UML Sequence Diagrams

Idaho State University | Computer Science

Isaac Griffith

CS 3321
Department of Computer Science
Idaho State University

ROAR

Idaho State University
Computer Science

Outline

- UML Sequence Diagrams

ROAR

Introduction

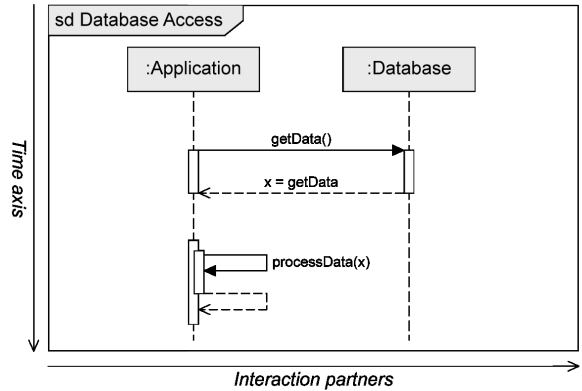
- Modeling inter-object behavior
  - interactions between objects
- Interaction
  - Specifies how messages and data are exchanged between interaction partners
- Interaction partners
  - Human (lecturer, administrator, ...)
  - Non-human (server, printer, executable software, ...)
- Examples of interactions
  - Conversation between persons
  - Message exchange between humans and a software system
  - Communication protocols
  - Sequence of method calls in a program
  - ...

Interaction Diagrams

- Used to specify interactions
- Modeling concrete scenarios
- Describing communication sequences at different levels of detail
- Interaction Diagrams show the following:
  - Interaction of a system with its environment
  - Interaction between system parts in order to show how a specific use case can be implemented
  - Interprocess communication in which the partners involved must observe certain protocols
  - Communication at class level (operation calls, inter-object behavior)

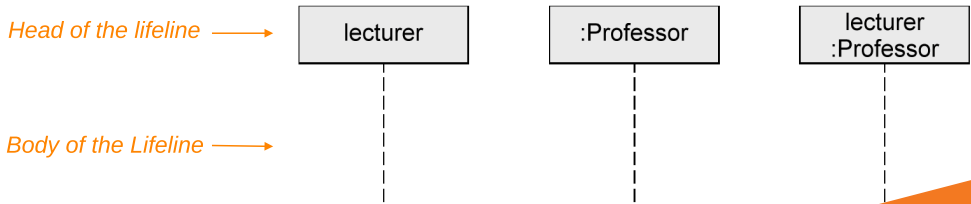
Sequence Diagram

- Two-dimensional diagram
  - Horizontal axis: involved interaction partners
  - Vertical axis: chronological order of the interaction
- Interaction = sequence of event specifications



Interaction Partners

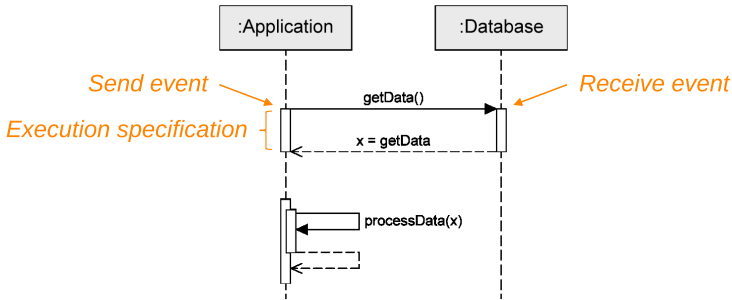
- Interaction partners are depicted as lifelines
- Head of the lifeline
  - Rectangle that contains the expression **roleName:Class**
  - Roles are a more general concept than objects
  - Object can take on different roles over its lifetime
- Body of the lifeline
  - Vertical, usually dashed line
  - Represents the lifetime of the object associated with it



Idaho State University
Computer Science

Exchanging Messages

- Interaction: sequence of events
- Message is defined via send event and receive event
- Execution specification
  - Continuous bar
  - Used to visualize when an interaction partner executes some behavior

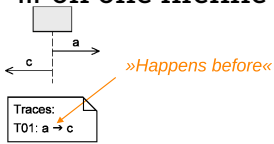


ROAR

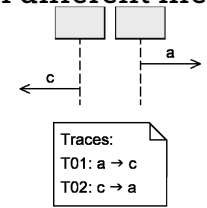
Exchanging Messages

Order of messages:

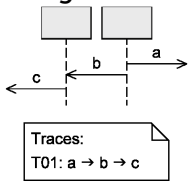
... on one lifeline



... on different lifelines

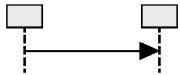


... on different lifelines which exchange messages

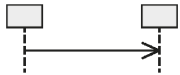


Messages

- Syncronous message
  - Sender waits until it has received a response message before continuing
  - Syntax of message name: `msg(par1, par2)`
    - `msg`: the name of the message
    - `par`: parameters separated by commas



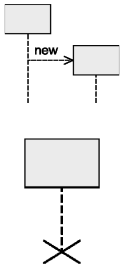
- Asynchronous message
  - Sender continues without waiting for a response message
  - Syntax of message name: `msg(par1, par2)`

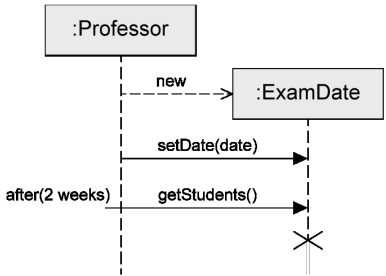


- Response message
  - May be omitted if content and location are obvious
  - Syntax: `att=msg(par1, par2) : val`
    - `att`: the return value can optionally be assigned to a variable
    - `msg`: the name of the message
    - `par`: parameters separated by commas
    - `val`: return value

Messages

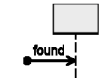
- Object creation
  - Dashed arrow
  - Arrowhead points to the head of the lifeline of the object to be created
  - Keyword **new**
- Object desctruction
  - Object is deleted
  - Large cross ($\times$) at the end of the lifeline

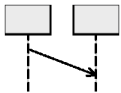


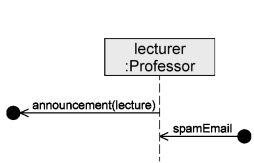


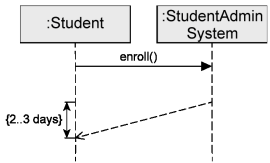
Messages

- Found message
  - Sender of a message is unknown or not relevant
- Lost message
  - Receiver of a message is unknown or not relevant
- Time-consuming message
  - "Message with duration"
  - Usually messages are assumed to be transmitted without any loss of time
  - Express that time elapses between the sending and the receipt of a message



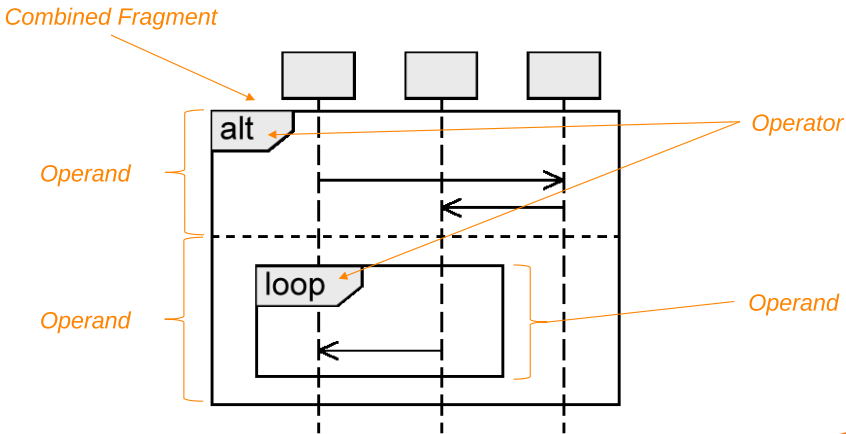






Combined Fragments

- Model various control structures
- 12 predefined types of operators



Branch/Loop Combined Fragments

Operator	Purpose
`alt`	Alternative interaction
`opt`	Optional interaction
`loop`	Repeated interaction
`break`	Exception interaction

Concurrency Combined Fragments

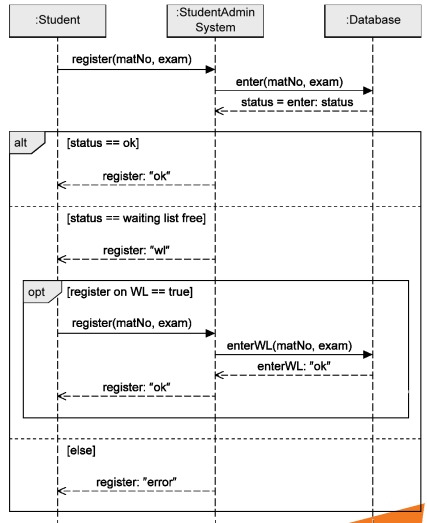
Operator	Purpose
`seq`	Weak order
`strict`	Strict order
`par`	Concurrent interaction
`critical`	Atomic interaction

Filters/Assertions Combined Fragments

Operator	Purpose
`ignore`	Irrelevant interaction
`consider`	Relevant interaction
`assert`	Asserted interaction
`neg`	Invalid interaction

`alt` Fragment

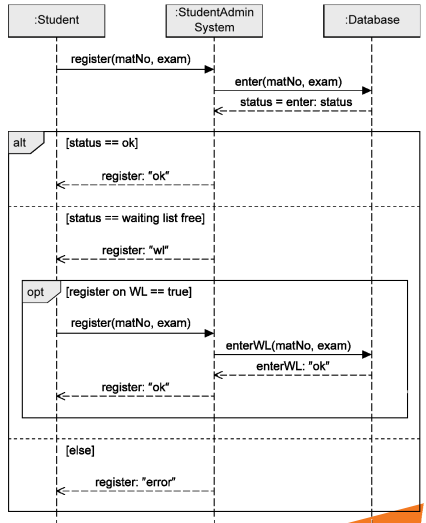
- To model alternative sequences
- Similar to switch stmt in Java
- Guards are used to select the one path to be executed
- Guards
  - Modeled in square brackets
  - default: `true`
  - predefined: `[else]`
- Multiple operands
- Guards have to be disjoint to avoid indeterministic behavior



Idaho State University
Computer Science

`opt` Fragment

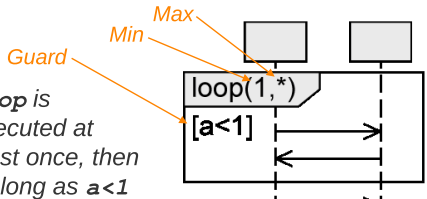
- To model an optional sequence
- Actual execution at runtime is dependent on the guard
- Exactly one operand
- Similar to `if` statement without `else` branch
- Equivalent to `alt` fragment with two operands, one of which is empty



ROAR

`loop` Fragment

- To express that a sequence is to be executed repeatedly
- Exactly one operand
- Keyword loop followed by the minimal/maximal number of iterations **(min..max)** or **(min, max)**
  - default: `(*)` .. no upper limit
- Guard
  - Evaluated as soon as the minimum number of iterations has taken place
  - Checked for each iteration within the `(min, max)` limits
  - If the guard evaluates to false, the execution of the loop is terminated

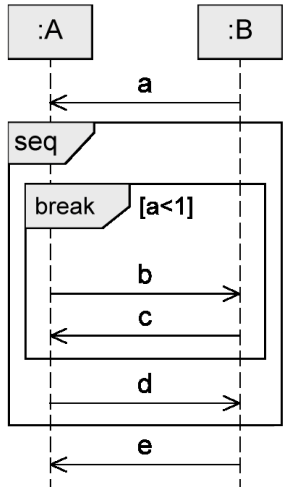


Notation alternatives:

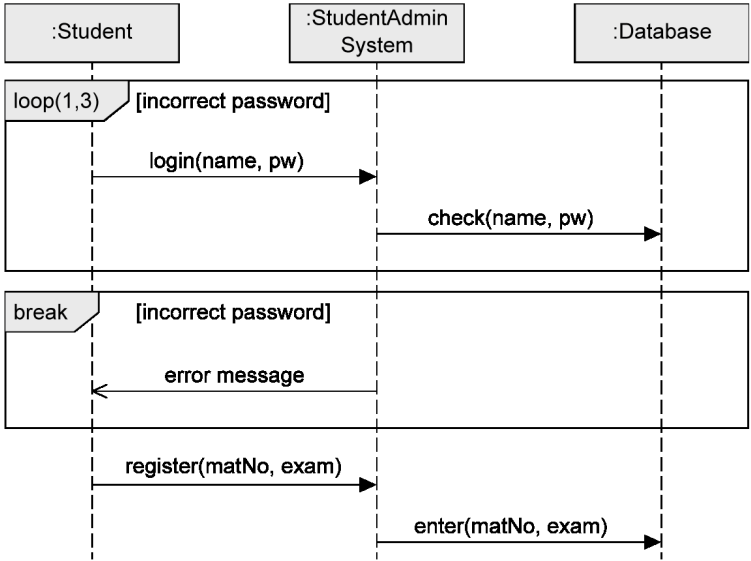
```
loop(3,8) = loop(3..8)
loop(8,8) = loop (8)
loop = loop (*) = loop(0,*)
```

`break` Fragment

- Simple form of exception handling
- Exactly one operand with a guard
- If the guard is true:
  - Interactions within this operand are executed
  - Remaining operations of the surrounding fragment are omitted
  - Interaction continues in the next higher level fragment
  - Different behavior than `opt` fragment

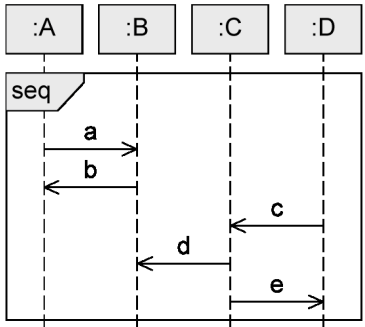


`loop/break` Example



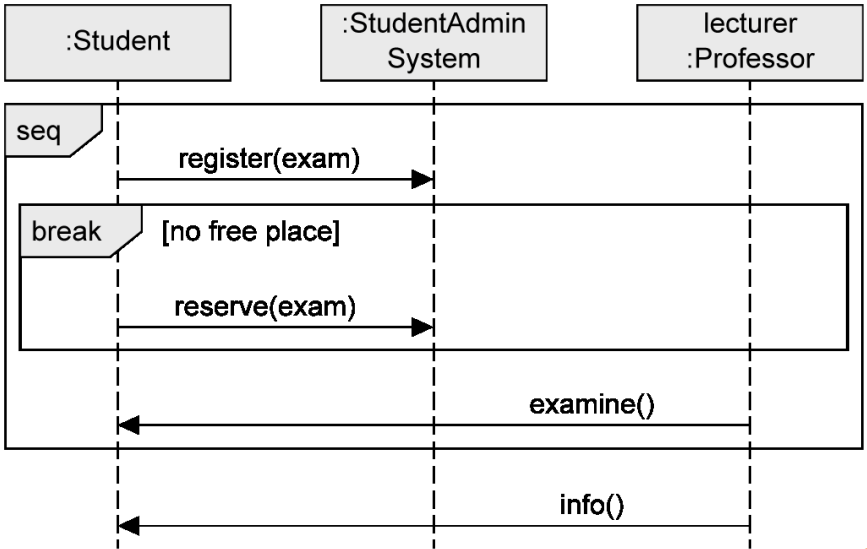
seq Fragment

- Default order of events
- Weak sequencing:
  1. The ordering of events within each of the operands is maintained in the result.
  2. Events on different lifelines from different operands may come in any order.
  3. Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand.



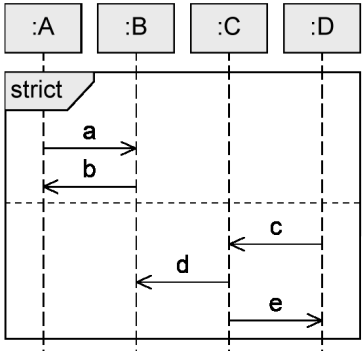
Traces:
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: c → a → b → d → e

`seq` Example



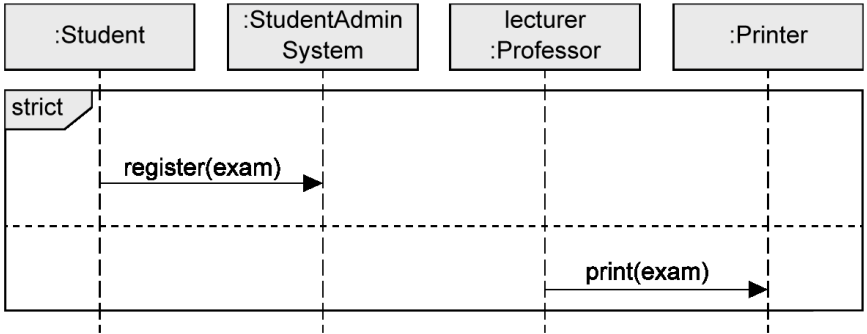
`strict` Fragment

- Sequential interaction with order
- Order of events occurrences on different lifelines between different operands is significant
  - Messages in an operand that is higher up on the vertical axis are always exchanged before the messages in an operand that is lower down on the vertical axis



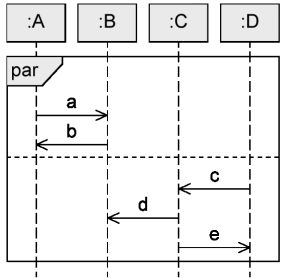
Traces:
T01: a → b → c → d → e

`strict` Example



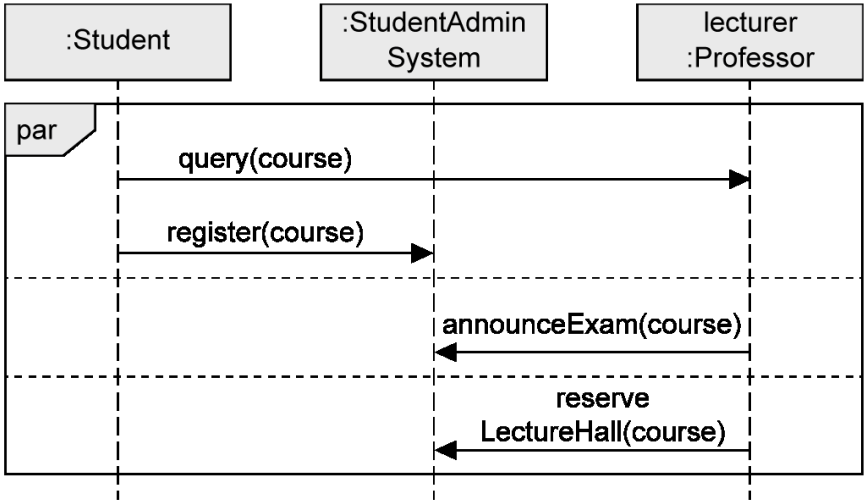
par Fragment

- To set aside chronological order between messages in different operands
- Execution paths of different operands can be interleaved
- Restrictions of each operand need to be respected
- Order of the different operands is irrelevant
- Concurrency, no true parallelism



Traces:
T01: a → b → c → d → e
T02: a → c → b → d → e
T03: a → c → d → b → e
T04: a → c → d → e → b
T05: c → a → b → d → e
T06: c → a → d → b → e
T07: c → a → d → e → b
T08: c → d → a → b → e
T09: c → d → a → e → b
T10: c → d → e → a → b

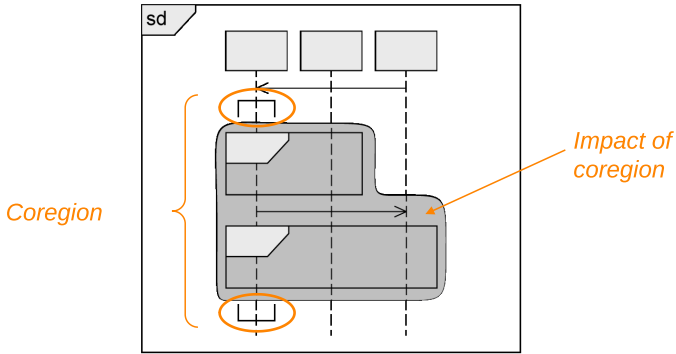
par Example



Idaho State University
Computer Science

Coregion

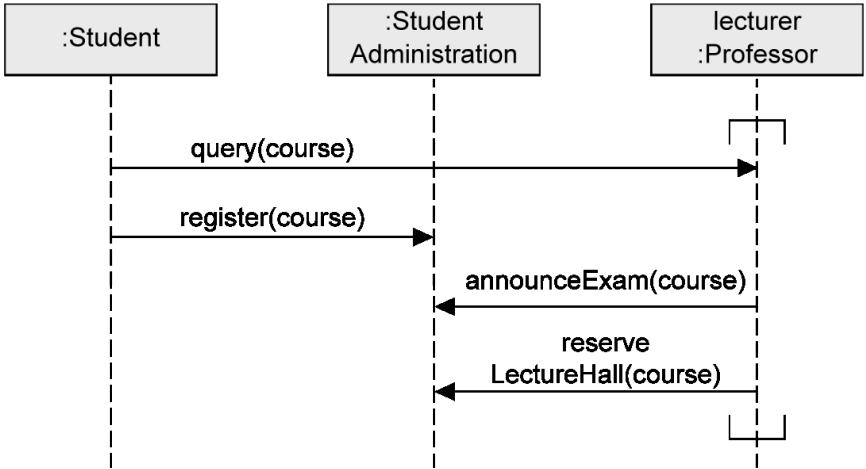
- To model concurrent events of a single lifeline
- Order of event occurrences within a coregion is not restricted
- Area of the lifeline to be covered by the coregion is marked by square brackets rotated 90 degrees



ROAR

Idaho State University
Computer Science

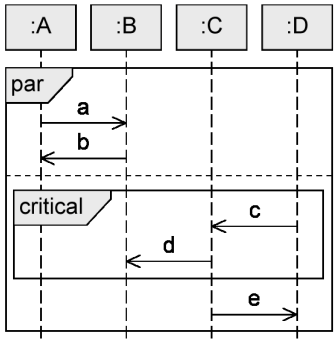
Coregion Example



ROAR

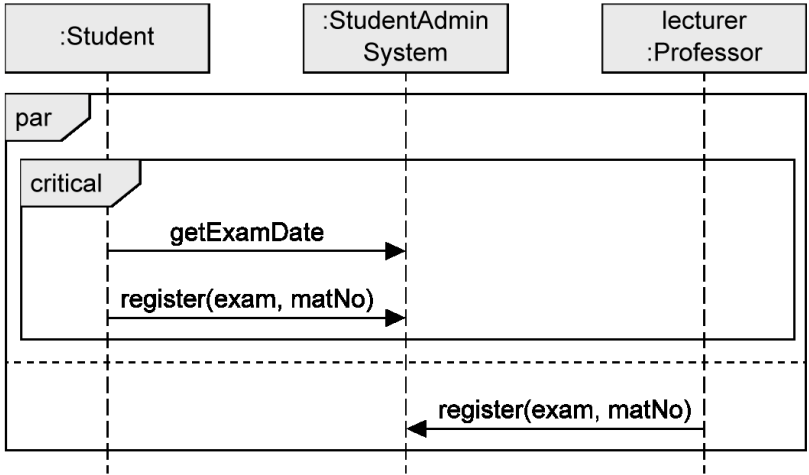
`critical` Fragment

- Atomic area in the interaction (one operand)
- To make sure that certain perats of an interaction are not interrupted by unexpected events
- Order within `critical:` default order `seq`



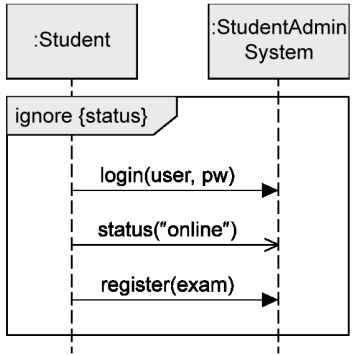
Traces:
T01: a → b → c → d → e
T02: a → c → d → b → e
T03: a → c → d → e → b
T04: c → d → a → b → e
T05: c → d → a → e → b
T06: c → d → e → a → b

`critical` Example



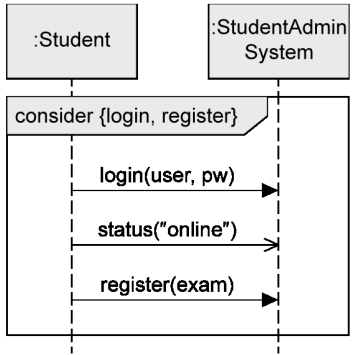
`ignore` Fragment

- To indicate irrelevant messages
- Messages may occur at runtime but have no further significance
- Exactly one operand
- Irrelevant messages in curly brackets after the keyword `ignore`



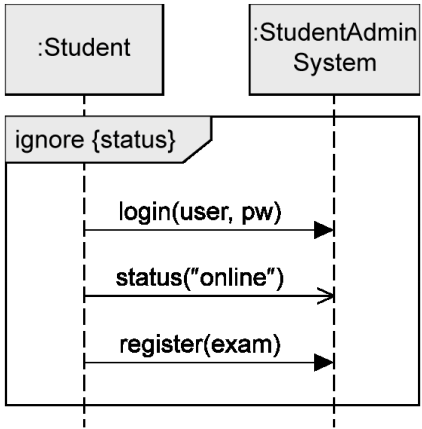
`consider` Fragment

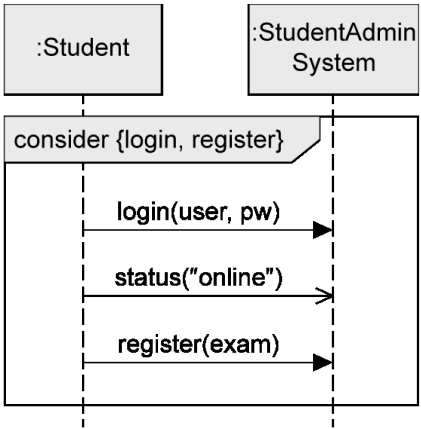
- To specify those messages that are of particular importance for the interaction under consideration
- Exactly one operand, dual to ignore fragment
- Considered messages in curly brackets after the keyword `consider`



Idaho State University
Computer Science

ignore vs. consider

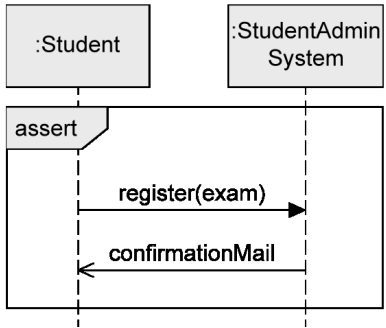




ROAR

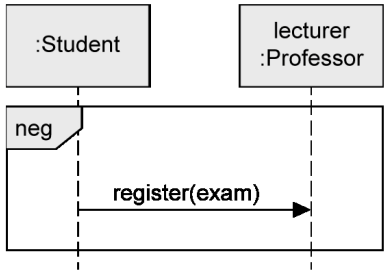
`assert` Fragment

- To identify certain modeled traces as mandatory
- Deviations that occur in reality but that are not included in the diagram are not permitted
- Exactly one operand



neg Fragment

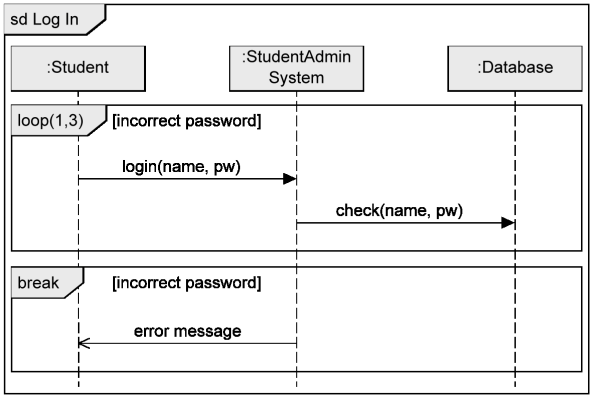
- To model invalid interactions
- Describing situations that must not occur
- Exactly one operand
- Purpose
  - Explicitly highlighting frequently occurring errors
  - Depicting relevant, incorrect sequences

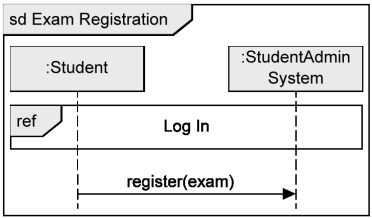


Idaho State University
Computer Science

Interaction Reference

- Integrates one sequence diagram in another sequence diagram

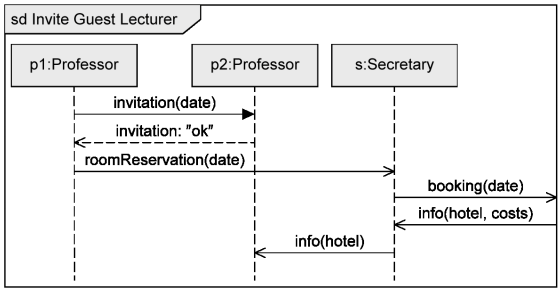


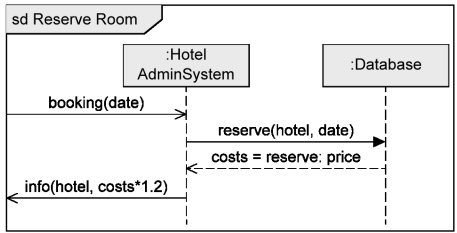


ROAR

Gate

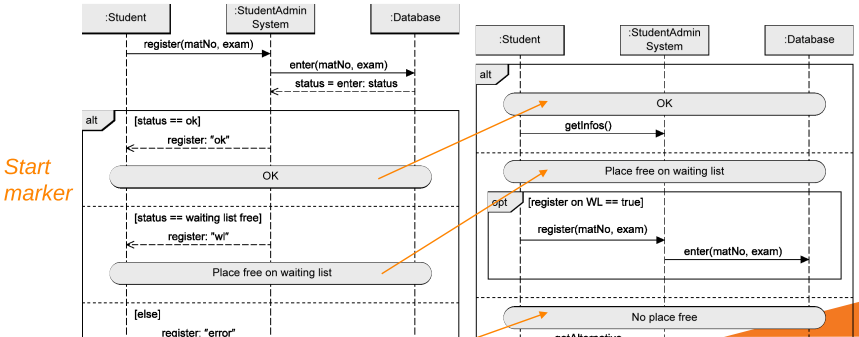
- Allows you to send and receive messages beyond the boundaries of the interaction fragment





Continuation Marker

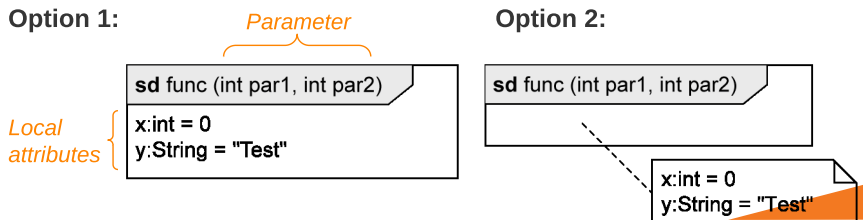
- Modularizes the operands of an `alt` fragment
- Breaks down complex interactions into parts and connect them to one another with markers
- Start marker points to target marker
- No return to the start marker (in contrast to an interaction reference)



Local Attributes and Parameters

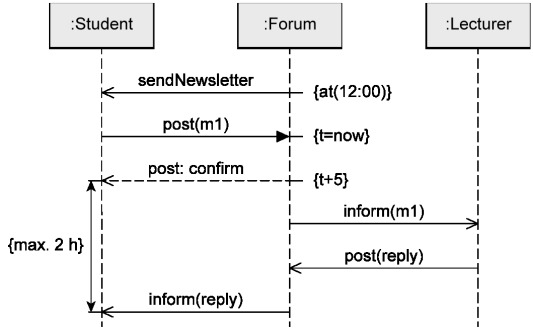
- Every sequence diagram is enclosed by a rectangular frame with a small pentagon in the upper left corner
- Keyword `sd`, name of the sequence diagram, parameters (optional)
- Example:

```
void func (int par1, int par2) {
    int x = 0;
    String y = "Test";
    ...
}
```



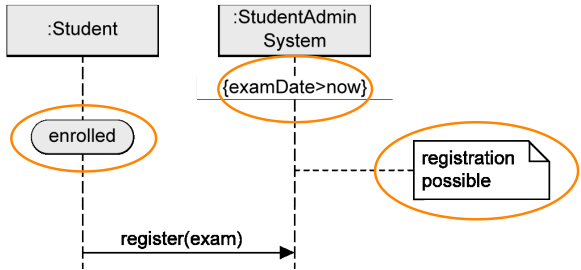
Time Constraints

- Types
  - Point in time for event occurrence
    - Relative: e.g., `after(5sec)`
    - Absolute: e.g., `at(12.00)`
  - Time period between two events
    - `{lower..upper}`
    - E.g., `{12.00..13.00}`
- Predefined actions
  - `now`: current time
    - Can be assigned to an attribute and then used in a time constraint
  - Duration: calculation of the duration of a message



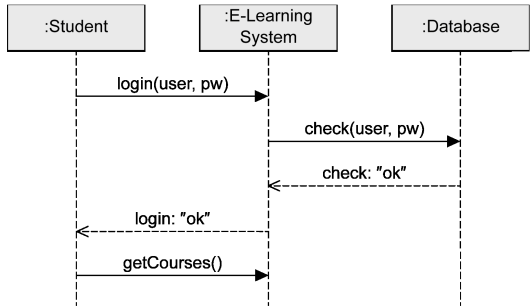
State Invariant

- Asserts that a certain condition must be fulfilled at a certain time
- Always assigned to a specific lifeline
- Evaluation before the subsequent event occurs
- If the state invariant is not true, either the model or the implementation is incorrect
- Three alternative notations:



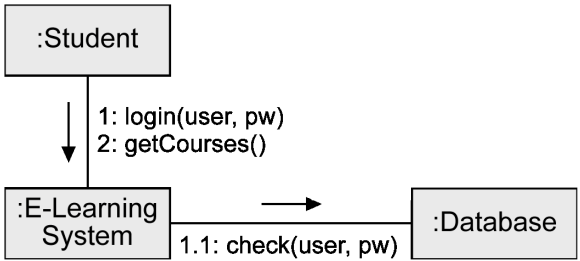
Four Types of Interaction Diagrams

- Based on the same concepts
- Generally equivalent for simple interactions, but different focus
- Sequence Diagram
  - Vertical axis: chronological order
  - Horizontal axis: interaction partners



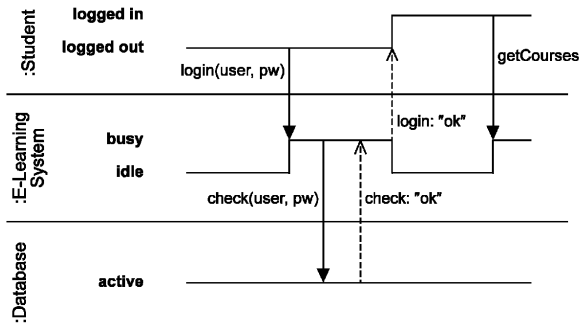
Four Types of Interaction Diagrams

- Communication Diagram
  - Models the relationships between communication partners
  - Focus: Who communicates with whom
  - Time is not a separate dimension
  - Message order via decimal classification



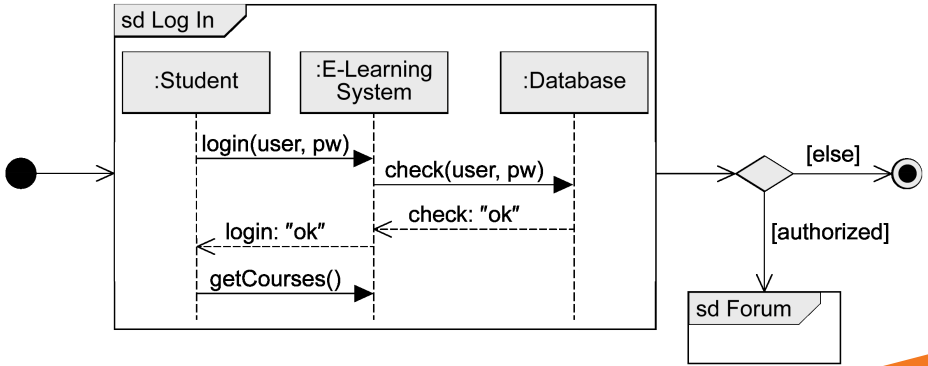
Four Types of Interaction Diagrams

- Timing Diagram
  - Shows state changes of the interaction partners that result from the occurrence of events
  - Vertical axis: interaction partners
  - Horizontal axis: chronological order



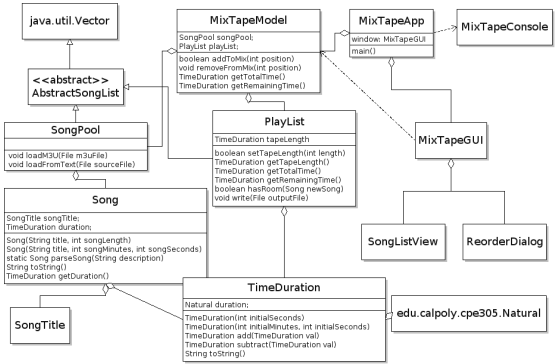
Four Types of Interaction Diagrams

- Interaction Overview Diagram
  - Visualizes order of different interactions
  - Allows to place various interaction diagram sin a logical order
  - Basic notation concepts of activity diagram



Sequence Diagram Exercise

Use the following class diagram and method definitions to create the sequence diagram for moving a song from the song pool to the play list.



In the MixTapeModel class:

```
/** Move a song from the pool to the playlist */
public boolean addToMix(int position)
```

In the PlayList class:

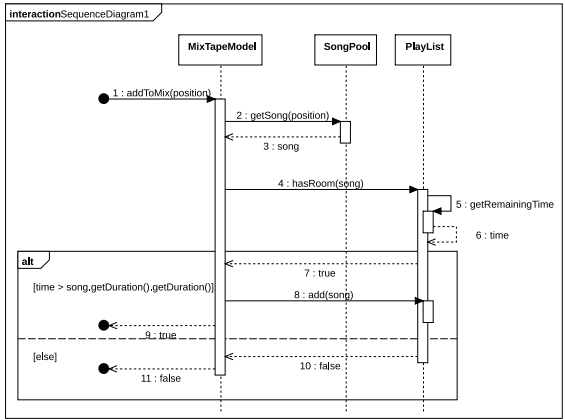
```
/** Determine if the tape has room for a specific song. */
public boolean hasRoom(Song newSong)

/** Accessor to the available time remaining on the tape,
that is, unused space. */
public TimeDuration getRemainingTime()

/** Adds up the total time taken by the current songs in
this play list. */
public TimeDuration getTotalTime()
```

Idaho State University
Computer Science

Sequence Diagram Exercise



ROAR

Are there any questions?