Class Diagram

What is a Class Diagram?

- A diagram that shows a set of classes, interfaces, and collaborations and their relationships

Why do we need Class Diagram?

- Focus on the conceptual and specification perspectives to avoid the premature implementation perspective all the time during your projects
- We can show the static structure of the things that exist, their internal structure, and their relationships to other things

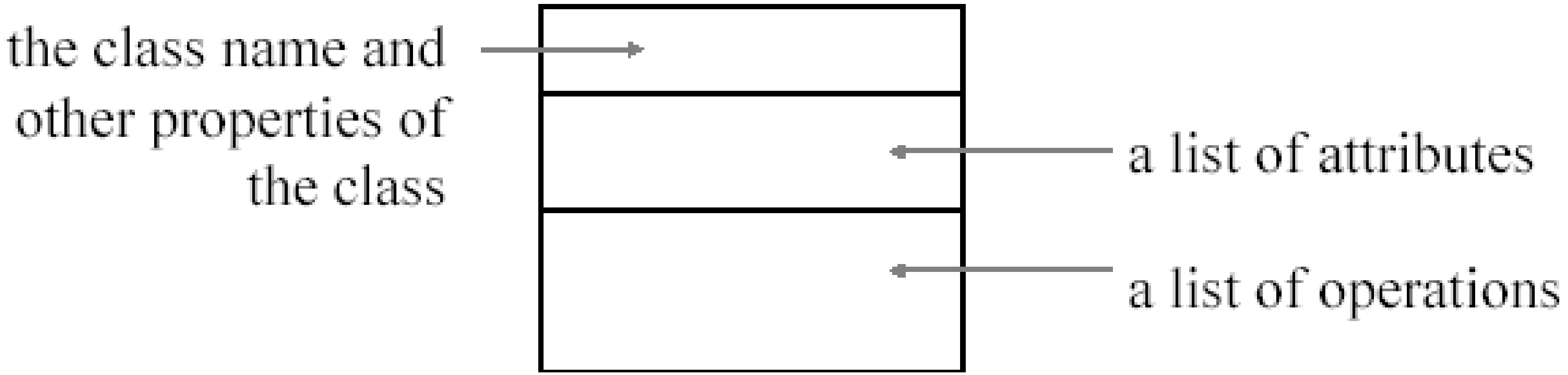
What are the main components of a Class Diagram?

- Class
  - Class name
  - Attribute
  - Operation
- Relationships
  - Generalization
  - Association
  - Aggregation
  - Dependency

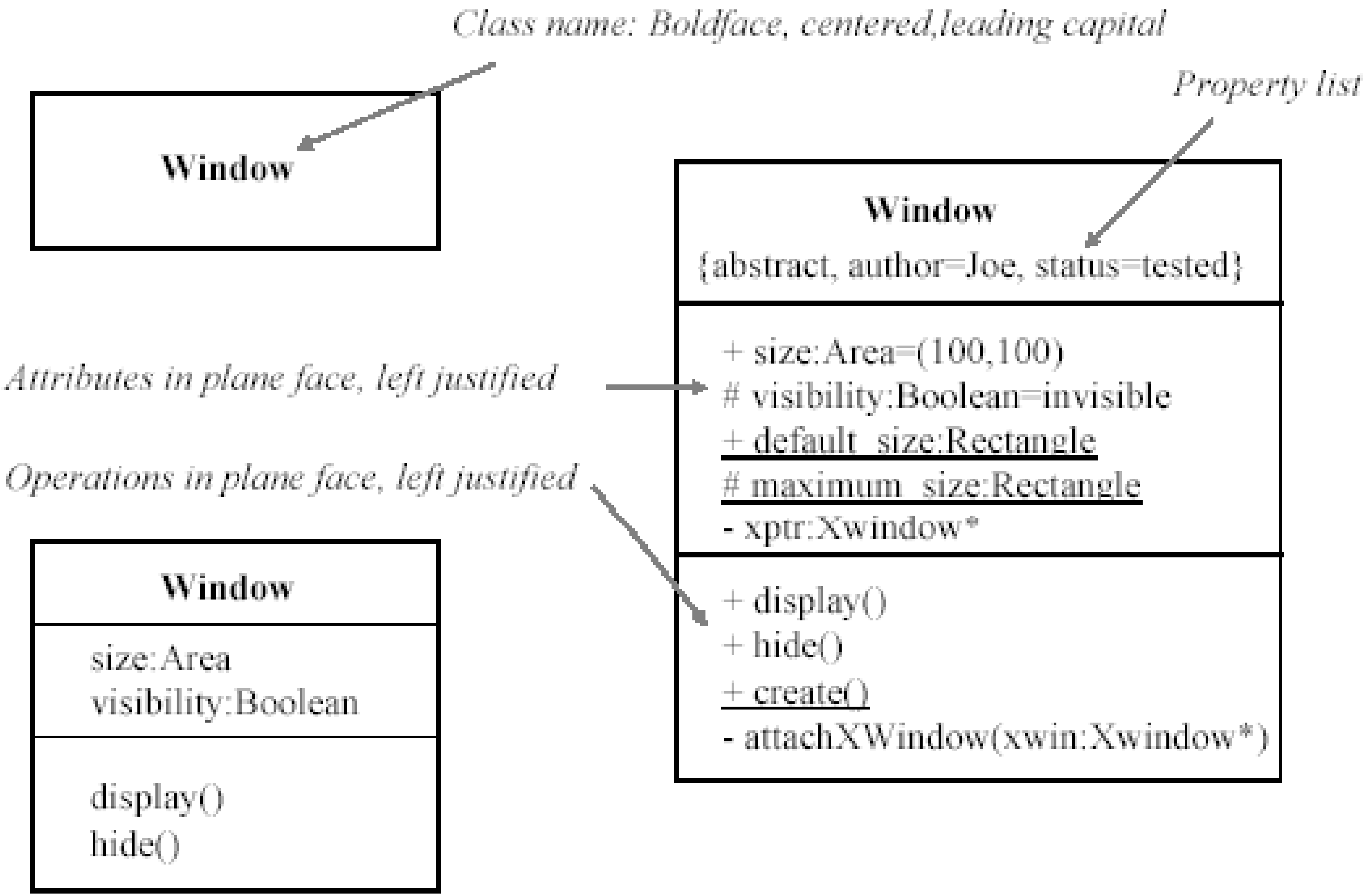
Class - Semantic

- A class is the descriptor for a set of objects with similar structure, behavior, and relationships
- Classes are declared in class diagrams and used in most other diagrams.
- The name of a class has scope within the package in which it is declared.
- The name must be unique among class names within its package

Class - Notation



Drawing a Class



Class name: Boldface, centered,leading capital

Window

Property list

Window

{abstract, author=Joe, status=tested}

+ size:Area=(100,100)
# visibility:Boolean=invisible
+ default_size:Rectangle
# maximum_size:Rectangle
- xptr:Xwindow*

+ display()
+ hide()
+ create()
- attachXWindow(xwin:Xwindow*)

Attributes in plane face, left justified

Operations in plane face, left justified

Window

size:Area
visibility:Boolean

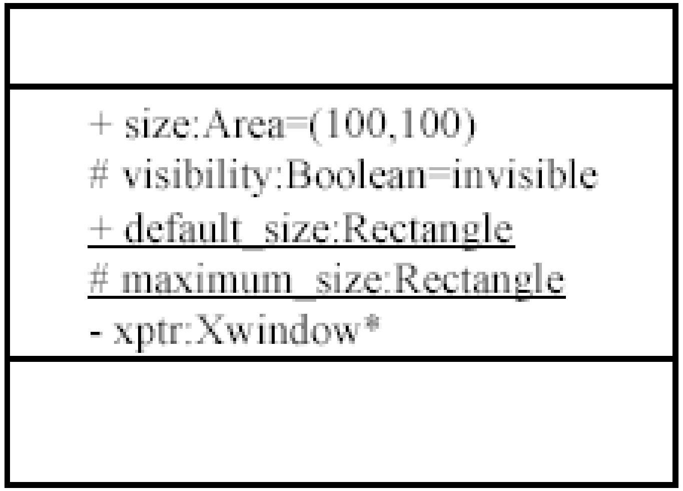
display()
hide()

Attributes - Semantic

- At the conceptual level, it is equivalent to a composition association
- Attributes are always single-valued
- Any type used as an attribute has value rather than references

Attribute - Notation

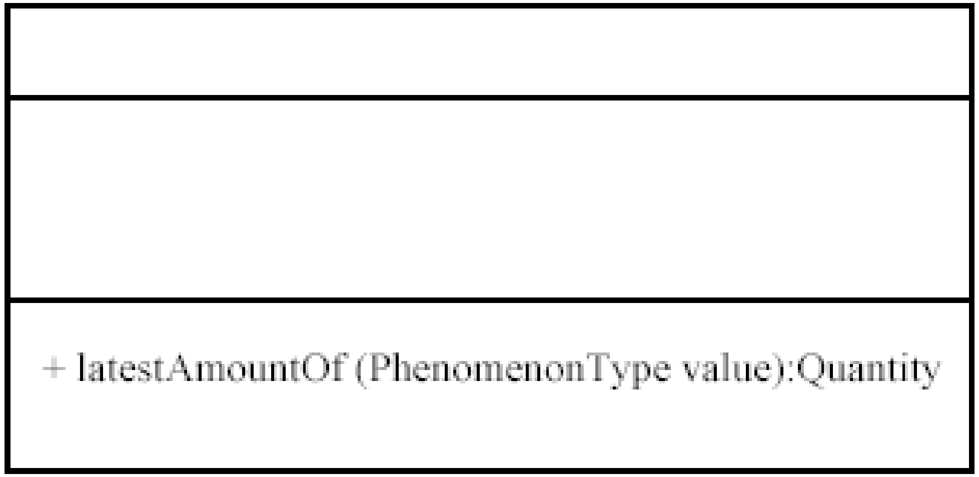
- Notation
  - visibility name:type-expression = initial-value {property-string}
    - Visibility: + public, # protected, - private
    - class-scope attribute and instance-scope attribute



+ size:Area=(100,100)
# visibility:Boolean=invisible
+ default_size:Rectangle
# maximum_size:Rectangle
- xptr:Xwindow*

Operation – Semantic & Notation

- Semantics
  - Public methods on a type
  - Within conceptual models, indicate the principal responsibilities of classes.
- Notation
  - visibility name (parameter-list):return-type-expression {property-string}



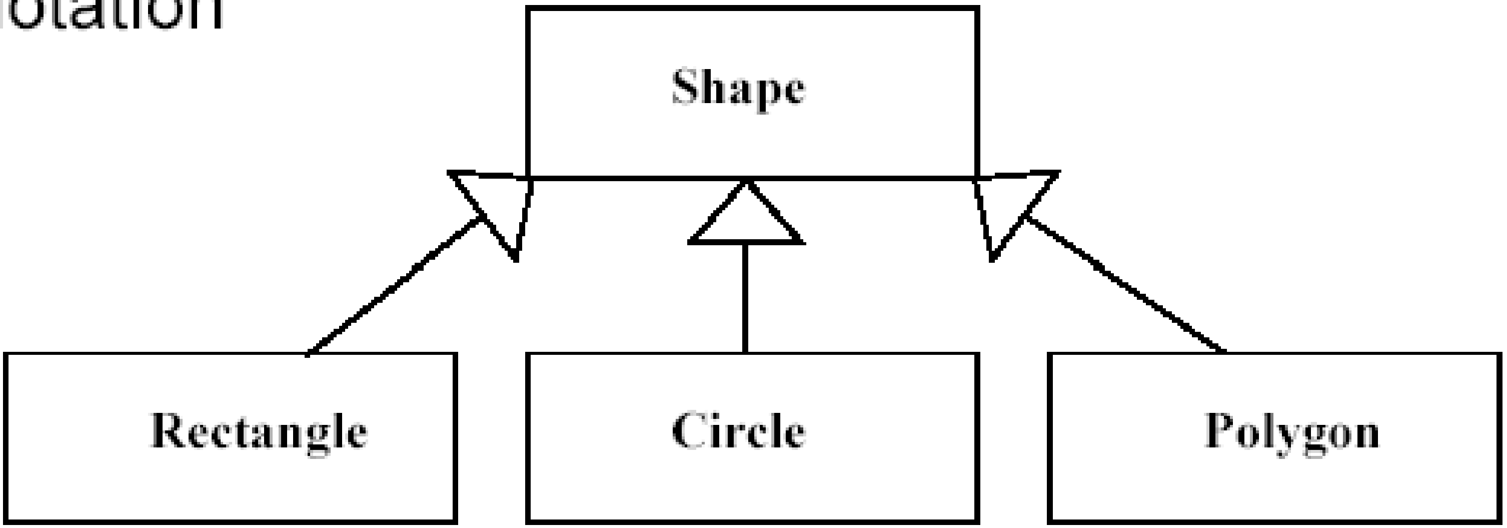
+ latestAmountOf (PhenomenonType value):Quantity

Generalization

- A relationship between a general thing and a more specific kind of that thing
- “is-a-kind-of” relationship

Generalization – Semantic & Notation

- Semantics
  - At the conceptual level, similarities among classes.
  - At the specification level, subtyping or interface-inheritance
  - At the implementation level, subclassing or implementation-inheritance.
- Notation

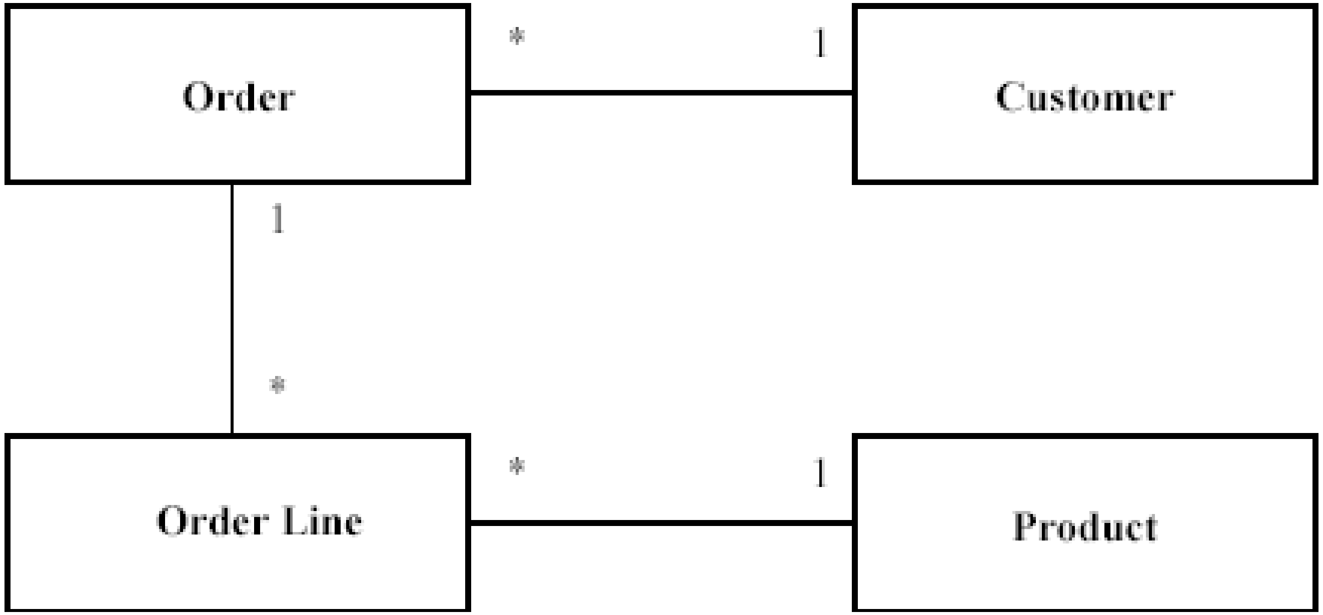


Association

- A structural relationship that specifies that objects of one thing are connected to objects of other
- At the conceptual level, associations represent conceptual relationships between classes
- At the specification level, associations represent responsibilities
- At the implementation level, associations represent navigability.

Association - Example

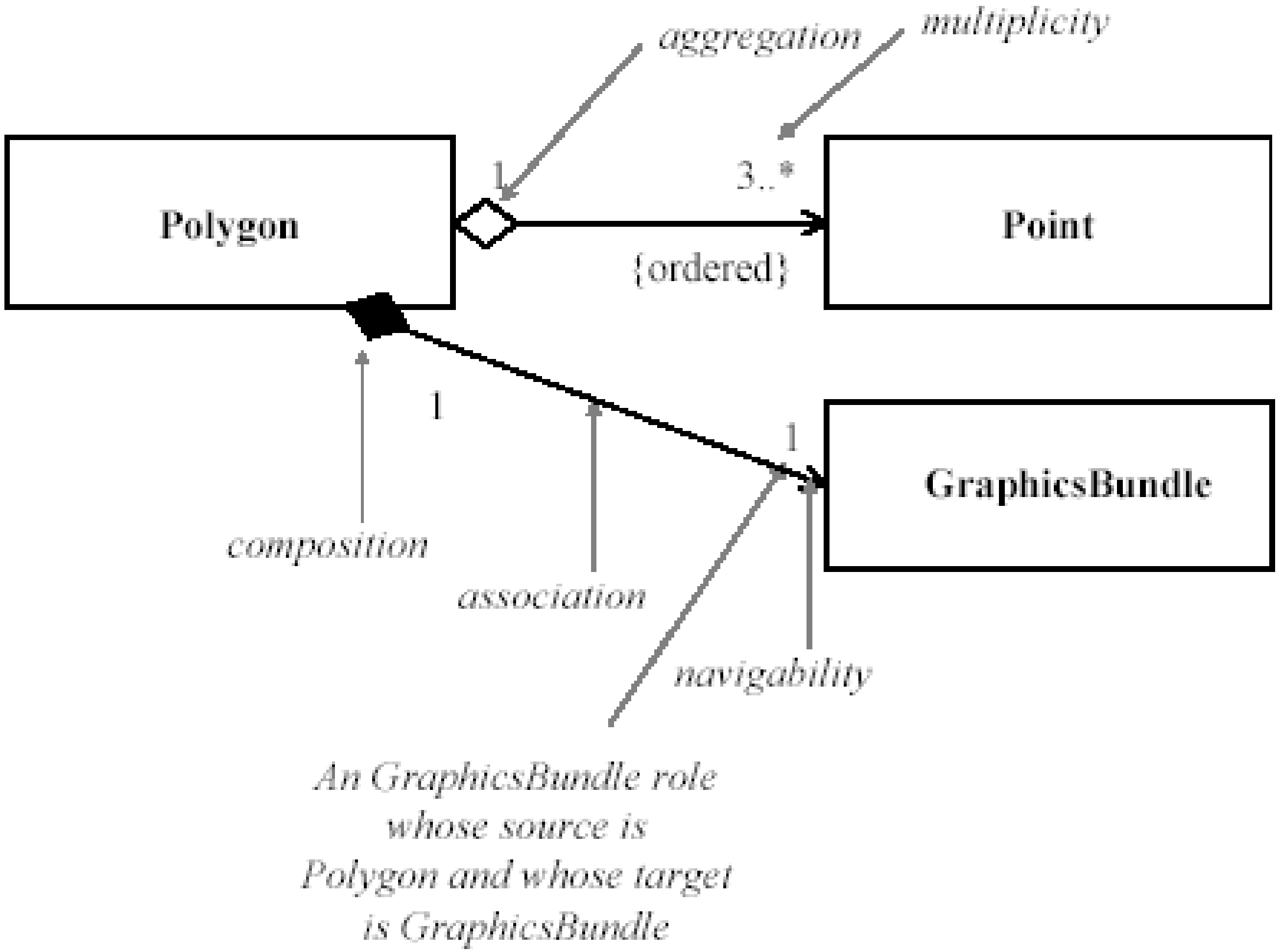
- Example
  - An Order has to come from a single Customer. A Customer may several Orders over time. Each of these Orders has several Order Lines, each of which refers to a single Product.



Association

- Role name
  - Each association has two roles; each role is a direction on the association
  - A role can be explicitly named with a label. It there is no label, you name a role after the target class
- Multiplicity
- Navigability
- Aggregation/Composition indicators
- Ordering

Association



Dependency

- A using relationship that states that a change in specification of one thing may affect another thing that uses it, but not necessarily the reverse
- One class uses another class as an argument in the signature of an operation

Sequence Diagram

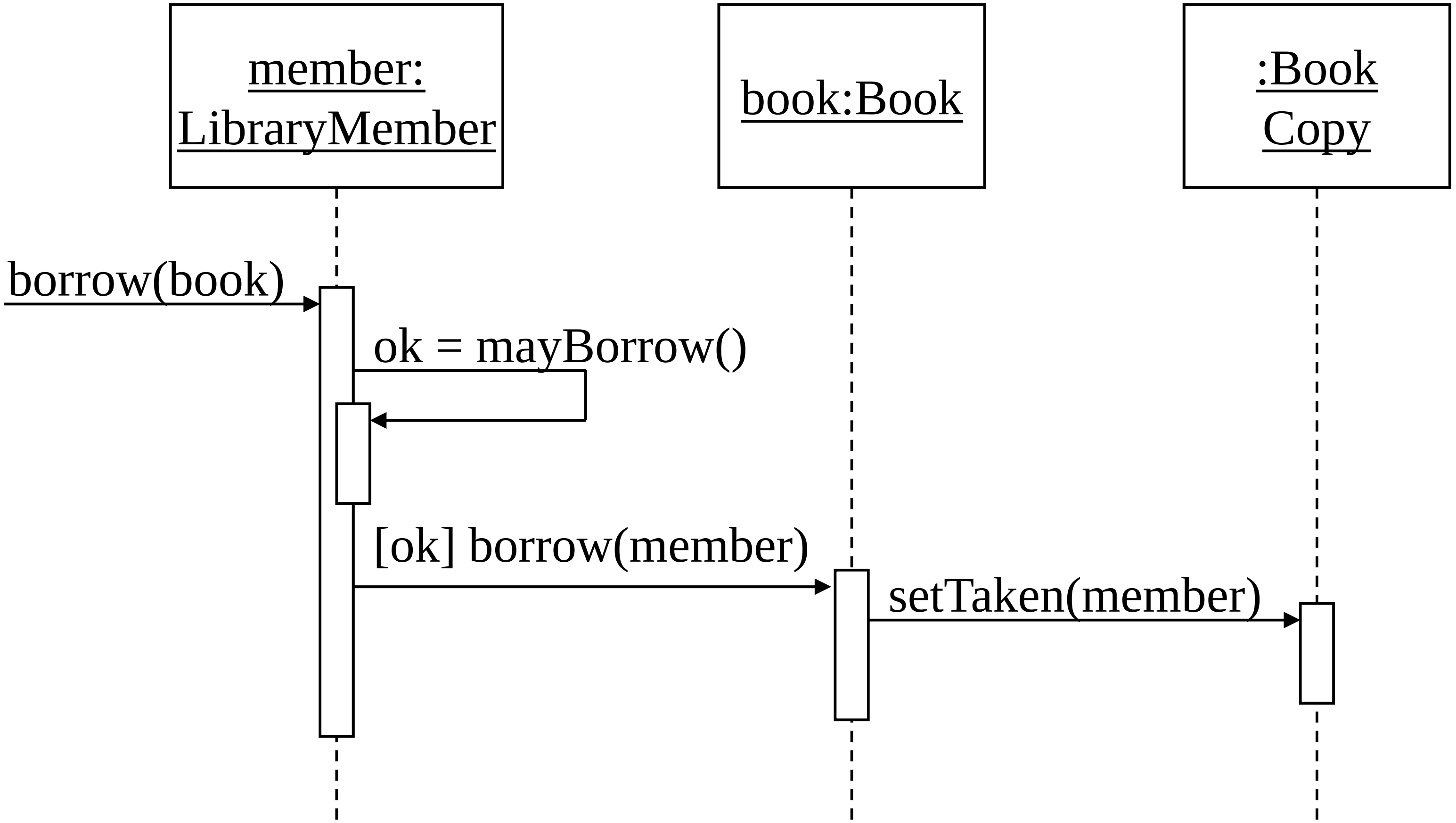
A First Look at Sequence Diagrams

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

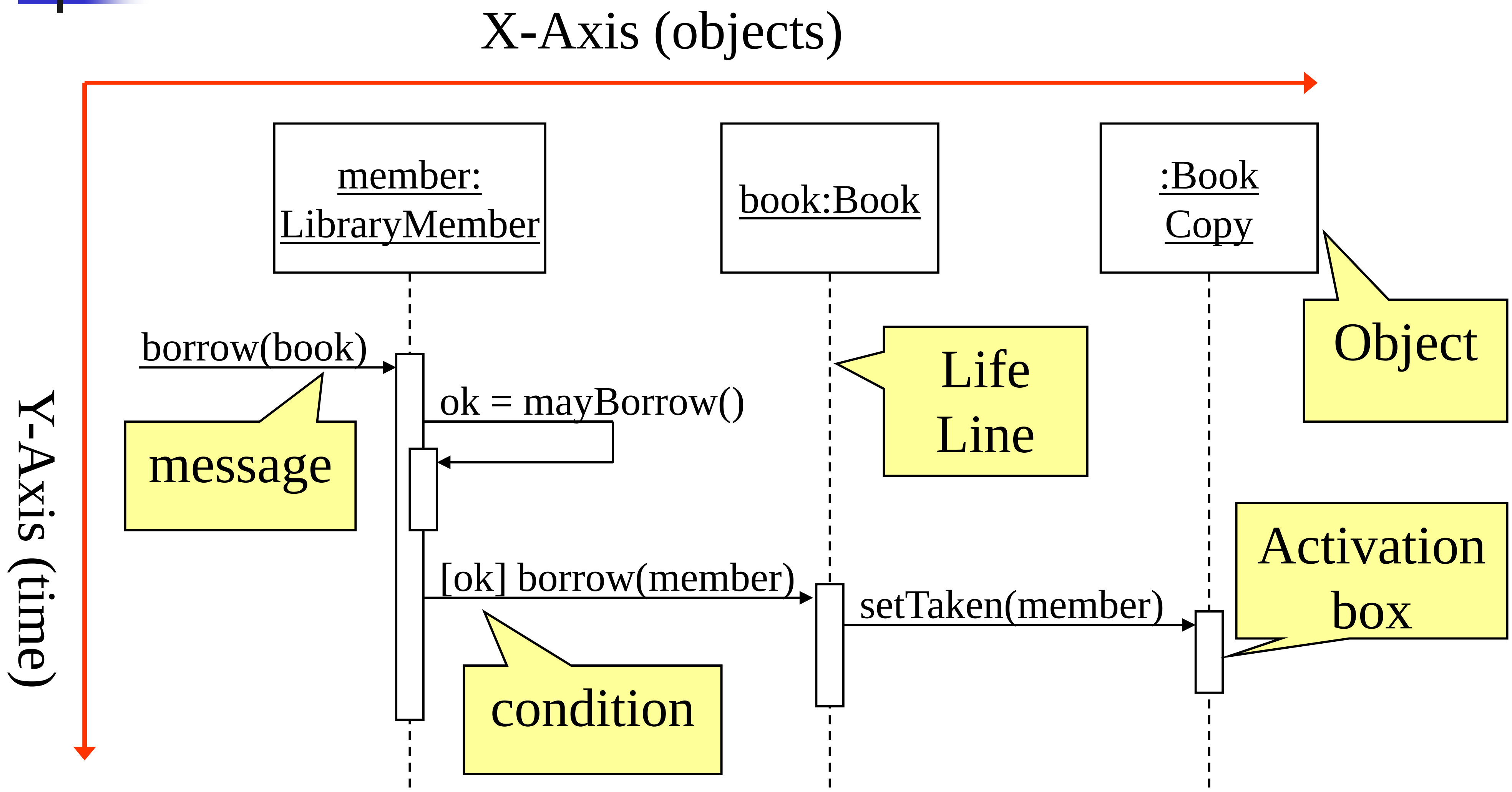
Sequence Diagram

- Interaction diagrams come in two forms, both present in the UML. The first form is the sequence diagram. In this form objects are shown as vertical lines with the messages as horizontal lines between them.
- This form was first popularized by Jacobson

A Sequence Diagram

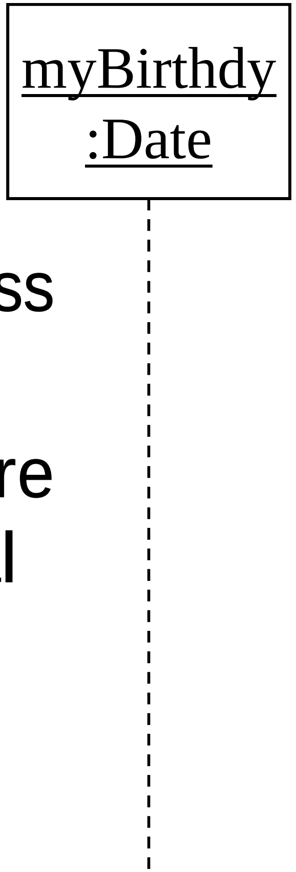


A Sequence Diagram

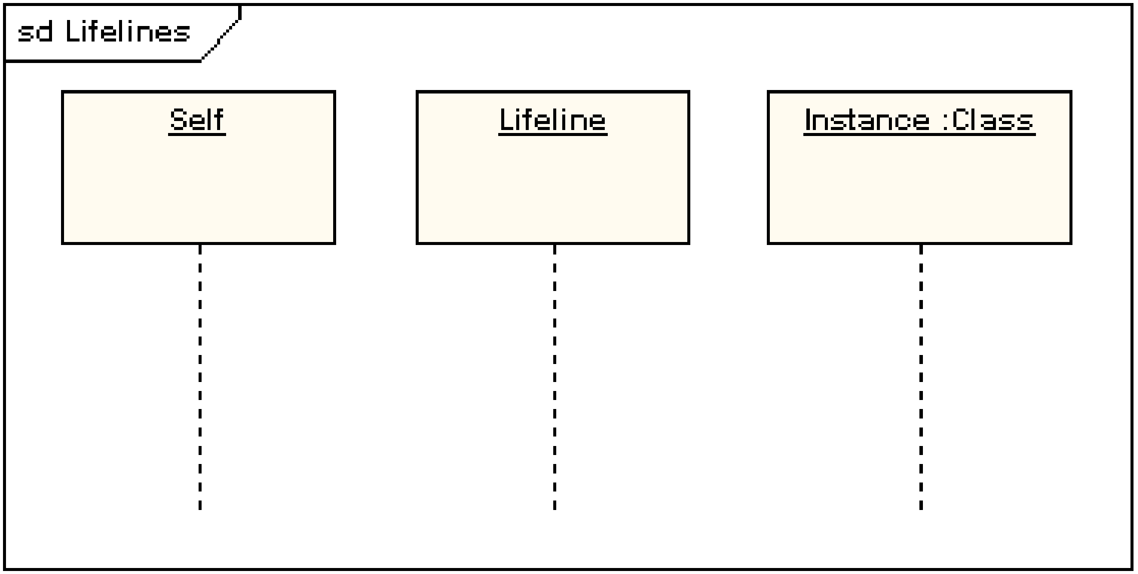


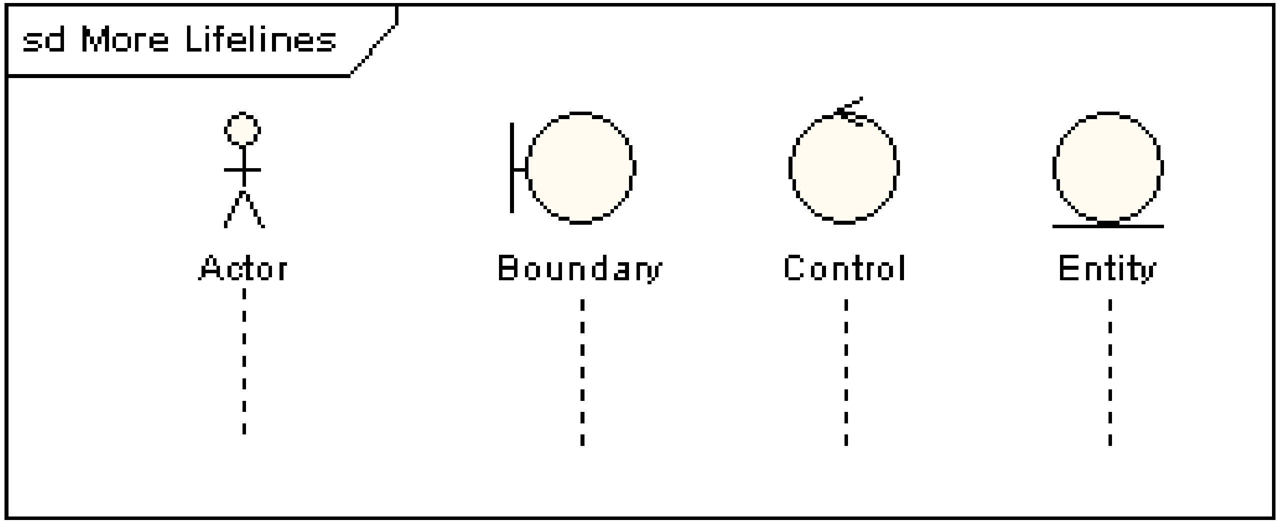
Object

- 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.
- The *Life-Line* represents the object's life during the interaction



Lifeline Samples





Messages

- An interaction between two objects is performed as a message sent from one object to another.
  - Most often implemented by a simple operation call.
  - Can be an actual message sent through some communication mechanism, either over the network or internally on a computer.
    - Inter-process communication (Signaling, ...)
    - Remote Procedure Call (RMI, CORBA, ...)

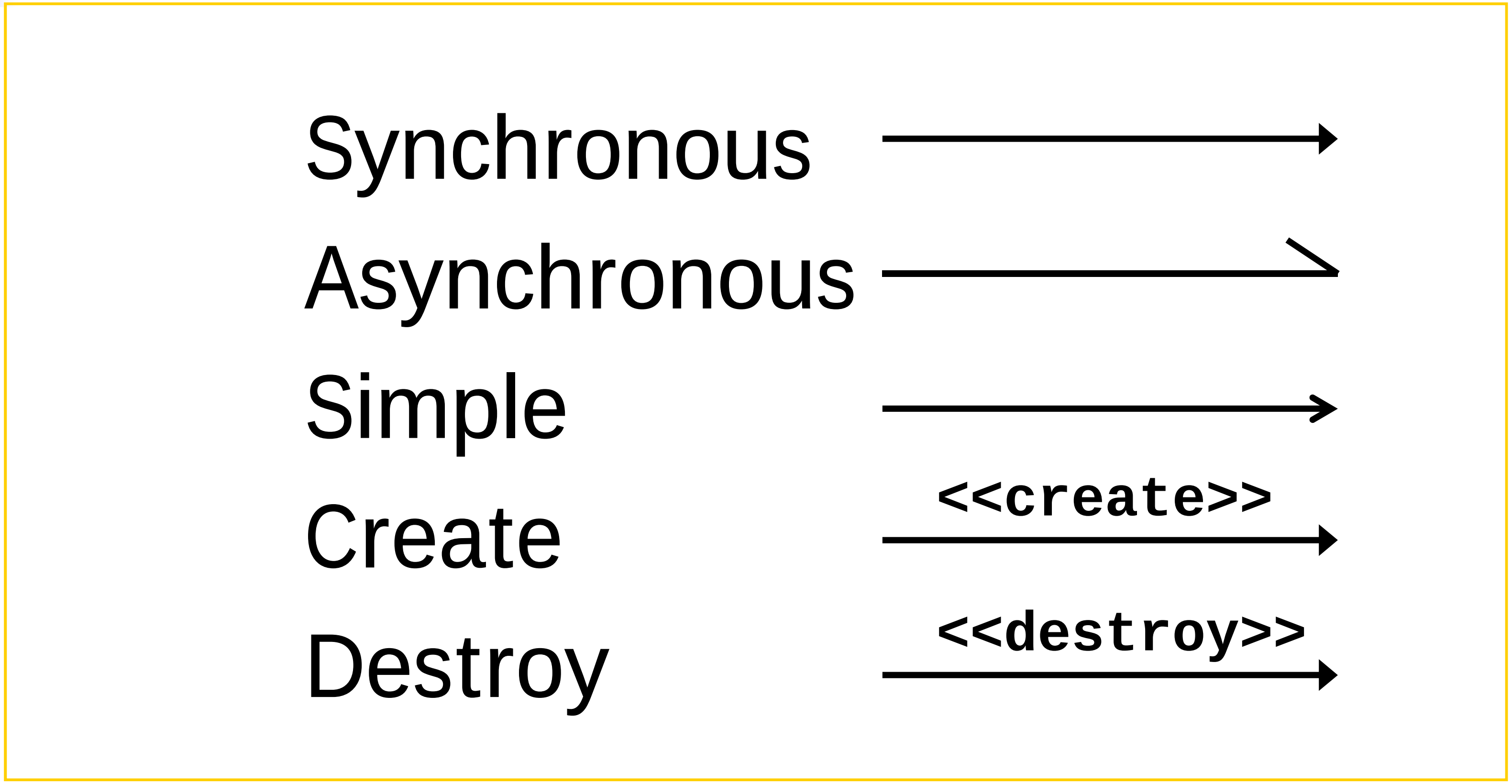
Messages (Cont.)

- If object obj_1 sends a message to another object obj_2 an association must exist between those two objects:
  - Structural dependency
  - obj_2 is in the global scope of obj_1
  - obj_2 is in the local scope of obj_1 (method argument)
  - obj_1 and obj_2 are the same object

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.
  - Prefer using a brief textual description whenever an *actor* is the source or the target of a message.

Message Types



Synchronous

Asynchronous

Simple

Create

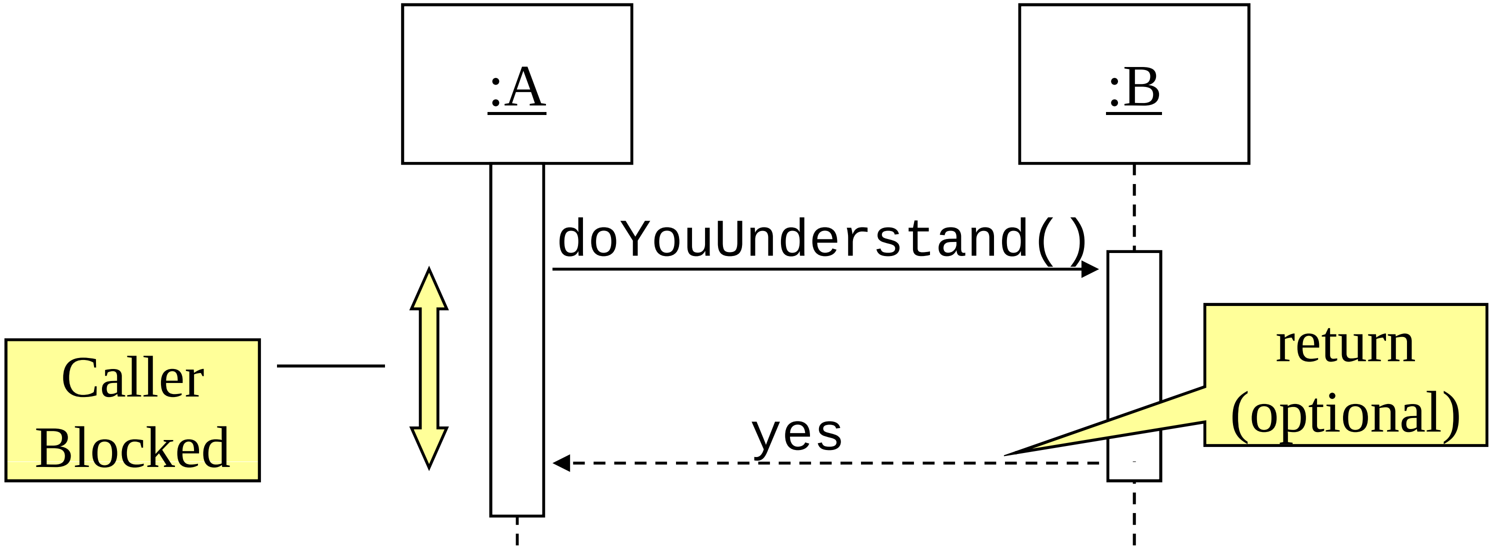
<<create>>

Destroy

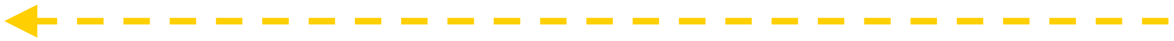
<<destroy>>

Synchronous Messages

- Nested flow of control, typically implemented as an operation call.
  - The routine that handles the message is completed before the caller resumes execution.



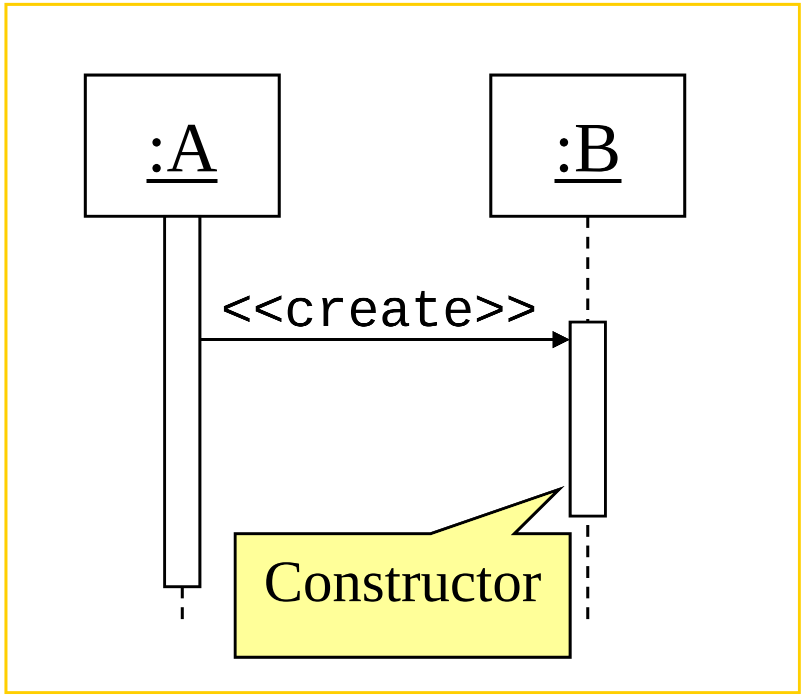
Return Values

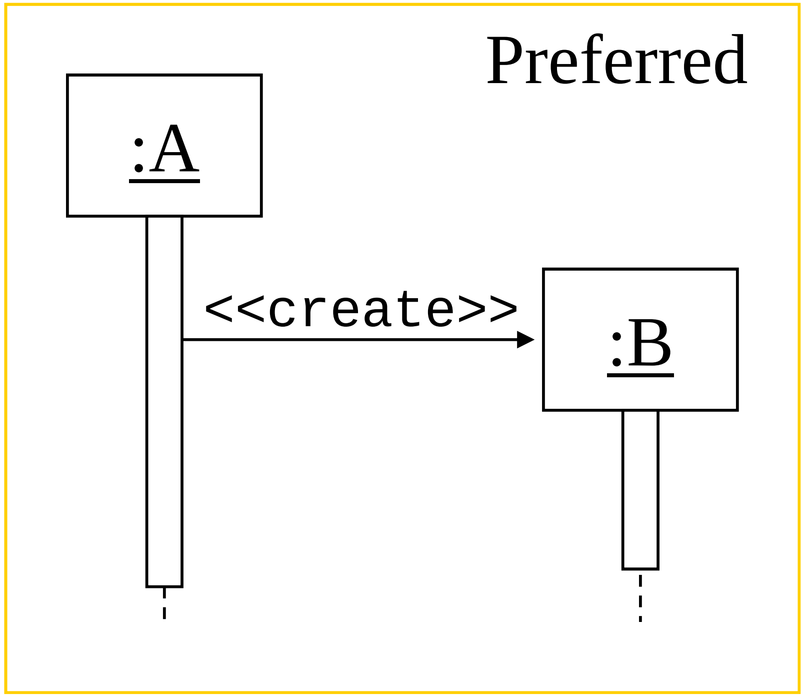


- Optionally indicated using a dashed arrow with a label indicating the return value.
  - Don’t model a return value when it is obvious what is being returned, e.g. getTotal()
  - Model a return value only when you need to refer to it elsewhere, e.g. as a parameter passed in another message.
  - Prefer modeling return values as part of a method invocation, e.g. `ok = isValid()`

Object Creation

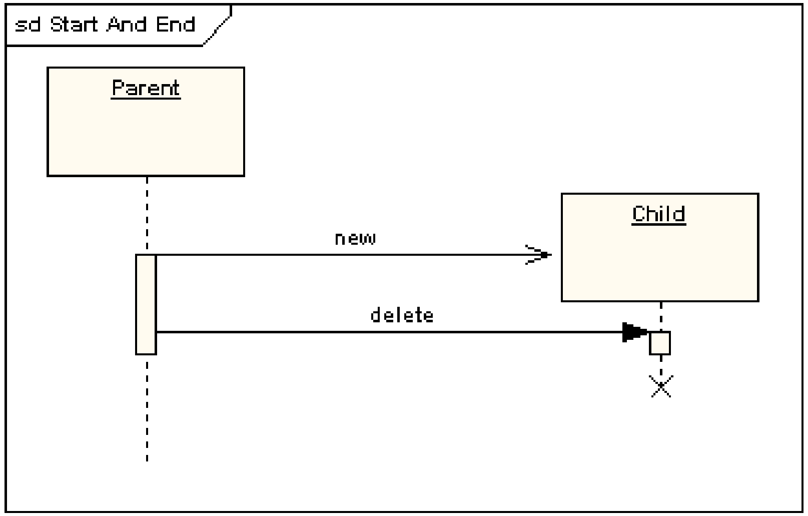
- An object may create another object via a `<<create>>` message.

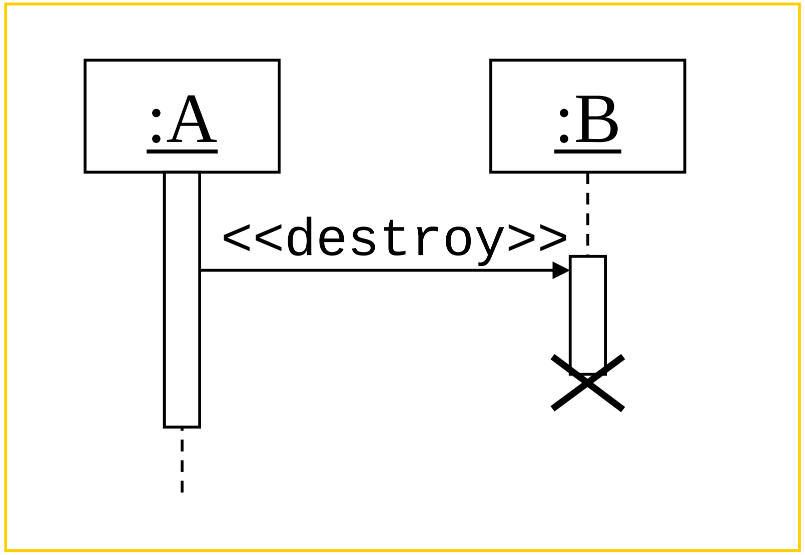




Object Destruction

- An object may destroy another object via a `<<destroy>>` message.
  - An object may destroy itself.
  - Avoid modeling object destruction unless memory management is critical.



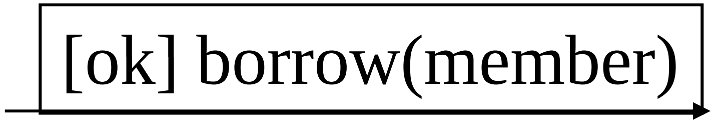


Asynchronous Messages

- Used for modeling concurrent systems.
- Caller does not wait for the message to be handled before it continues to execute.
  - As if the call returns immediately
- *Active objects* own an execution thread and can initiate control activity.
- An asynchronous message can:
  - Create a new thread (a new activation record)
  - Create a new object
  - Communicate with a thread that is already running.

Control information

- Condition
  - syntax: ‘[‘ expression ’]’ message-label
  - The message is sent only if the condition is true
  - example:

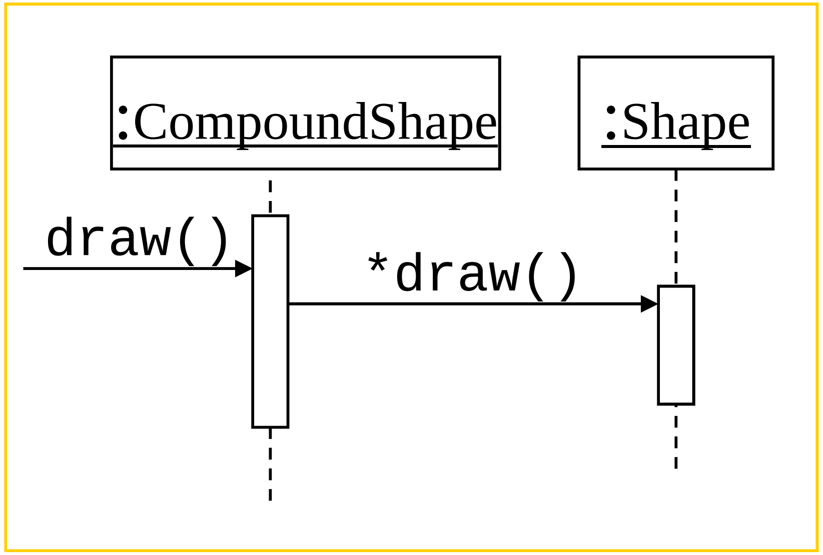


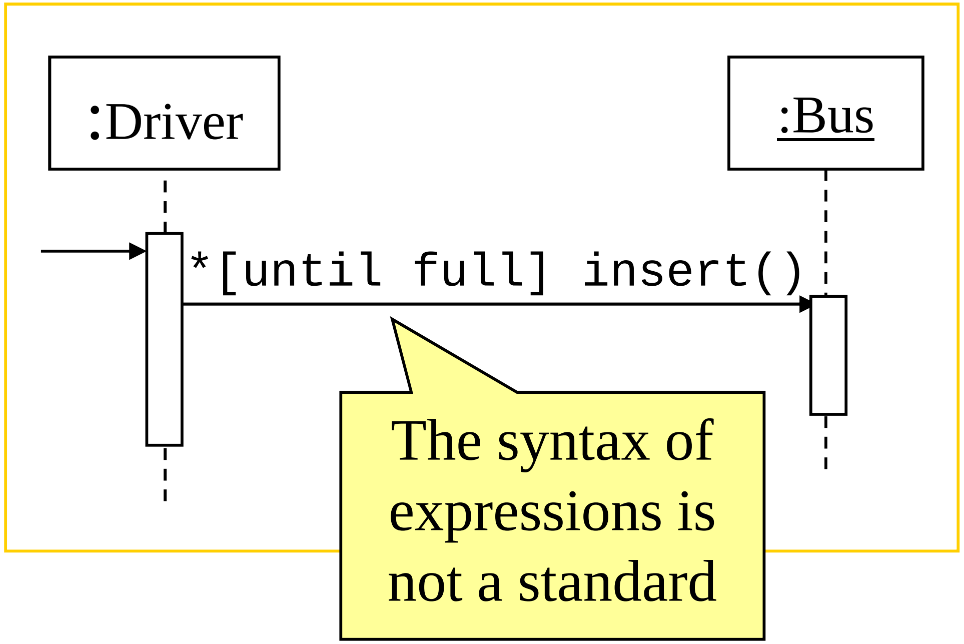
[ok] borrow(member)

- Iteration
  - syntax: * [‘[‘ expression ‘]’] message-label
  - The message is sent many times to possibly multiple receiver objects.

Control Information (Cont.)

- Iteration examples:





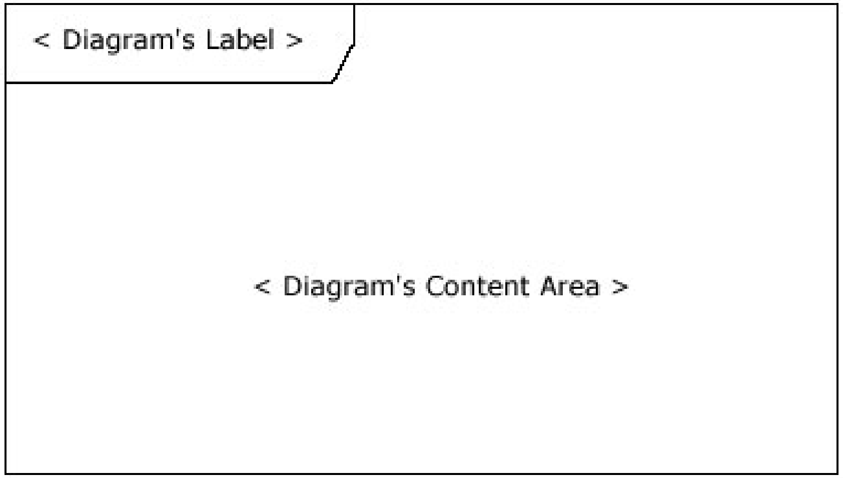
The syntax of expressions is not a standard

Control Information (Cont.)

- The control mechanisms of sequence diagrams suffice only for modeling simple alternatives.
  - Consider drawing several diagrams for modeling complex scenarios.
  - Don’t use sequence diagrams for detailed modeling of algorithms (this is better done using *activity diagrams, pseudo-code* or *state-charts*).

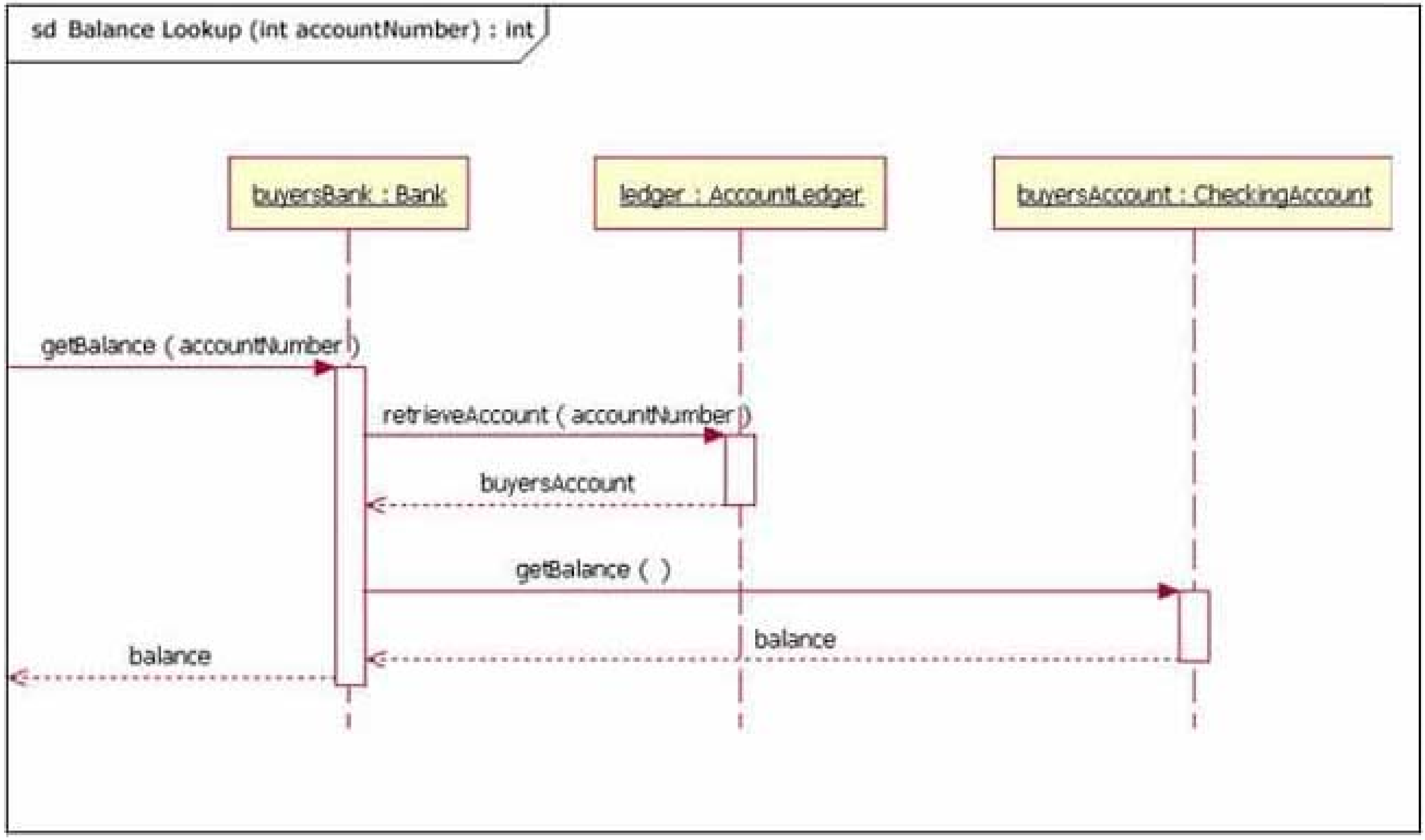
Frame Element

- Frame Element is the optional graphical boundary of a diagram
- A frame element provides a consistent place for a diagram's label, while providing a graphical boundary for the diagram
- The diagram's label is placed in the top left corner, called the frame's "namebox," a sort of dog-eared rectangle, and the actual UML diagram is defined within the body of the larger enclosing rectangle.

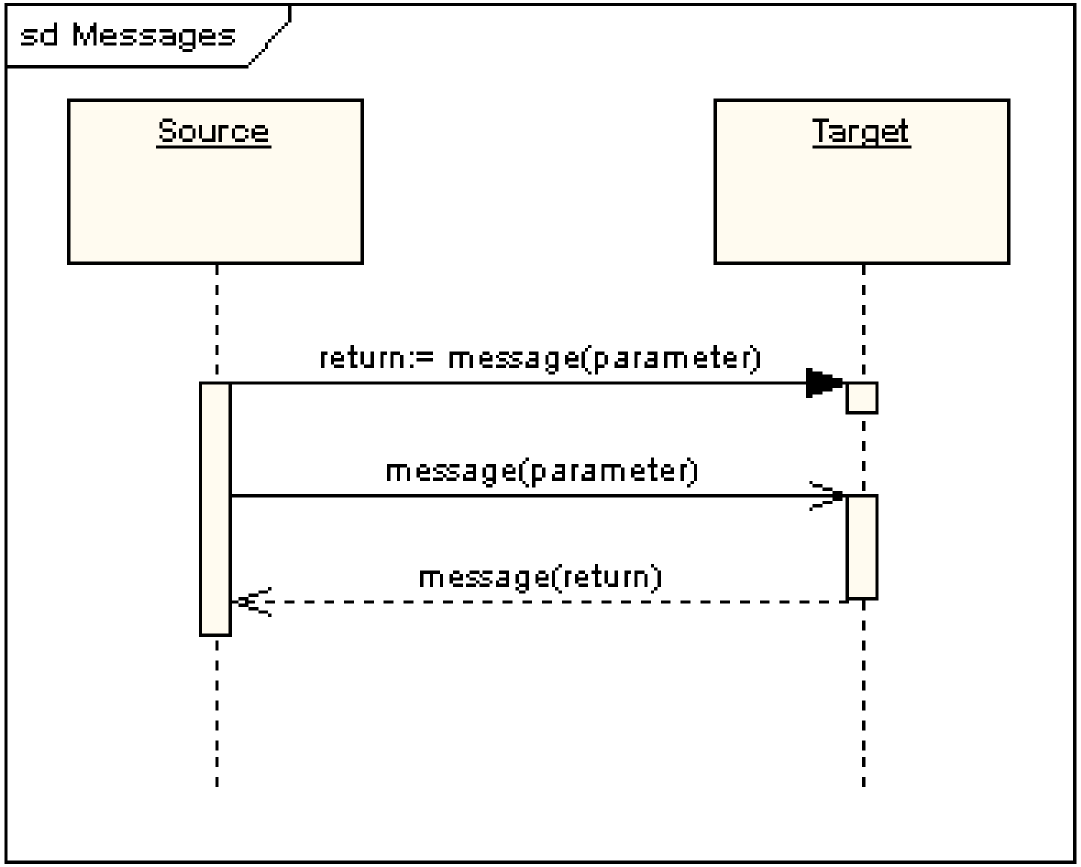


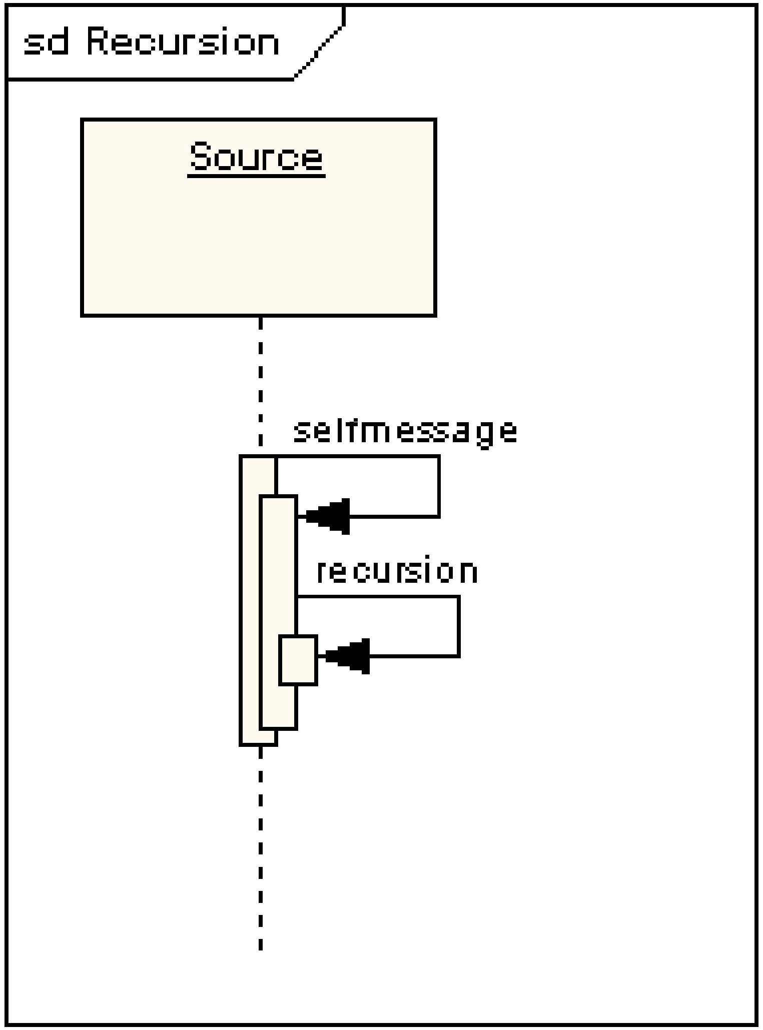
Incoming and Outgoing Message at the border of the frame element

- On sequence diagrams incoming and outgoing messages (a.k.a. interactions) for a sequence can be modeled by connecting the messages to the border of the frame element

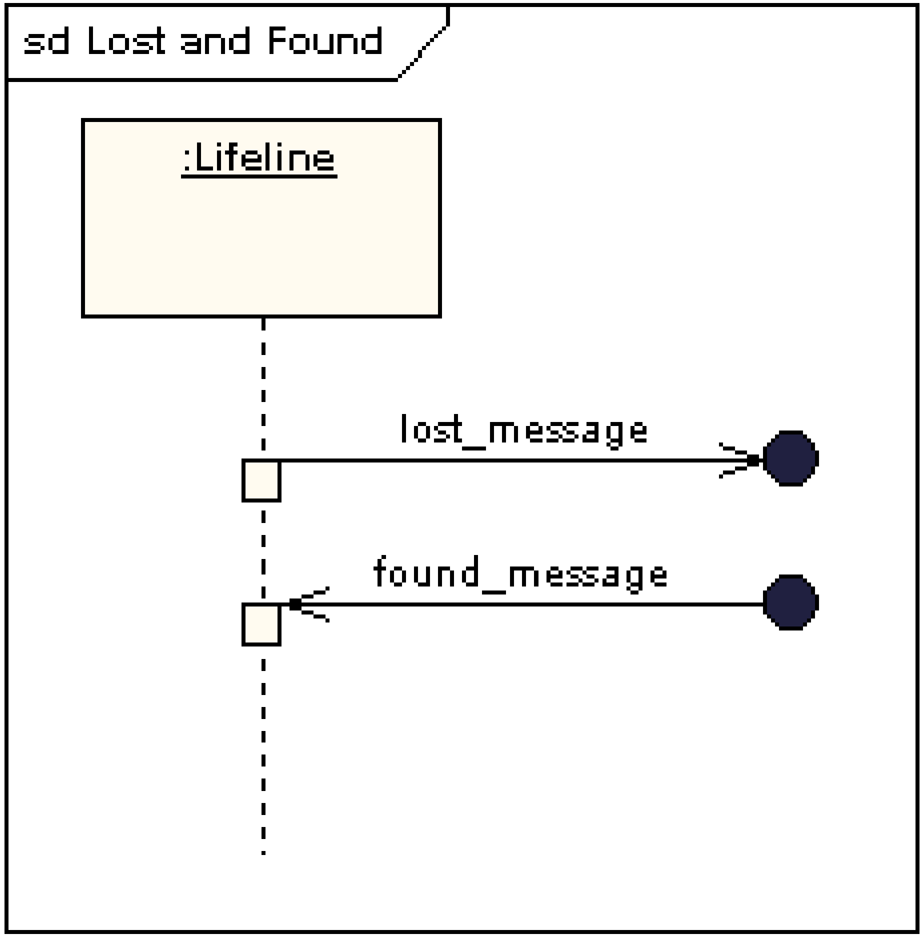

sd Balance Lookup (int accountNumber) : int
buyersBank : Bank
ledger : AccountLedger
buyersAccount : CheckingAccount
getBalance (accountNumber)
retrieveAccount (accountNumber)
buyersAccount
getBalance ( )
balance
balance

Message Sample

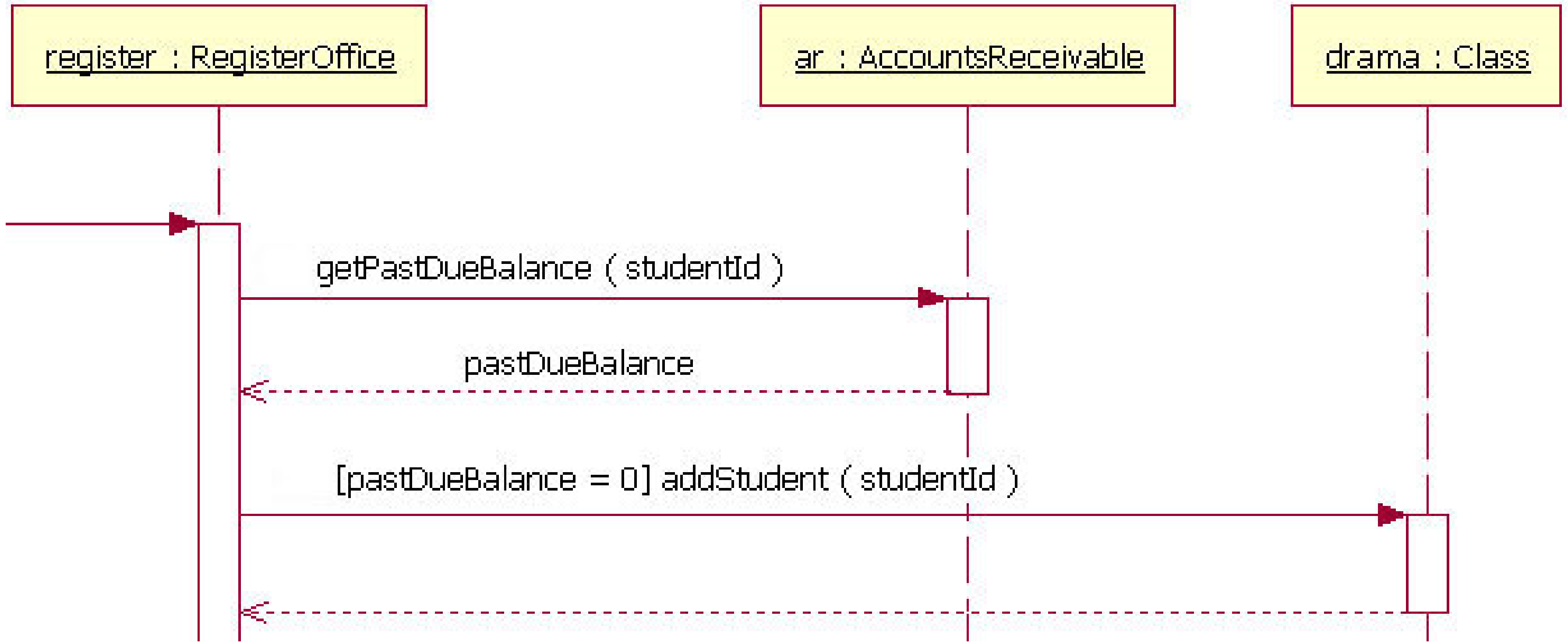




Lost and Found Message



- UML 2.0 introduced the concept of messages that do not reach their destination (lost messages)or are from unknown sources (found messages)
- Note that lost and found are relative terms ; the receiver or sender might only be unknown with respect to your current sequence diagram
- Lost messages are commonly used to show how a system handles a network failure resulting in an undelivered message
- Found messages are commonly used for modeling exception handling —you don ’t necessarily care who threw the exception ; you simply want to show how it is handled

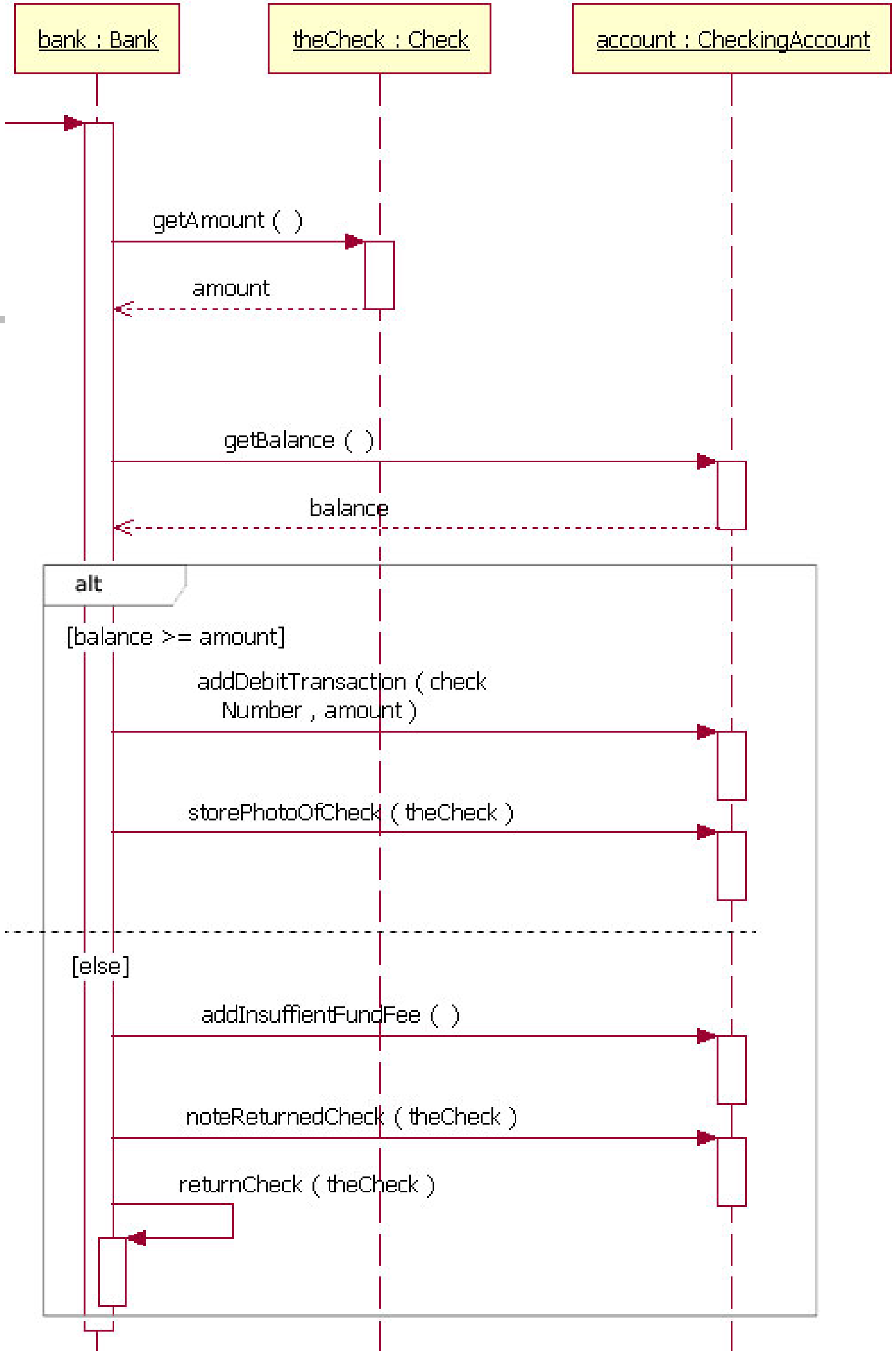

register : RegisterOffice
ar : AccountsReceivable
drama : Class
getPastDueBalance (studentId)
pastDueBalance
[pastDueBalance = 0] addStudent (studentId)

Combined Fragments

- The sequence diagrams are not intended for showing complex procedural logic
- However, there are a number of mechanisms that do allow for adding a degree of procedural logic to diagrams and which come under the heading of combined fragments.
- A combined fragment is one or more processing sequence enclosed in a frame and executed under specific named circumstances. The fragments available are:
- Alternative fragment (denoted “alt”) models if...then...else constructs.
- Option fragment (denoted “opt”) models switch constructs.
- Break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram.
- Parallel fragment (denoted “par”) models concurrent processing.
- Consider fragment is in effect the opposite of the ignore fragment: any message not included in the consider fragment should be ignored.
- Loop fragment encloses a series of messages which are repeated.

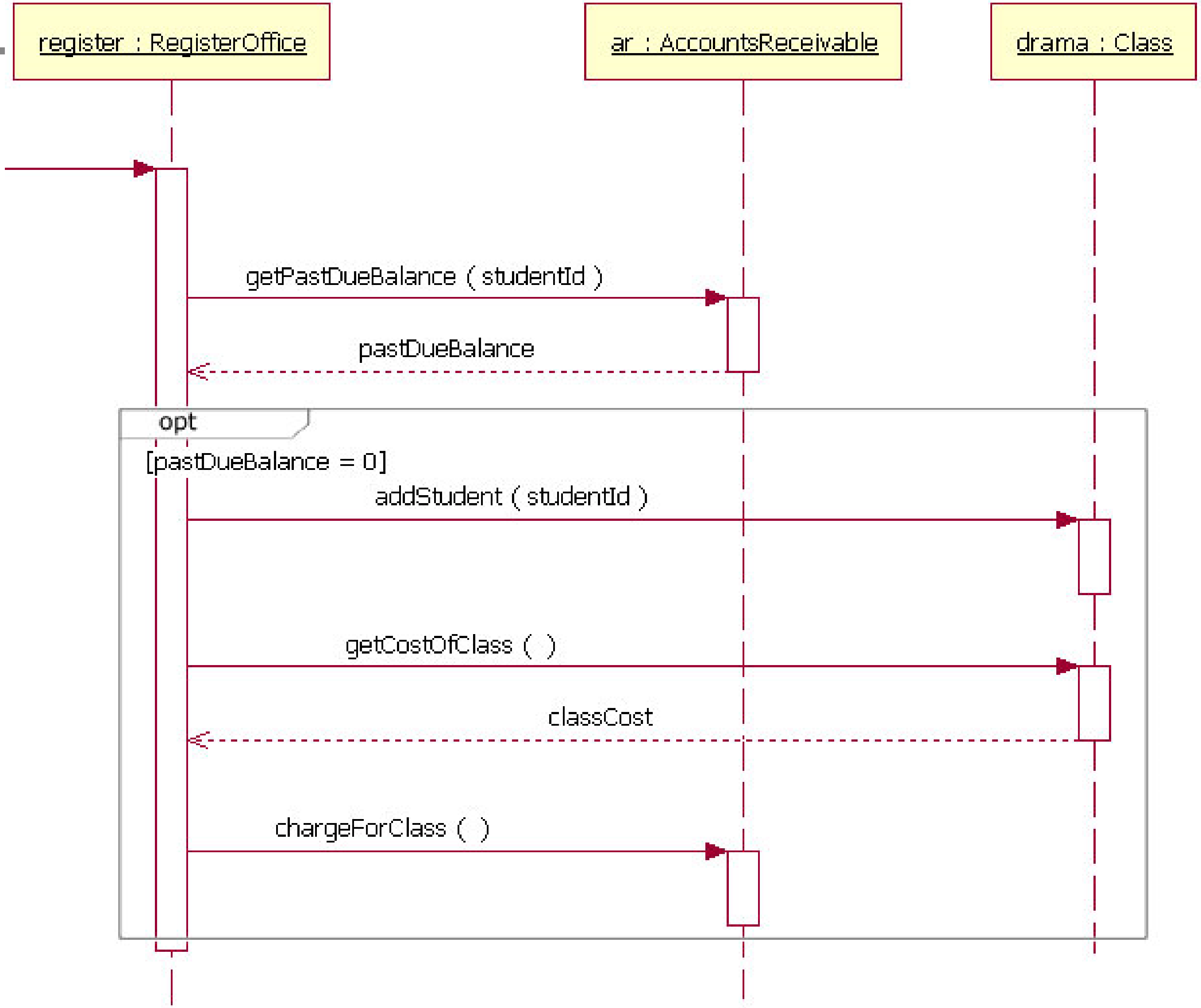
Alternative Fragments

- 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



Option Fragments

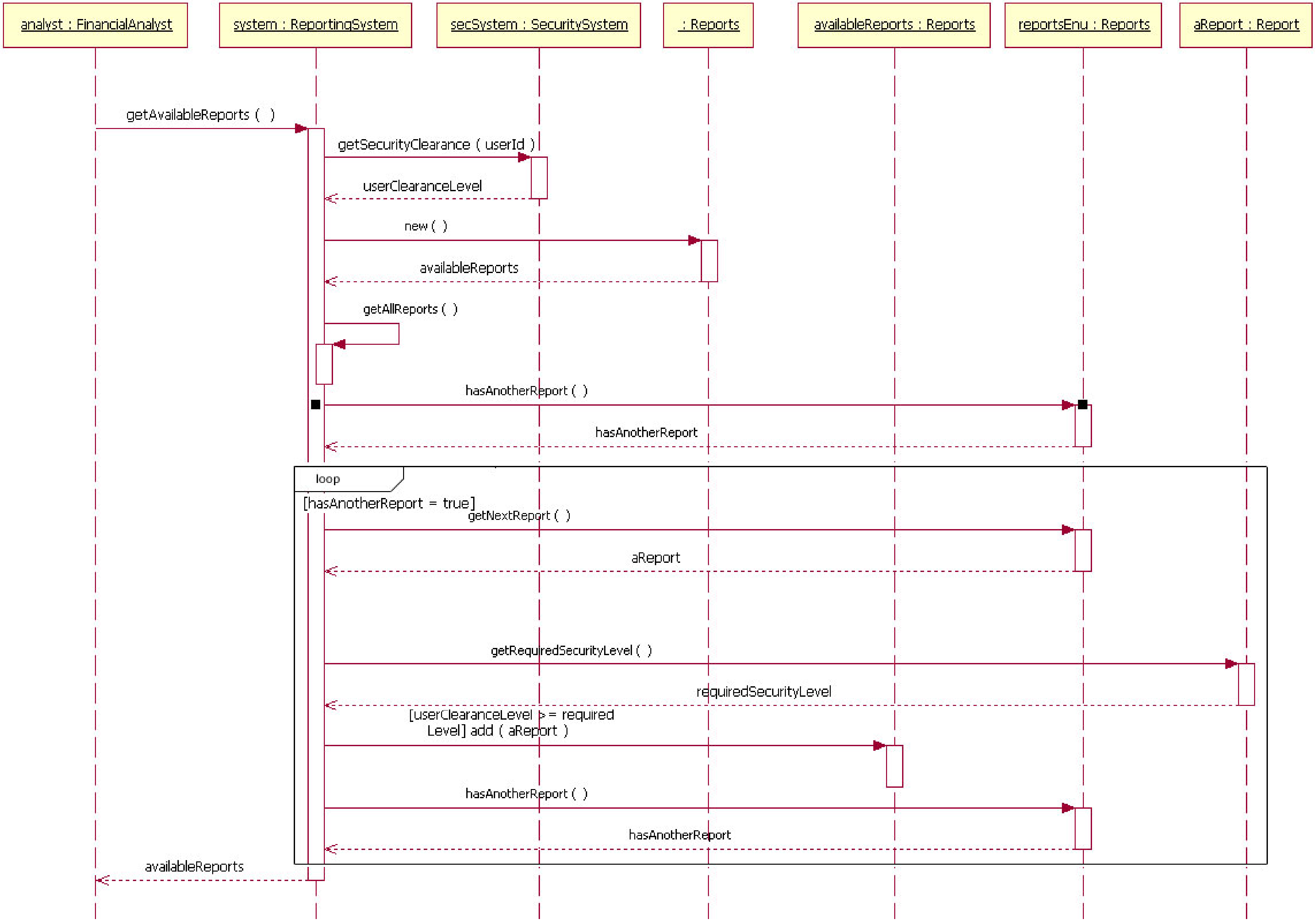
- The option combination fragment is used to model a sequence that, given a certain condition, will occur; otherwise, the sequence does not occur
- An option is used to model a simple "if then" statement



Loop Fragments

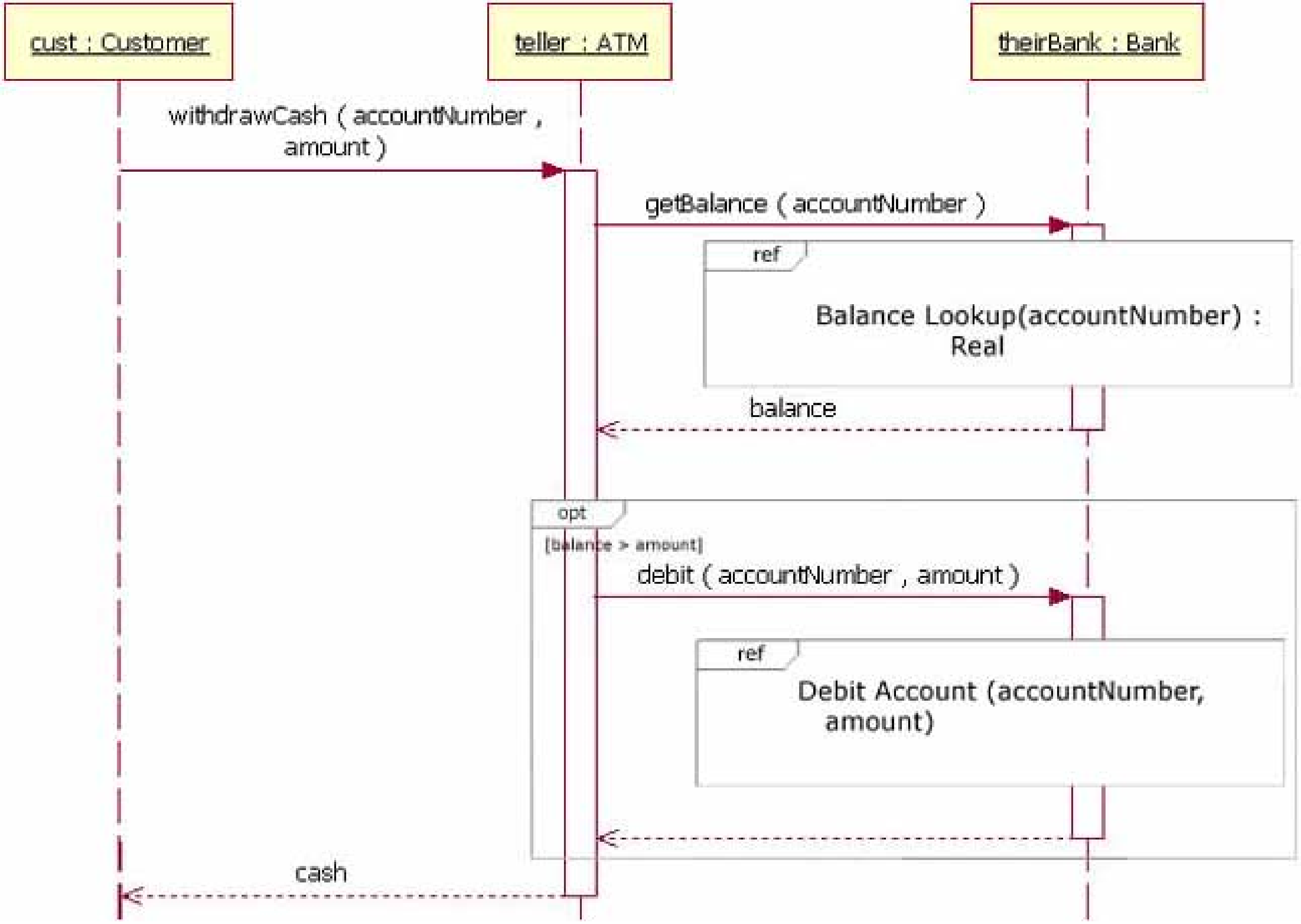
- The loop in the sequence diagram uses a Boolean test to verify if the loop sequence should be run
- When you get to the loop combination fragment a test is done to see if the value of the entry condition is true. Then the sequence goes into the loop fragment

Loop Fragment

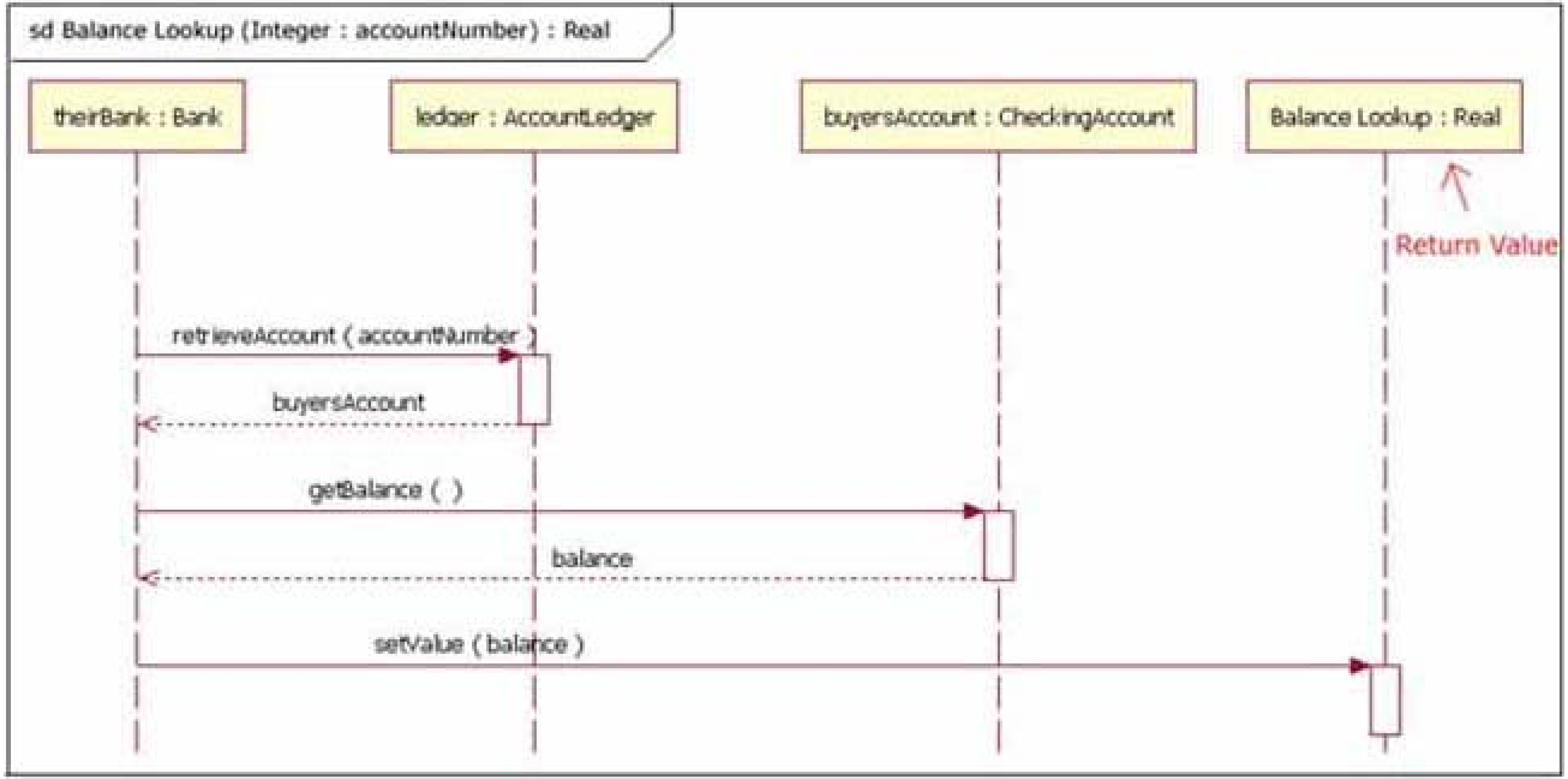


Reference Fragments

- The text "ref" is placed inside the frame's namebox, and the name of the sequence diagram being referenced is placed inside the frame's content area along with any parameters to the sequence diagram

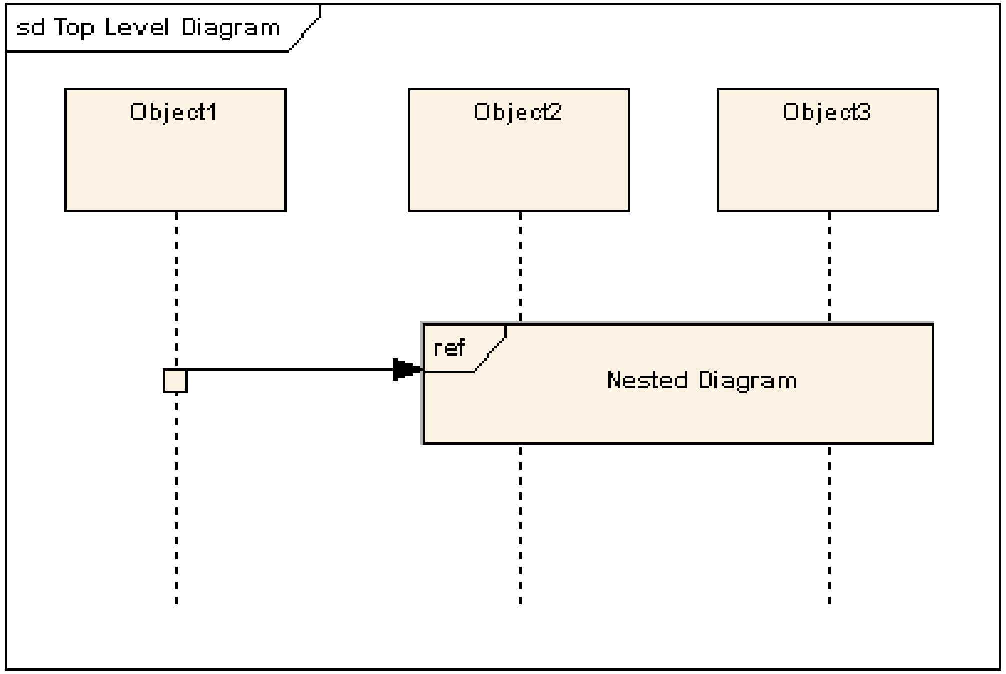

cust : Customer
teller : ATM
theirBank : Bank
withdrawCash (accountNumber , amount)
getBalance (accountNumber)
ref
Balance Lookup(accountNumber) : Real
balance
opt
[balance > amount]
debit (accountNumber , amount)
ref
Debit Account (accountNumber, amount)
cash

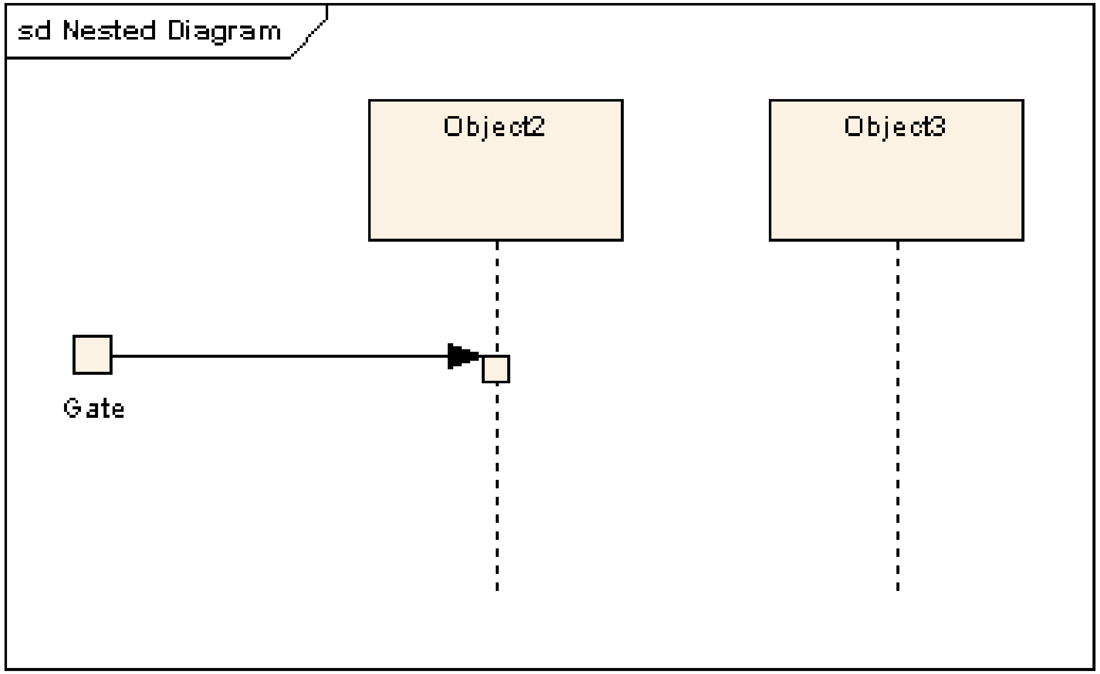
Referenced Sequence Diagram (It is reusable)



Gate

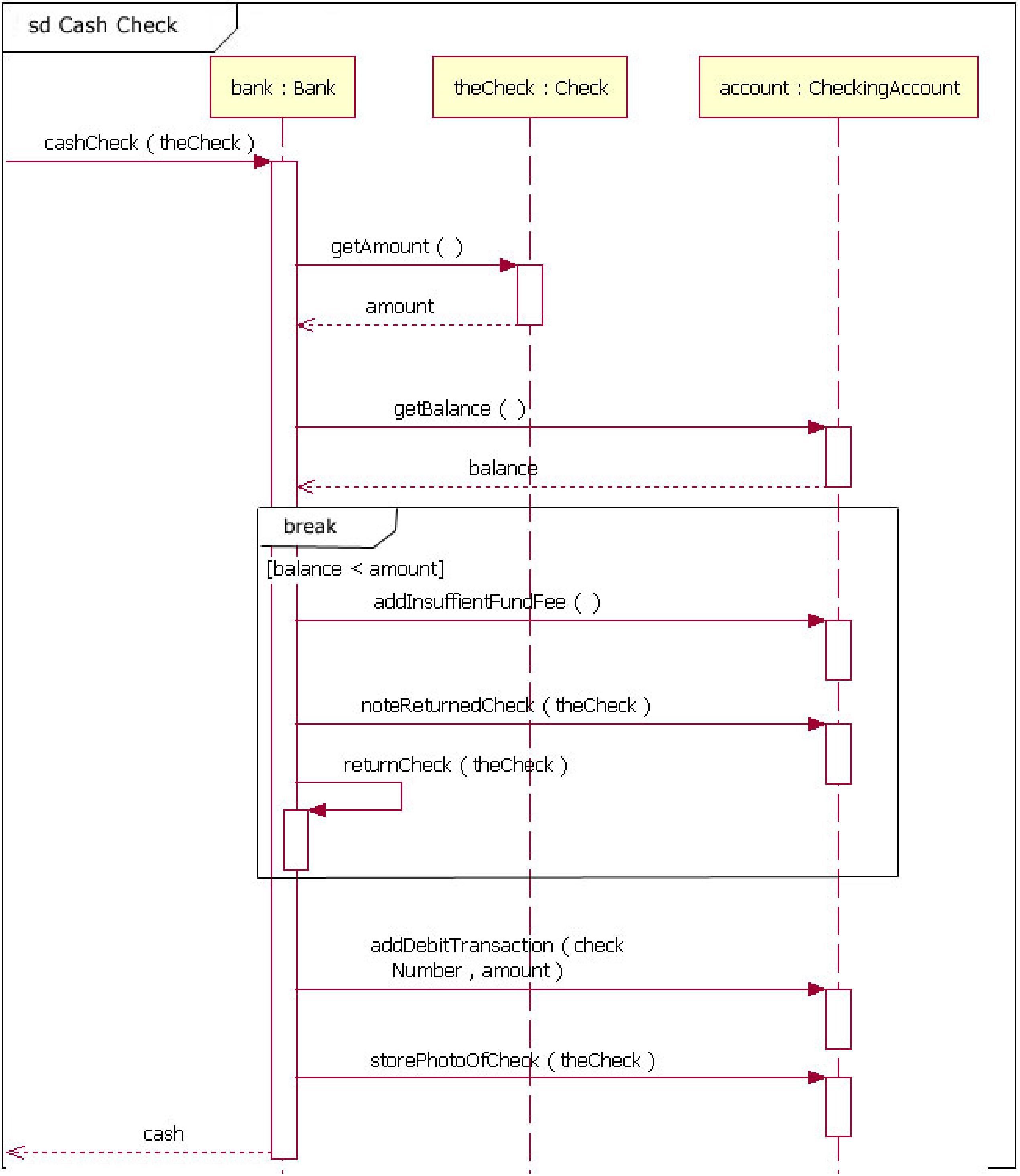
- A gate is a connection point for connecting a message inside a fragment with a message outside a fragment
- A gate as a small square on a fragment frame
- Diagram gates act as off-page connectors for sequence diagrams, representing the source of incoming messages or the target of outgoing messages





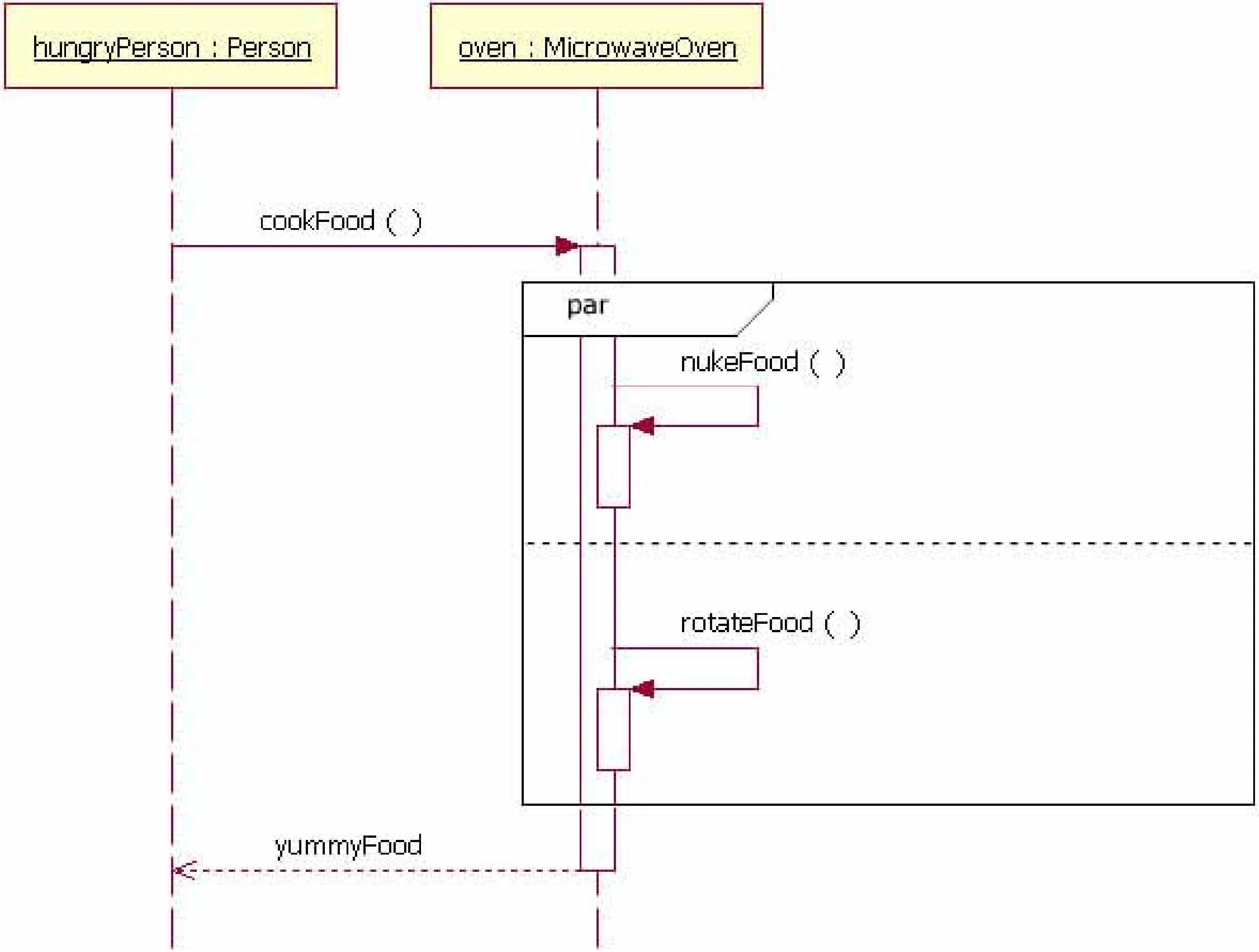
Break Fragments

- 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
- Once all the messages in the break combination have been sent, the sequence exits without sending any of the remaining messages
- In this way the break combined fragment is much like the break keyword in a programming language like C++ or Java.



Parallel Fragments

- The parallel combination fragment is drawn using a frame, and you place the text "par" in the frame's namebox
- The frame is divided into content sections, separated by a dashed line
- Each interaction in the frame represents a thread of execution done in parallel


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