Software Performance Analysis
- UML Extensions

Critical Use Cases

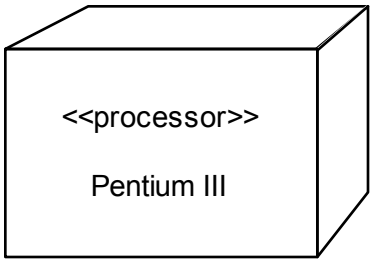
- Use cases are employed to
  - model the context of the system: the system boundary indicates which ***features*** are part of (inside) the system, which ***actors*** interact with system, and the ***meaning of interaction***
  - specify the requirements for the system (i.e., what the system should do from the point of view of actors)
- From a performance point of view, use case diagrams are used to identify ***the critical functions of the system that are most important to performance***
- The ***critical use cases*** are considered, including
  - are ***critical*** to the ***operation*** of the system
  - influence users' perception of ***responsiveness***
  - represent a ***risk*** that performance goals might not be met

Extending the UML

- UML provides built-in extension mechanisms that allow you to tailor the notation for particular purposes.
- These mechanisms are
  - stereotypes
  - tagged values
  - constraints

Stereotypes

- A stereotype allows you to create new model elements
  - derived from existing UML elements
  - specific to a problem domain
- The stereotype is represented as
  - a string enclosed in guillemets (<< >>), or
  - a graphic elements, such as icon



stereotypes

:Processor

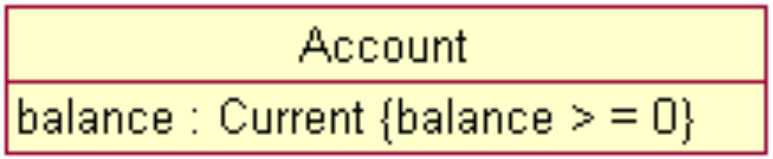
Tagged Values

- A tagged value allows you to include new properties for model elements
- A tagged value is a pair of strings -- a tag and a value
  - {name of a property = value of the property}

Tagged Values

Constraints

- A constraint is a condition or restriction that defines additional model semantics
- A constraint may be attached to an individual model element or a collection of elements
- A constraint is written as a string enclosed in braces ({})



Constraints

Stereotypes, Tagged Values, Constraints

- We use stereotypes and tagged values to capture information about the software execution environment
  - e.g., processor type, processor speed, network speed
- We use constraints to specify performance objectives
  - e.g., response time or throughput

Extensions to Sequence Diagram Notation

- Sequence diagram notation is extended to represent
  - hierarchical structure (instance decomposition and references)
  - looping
  - alternation
  - concurrency

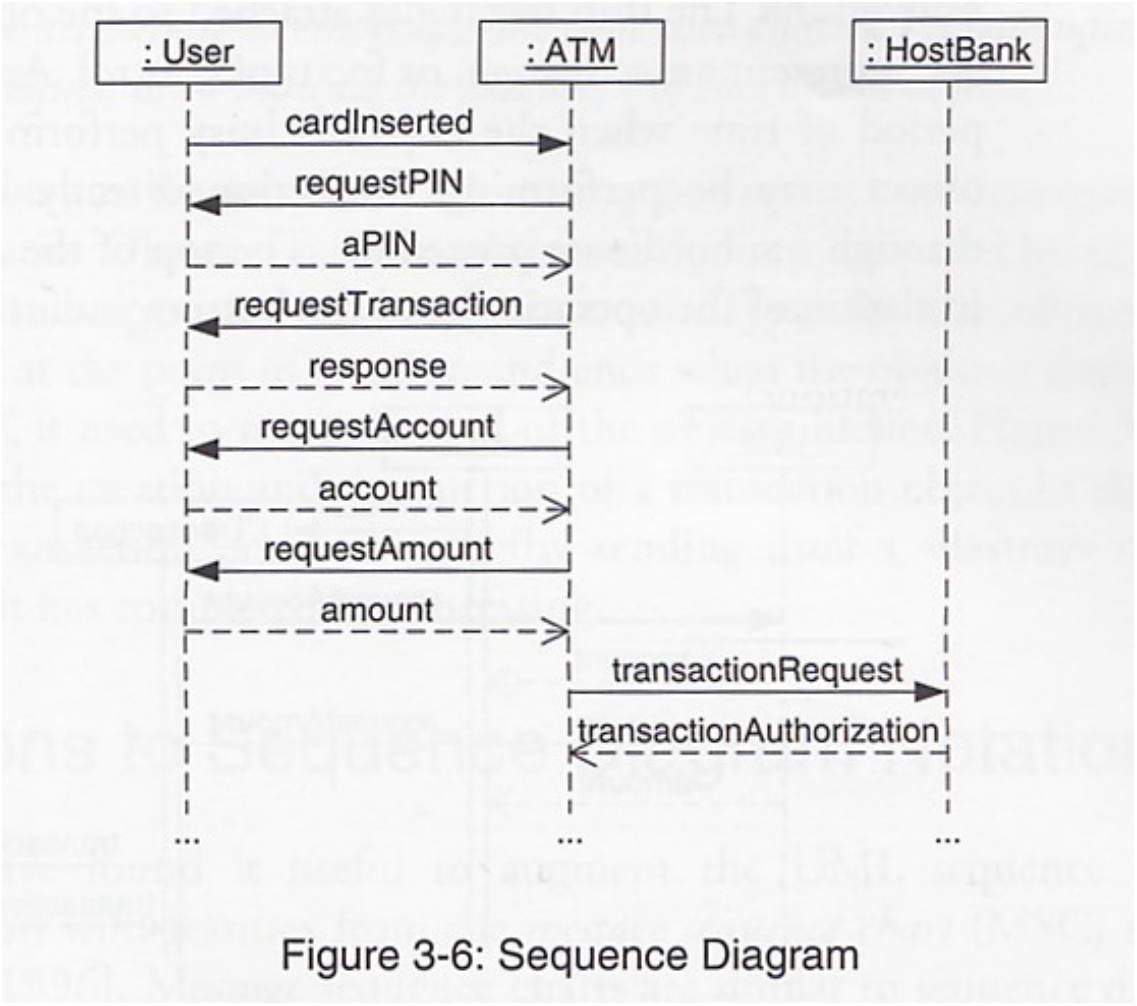


Figure 3-6: Sequence Diagram

Instance Decomposition

- Uses instance decomposition to indicate the refinement of sequence diagrams
- Makes it possible to attach another sequence diagram to an object lifeline
- Allows expansion of a high-level sequence diagram to show lower-level interactions

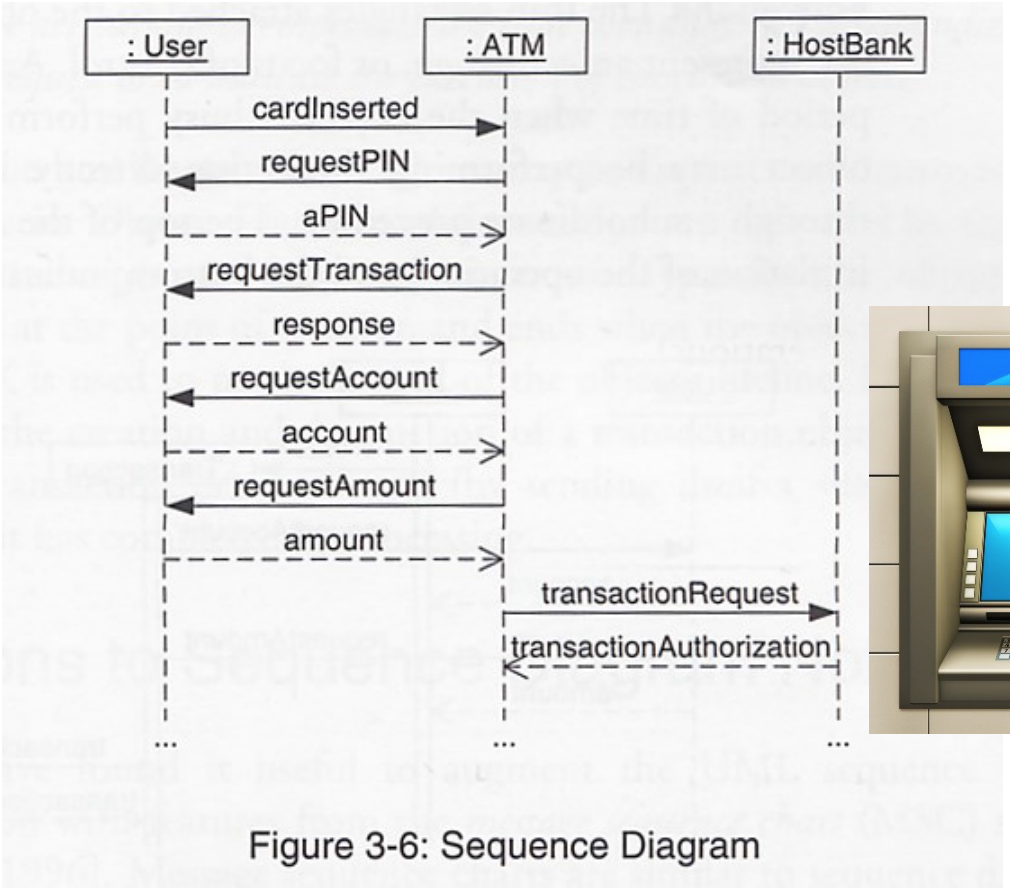


Figure 3-6: Sequence Diagram

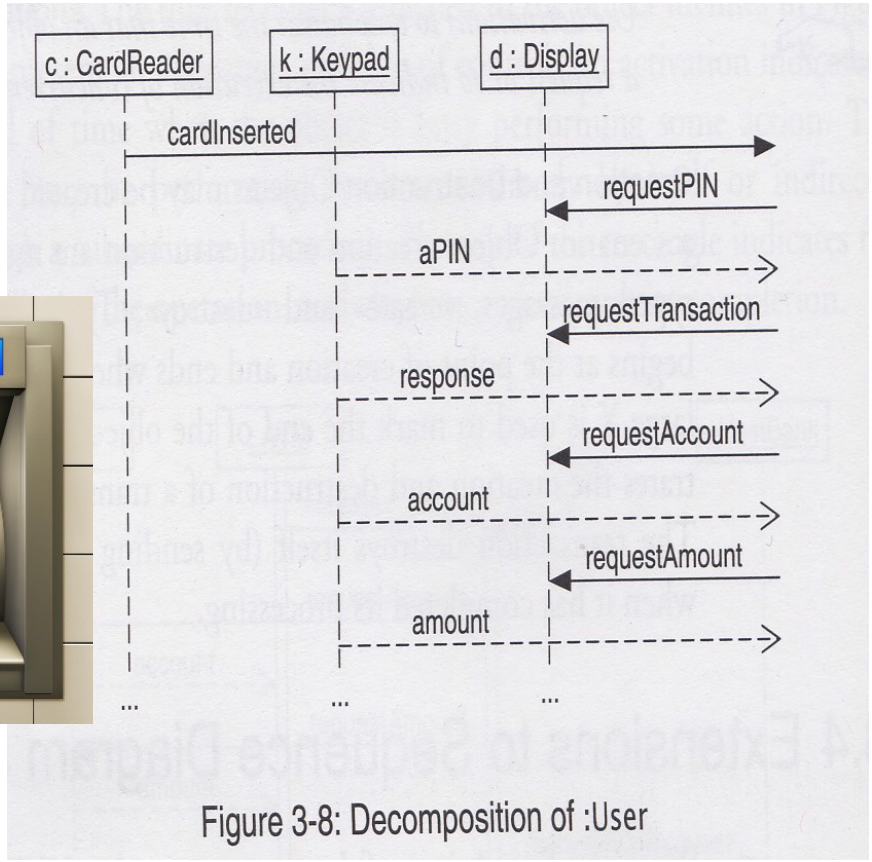


Figure 3-8: Decomposition of :User

ATTENTION!

For the decomposition to be meaningful, ***the order of messages on the decomposed instance must be preserved***

Benefits of Instance Decomposition

- Elaborate the sequence diagram as we learn more about the system, without having to re-draw the diagram each time
- Ensure the consistency with the scenario as it was originally described

Use instance decomposition to elaborate high-level objects as the design evolves

Loop and Alternation

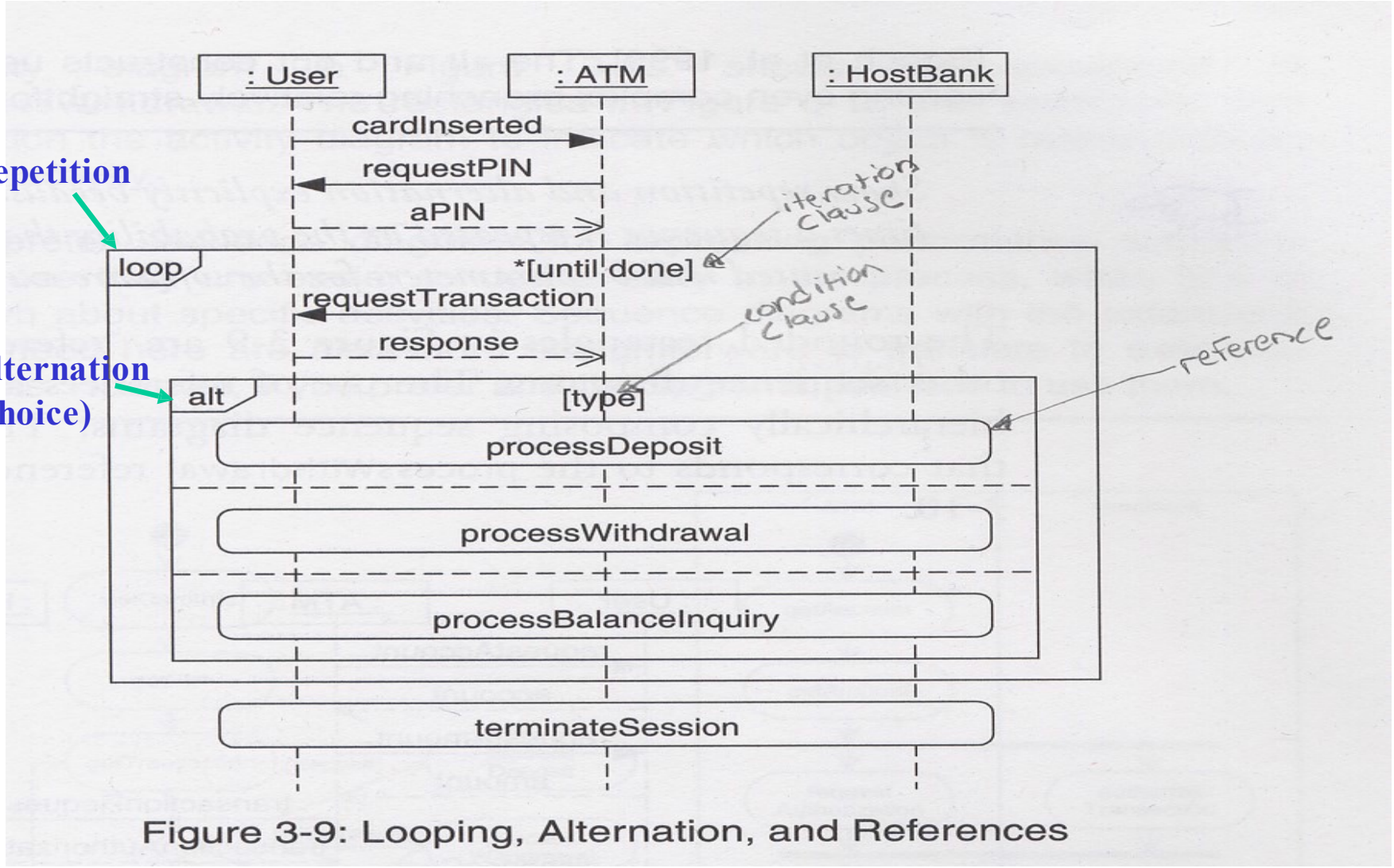


Figure 3-9: Looping, Alternation, and References

References

- ***References*** allows for referring to other sequence diagrams
- Use references to reduce the complexity of sequence diagrams.

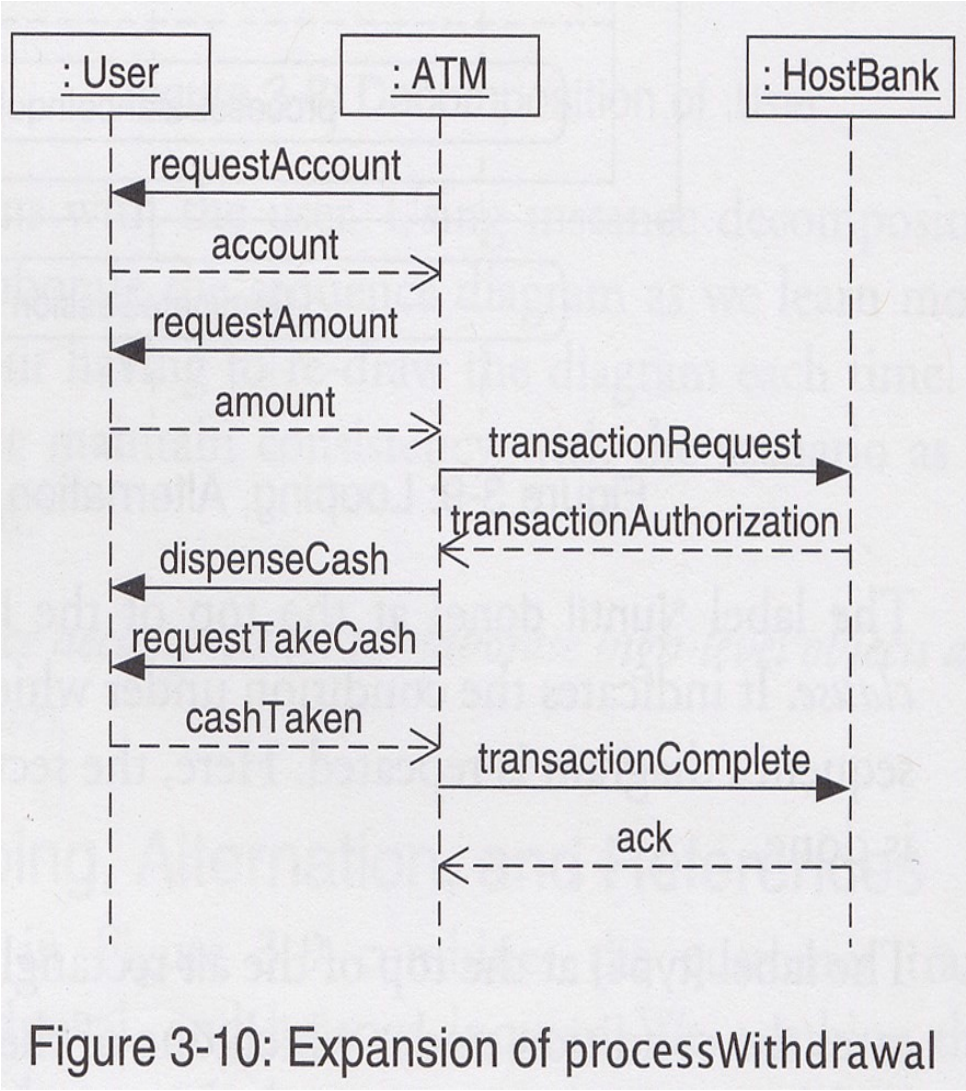
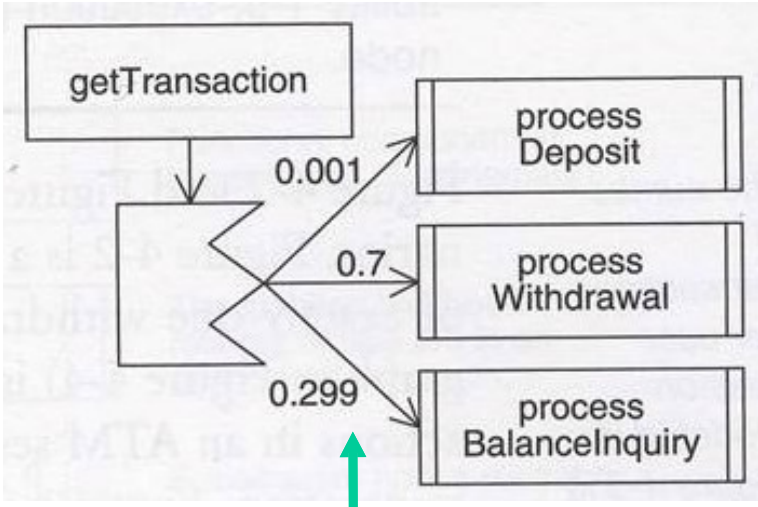


Figure 3-10: Expansion of processWithdrawal

Loop and Alternation (con't)

- ***Loop*** can be used when a sequence is repeated while, loop, etc.
- ***Alternation*** can be used when several possible transitions will be executed If-then, switch-case, etc.
- A probability of execution can be attached to a given sequence



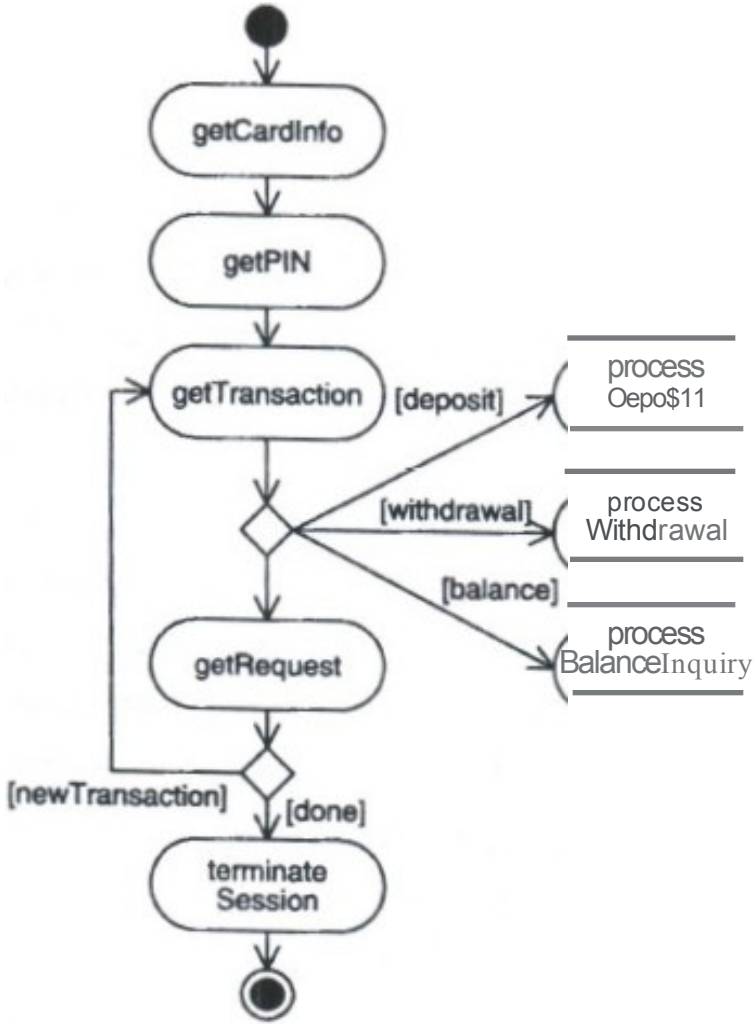


Figure 3-11: Activity Diagram for ATM

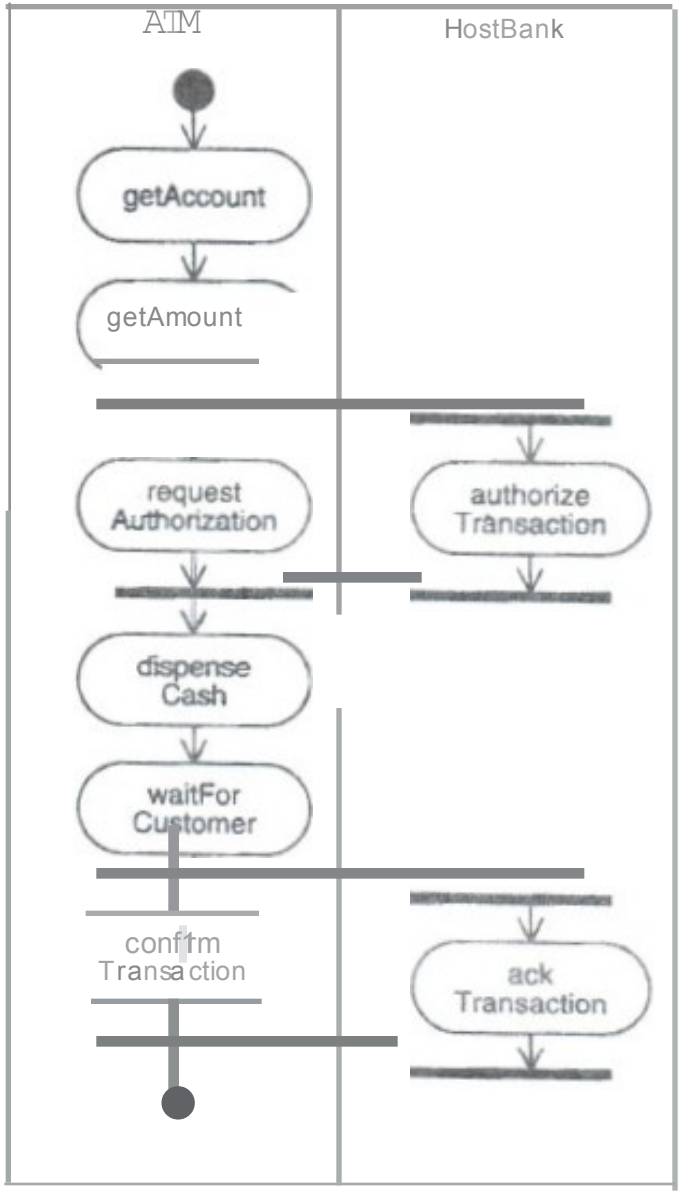
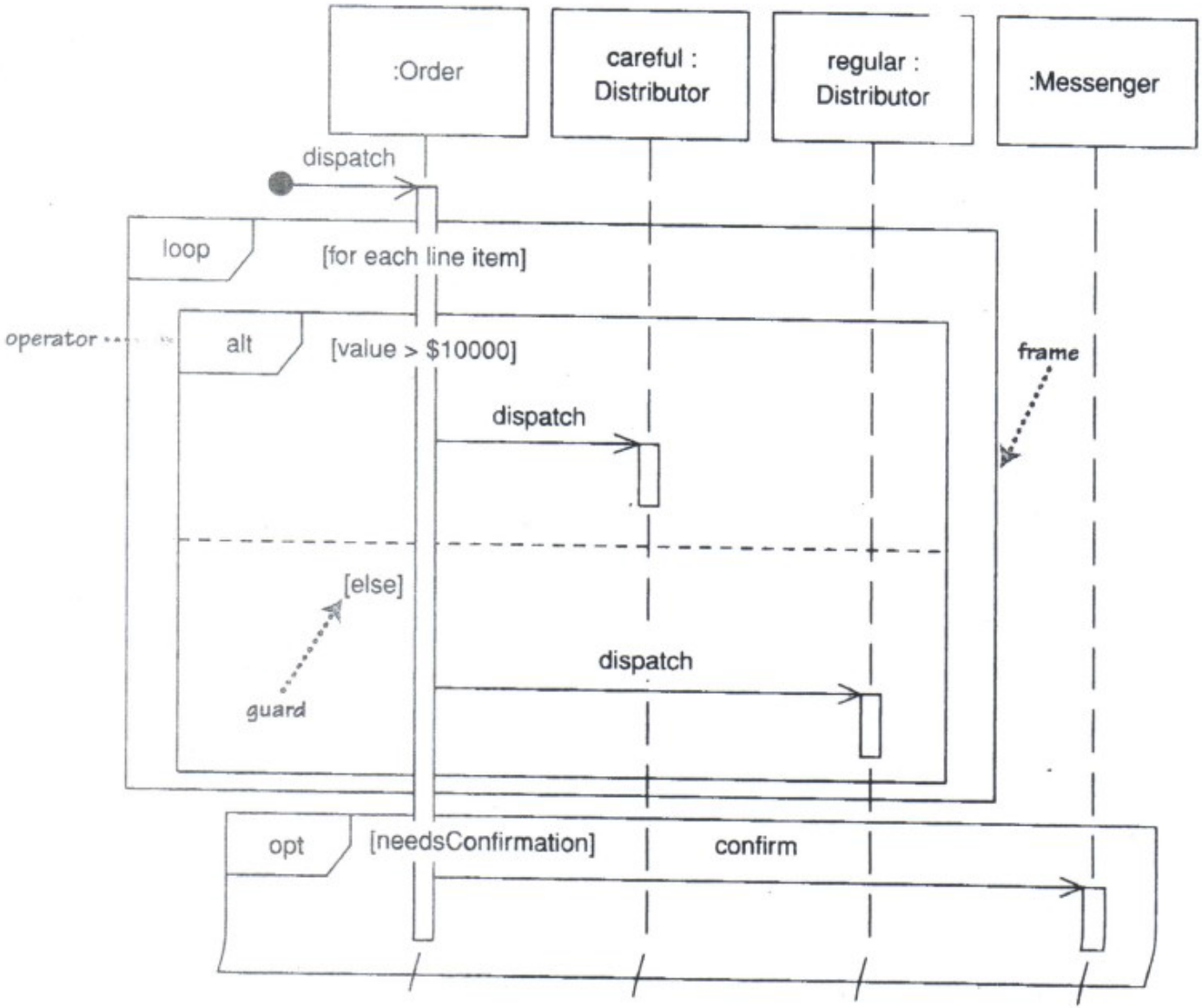


Figure 3-12: Expansion of processWithdrawal

Example

- Once an order is made, a dispatch message is sent.
- The following algorithm describes the dispatch steps:

```
procedure dispatch
    foreach (order.lineitem)
        if (product.value > $10K)
                careful.dispatch
        else
            regular.dispatch
        endif
    endfor
    if (needsConfirmation) meesenger.confirm
end procedure
```


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

Specifying Time

- The UML allows you to specify timing requirements through the use of
  - timing marks
  - time expressions
  - timing constraints

Timing Marks

- Denote the time at which a message or an event occurs
- For example:
  - message.sendTime() -- The time that the message is sent
  - message.recieveTime() -- The time that the message is received
  - where message is the name of the message

Time Expressions

- Evaluate to an absolute or relative value of time
- Express an elapsed time or the occurrence of some particular time
- For example,
  - after(500msec) -- time elapsed after a particular state is entered
  - when(t=08:00) -- the occurrence of an event at the time 08:00

Time Constraints

- Express a constraint based on the absolute or relative value of time
- For example
  - {a.recieveTime() - b.sendTime() < 10 msec}

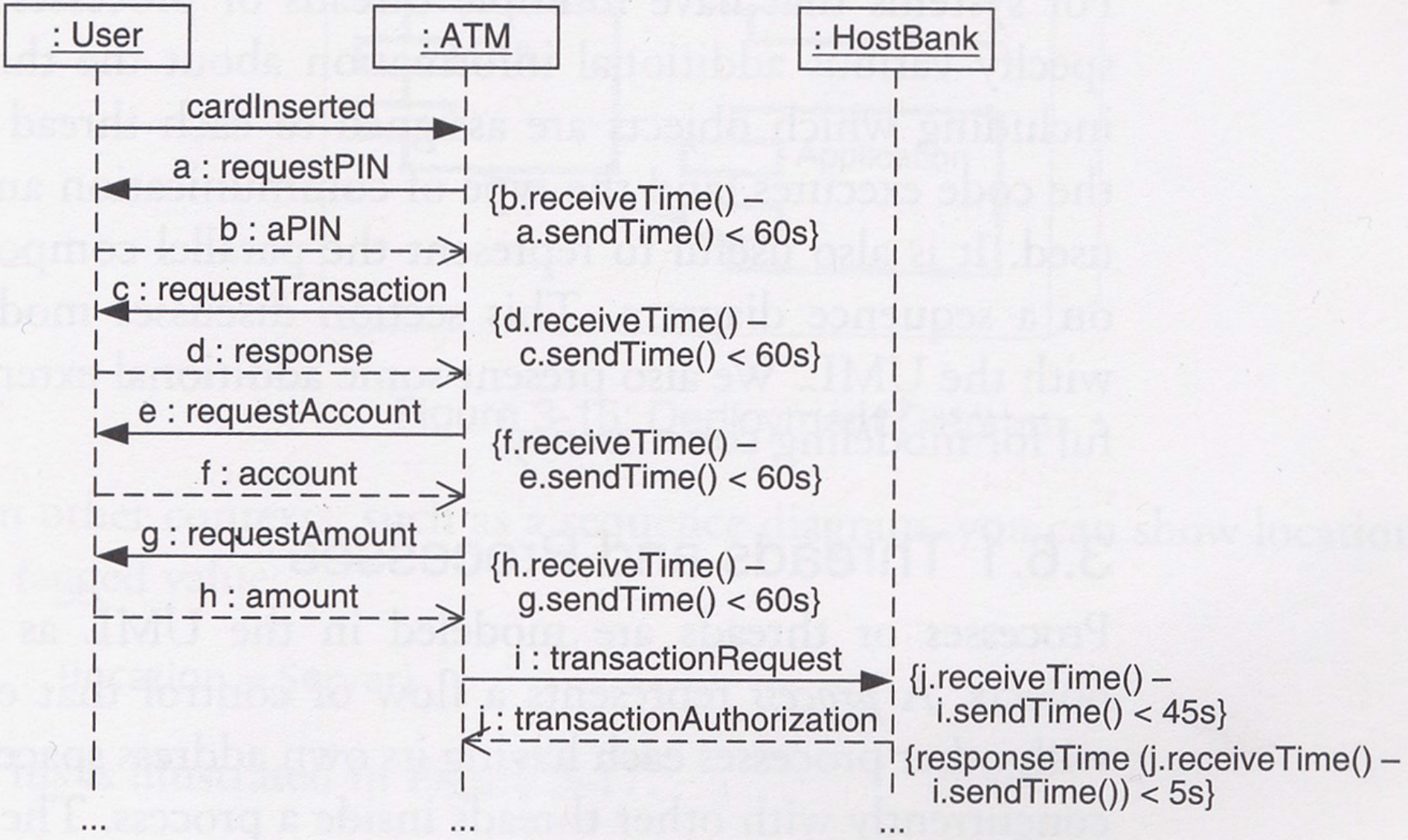


Figure 3-13: Time in Sequence Diagrams

Time Constraints (con't)

- Timeout conditions are not particularly useful from a performance perspective
- When specifying performance, we are more interested in response time
  - responseTime(j.receiveTime() – i.sendTime())
  - {responseTime(j.receiveTime() – i.sendTime()) < 5s} – a time constraint example

Use time expressions that are meaningful from a performance perspective, such as responseTime(), to specify performance objectives

Concurrency

- Modeling concurrency is important in the later stages of SPE for evaluating contention effects
- Concurrency issues are expressed by UML notations
  - Threads and Processes
  - Coregions
  - Parallel Composition
  - Synchronization

Threads and Processes

- A process represents a flow of control that executes in parallel with other processes
  - Each process has its own address space
  - represented by a standard stereotype ***<<process>>***
- A thread executes concurrently with other threads inside a process
  - all threads belonging to a process all share the same address space
  - represented by a standard stereotype ***<<thread>>***

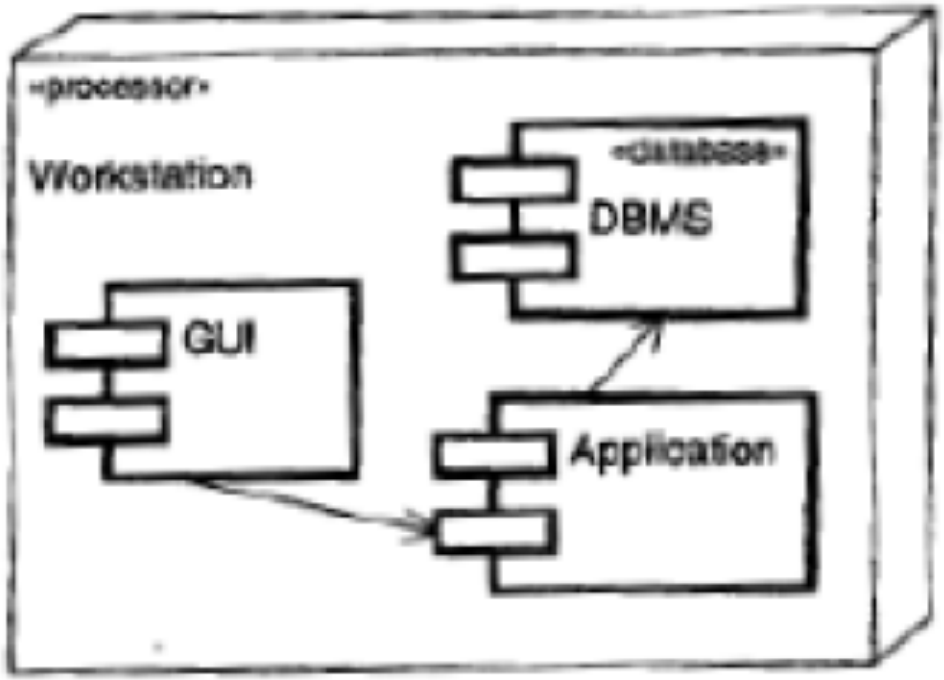


Figure 3-15: Deployment Diagram

Coregions

- A sequence diagram are strictly ordered in time
- ***Coregions*** allow an exception to total ordering whereby messages within a coregion are ***unordered***
- Coregions allow you to show the interleaving of messages that occur in parallel processing

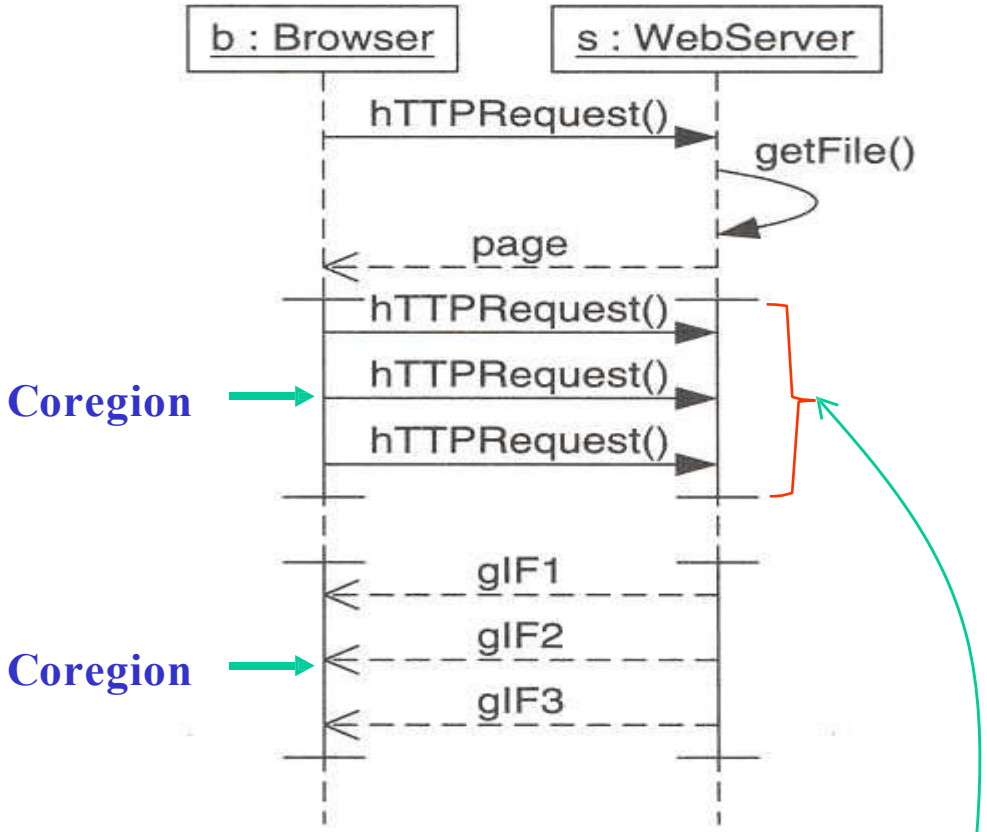


Figure 3-16: Coregions

{represented by a solid line with horizontal dividers}

Parallel Composition

- Indicates sections of the sequence diagram that are executed in parallel
- Shows the interleaving of messages that occur in parallel processing
- Allows more flexible representation of parallel processing

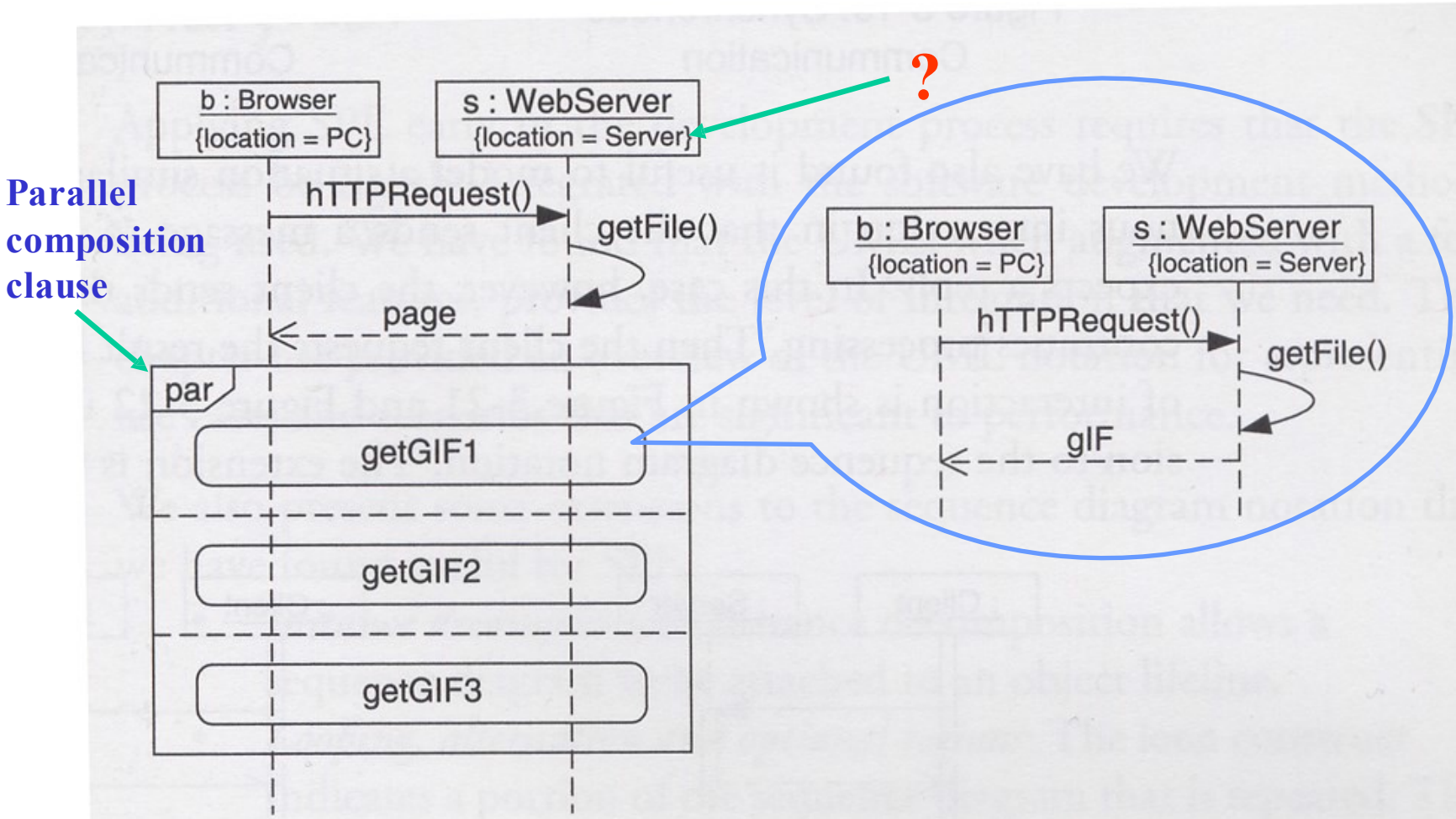


Figure 3-17: Parallel Composition

Figure 3-18: Expansion of getGIF

Synchronization

- UML provides different types of arrowheads to represent communications among objects

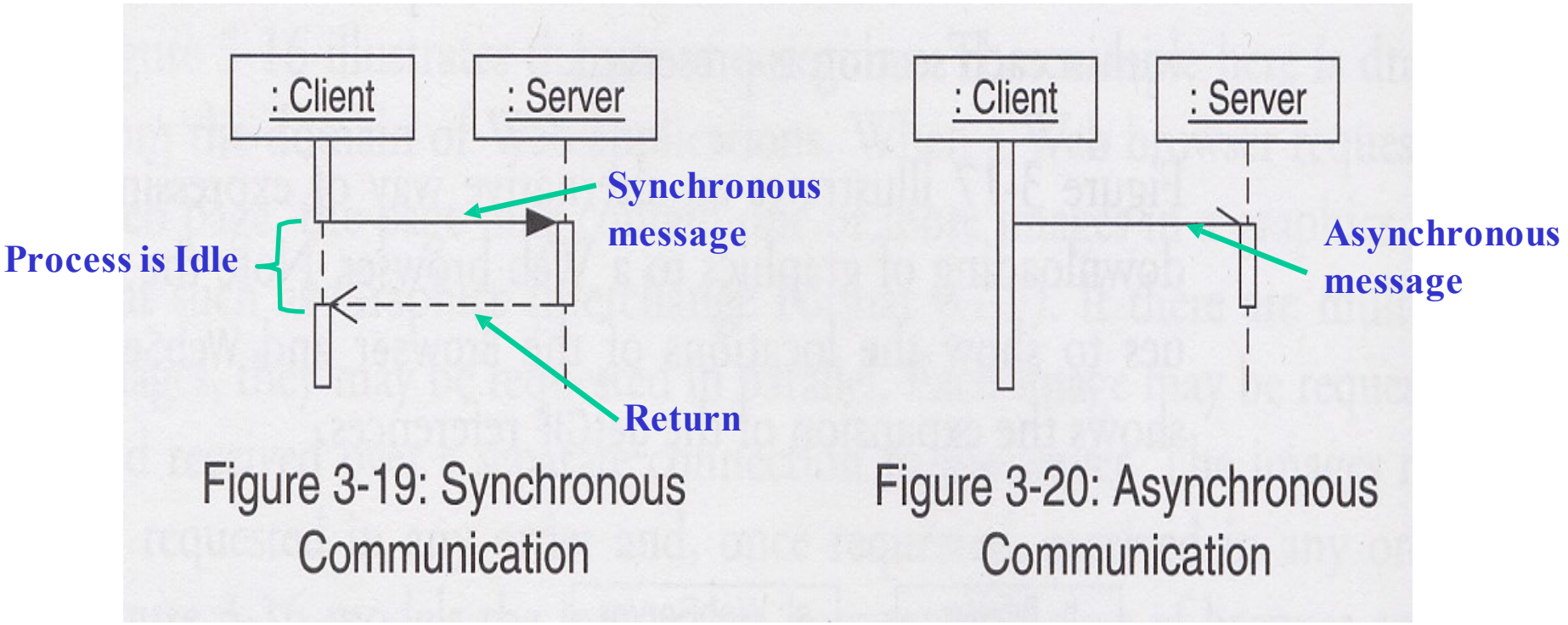


Figure 3-19: Synchronous Communication

Figure 3-20: Asynchronous Communication

Synchronization (con't)

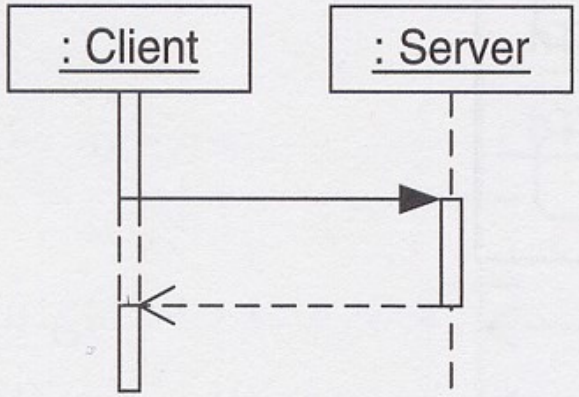


Figure 3-21: Deferred Synchronous Communication

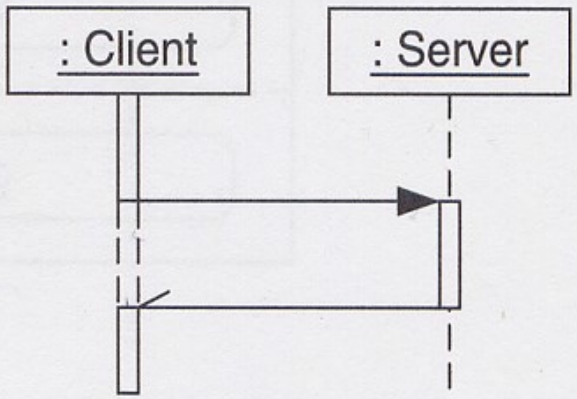


Figure 3-22: Asynchronous Callback

Contention Effects

- Modeling concurrency is important in the later stages of SPE for evaluating contention effects
- Early stages of the development process focus on software model without contention
- Concurrency and synchronization properties of the proposed software are considered later when your knowledge of the software system increases