Unified Modeling Language

Dynamic Modeling

Interaction Diagrams: Sequence and Collaboration Diagrams

Dynamic modeling

- Why?

How do you find classes?

- In previous lectures we have already established the following sources
  - Application domain analysis: Talk to client to identify abstractions
  - Application of general world knowledge and intuition
  - Scenarios
    - » Natural language formulation of a concrete usage of the system
  - Use Cases
    - » Natural language formulation of the functions of the system
  - Textual analysis of problem statement (Abbott)
- Today we show how identify classes from dynamic models
  - Events in a sequence diagram as well as actions and activities in state chart diagrams are candidates for public operations in classes
  - Activity lines in sequence diagrams are also candidates for objects

Modelli Dinamici

- I modelli dinamici descrivono il comportamento del sistema in funzione del tempo
  - I modelli dinamici sono un tipo di modello operazionale (modello che descrive il comportamento desiderato)
  - Utili soprattutto per sistemi orientati al controllo (embedded systems, sistemi interattivi, traduttori, sistemi operativi, software di comunicazione)

Dynamic Modeling with UML

- Diagrams for dynamic modeling
  - *Interaction diagrams* describe the dynamic behavior between objects
  - *Statecharts* describe the dynamic behavior of a single object
- Interaction diagrams
  - Sequence Diagram:
    - » Dynamic behavior of a set of objects arranged in time sequence.
  - Collaboration Diagram :
    - » Shows the relationship among objects. Does not show time
- State Chart Diagram:
  - A state machine that describes the response of an object of a given class to the receipt of outside stimuli (Events).
  - *Activity Diagram:* A special type of statechart diagram, where all states are action states

Dynamic Modeling

- Purpose:
  - Detect and supply methods for the object model
- How do we do this?
  - Start with use case or scenario
  - Model interaction between objects => sequence diagram
  - Model dynamic behavior of a single object => statechart diagram

Start with Flow of Events from Use Case

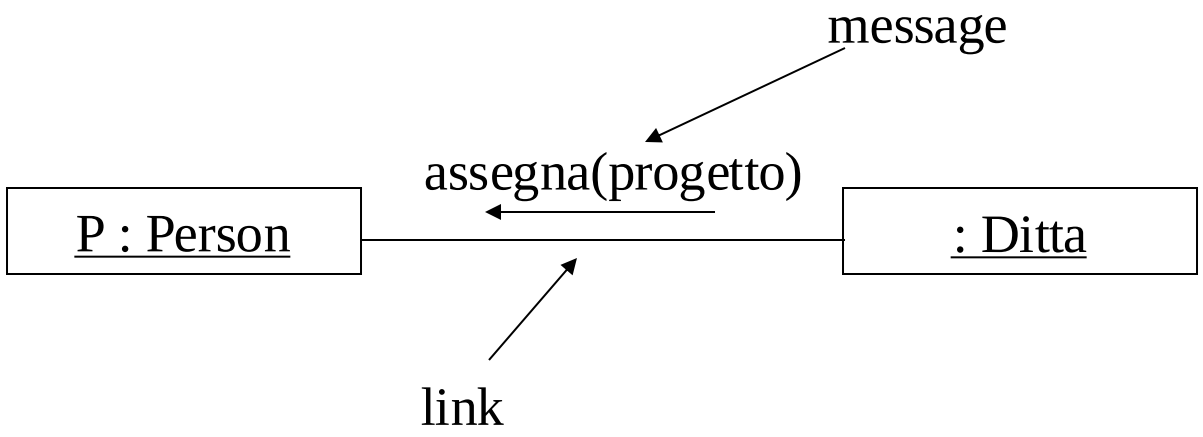
- Flow of events from “Dial a Number” Use case:
  - Caller lifts receiver
  - Dial tone begins
  - Caller dials
  - Phone rings
  - Callee answers phone
  - Ringing stops
  - ....
- A scenario can be the historical record of executing a system or a thought experiment of executing a proposed system.
- Each **event** transmits information from one object to another. For example, ***dial tone begins*** transmits a signal from the phone line to the caller.

Interaction Diagrams

- Descrivono il comportamento dinamico di un gruppo di oggetti che “interagiscono” per risolvere un problema
- Una ***interazione*** è un comportamento che comprende un insieme di messaggi scambiati tra un insieme di oggetti nell’ambito di un contesto per raggiungere uno scopo
- UML propone due diversi tipi di Interaction Diagram
  - Sequence Diagram
  - Collaboration Diagram
- Sequence e Collaboration diagrams esprimono informazioni simili, ma le evidenziano in modo diverso

Interazioni e Messaggi

- Una interazione, tipicamente, avviene tra oggetti tra cui esiste un link (istanza di un'associazione)
- Un messaggio è una specificazione di una comunicazione tra oggetti che trasmette informazione con l'aspettativa che ne conseguirà una attività.
  - La ricezione di un messaggio può essere considerata una istanza di un evento



Messaggi

Ricordate?

- Gli oggetti sono gli elementi attivi di un programma. Come fanno gli oggetti a compiere le azioni desiderate ?

- Gli oggetti sono attivati dalla ricezione di un messaggio
- Una classe determina i messaggi a cui un oggetto può rispondere
- I messaggi sono inviati da altri oggetti

Messaggi

Ricordate?

- Per l'invio di un messaggio è necessario specificare:
  - Ricevente
  - Messaggio
  - Eventuali informazioni aggiuntive

Sequence Diagrams

- I diagrammi di sequenza descrivono le interazioni tra oggetti che collaborano per svolgere un compito
- Sono utili per evidenziare la distribuzione del controllo nel sistema (“chi” fa “che cosa” …)
- Gli oggetti collaborano scambiandosi messaggi
- Lo scambio di un messaggio in OOP equivale all’invocazione di un metodo

Sequence Diagrams

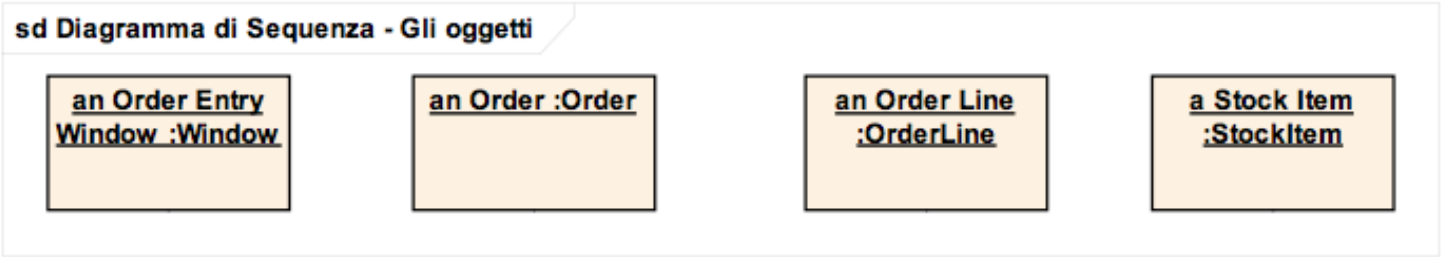
- I diagrammi di sequenza possono essere utilizzati nei seguenti modi:
  - Per modellare le interazioni ad alto livello tra oggetti attivi all'interno di un sistema
  - Per modellare l'interazione tra istanze di oggetti nel contesto di una collaborazione che realizza un caso d'uso
  - Per modellare l'interazione tra oggetti in una collaborazione che realizza una operazione

Sequence Diagrams

- Evidenziano la sequenza temporale delle azioni
- Non si vedono le associazioni tra oggetti
- Le attività svolte dagli oggetti sono mostrate su linee verticali
- La sequenza dei messaggi scambiati tra gli oggetti è mostrata su linee orizzontali
- Possono corrispondere a uno scenario specifico o a un intero caso d'uso (aggiungendo salti e iterazioni)
- Si possono annotare con vincoli temporali

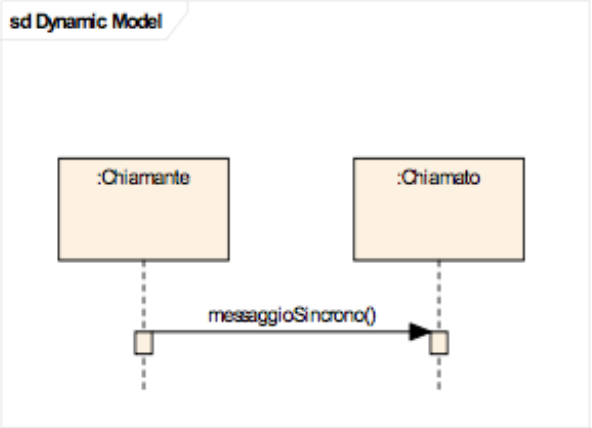
Gli oggetti

- Asse x (asse degli oggetti):
  - Gli oggetti sono disposti orizzontalmente
  - Un oggetto è un'istanza di una classe
  - Sintassi: nomeOggetto : NomeClasse
- Asse t (asse del tempo):
  - Il flusso del tempo è descritto verticalmente



Scambio di messaggi sincroni

- Si disegna con una freccia chiusa da chiamante a chiamato.
  La freccia è etichettata col nome del metodo invocato e, opzionalmente, con i suoi parametri e il suo valore di ritorno

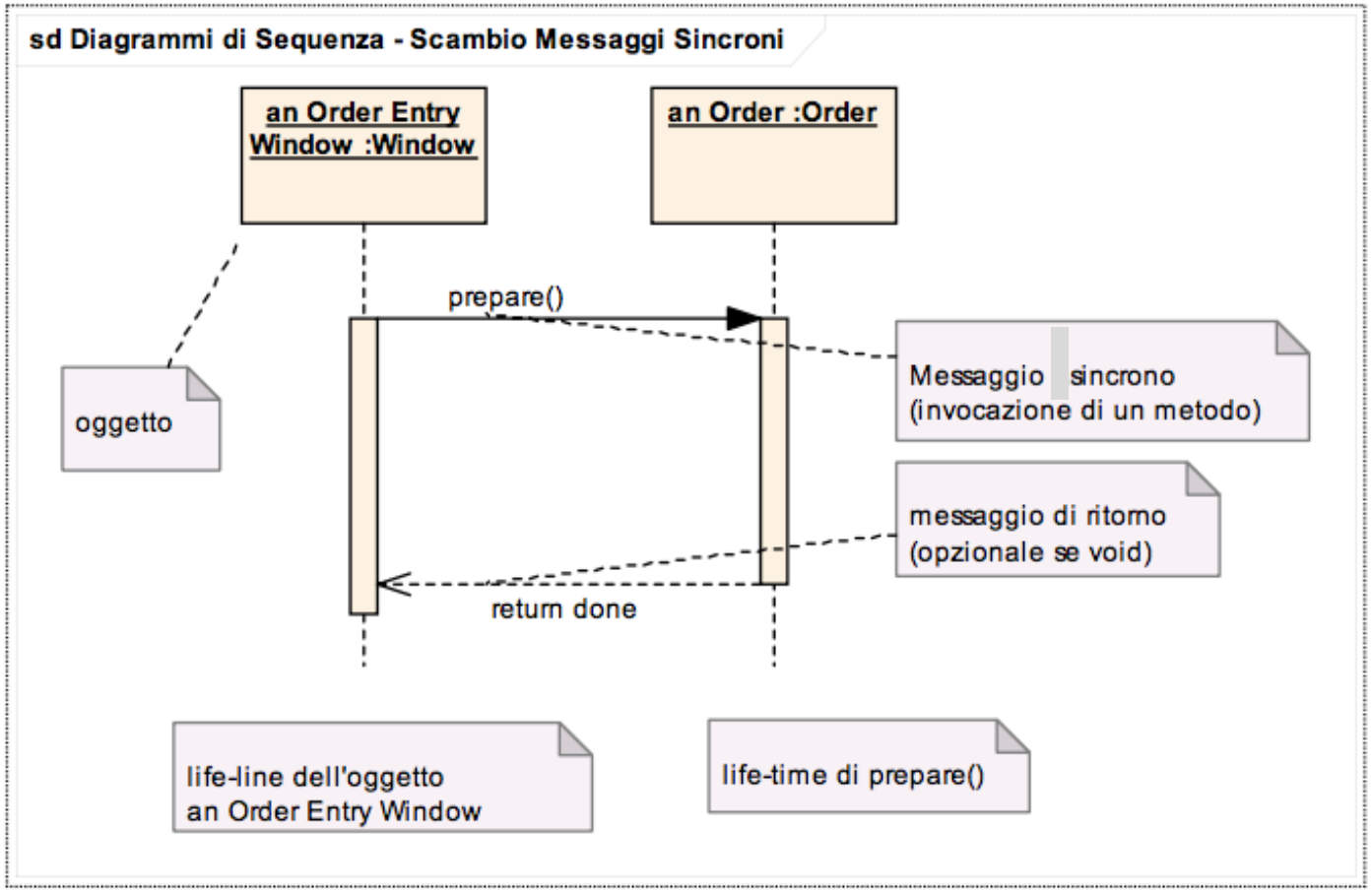


- Il chiamante attende la terminazione del metodo del chiamato prima di proseguire
- Il life-time (durata, vita) di un metodo è rappresentato da un rettangolino che collega la freccia di invocazione con la freccia di ritorno

Scambio di messaggi

- Life-time corrisponde ad avere un record di attivazione di quel metodo sullo stack di attivazione
- Il ritorno è rappresentato con una freccia tratteggiata
- Il ritorno è sempre opzionale. Se si omette, la fine del metodo è decretata dalla fine del life-time

Scambio di messaggi: un esempio



Messaggi asincroni

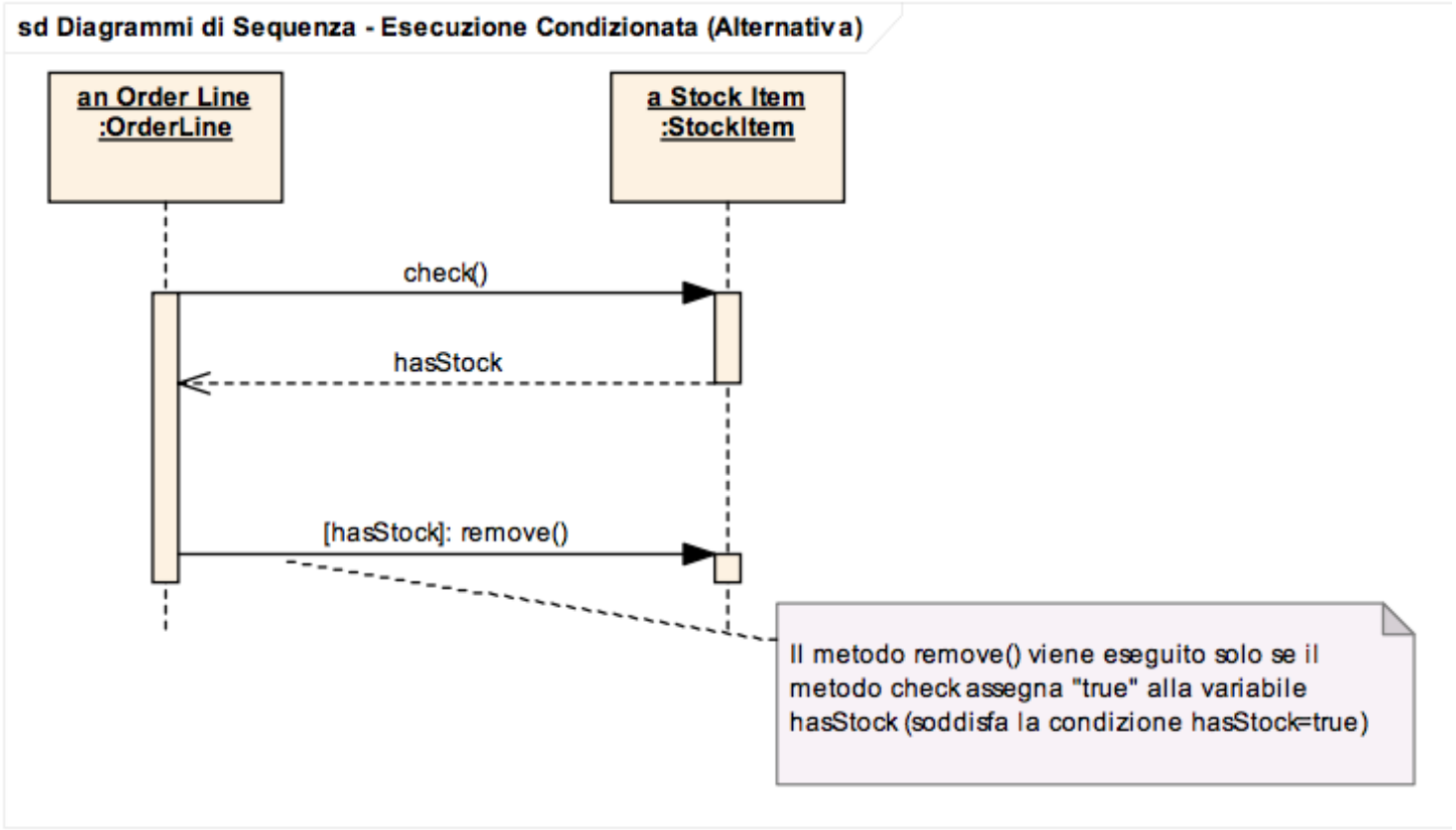
- Si usano per descrivere interazioni concorrenti
- Si disegna con una freccia aperta da chiamante a chiamato. La freccia è etichettata col nome del metodo invocato e, opzionalmente, con i suoi parametri e il suo valore di ritorno
- Il chiamante non attende la terminazione del metodo del chiamato, ma prosegue subito dopo l'invocazione
- Il ritorno non segue quasi mai la chiamata

Esecuzione condizionata di un messaggio

- L'esecuzione di un metodo può essere assoggettata ad una condizione. Il metodo viene invocato solo se la condizione risulta verificata a run-time
- Se la condizione non è verificata, il diagramma *non* dice cosa succede (a meno che non venga esplicitamente modellato ciascun caso)
- La condizione si rappresenta sulla freccia di invocazione del metodo, racchiusa tra parentesi quadre
- Sintassi:

  [cond] : nomeMetodo()

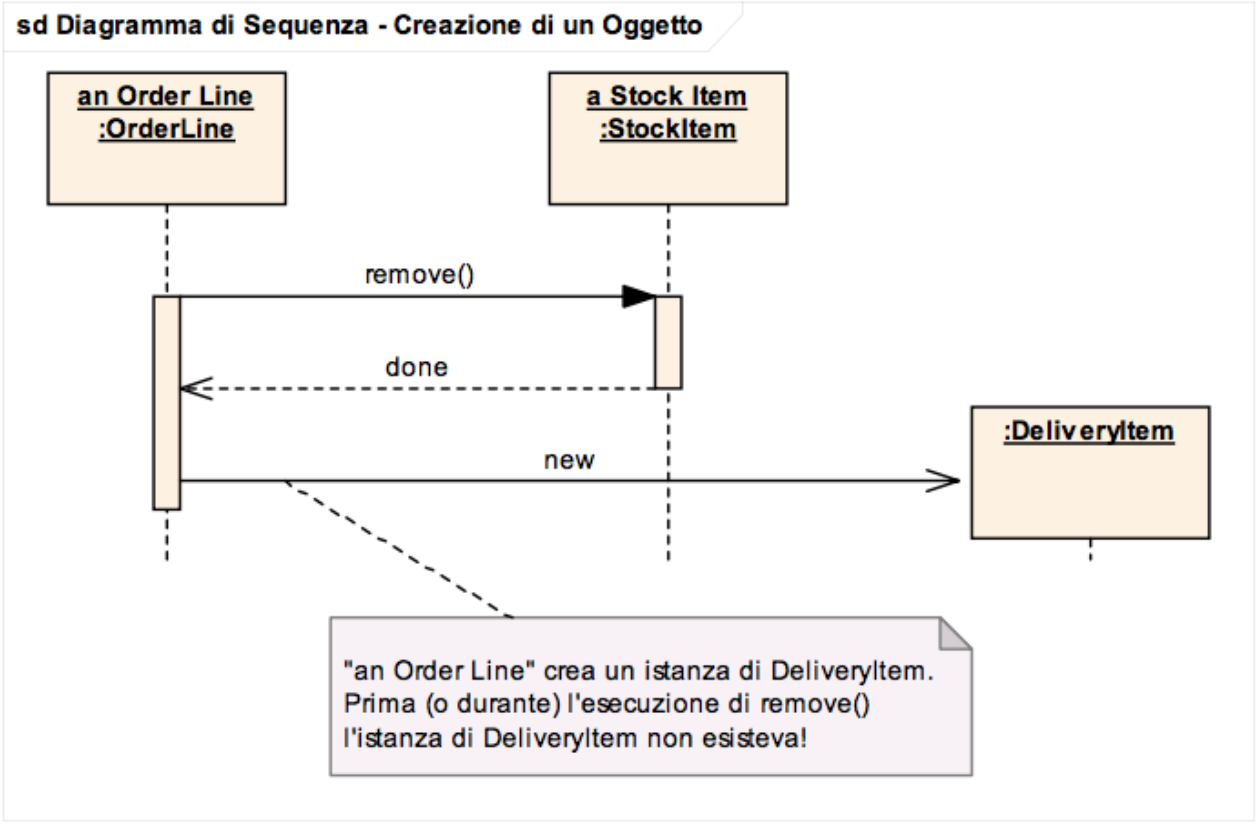
Esecuzione condizionata di un messaggio



Costruzione di un nuovo oggetto

- Rappresenta la costruzione di un nuovo oggetto non presente nel sistema fino a quel momento
- Corrisponde all'allocazione dinamica (allocazione nello heap di sistema, istruzione new)
- Messaggio etichettato new, create,
- L'oggetto viene collocato nell'asse temporale in corrispondenza dell'invocazione nel metodo new (o create)

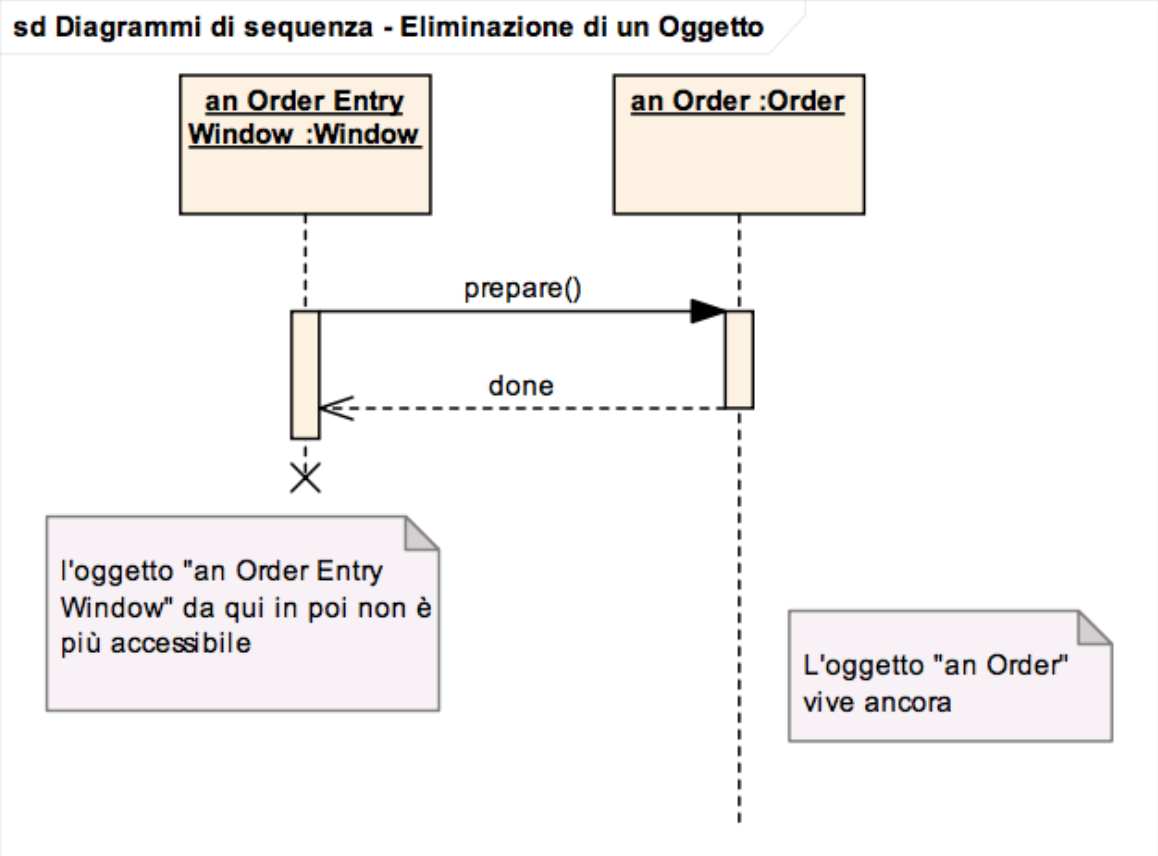
Costruzione di un nuovo oggetto



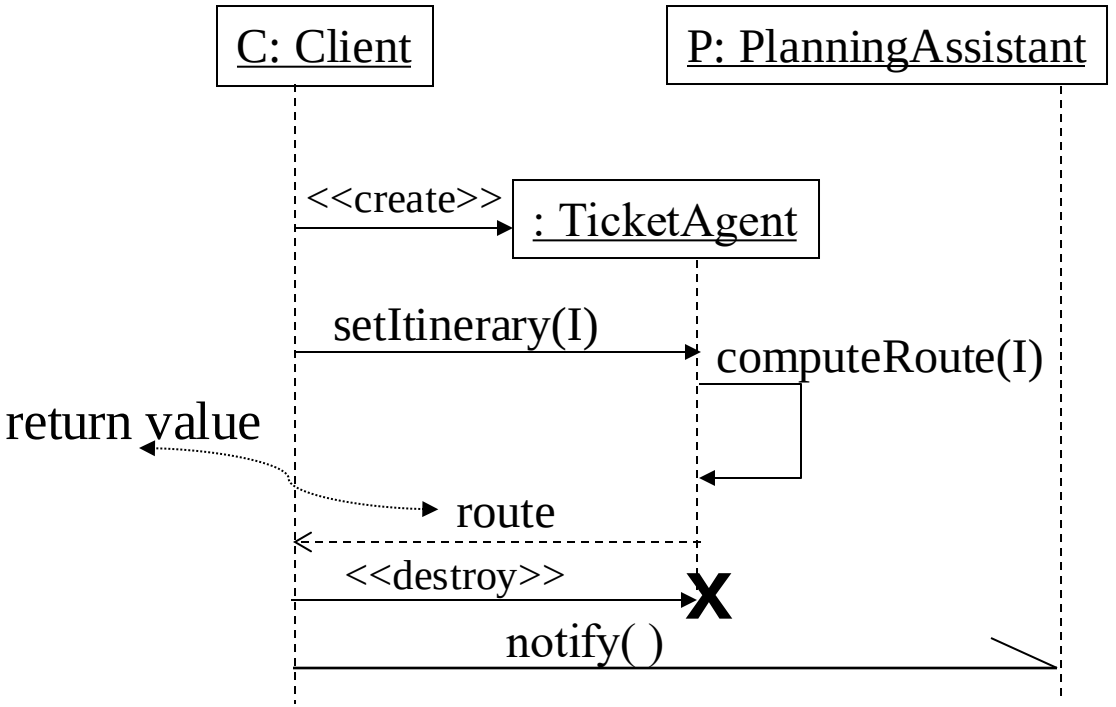
Distruzione di un oggetto (preesistente)

- Rappresenta la distruzione di un oggetto presente nel sistema fino a quel momento
- Corrisponde alla deallocazione dinamica (deallocazione dallo heap di sistema, istruzione delete/dispose/)
- Si rappresenta con una X posta in corrispondenza della life-line dell'oggetto
- Da quel momento in poi non è "legale" invocare alcun metodo dell'oggetto distrutto

Distruzione di un oggetto (preesistente)



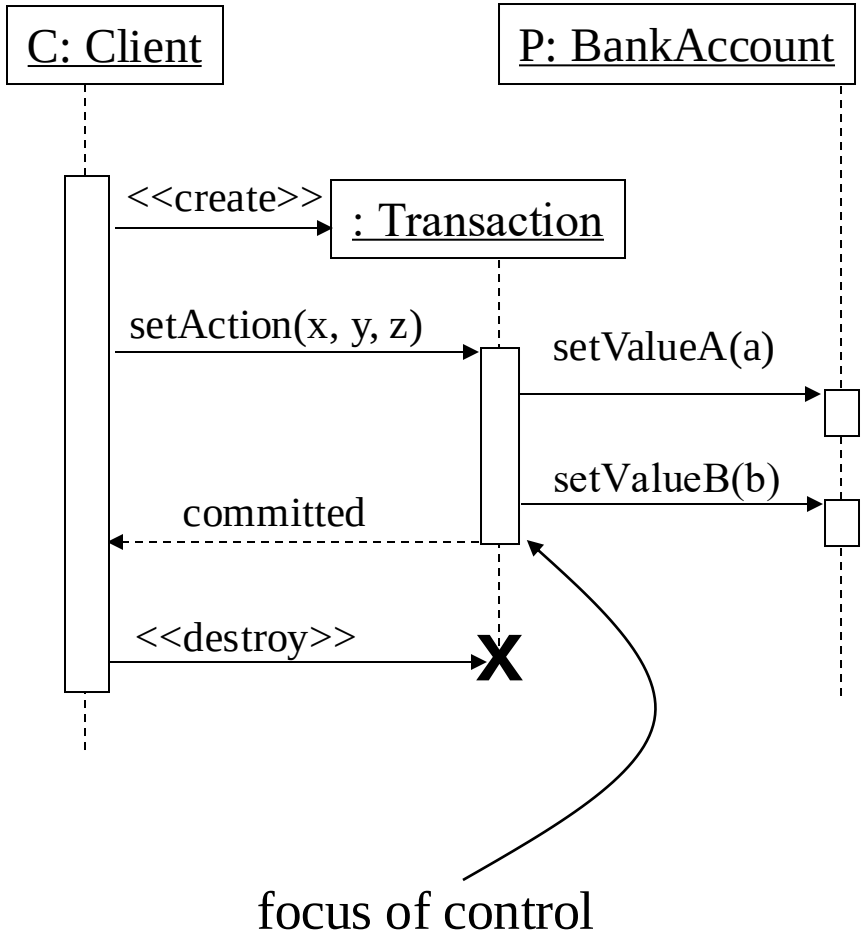
Creazione e distruzione di oggetti



- ***Lifeline di un oggetto***: linea tratteggiata rappresentante l'esistenza (vita) di un oggetto in un periodo di tempo

- gli oggetti che sono creati durante l'interazione vanno posti all'altezza della action di creazione
  - in cima vanno gli oggetti che partecipano al sequence diagram per tutta la durata;

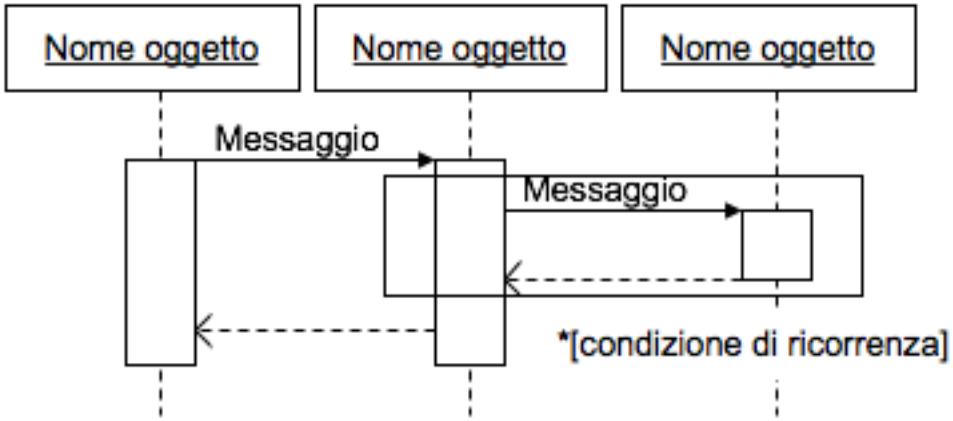
Box di attivazione



- Indica il periodo di tempo durante il quale un oggetto sta eseguendo una action, direttamente o indirettamente
- La cima del rettangolo è allineata con lo start della action (ricezione del messaggio); il fondo è allineato con il completamento della action, ed eventualmente con un messaggio di ritorno

Iterazione (ricorrenze)

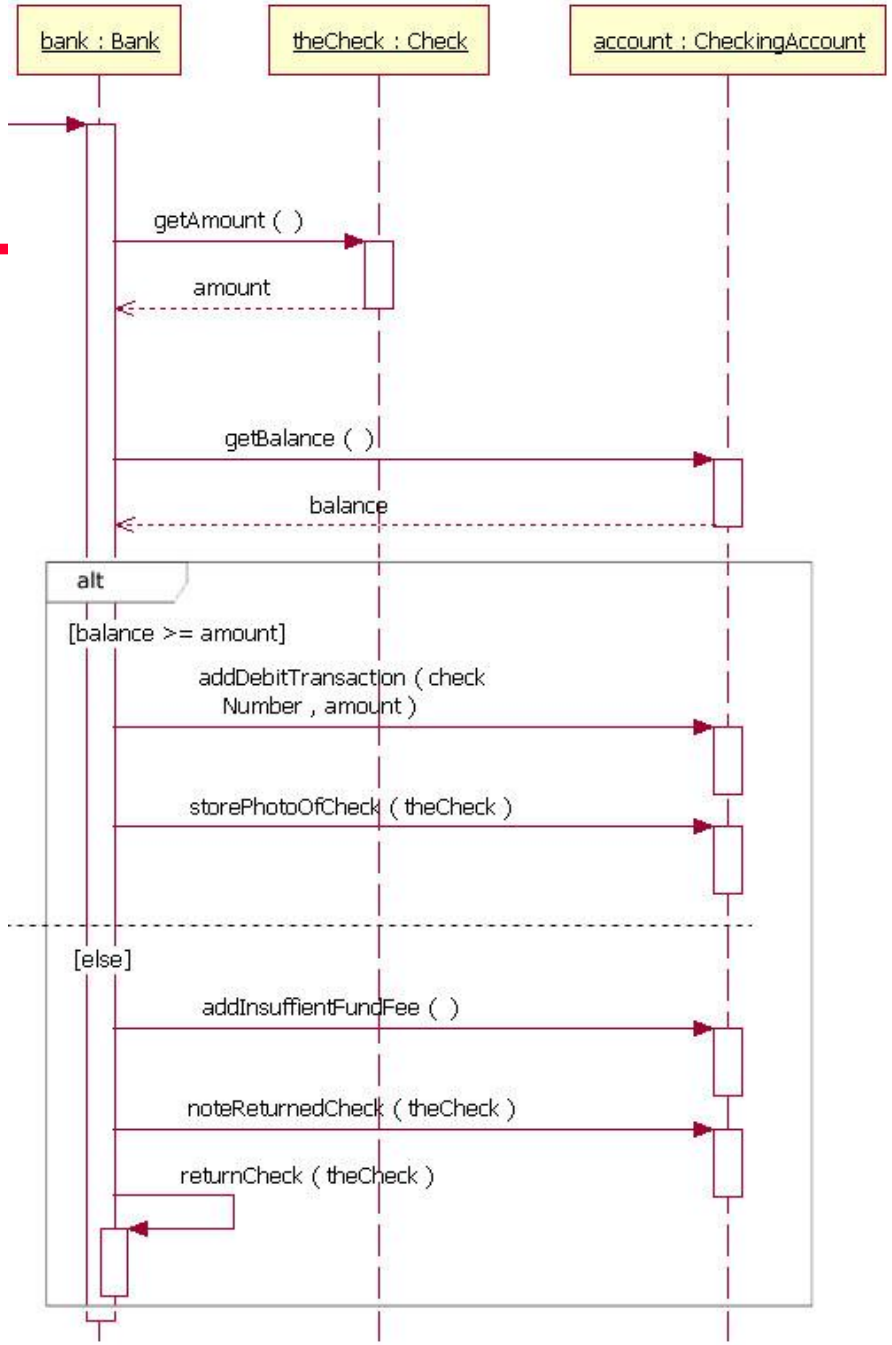
- Rappresenta l'esecuzione ciclica di più messaggi
- Si disegna raggruppando con un blocco (riquadro, box) i messaggi (metodi) su cui si vuole iterare
- Si può aggiungere la condizione che definisce l'iterazione sull'angolo in alto a sinistra del blocco
- La condizione si rappresenta al solito tra parentesi quadre



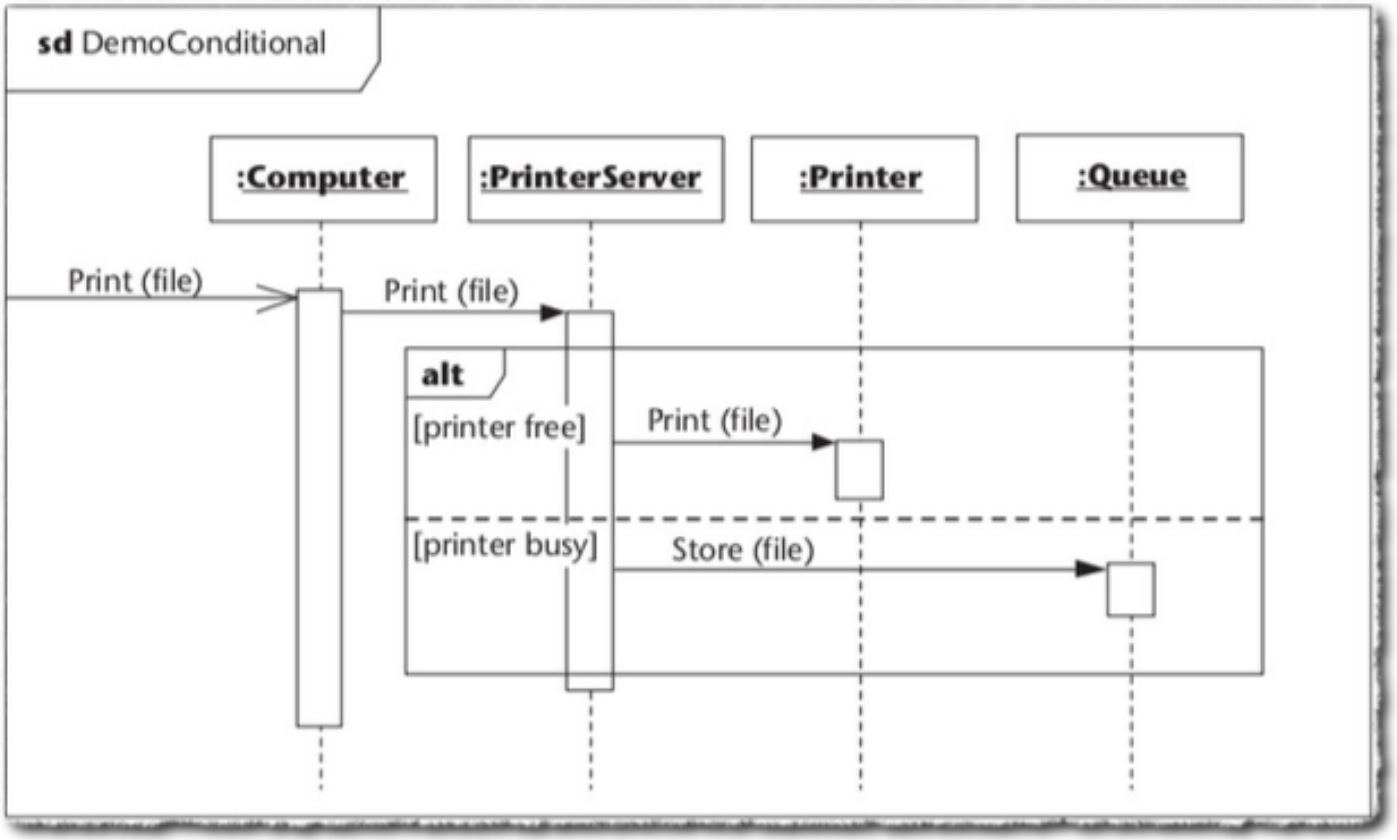
Notazione per le iterazioni (ricorrenze)

Sequence diagrams: cicli e condizioni

- Cicli e condizioni si indicano con un riquadro (frame) che racchiude una sottosequenza di messaggi.
- Nell'angolo in alto è indicato il costrutto. Tra i costrutti possibili:
  - Loop (ciclo while-do o do-while): la condizione è indicata tra parentesi quadra all'inizio o alla fine;
  - Alt (if-then-else): la condizione si indica in cima; se ci sono anche dei rami else allora si usa una linea tratteggiata per separare la zona then dalla zona else indicando eventualmente un'altra condizione accanto alla parola else;
  - Opt (if-then): racchiude una sottosequenza che viene eseguita solo se la condizione indicata in cima è verificata
    - » Sono possibili anche altri costrutti per indicare parallelismo, regioni critiche, etc..
  - In realtà, è buona norma utilizzare altri tipi di diagramma quando l'algoritmo da modellare si fa complesso.

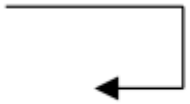

bank : Bank
theCheck : Check
account : CheckingAccount
getAmount ( )
amount
getBalance ( )
balance
alt
[balance >= amount]
addDebitTransaction (check Number , amount)
storePhotoOfCheck (theCheck)
[else]
addInsuffientFundFee ( )
noteReturnedCheck (theCheck)
returnCheck (theCheck)

Alt (if-then-else):

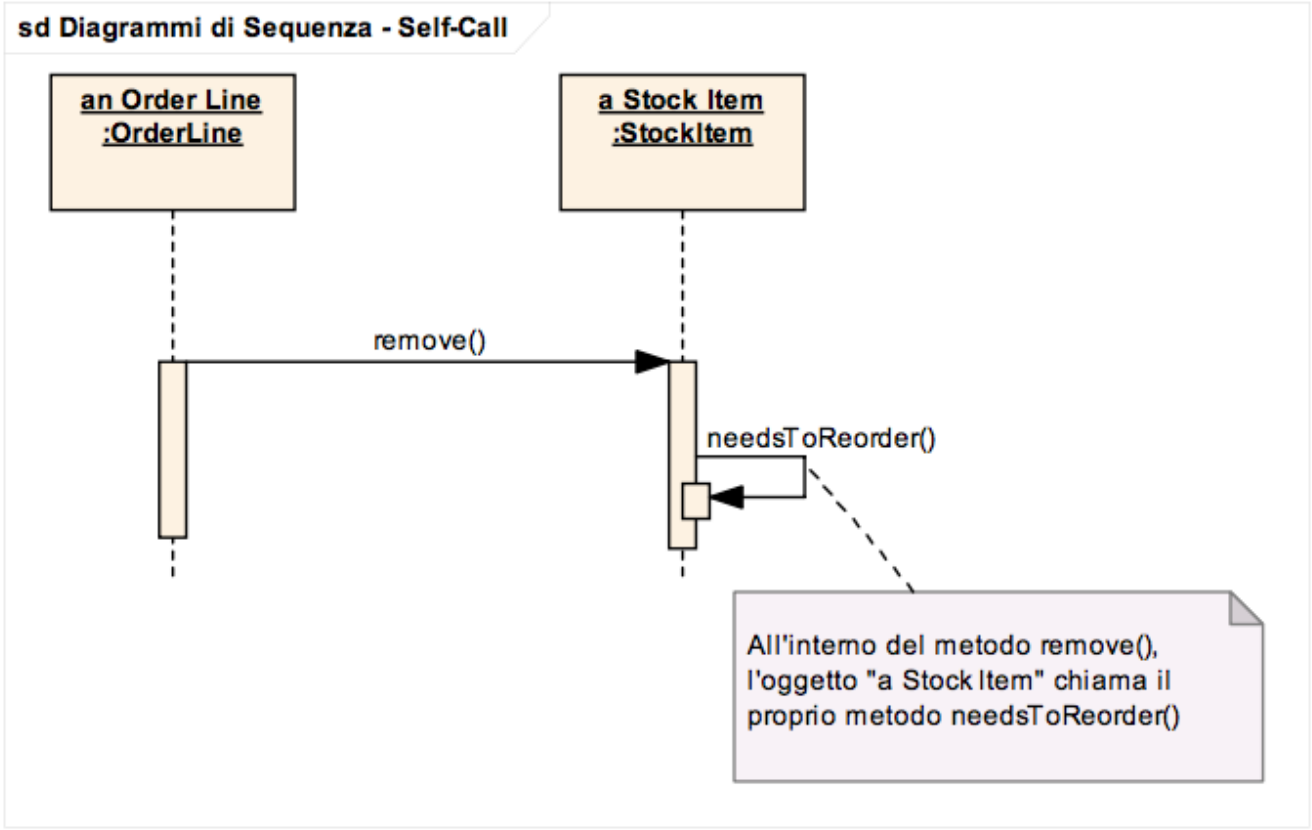


Auto-chiamata

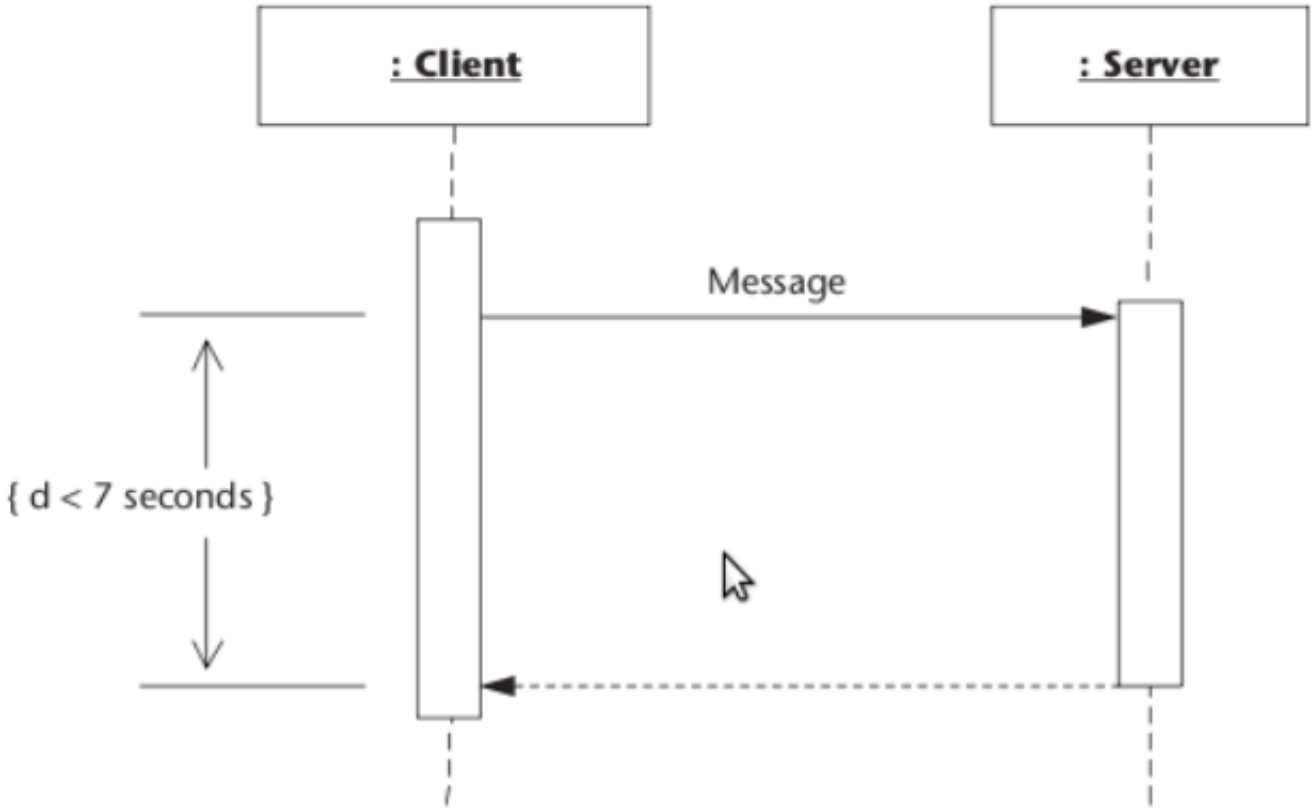
- Descrive un oggetto che invoca un proprio metodo
- Chiamante e chiamato in questo caso coincidono
- Si rappresenta con una "freccia circolare" che rimane all'interno del life time di uno stesso metodo
- Viene usata anche per rappresentare la ricorsione



Auto-chiamata

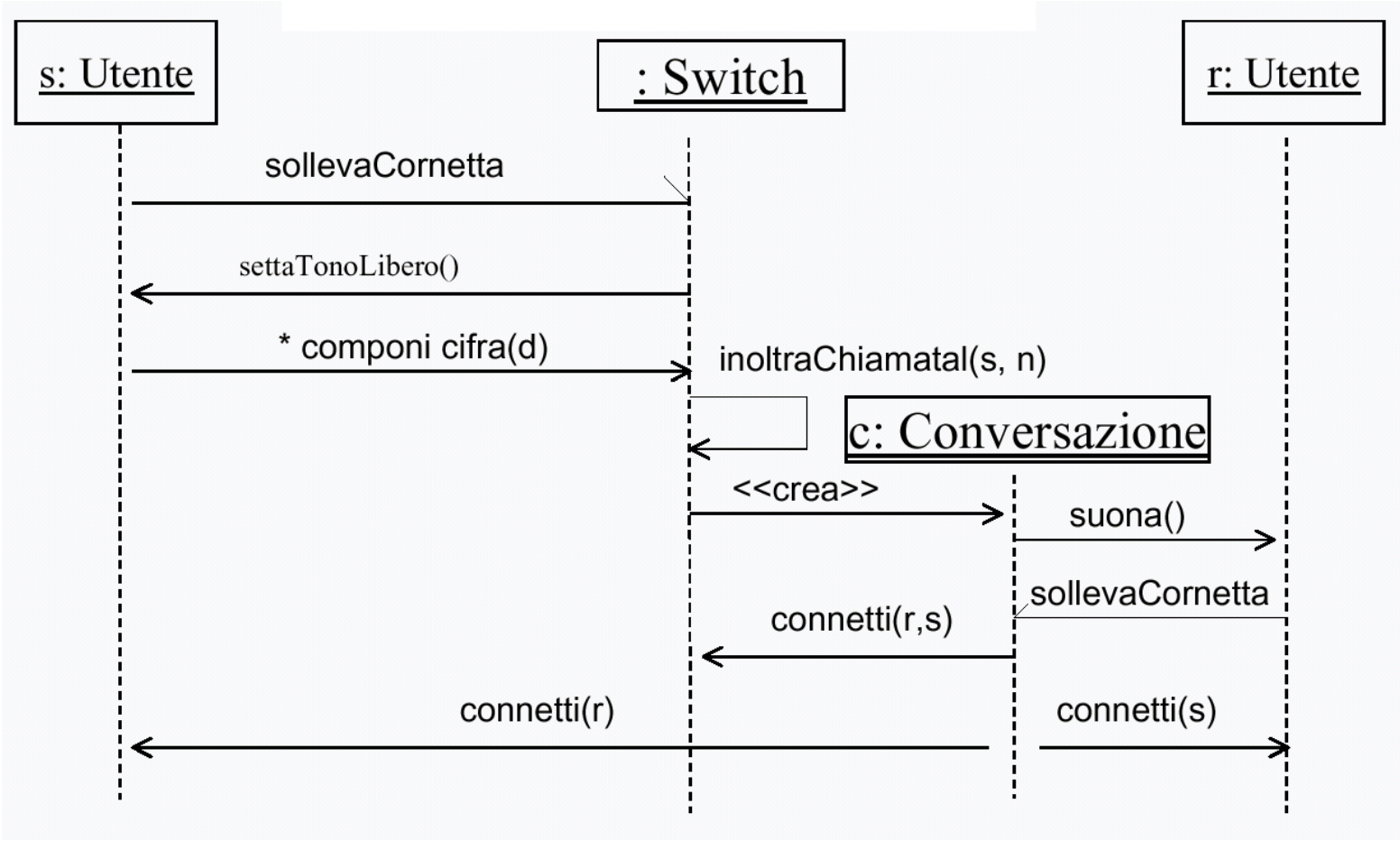


Esprimete vincoli sul tempo di risposta



Un esempio

Scenario “chiamata effettuata con successo”
del caso d’uso “effettua chiamata interna”



Sequence Diagrams: riassumendo

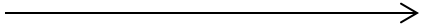
- Istanze di classi
  - Rappresentate da rettangoli col nome della classe e l'identificatore dell'oggetto sottolineati oppure semplicemente con un nome dal quale si evinca che si sta considerando un'istanza della classe (ad esempio anOrder oppure aProduct).
- Attori
  - Rappresentati come negli use case diagrams;
  - Sono riportati sulla sinistra, con frecce di interazione verso oggetti del sistema;
  - Possono anche non essere riportati, nel caso in cui lo scenario venga avviato a sua volta da un altro scenario.
- Messaggi
  - Rappresentati come frecce da un attore ad un oggetto, o fra due oggetti;
  - Un messaggio può anche insistere all'interno di uno stesso oggetto: in tal caso è indicato da una freccia circolare;
  - L'ordine dei messaggi (dall'alto verso il basso) ricalca l'ordine sequenziale con il quale vengono scambiati.

Messaggi e Azioni: riassumendo

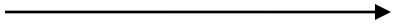
- Ad un messaggio possono corrispondere diversi tipi di azioni
  - ***Call***: invoca una operazione di un oggetto; un oggetto può inviare un messaggio a se stesso
  - ***Return***: restituisce un valore al chiamante
  - ***Send***: invia un segnale ad un oggetto (inizio esecuzione)
  - ***Create***: crea un oggetto
  - ***Destroy***: distrugge un oggetto; un oggetto può distruggere se stesso
- I messaggi possono essere preceduti da condizioni
  - [x > 0] messaggio()
- I messaggi possono indicare iterazioni
  - * messaggio()

Tipi di messaggi: riassumendo

- ***Semplice***: rappresenta un flat flow of control; il controllo è passato dal mittente al ricevente



- ***Sincrono***: rappresenta un nested flow of control; il controllo è passato dal mittente al ricevente ed il mittente aspetta che il ricevente gli restituisca il controllo



  - possono generarsi sequenze innestate di messaggi: il ricevente invia un altro messaggio ad un altro oggetto

- ***Asincrono***: rappresenta un non-nested flow of control tramite un signal; il mittente 'segnala' il ricevente tramite il message e continua senza aspettare il ricevente, che può o meno ritornare informazioni

A volte:

Sequence Diagrams

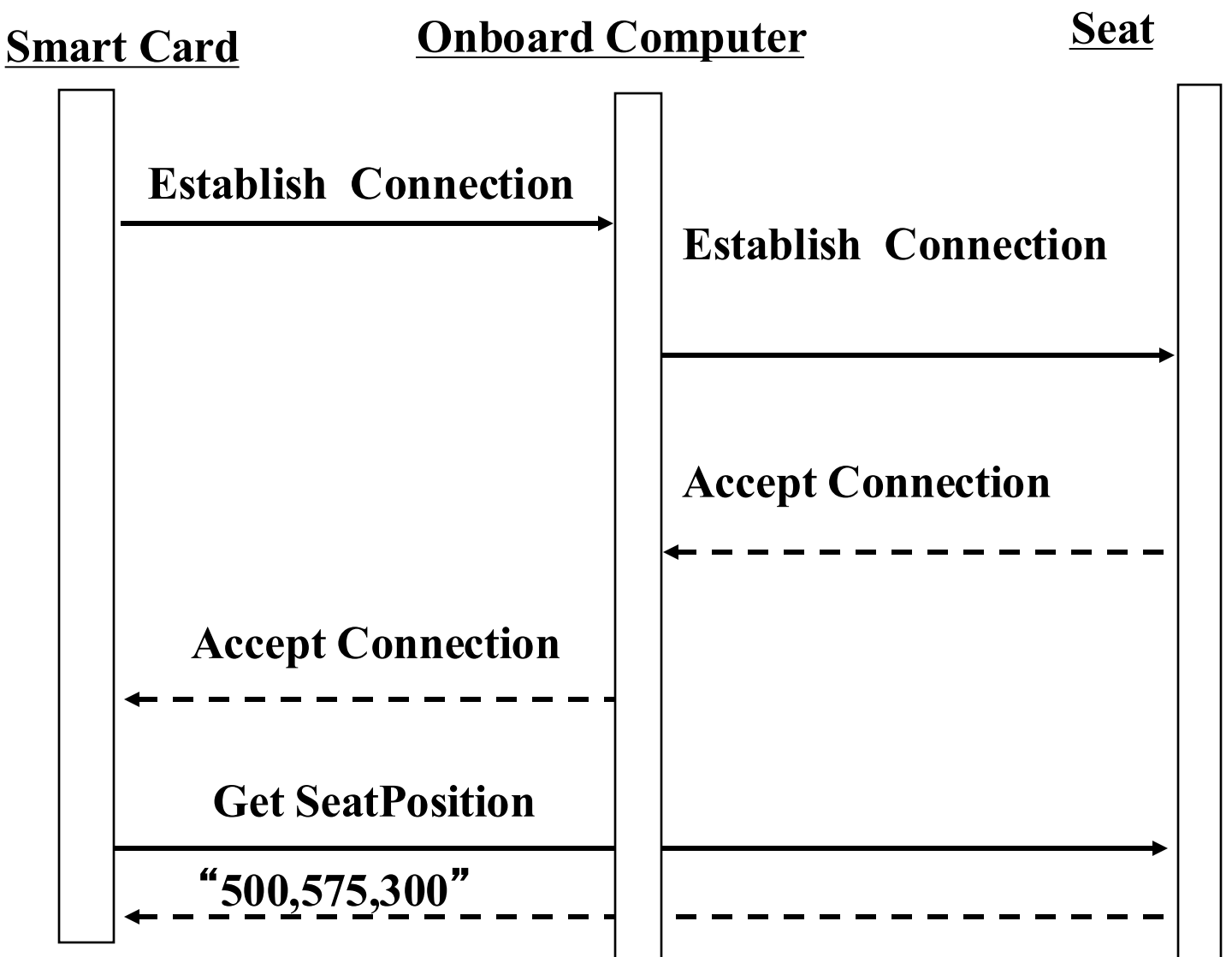
- From the flow of events in the use case or scenario proceed to the sequence diagram
- A sequence diagram is a graphical description of objects participating in a use case or scenario using a DAG (direct acyclic graph) notation
- Relation to object identification:
  - Objects/classes have already been identified during object modeling
  - Objects are identified as a result of dynamic modeling
- Heuristic:
  - A event always has a sender and a receiver.
  - The representation of the event is sometimes called a message
  - Find them for each event => These are the objects participating in the use case

Heuristics for Sequence Diagrams

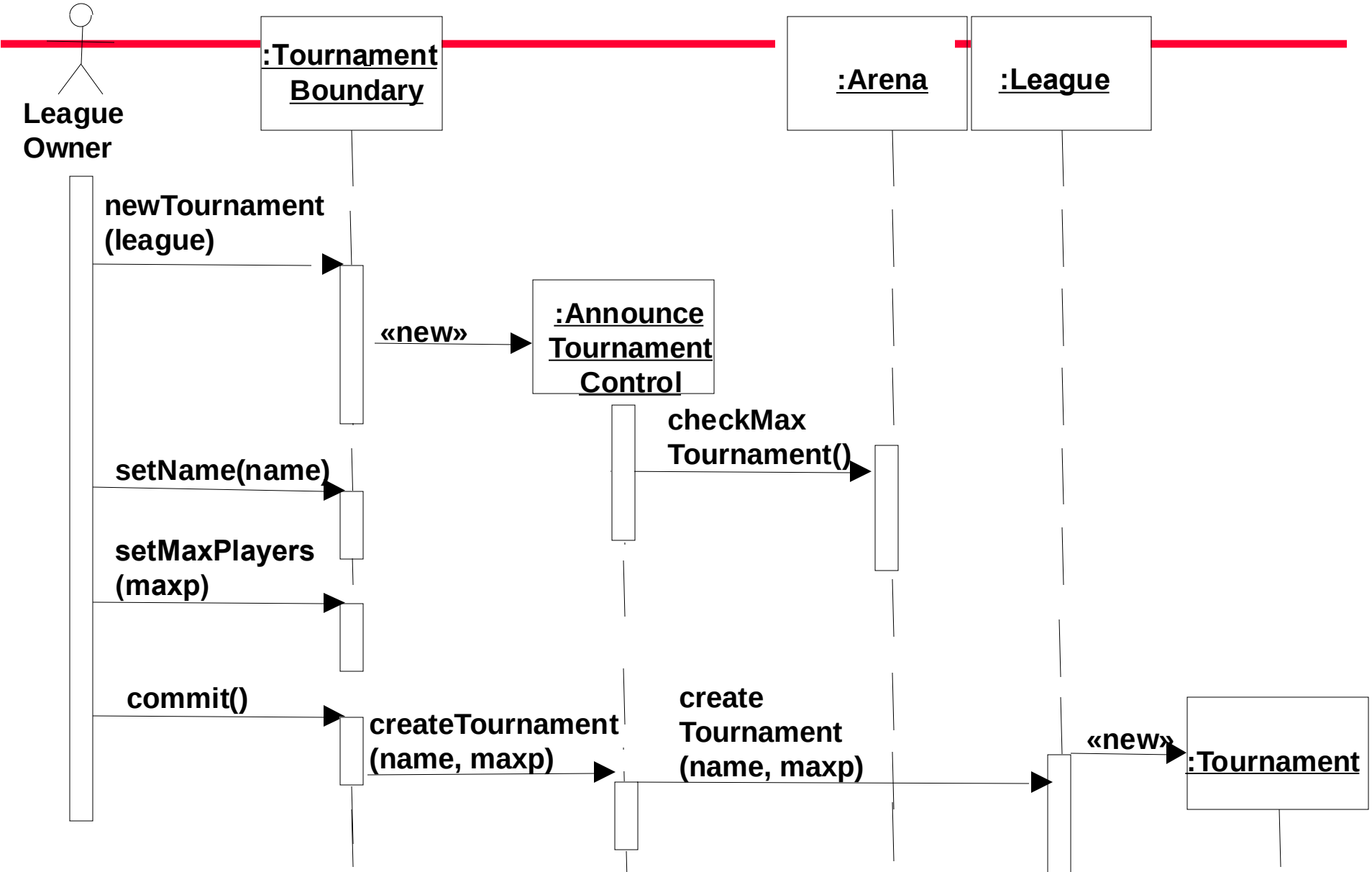
- Layout:
  - 1st column: Should correspond to the actor who initiated the use case
  - 2nd column: Should be a boundary object
  - 3rd column: Should be the control object that manages the rest of the use case
- Creation:
  - Control objects are created at the initiation of a use case
  - Boundary objects are created by control objects
- Access:
  - Entity objects are accessed by control and boundary objects,
  - Entity objects should never call boundary or control objects: This makes it easier to share entity objects across use cases and makes entity objects resilient against technology-induced changes in boundary objects.

Is this a good Sequence Diagram?

- First column is not the actor
- It is not clear where the boundary object is
- It is not clear where the control object is

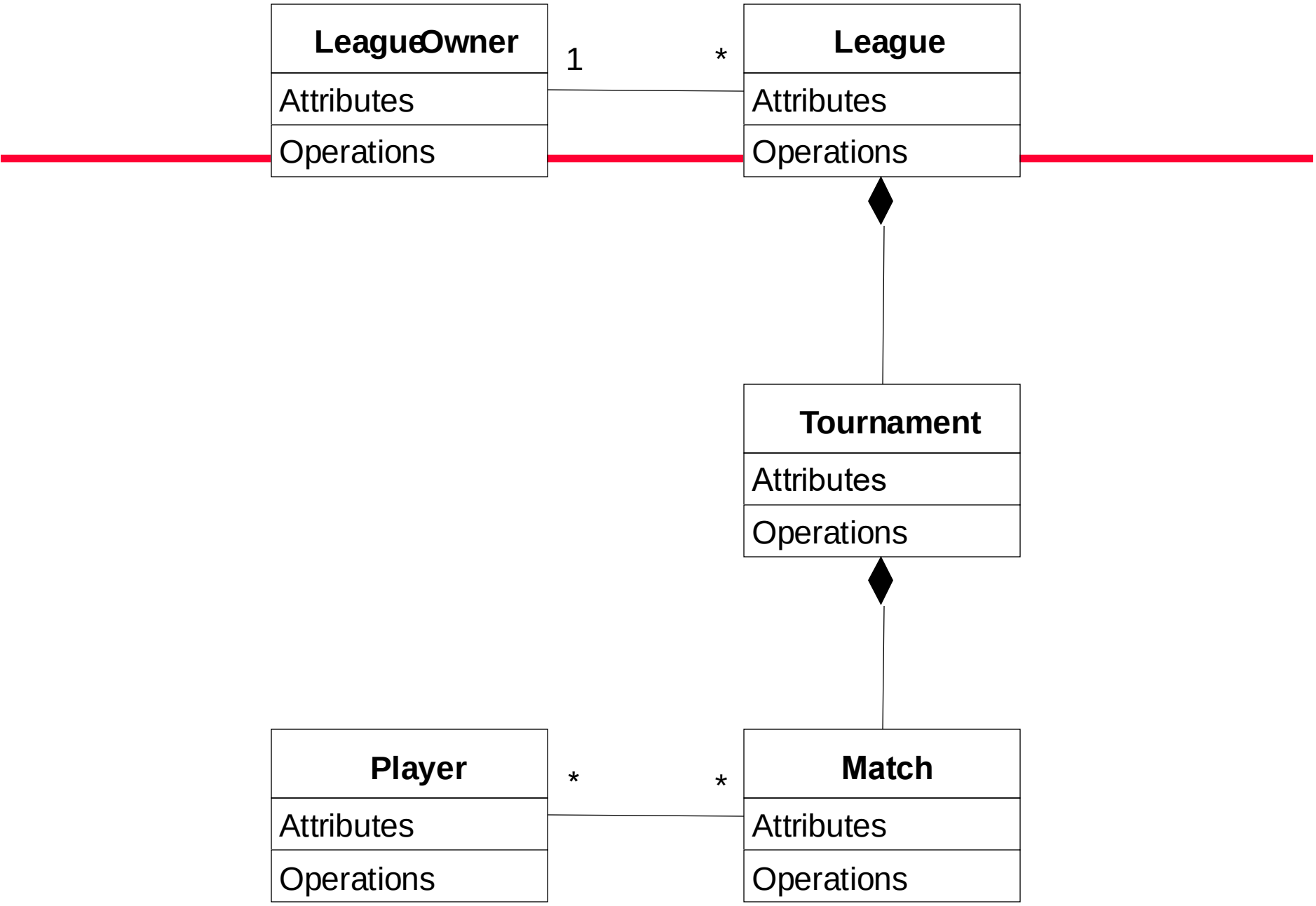


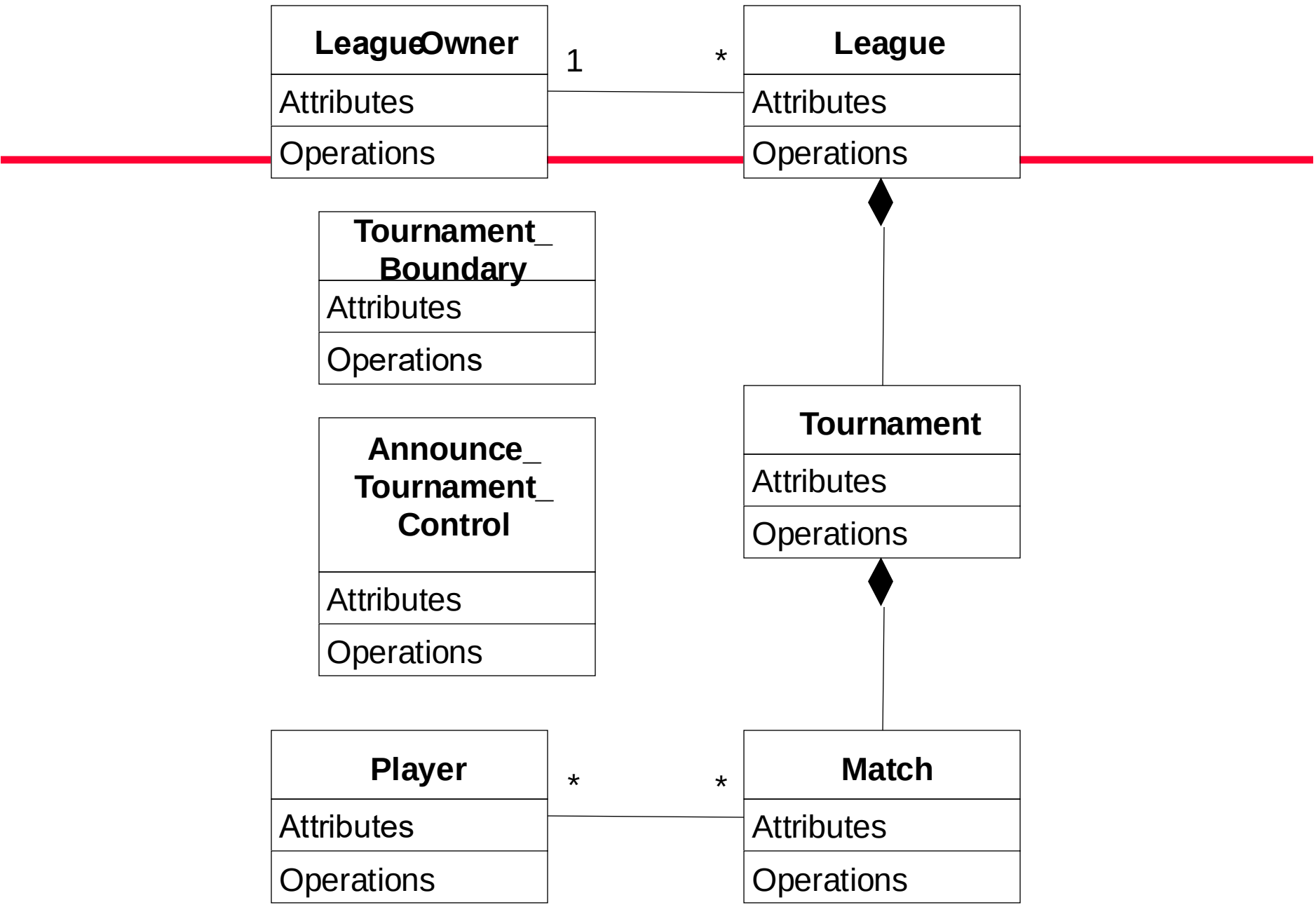
An ARENA Sequence Diagram : Create Tournament



Impact on ARENA's Object Model

- Let's assume, before we formulated the previous sequence diagram, ARENA's object model contained the objects
  - League Owner, Arena, League, Tournament, Match and Player
- The Sequence Diagram identified new Classes
  - Tournament Boundary, Announce_Tournament_Control

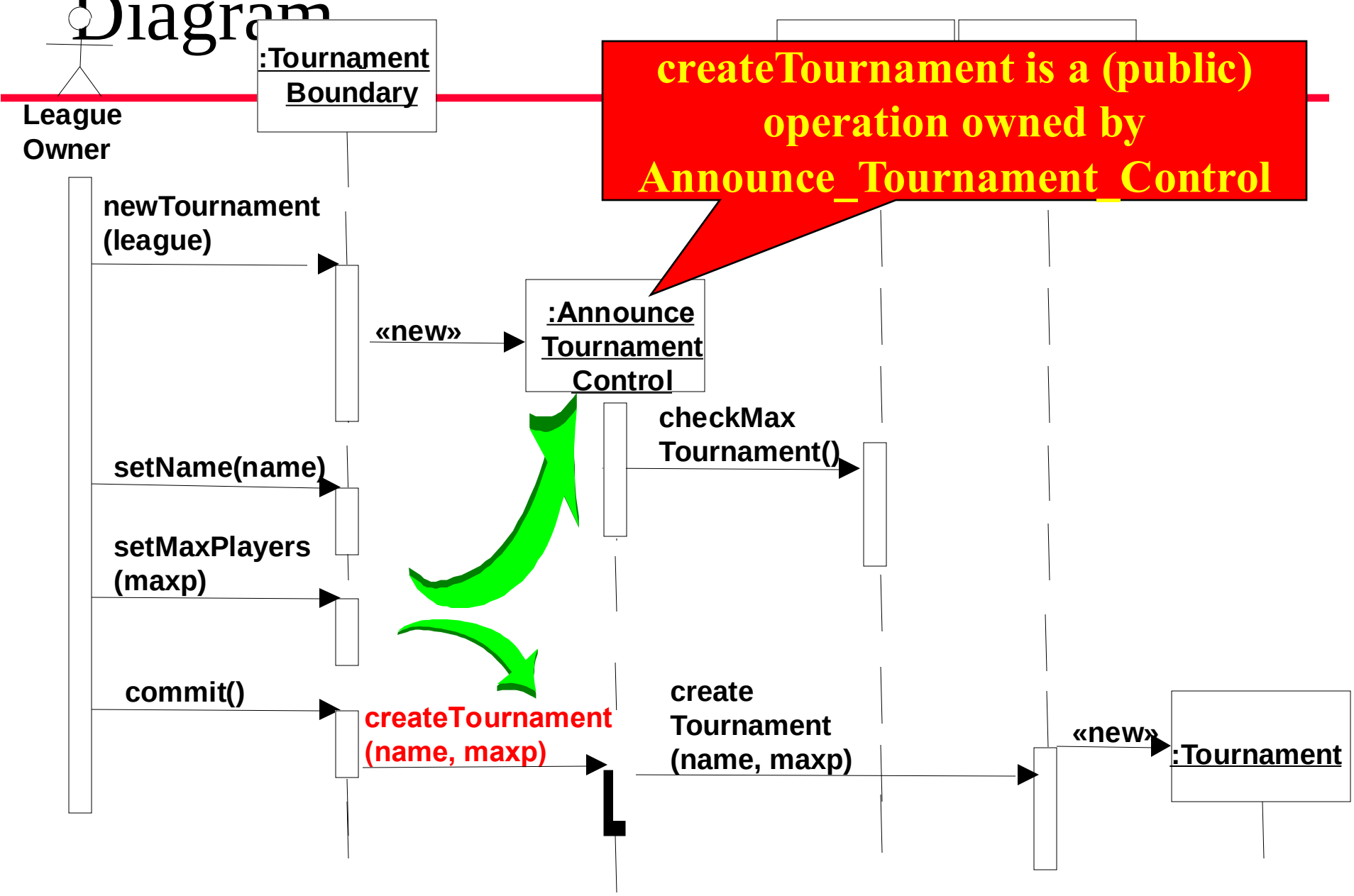

LeagueOwner
Attributes
Operations
1
*
League
Attributes
Operations
Tournament
Attributes
Operations
Player
Attributes
Operations
*
*
Match
Attributes
Operations

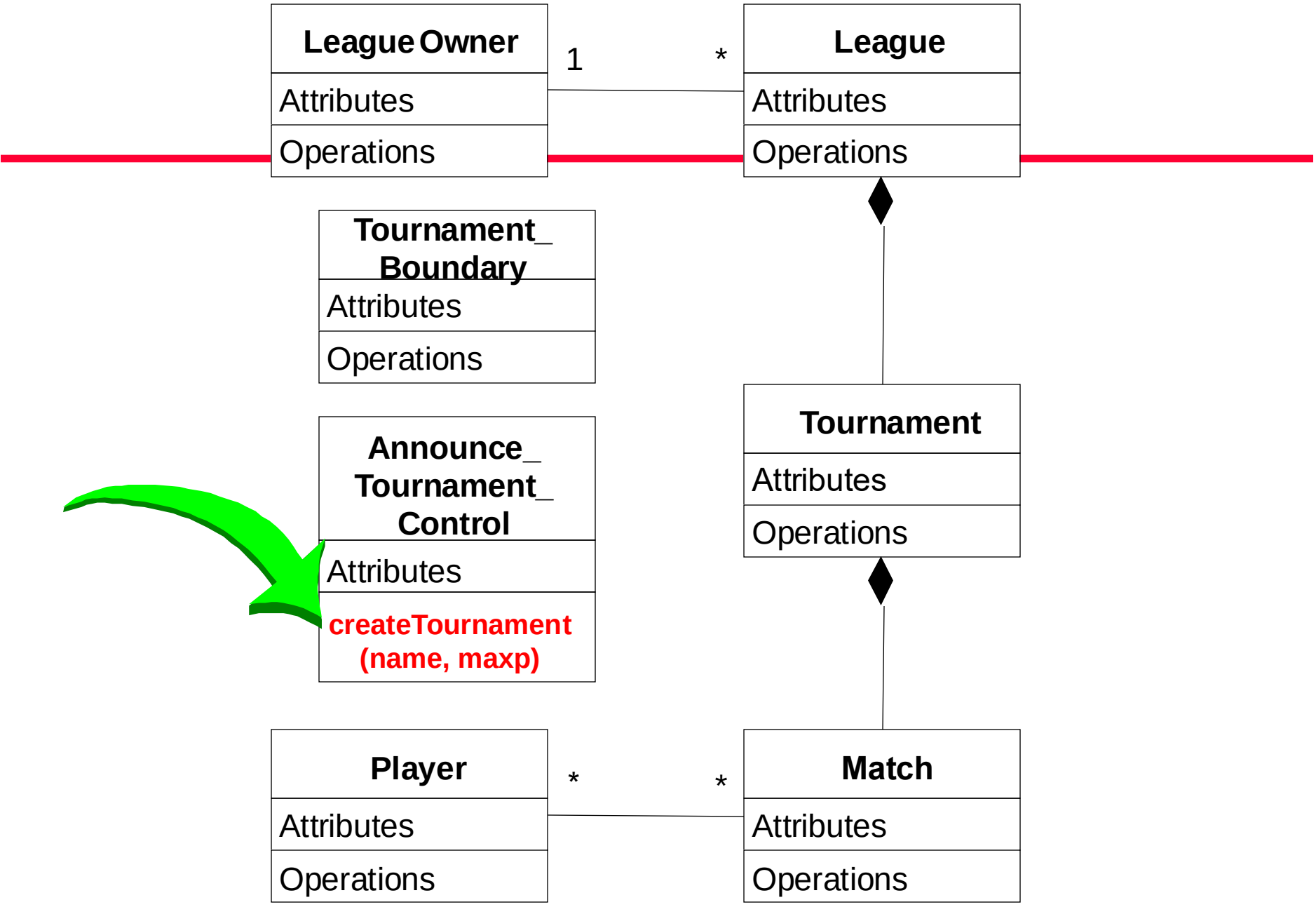

LeagueOwner
Attributes
Operations
1
*
League
Attributes
Operations
Tournament_ Boundary
Attributes
Operations
Announce_ Tournament_ Control
Attributes
Operations
Tournament
Attributes
Operations
Player
Attributes
Operations
*
*
Match
Attributes
Operations

Impact on ARENA's Object Model (ctd)

- The Sequence Diagram also supplied us with a lot of new events
  - newTournament(league)
  - setName(name)
  - setMaxPlayers(max)
  - Commit
  - checkMaxTournaments()
  - createTournament
- Question: Who owns these events?
- Answer: For each object that receives an event there is a public operation in the associated class.
  - The name of the operation is usually the name of the event.

Example from the Sequence Diagram



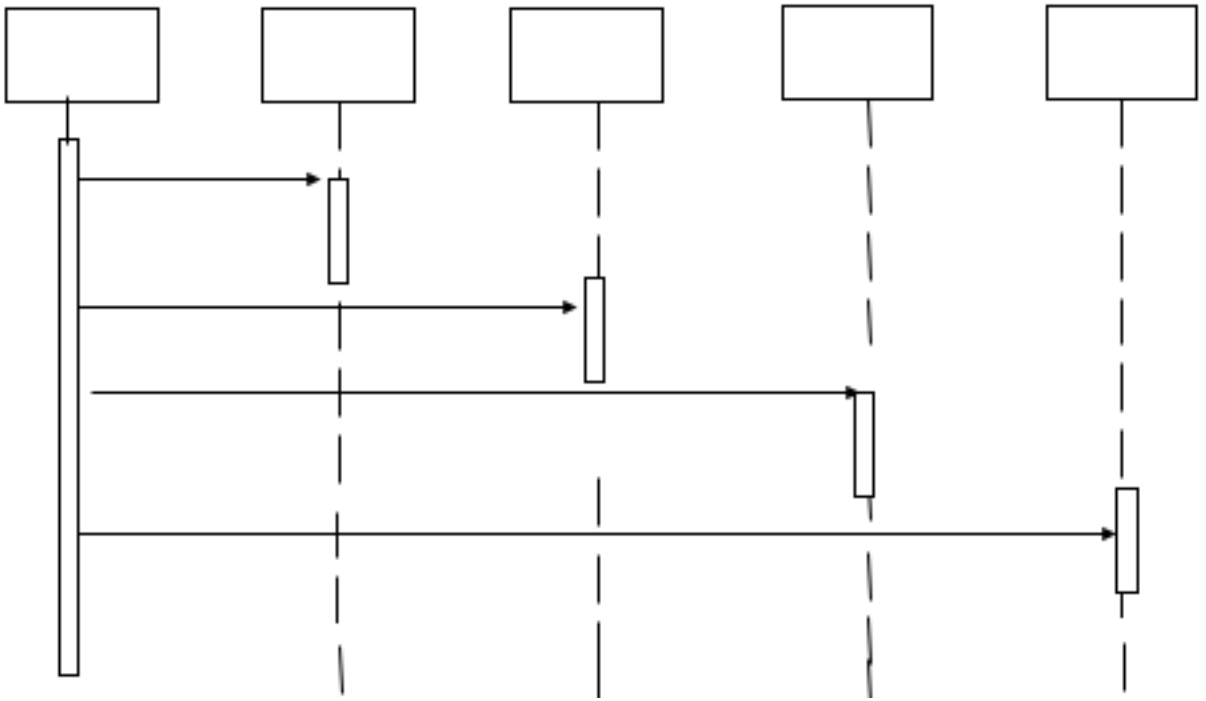

League Owner
Attributes
Operations
1
*
League
Attributes
Operations
Tournament_ Boundary
Attributes
Operations
Announce_ Tournament_ Control
Attributes
createTournament (name, maxp)
Tournament
Attributes
Operations
Player
Attributes
Operations
*
*
Match
Attributes
Operations

What else can we get out of sequence diagrams?

- Sequence diagrams are derived from the use cases. We therefore see the structure of the use cases.
- The structure of the sequence diagram helps us to determine how decentralized the system is.
- We distinguish two structures for sequence diagrams: Fork and Stair Diagrams (Ivar Jacobsen)

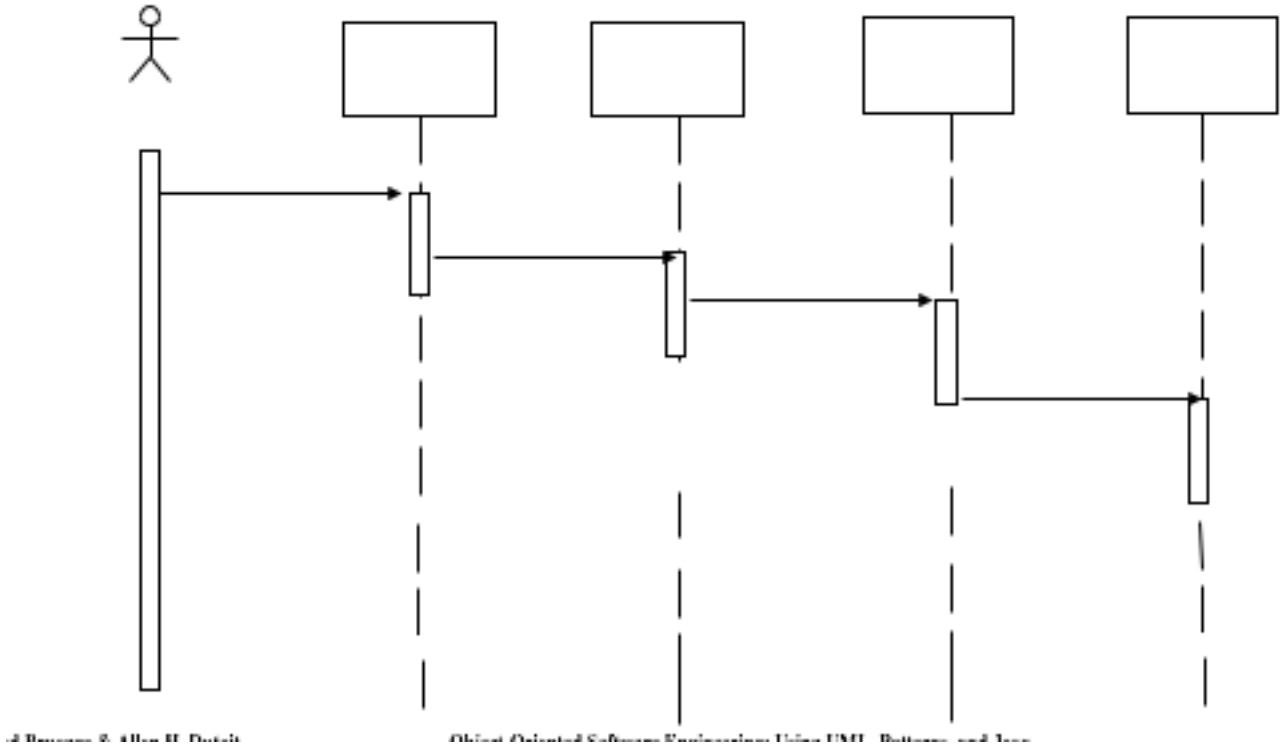
Fork Diagram

- Much of the dynamic behavior is placed in a single object, usually the control object. It knows all the other objects and often uses them for direct questions and commands.



Stair Diagram

- The dynamic behavior is distributed. Each object delegates some responsibility to other objects. Each object knows only a few of the other objects and knows which objects can help with a specific behavior.



Fork or Stair?

- Which of these diagram types should be chosen?
- Object-oriented fans claim that the stair structure is better
  - The more the responsibility is spread out, the better
- However, this is not always true.
- Decentralized control structure
  - The operations have a strong connection
  - The operations will always be performed in the same order
- Centralized control structure (better support of change)
  - The operations can *change* order
  - New operations can be inserted as a result of new requirements

Da use case a sequence diagram

- Un sequence diagram per use-case
- Partire da un sequence di base, con un solo attore e un solo oggetto
  - L'oggetto rappresenta l'intero sistema, così come lo vede l'attore, dall'esterno
  - Punto di partenza: black-box

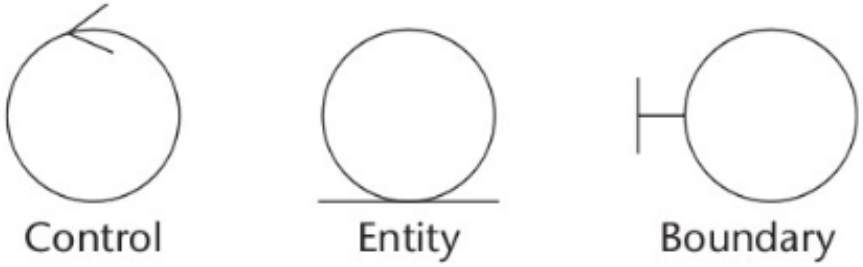
Da use case a sequence diagram

- Un messaggio dall'attore al sistema con cui il caso d'uso viene attivato
- Trattare il caso base, in modo “ottimistico”
  - Tralasciare eccezioni e ogni volta che ci sono scelte, descrivere solo ciò che accade nel caso “positivo”
  - Ulteriori dettagli possono essere aggiunti in seguito (descrivendo frammenti alternativi)

Da use case a sequence diagram

- Quali oggetti/entità mettere nel sequence diagram?
  - Un attore che attiva il caso d'uso
  - Un oggetto che si occupa della presentazione (oggetto “boundary”)
  - Un oggetto che si occupa di coordinare l'attività svolta dal sistema (oggetto “control”)
  - Un oggetto responsabile della rappresentazione/gestione/memorizzazione dei dati (oggetto “entity”)

Da use case a sequence diagram



- Queste regole non sono la soluzione universale, e l'esperienza suggerisce eccezioni, ma...
- ...se non si da dove mettere le mani, le semplici regole delle slide precedenti sono già un ottimo punto di partenza..

Alcuni suggerimenti

- Assicurarsi che i metodi rappresentati nel diagramma siano gli stessi definiti nelle corrispondenti classi (con lo stesso numero e lo stesso tipo di parametri)
- Documentare ogni assunzione nella dinamica con note o condizioni (ad es. il ritorno di un determinato valore al termine di un metodo, il verificarsi di una condizione all'uscita da un loop, ecc.)
- Mettere un titolo per ogni diagramma (ad es. "sd Diagrammi di Sequenza – Eliminazione di un Oggetto")

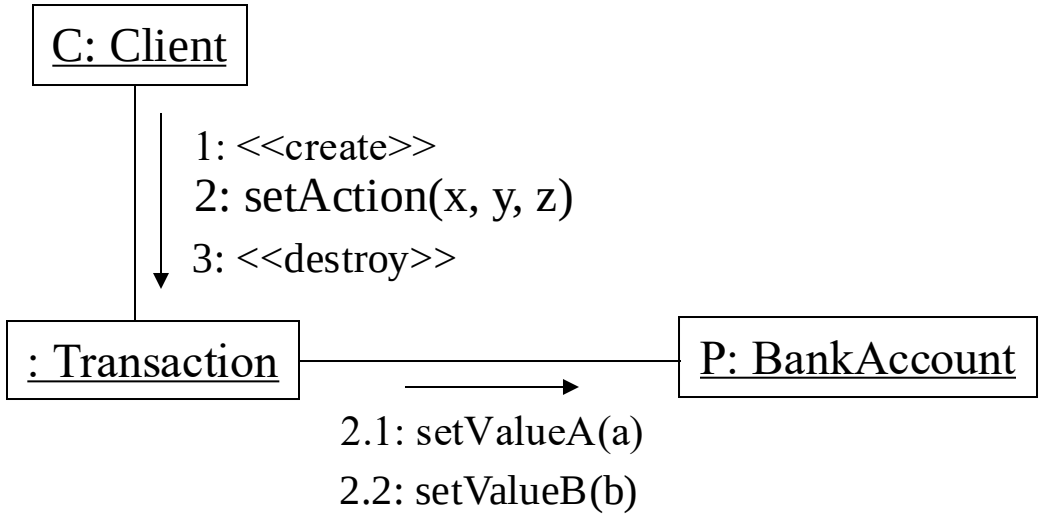
Alcuni suggerimenti

- Scegliere nomi espressivi per le condizioni e per i valori di ritorno
- Non inserire troppi dettagli in un unico diagramma (flussi condizionati, condizioni, logica di controllo)
- Non bisogna rappresentare tutto quello che si rappresenta nel codice
- Se il diagramma è complesso, scomporlo in più diagrammi semplici (ad es. uno per il ramo if, un altro per il ramo else, ecc.)

Collaboration Diagram

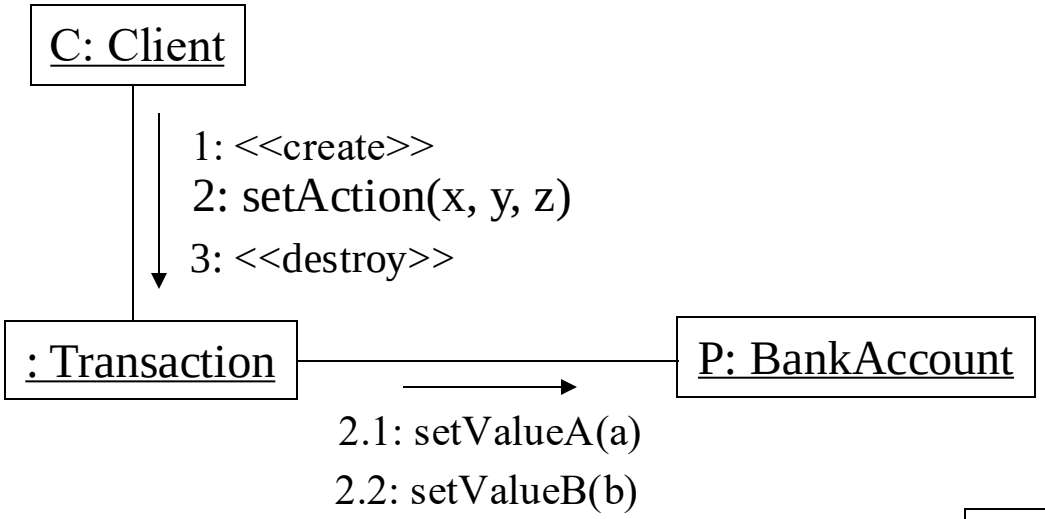
- Specifica gli oggetti che collaborano tra loro in un dato scenario, ed i messaggi che si indirizzano
- La sequenza dei messaggi è meno evidente che nel diagramma di sequenza, mentre sono più evidenti i legami tra gli oggetti
  - Per meglio visualizzare l'ordine sequenziale dello scambio dei messaggi è possibile 'numerare' i message anteponendo al loro nome un numero che indica l'ordine nella sequenza
- Può essere utilizzato in fasi diverse (analisi, disegno di dettaglio) e rappresentare diverse tipologie di oggetti
- Adatti per concorrenza e thread, invocazioni innestate

Collaboration Diagrams: Messaggi e Link



- Per le sequenze di messaggi innestati, la numerazione è espressa in una dot-notation

Collaboration Diagrams: Messaggi e Link



- Per le sequenze di messaggi innestati, la numerazione è espressa in una dot-notation