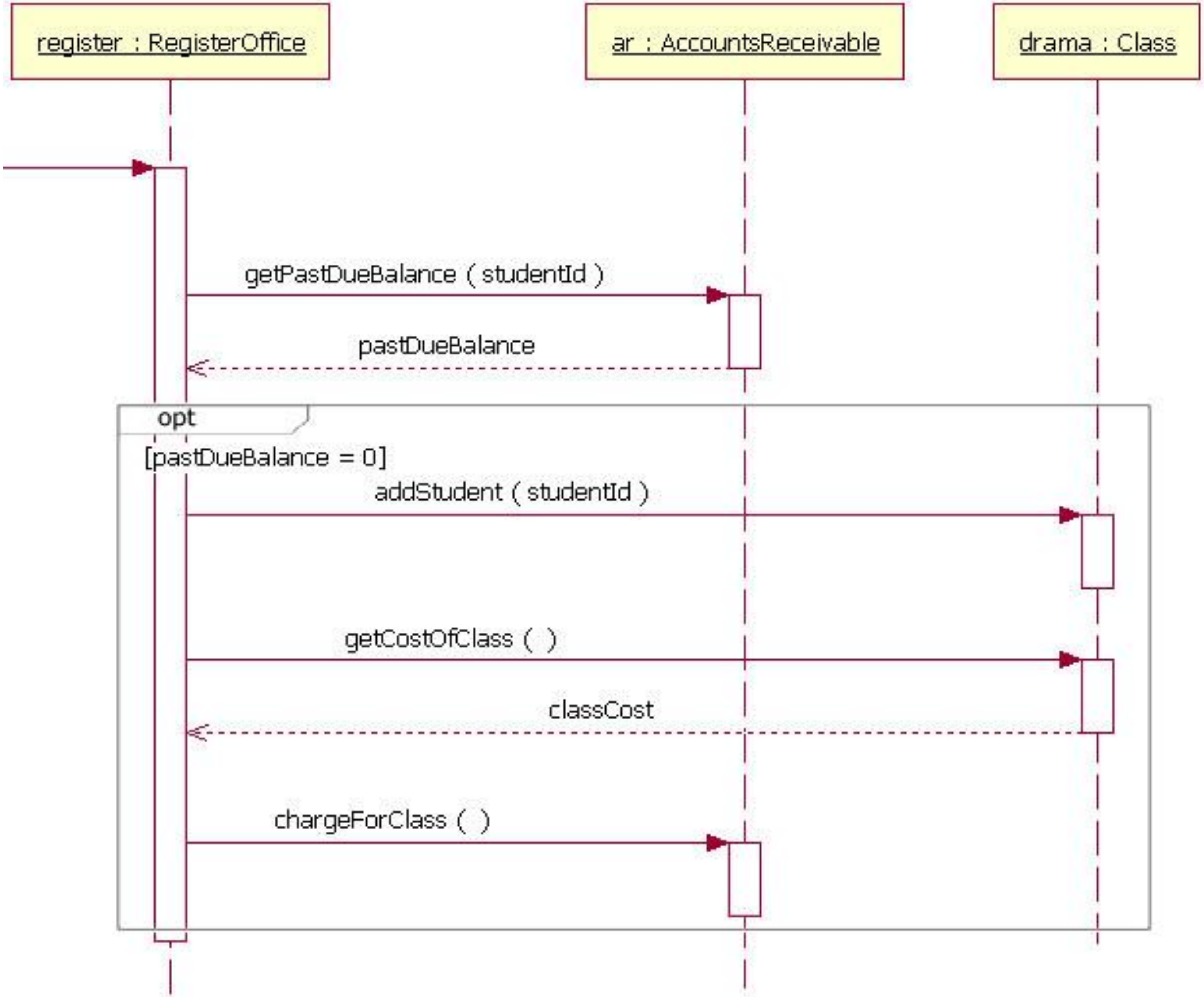
- Alternatives are used to designate a mutually exclusive choice between two or more message sequences.
- Alternatives allow the modeling of the classic "if then else" logic.
  - e.g., **if** I buy three items, **then** I get 20% off my purchase; **else** I get 10% off my purchase.
- An alternative combination fragment element is drawn using a frame.
- The word "alt" is placed inside the frame's namebox.
- Operands are separated by a dashed line.
- Each operand is given a guard to test against, and this guard is placed towards the top left section of the operand on top of a lifeline.
- If an operand's guard equates to "true," then that operand is the operand to follow.
- There can be as many alternative paths as are needed.
  - Add an operand to the rectangle with that sequence's guard and messages.


bank : Bank
theCheck : Check
account : CheckingAccount
getAmount ( )
amount
getBalance ( )
balance
alt
[balance >= amount]
addDebitTransaction (check Number , amount)
storePhotoOfCheck (theCheck)
[else]
addInsuffientFundFee ( )
noteReturnedCheck (theCheck)
returnCheck (theCheck)


bank:Bank
theCheck:Check
account:CheckingAccount
getAmount( )
getBalance( )
balance
Alternative combination fragment
Fragment operator
alt
[balance>=amount]
addDebitTransaction(check,Number,account)
Guard condition
storePhotoOfCheck(theCheck)
Interaction operands
[else]
addInsufficientFundFee ( )
noteReturnedcheck(theCheck)
returnCheck(theCheck)

Option

- The option combination fragment is used to model a sequence
    - given a certain condition, sequence will occur otherwise, the sequence does not occur.
- An option is used to model a simple "if then" statement.
- The option combination fragment notation is similar to the alternation combination fragment, except that it only has one operand and there never can be an "else" guard.
- To draw an option combination you draw a frame.
- The text "opt" is placed inside the frame's namebox
- In the frame's content area the option's guard is placed towards the top left corner on top of a lifeline.
- Then the option's sequence of messages is placed in the remainder of the frame's content area

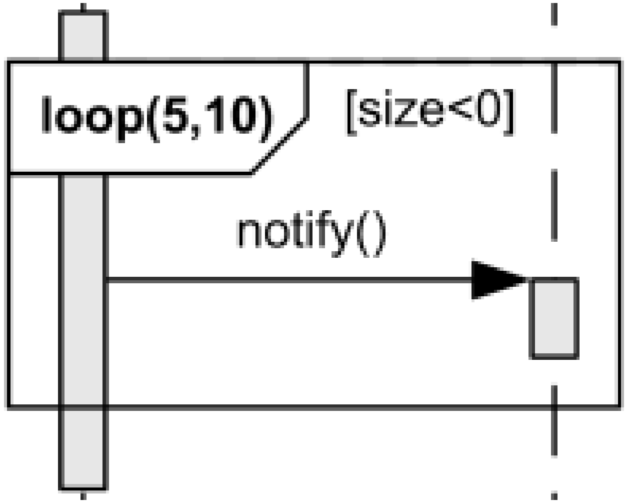

register : RegisterOffice
ar : AccountsReceivable
drama : Class
getPastDueBalance (studentId)
pastDueBalance
opt
[pastDueBalance = 0]
addStudent (studentId)
getCostOfClass ( )
classCost
chargeForClass ( )

Loops

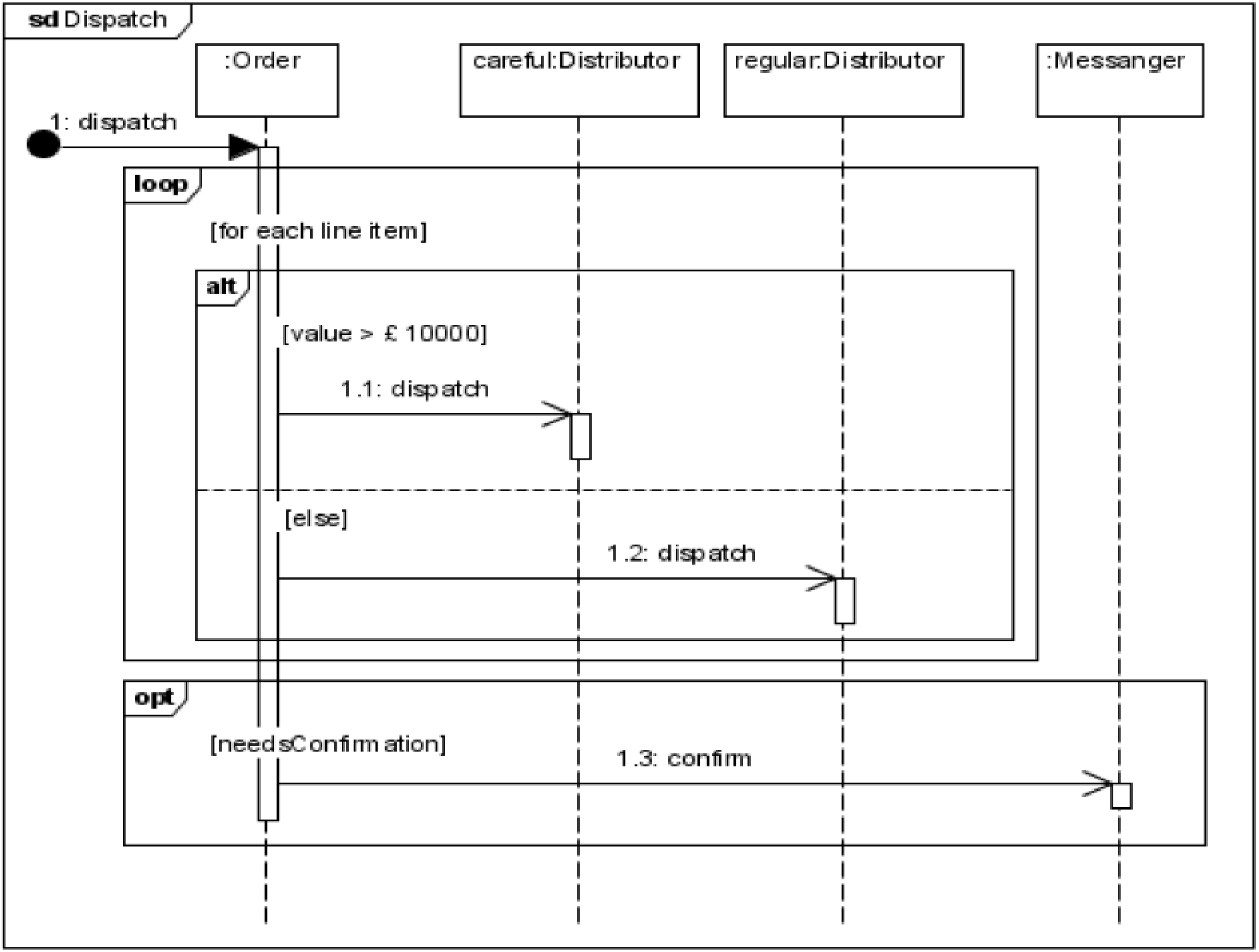
- Occasionally you will need to model a repetitive sequence.
- In UML 2, modeling a repeating sequence has been improved with the addition of the loop combination fragment.
- The loop combination fragment is very similar in appearance to the option combination fragment.
- You draw a frame, and in the frame's namebox the text "loop" is placed.
- Inside the frame's content area the loop's guard is placed towards the top left corner, on top of a lifeline.
- Then the loop's sequence of messages is placed in the remainder of the frame's content area.

Loops

- In a loop, a guard can have two special conditions tested against in addition to the standard Boolean test.
- The special guard conditions are
  - minimum iterations written as "minint = [the number]" (e.g., "minint = 1")
  - maximum iterations written as "maxint = [the number]" (e.g., "maxint = 5“)
  - With a minimum iterations guard, the loop must execute at least the number of times indicated, whereas with a maximum iterations guard the number of loop executions cannot exceed the number.

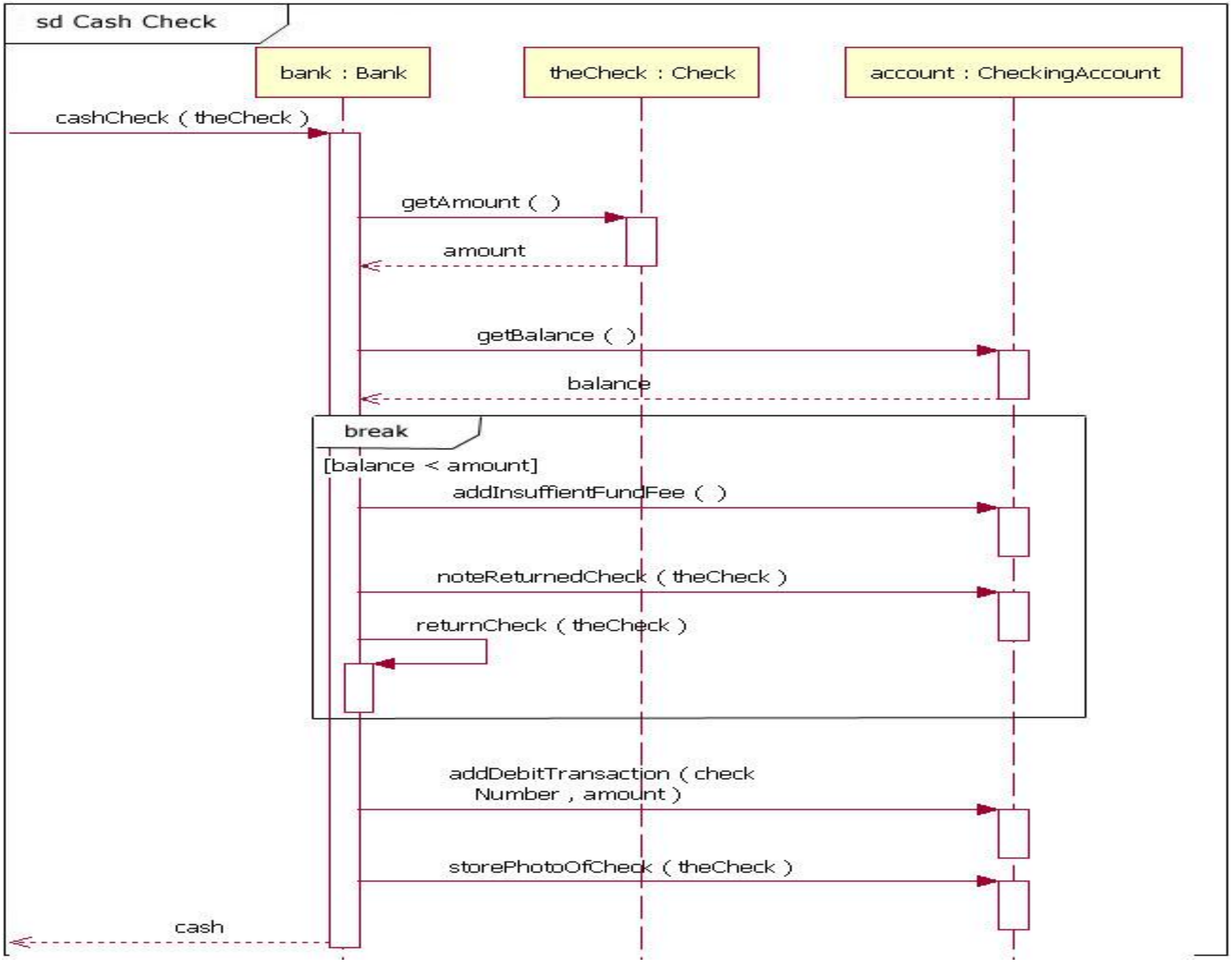


Interaction Frames



Break

- The break combined fragment is almost identical in every way to the option combined fragment, with two exceptions.
- First, a break's frame has a namebox with the text "break" instead of "option."
- Second, when a break combined fragment's message is to be executed, the enclosing interaction's remainder messages will not be executed because the sequence breaks out of the enclosing interaction.
- Breaks are most commonly used to model exception handling


sd Cash Check
bank : Bank
theCheck : Check
account : CheckingAccount
cashCheck (theCheck)
getAmount ( )
amount
getBalance ( )
balance
break
[balance < amount]
addInsuffientFundFee ( )
noteReturnedCheck (theCheck)
returnCheck (theCheck)
addDebitTransaction (check Number , amount)
storePhotoOfCheck (theCheck)
cash

Parallel

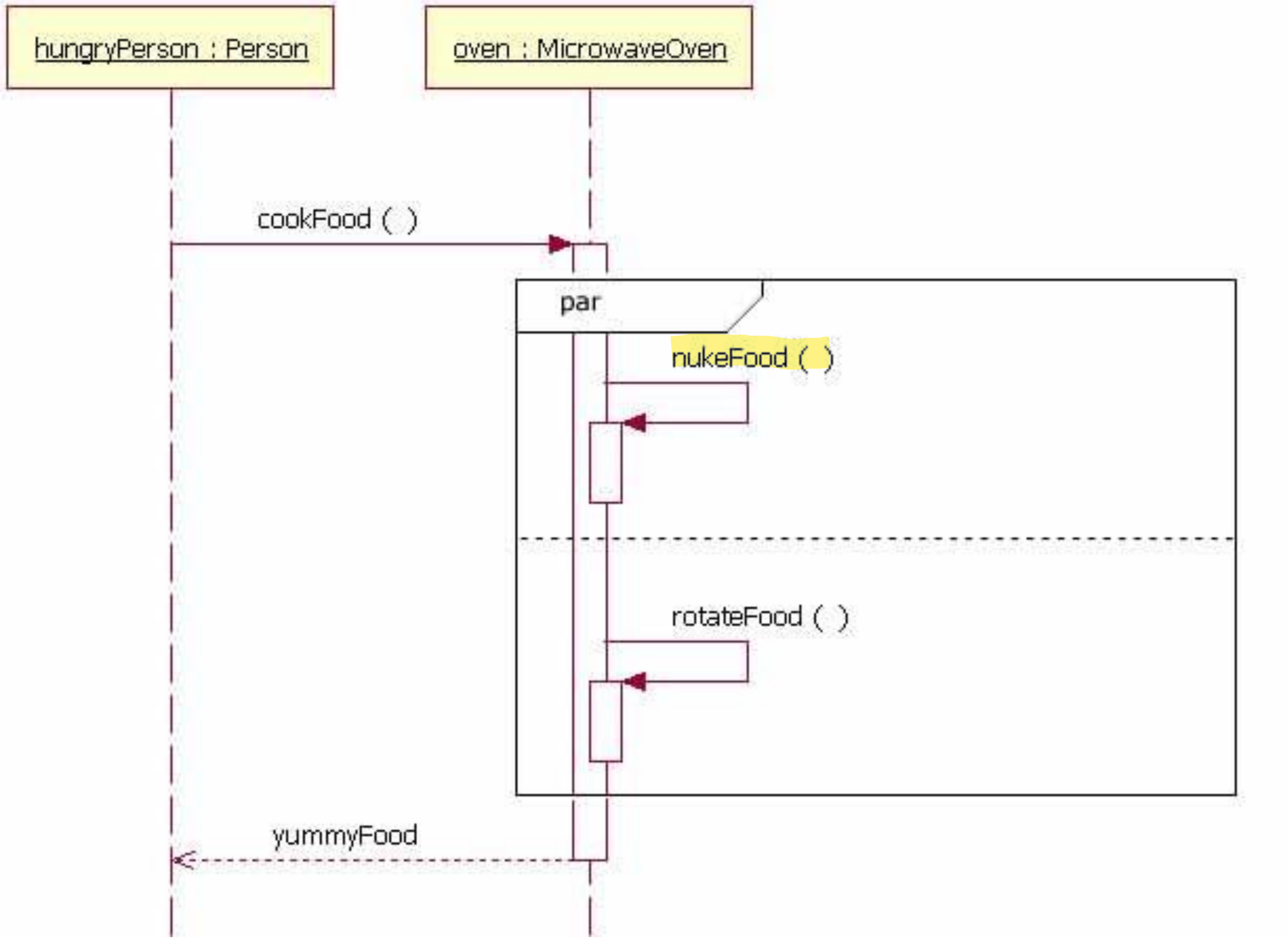
- The parallel combination fragment element needs to be used when creating a sequence diagram that shows parallel processing activities.

- The parallel combination fragment is drawn using a frame, and you place the text "par" in the frame's namebox.

- You then break up the frame's content section into horizontal operands separated by a dashed line.

- Each operand in the frame represents a thread of execution done in parallel.

Parallel


hungryPerson : Person
oven : MicrowaveOven
cookFood ( )
par
nukeFood ( )
rotateFood ( )
yummyFood