UML Sequence Diagrams

Software Design (40007) – 2023/2024

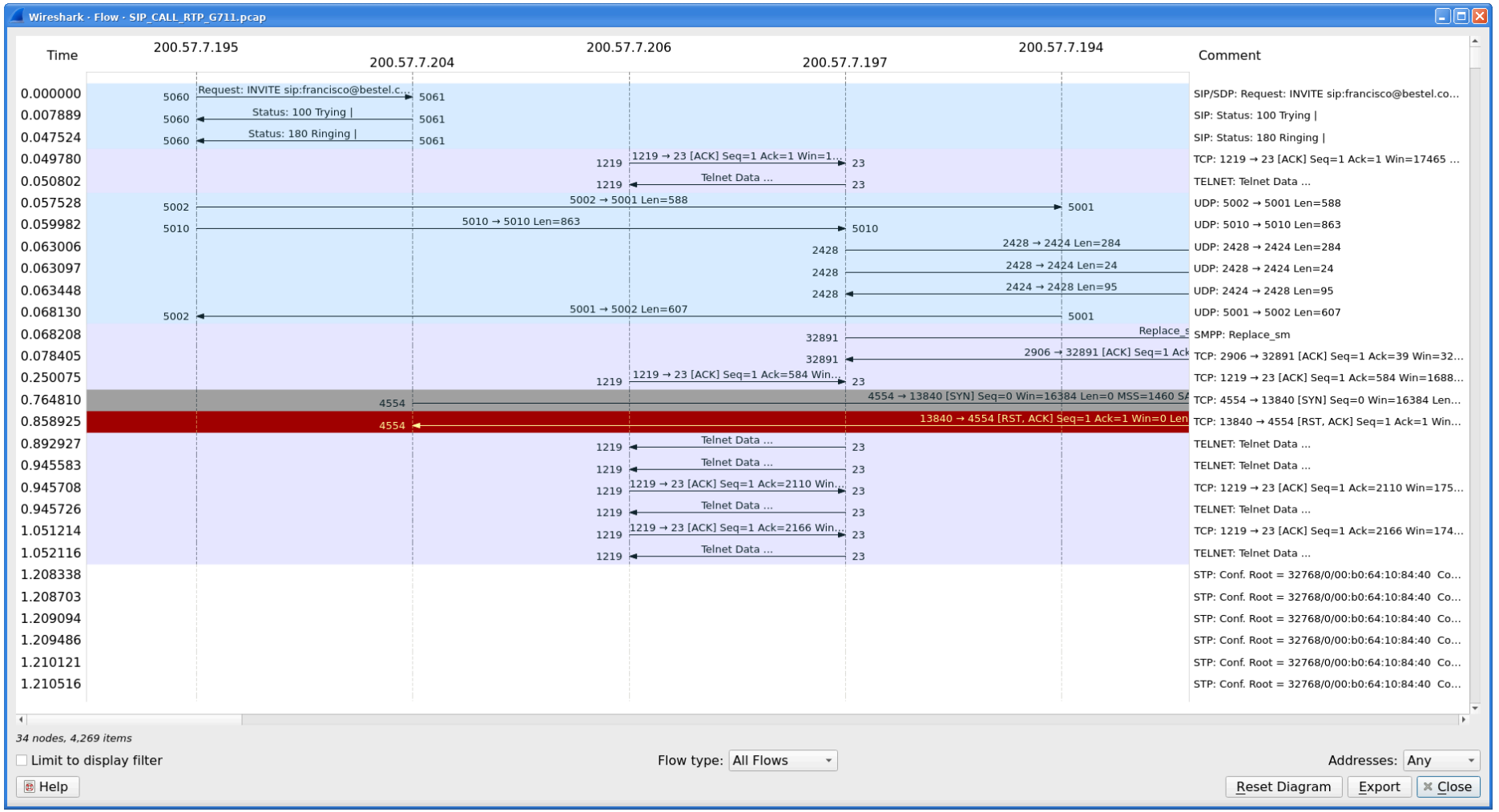
Justus Bogner
Ivano Malavolta

Software and Sustainability research group (S2)
Department of Computer Science, Faculty of Sciences

Roadmap

- Basics
  - Interactions and interaction partners
  - Messages
- Combined fragments
  - Branches and loops
  - Concurrency and order
  - Filters and assertions
- Further modelling elements

Example of interactions: Wireshark



https://www.wireshark.org/docs/wsug_html/#ChStatFlowGraph

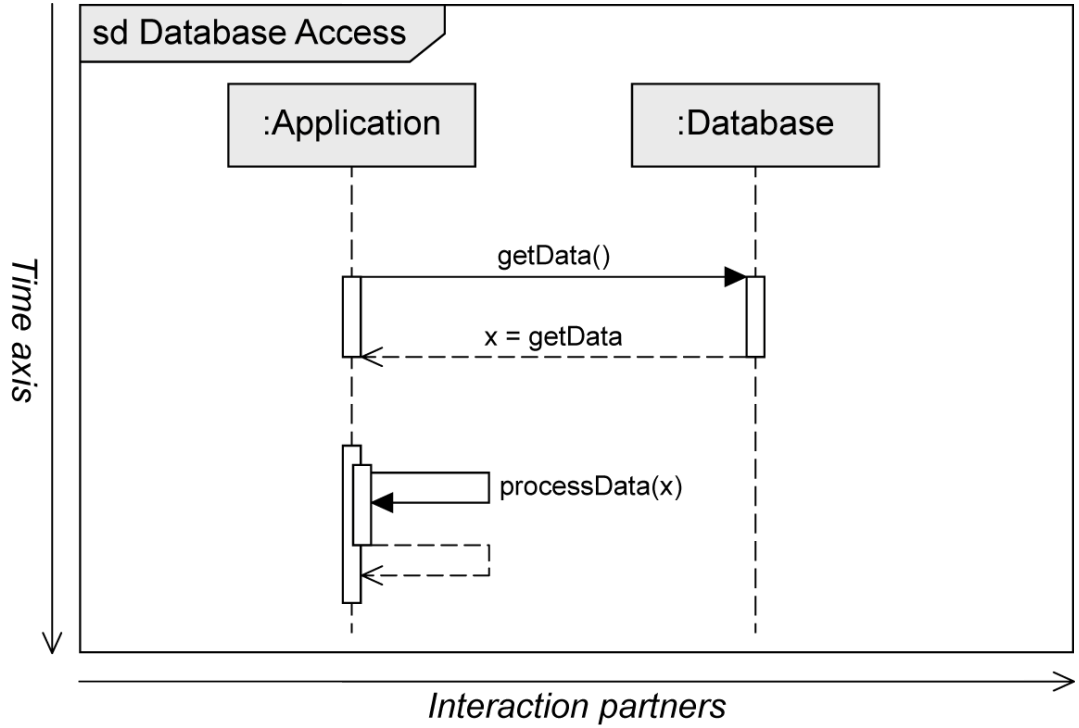
Why UML sequence diagrams?

Main Goal:
To model interactions between partners

- Interaction
  - Specifies how messages and data are exchanged among partners
- Interaction partners
  - Human (lecturers, videogame players, users, administrators, …)
  - Non-human (servers, printers, executable software, **objects**, …)
- Examples of interactions
  - Conversation between persons
  - Message exchange between humans and a software system
  - Communication protocols
  - **Sequence of method calls in a program**
  - …
- State machine diagrams: internal behavior of a single object
- Sequence diagrams: joint behavior of several objects

Sequence diagram

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

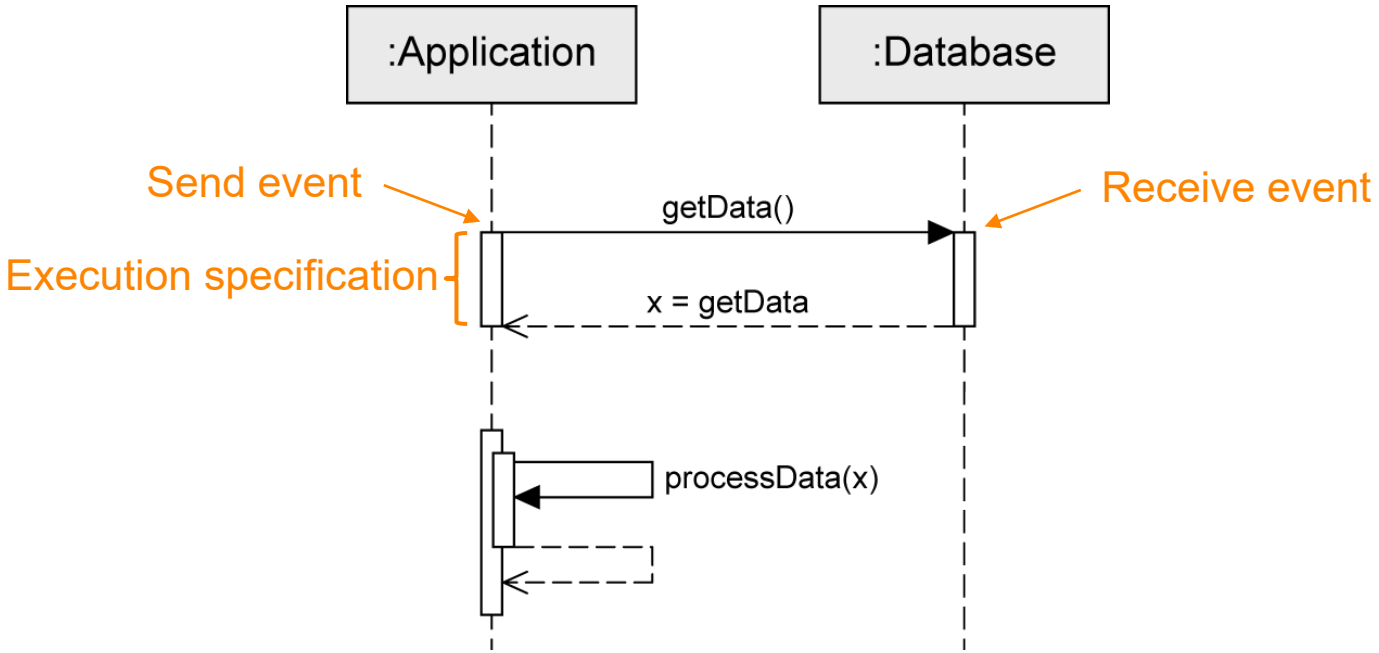


Interaction partners

- Interaction partners are depicted as **lifelines**
- Head of the lifeline
  - Rectangle that contains the expression **object:Class**
- Body of the lifeline
  - Vertical dashed line
  - Represents the lifetime of the object associated with it

Exchanging Messages (1/2)

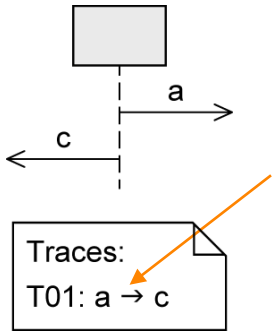
- A message is defined via a **send event** and a **receive event**
- Events are optionally linked via an **execution specification**
  - Visualized as a continuous bar
  - Indicates when the receiving partner executes some behavior



Exchanging Messages (2/2)

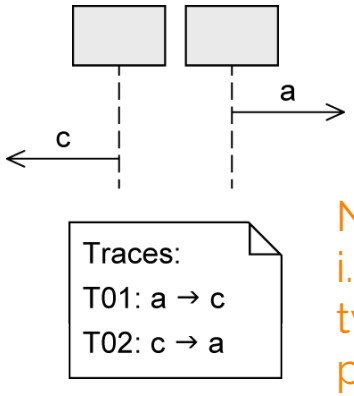
3 rules about the order of messages:

1. on one lifeline



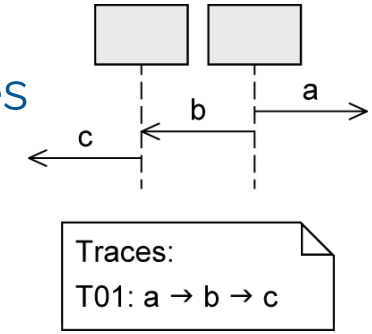
= "happens before"

2. on different lifelines without message exchanges



Not specified, i.e., one of two traces is possible

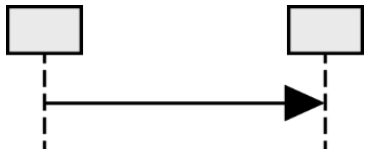
3. on different lifelines that exchange messages



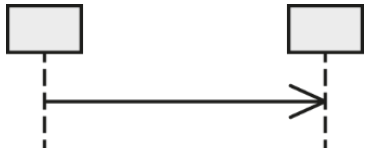
Careful: non-deterministic behavior!

Types of messages (1/3)

- Synchronous message
  - Sender waits until it has received a response message before continuing
  - Syntax of message name: `msg(par1,...)`
    - `msg`: name of the message
    - $\texttt{par}_i$: parameters separated by commas



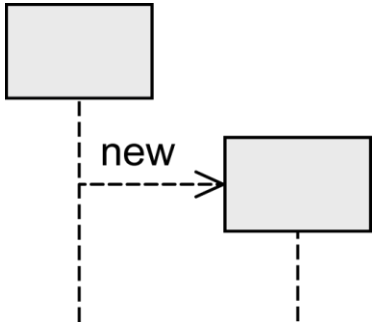
- Asynchronous message
  - Sender continues without waiting for a response message
  - Syntax of message name: `msg(par1,...)`



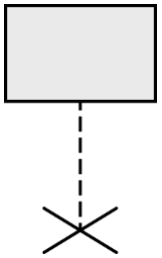
- Response message
  - May be omitted if content and location are obvious
  - Syntax: `res = msg(par1,...):val`
    - `res`: response can optionally be assigned to a variable
    - `msg`: name of the message
    - $\texttt{par}_i$: parameters separated by commas
    - `val`: return value

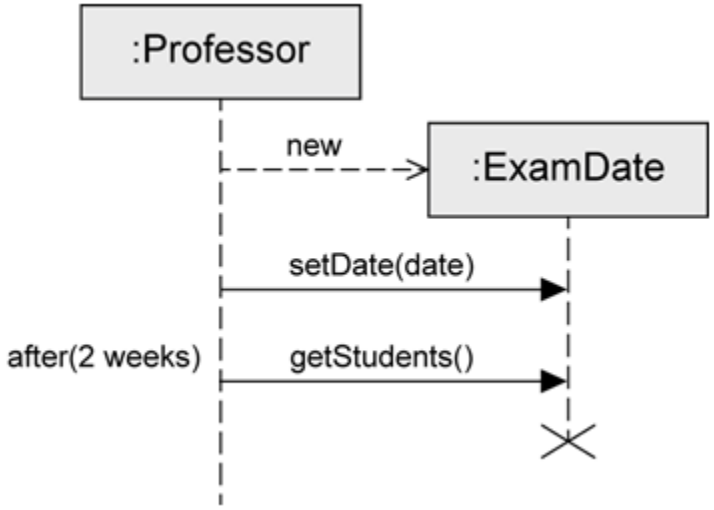
Types of messages (2/3)

- Object creation
  - Dashed arrow with keyword `new`
  - Arrowhead points to the head of the lifeline of the object to be created



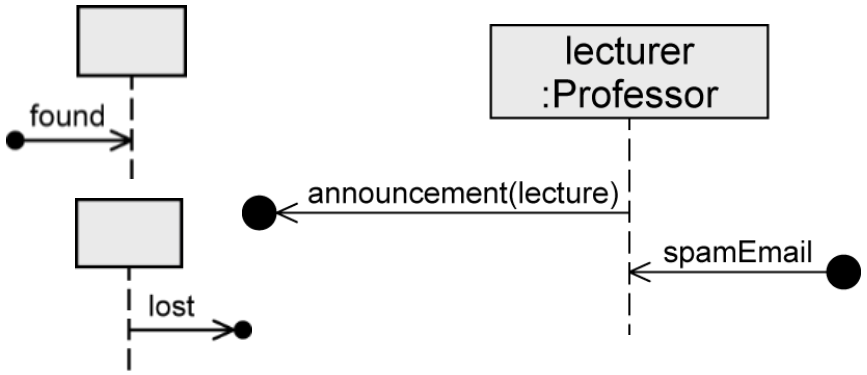
- Object destruction
  - Object is deleted
  - Large cross (×) at the end of the lifeline

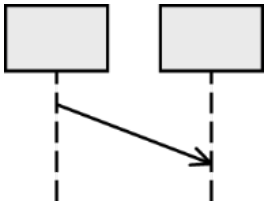


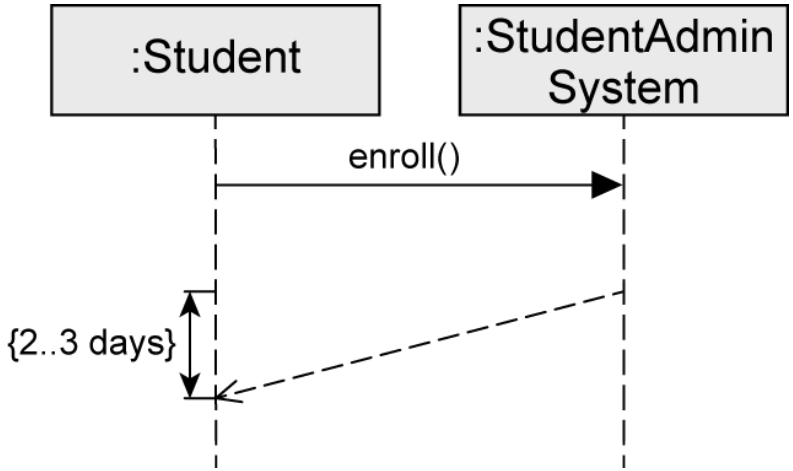


Types of messages (3/3)

- Found message
  - Sender unknown or not relevant
- Lost message
  - Receiver unknown or not relevant
- Time-consuming message
  - "Message with duration"
  - Standard messages are transmitted instantly
  - Here: time elapses between sending and receiving

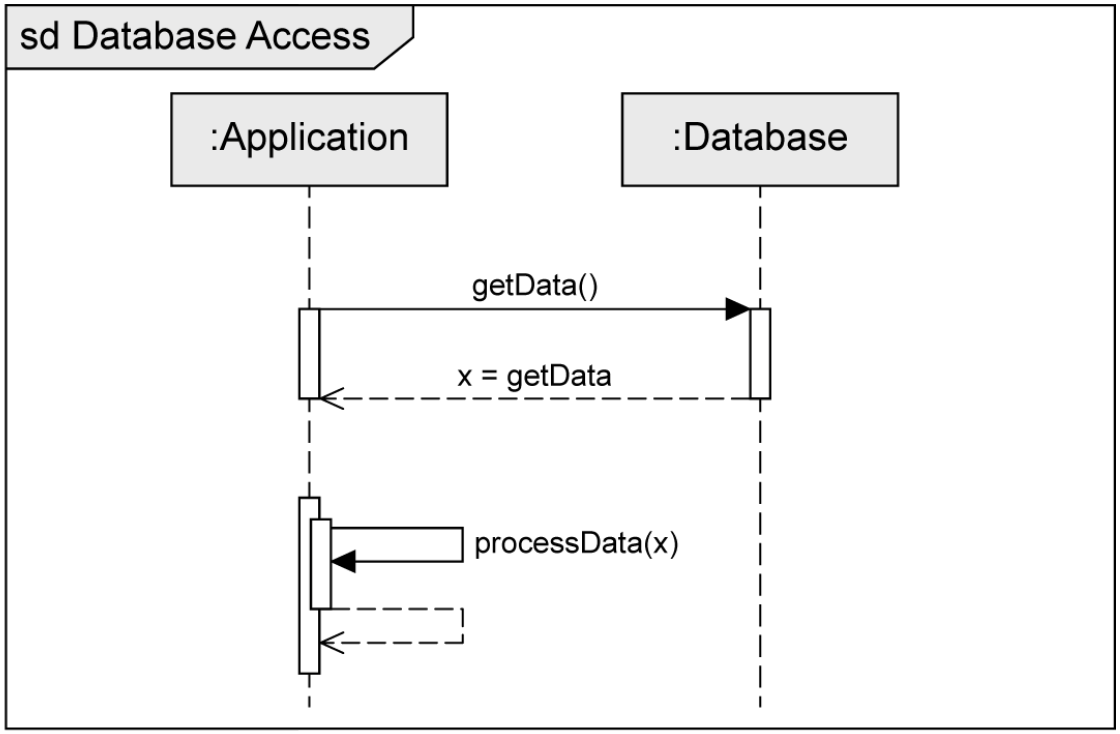






Combined Fragments

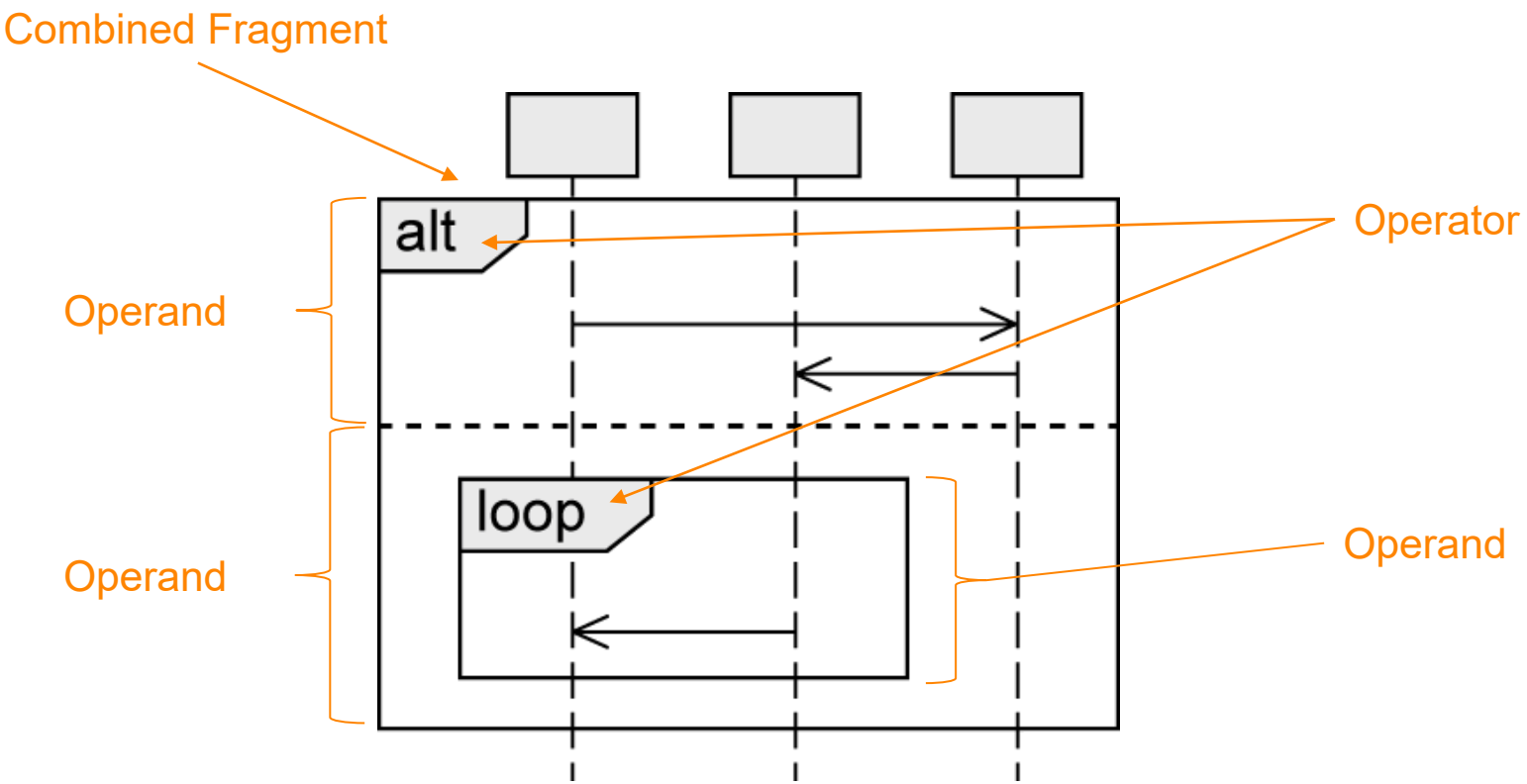
Why combined fragments?



These modelling constructs are suitable for simple, linear interactions but what about more complex interaction flows?

Combined fragments

- Allow modelling various control structures

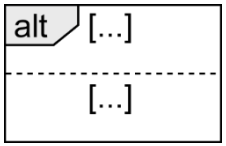


Types of combined fragments

	Operator	Purpose
Branches and loops	`alt`	Alternative interaction
	`opt`	Optional interaction
	`loop`	Repeated interaction
	`break`	Exception interaction
Concurrency and order	`seq`	Weak order
	`strict`	Strict order
	`par`	Concurrent interaction
	`critical`	Atomic interaction
Filters and assertions	`ignore`	Irrelevant interaction
	`consider`	Relevant interaction
	`assert`	Asserted interaction
	`neg`	Invalid interaction

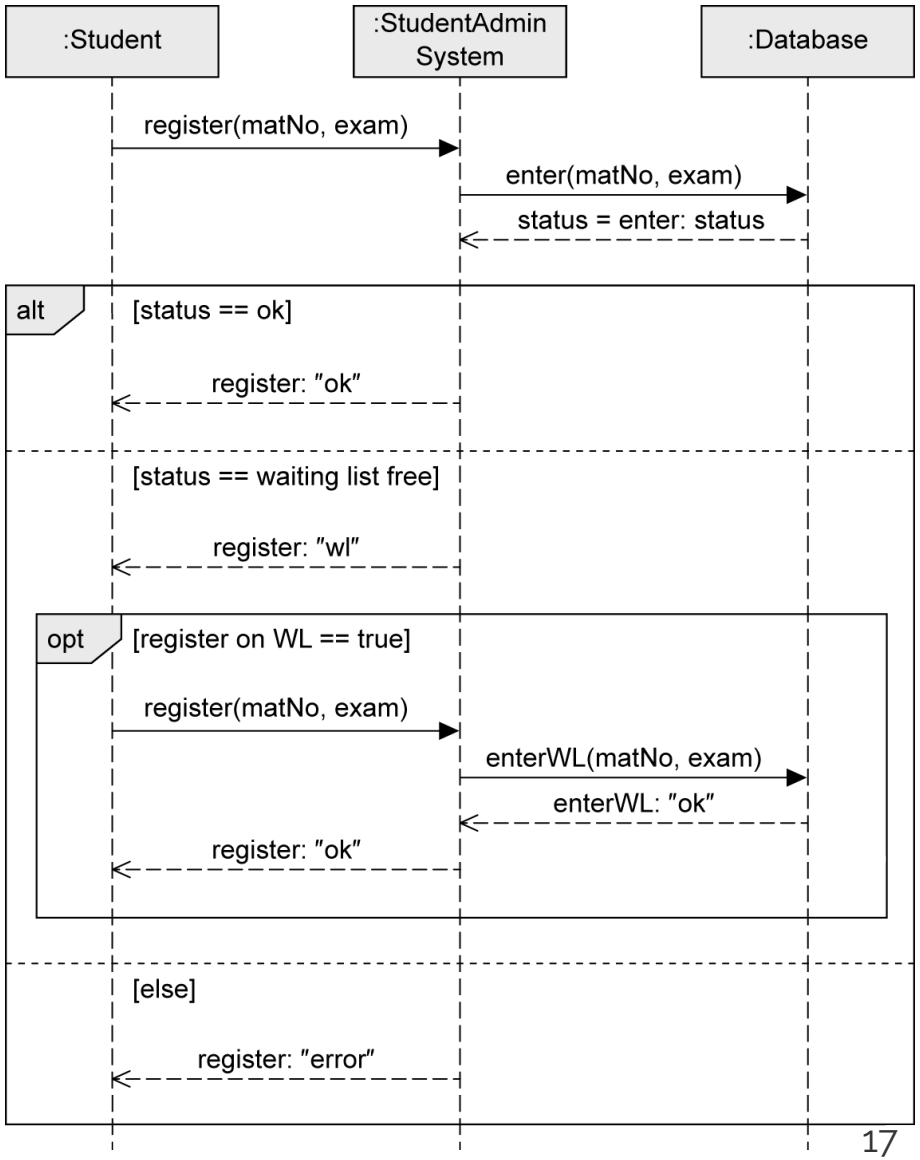
Branches and Loops

`alt` fragment



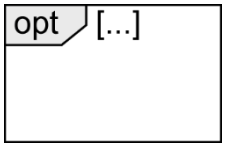
Model alternative sequences

- Similar to **`switch`** or **`if`** statement in Java
- At least 2 operands
- Guards are used to select the path to be executed
  - Modeled in square brackets
  - default: `true`
  - Special guard: [else]
  - Must be mutually exclusive to avoid indeterministic behavior



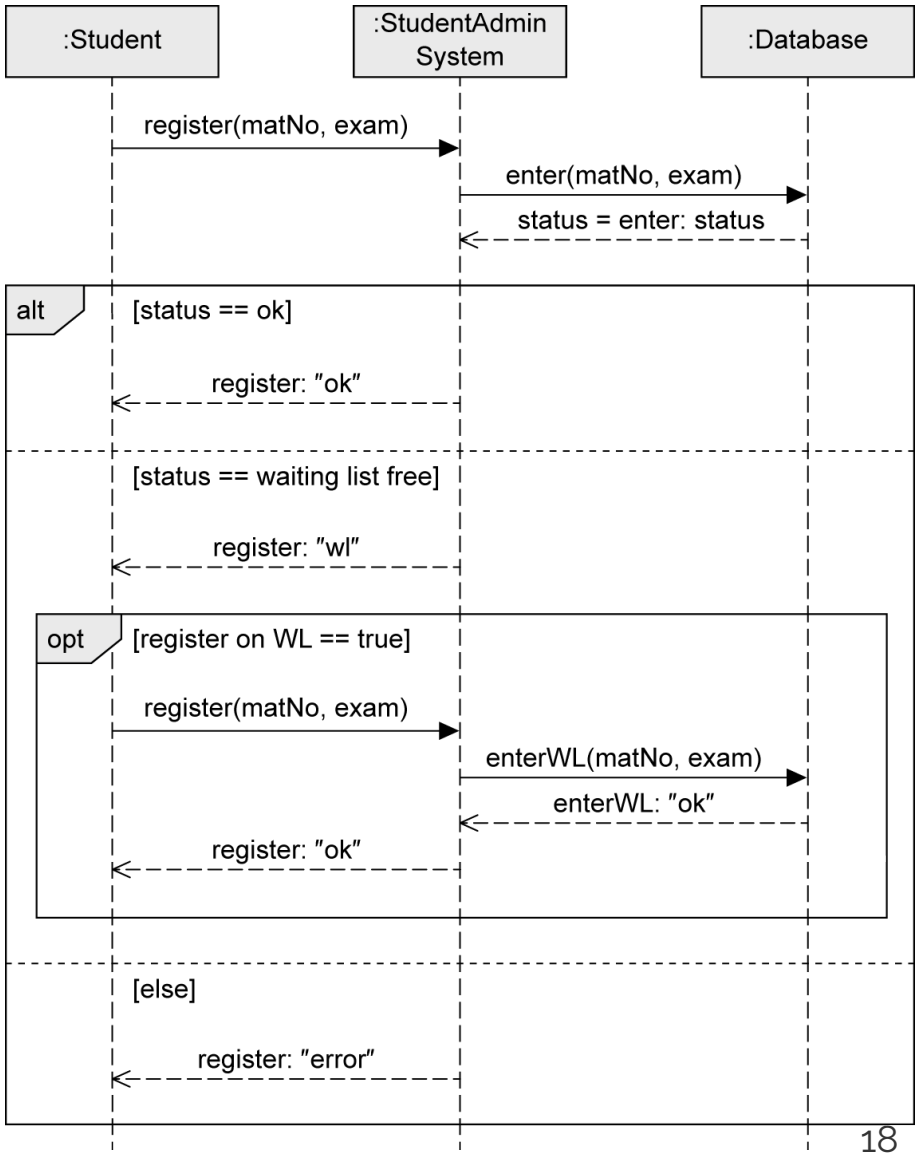
VU VRIJE UNIVERSITEIT AMSTERDAM

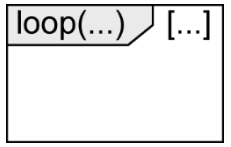
`opt` fragment



Model an optional sequence

- Actual execution at runtime is dependent on a (required) guard
- Similar to **`if`** statement without **`else`** branch(es)
- Exactly one operand



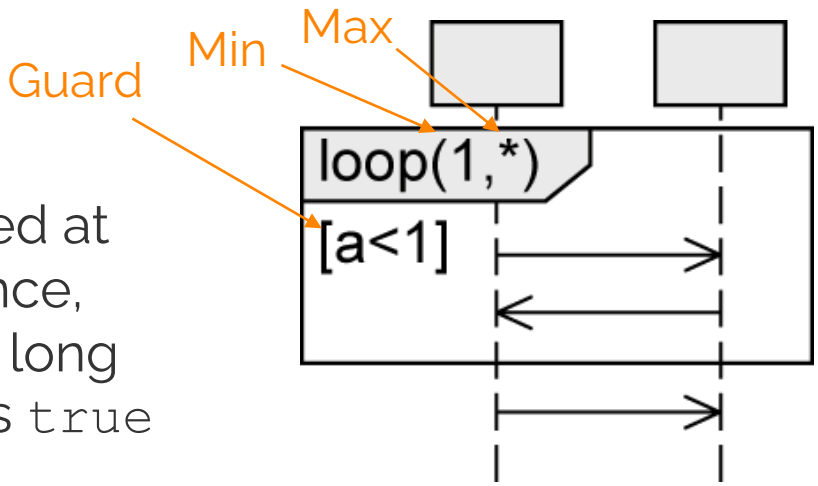


`loop` fragment

- Exactly one operand
- Minimum / maximum # of iterations
  - (**`min..max`**) or (**`min,max`**)
  - default: `(*)` →no min and max, similar to `while(true)`
- Guard
  - Evaluated right after the minimum number of iterations
  - Checked for each subsequent iteration within the **`max`** limit
  - If the guard evaluates to `false`, the loop is terminated

Model repeated sequences

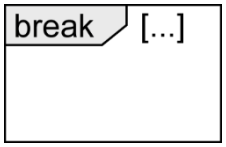
loop is executed at least once, then as long as a<1 is `true`



Notation alternatives:

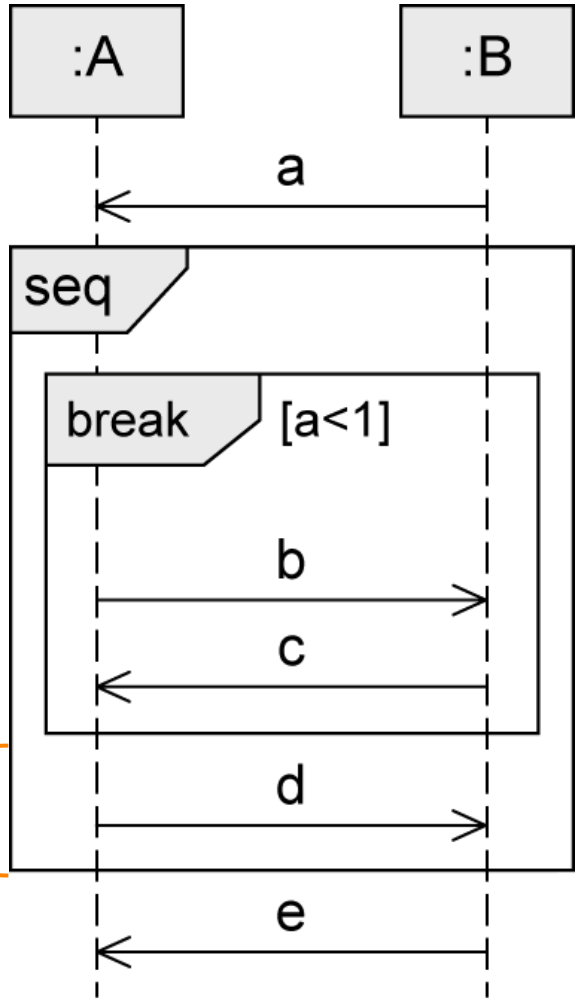
loop(3,8) = loop(3..8)
loop(8,8) = loop (8)
loop = loop (*) = loop(0,*)

`break` fragment



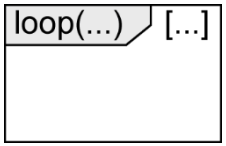
Model exception handling

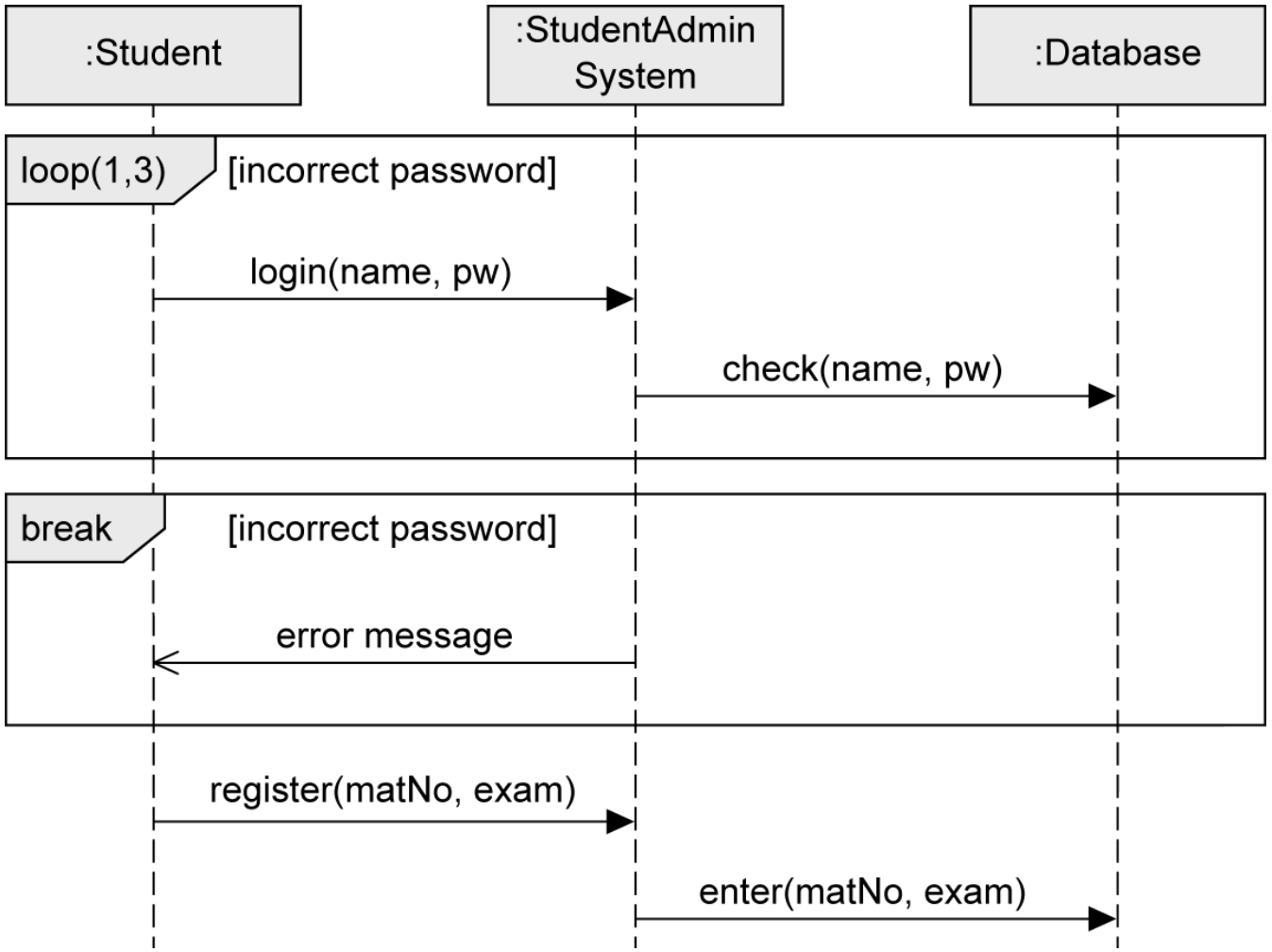
- Exactly one operand with a guard
- If the guard is `true`:
  - Interactions within this operand are executed (b → c)
  - Remaining operations of the surrounding fragment are omitted (d)
  - Interaction continues in the next higher level fragment (e)



Not executed if break is executed

Example: `loop` and `break` fragments





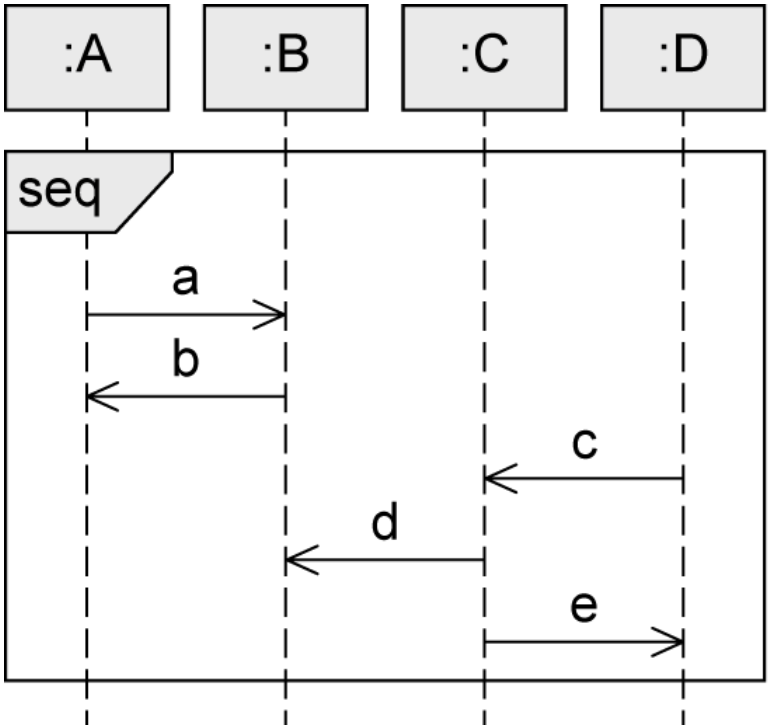
Concurrency and Order

Types of combined fragments

	Operator	Purpose
Branches and loops	`alt`	Alternative interaction
	`opt`	Optional interaction
	`loop`	Repeated interaction
	`break`	Exception interaction
Concurrency and order	`seq`	Weak order
	`strict`	Strict order
	`par`	Concurrent interaction
	`critical`	Atomic interaction
Filters and assertions	`ignore`	Irrelevant interaction
	`consider`	Relevant interaction
	`assert`	Asserted interaction
	`neg`	Invalid interaction

seq fragment

- Weak sequencing: the default order of events



What are the 3 possible traces of messages?

Traces:

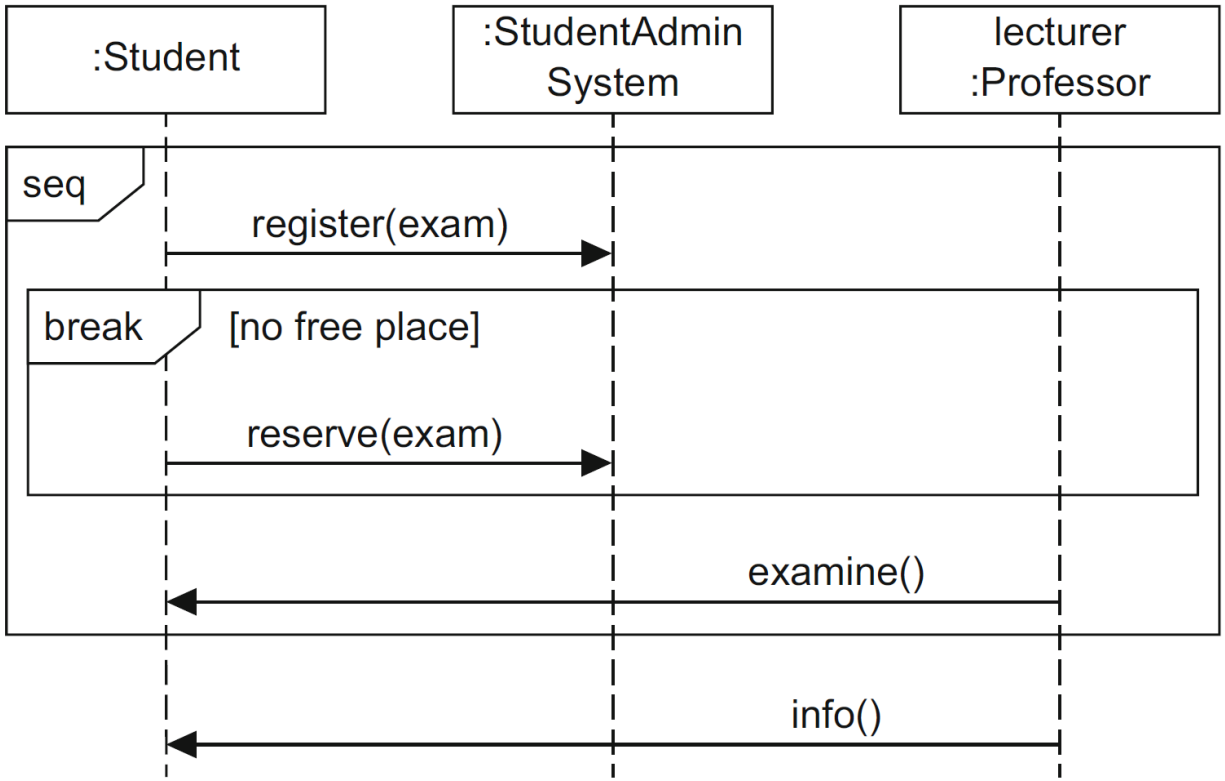
T01: a → b → c → d → e

T02: a → c → b → d → e

T03: c → a → b → d → e

Example: `seq` fragment combined with `break`

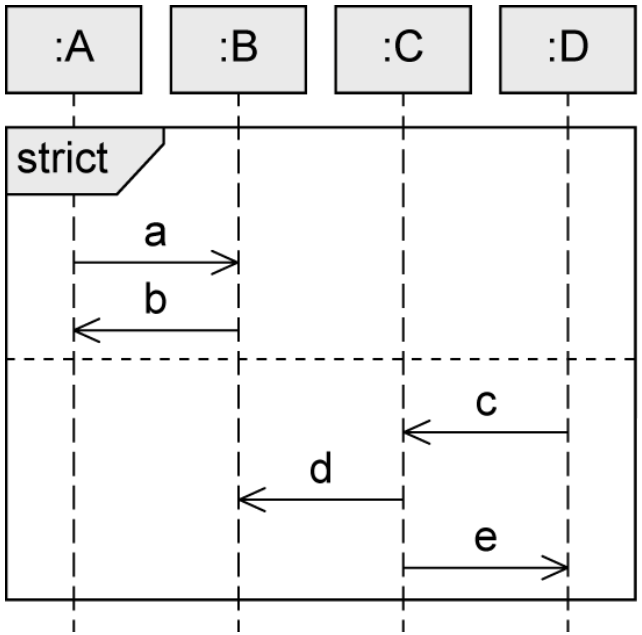
- `seq` is rarely useful on its own (default order), but is often valuable in combination with a `break` fragment



`strict` fragment

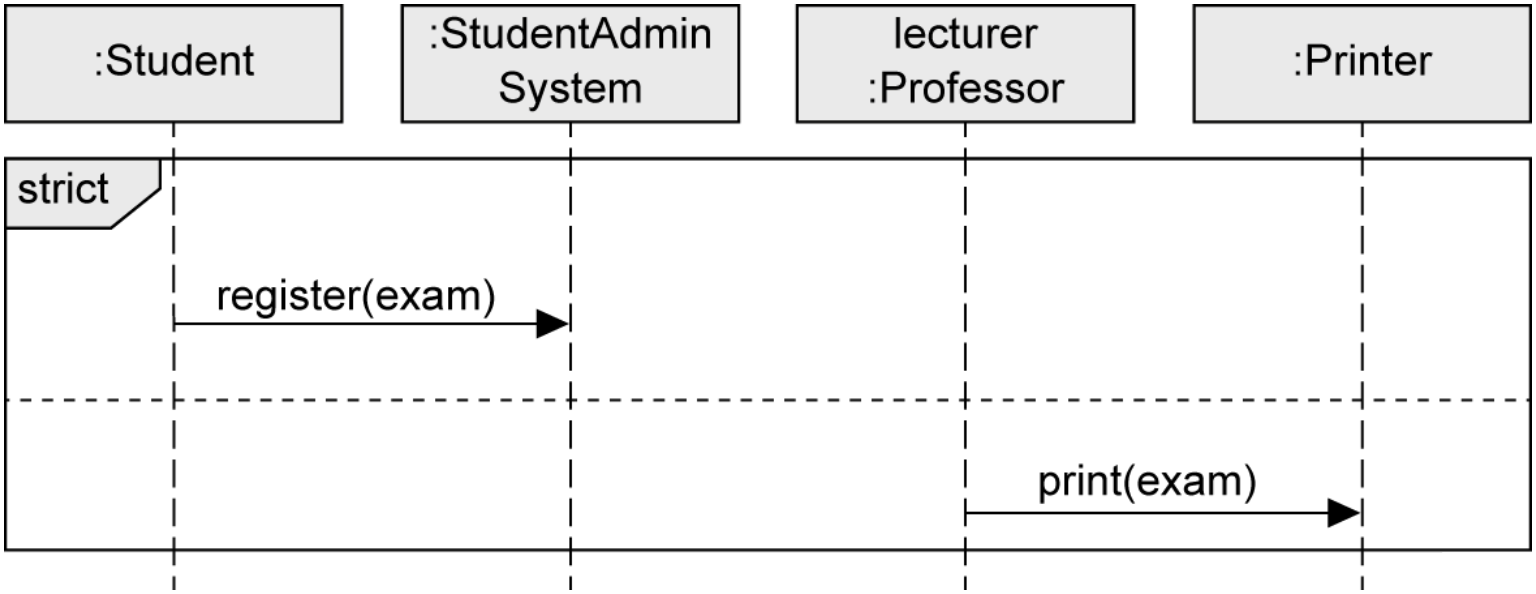
Model a fixed sequence of events across lifelines

- Between different operands, the event order of the vertical axis becomes deterministic **even for unconnected lifelines**
- Messages in a higher-up operand are always exchanged **before** messages in a lower operand

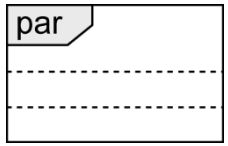


Traces:
T01: a → b → c → d → e

Example: `strict` fragment

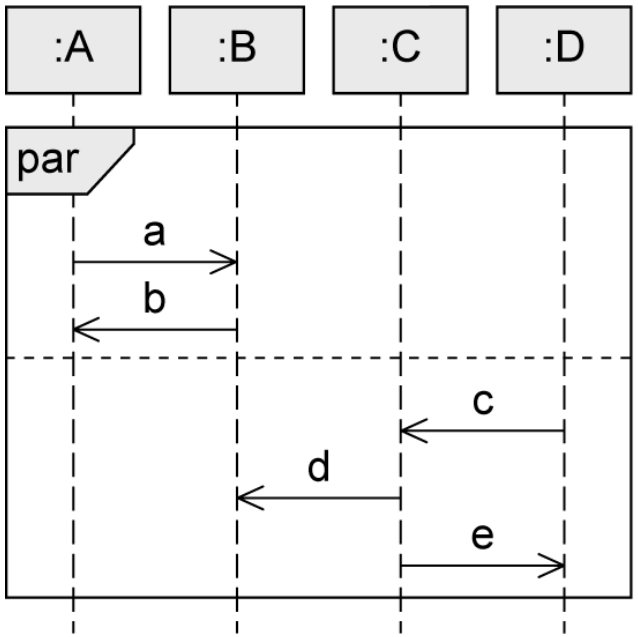


`par` fragment



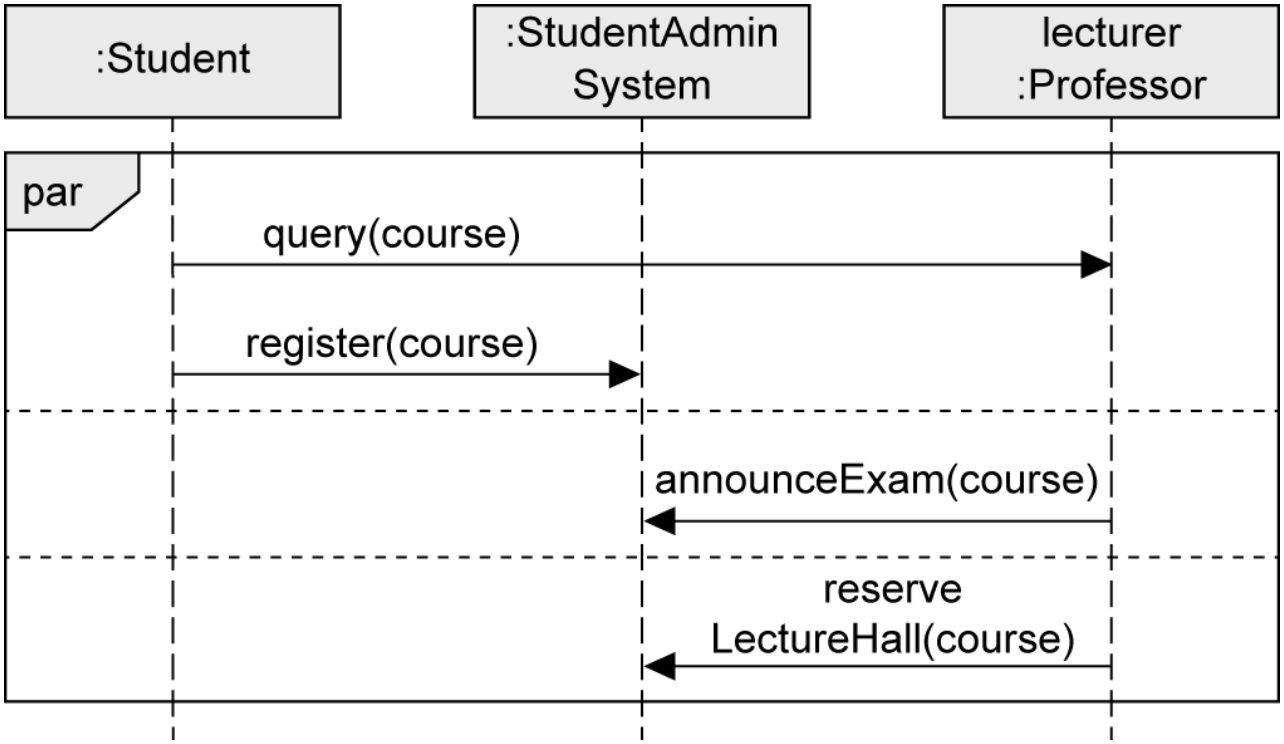
Relax the chronological order between messages in different operands (concurrency)

- Execution paths of different operands can be interleaved
- Restrictions within each operand need to be respected
- Order of the different operands is irrelevant



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

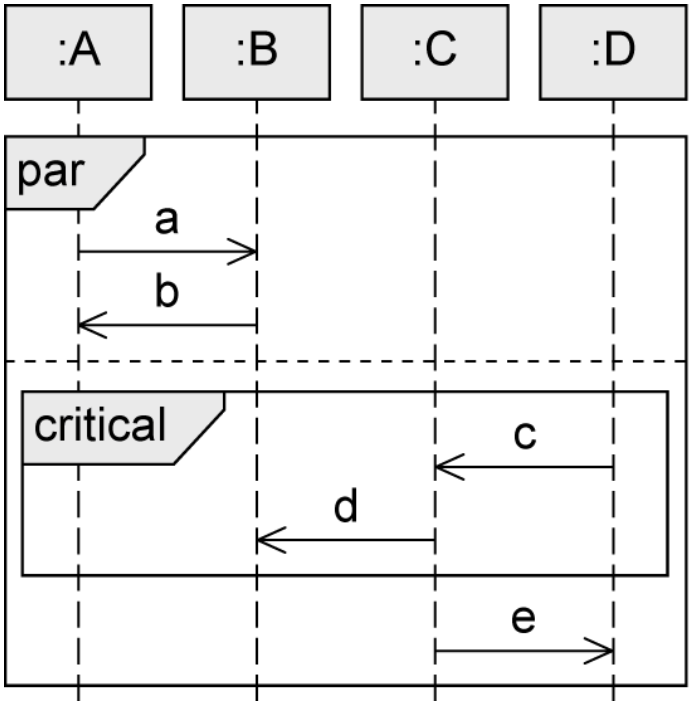
Example: `par` fragment



`critical` fragment

Model atomic areas within indeterministic execution

- To make sure that certain parts of an interaction are not interrupted by unexpected events



Traces:

T01: a → b → c → d → e

T02: a → c → d → b → e

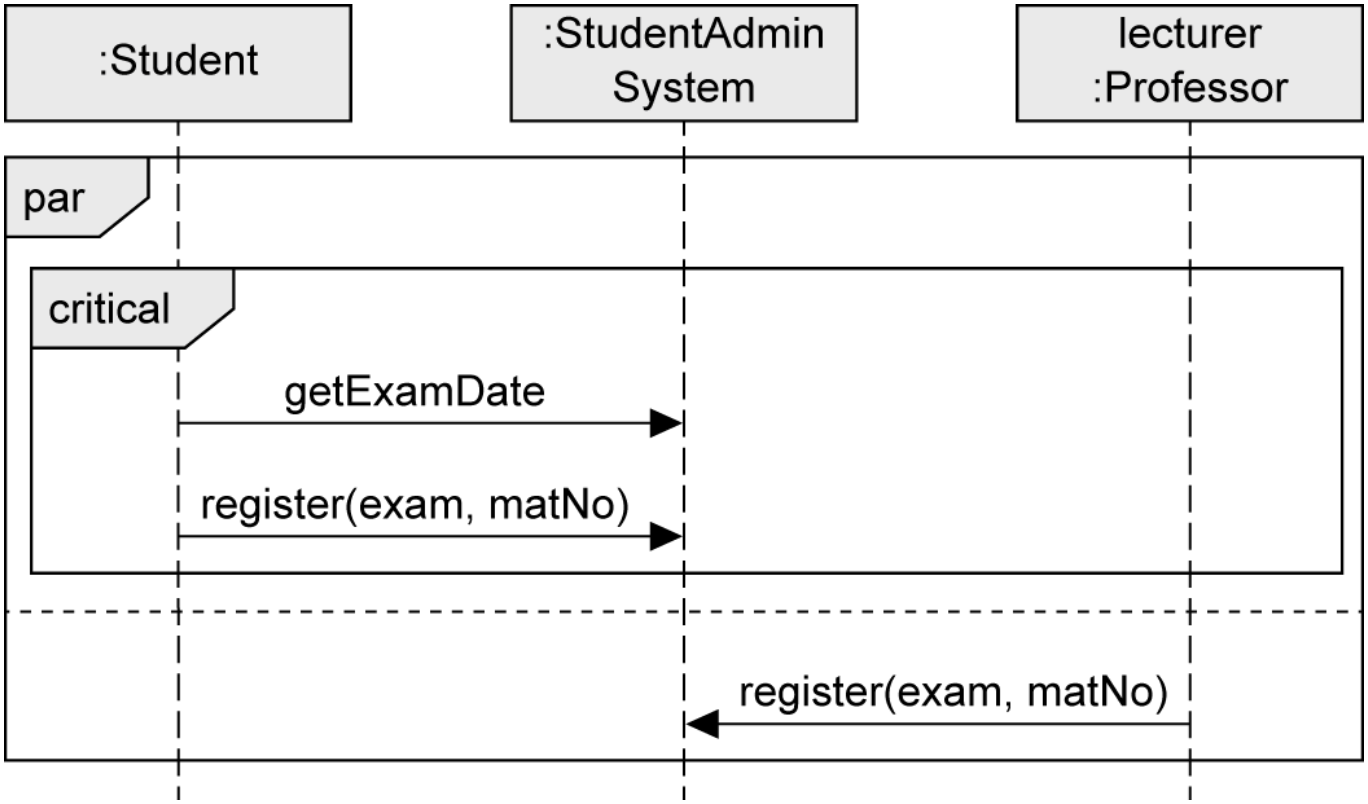
T03: a → c → d → e → b

T04: c → d → a → b → e

T05: c → d → a → e → b

T06: c → d → e → a → b

Example: `critical` fragment

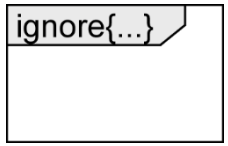


Filters and Assertions

Types of combined fragments

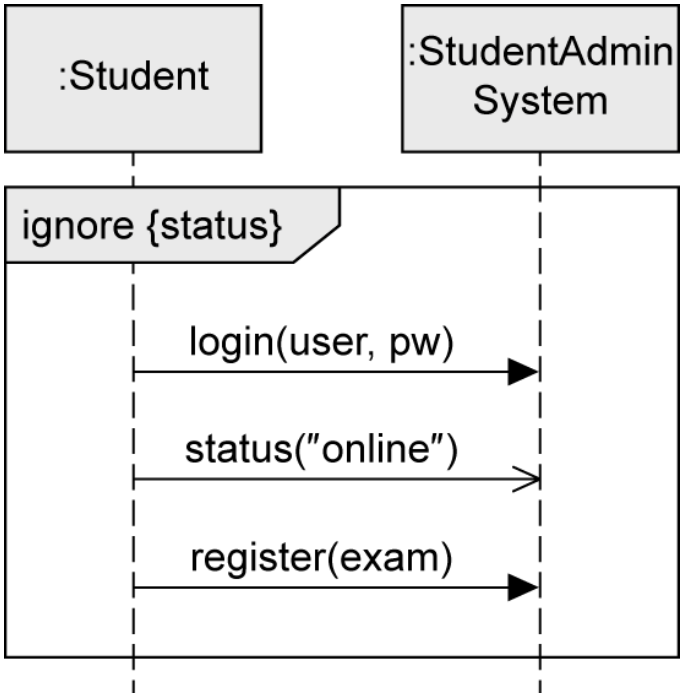
	Operator	Purpose
Branches and loops	`alt`	Alternative interaction
	`opt`	Optional interaction
	`loop`	Repeated interaction
	`break`	Exception interaction
Concurrency and order	`seq`	Weak order
	`strict`	Strict order
	`par`	Concurrent interaction
	`critical`	Atomic interaction
Filters and assertions	`ignore`	Irrelevant interaction
	`consider`	Relevant interaction
	`assert`	Asserted interaction
	`neg`	Invalid interaction

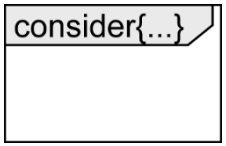
`ignore` fragment



Model irrelevant messages

- Messages that occur at runtime without notable significance
- Exactly one operand
- Irrelevant messages in curly brackets after the keyword

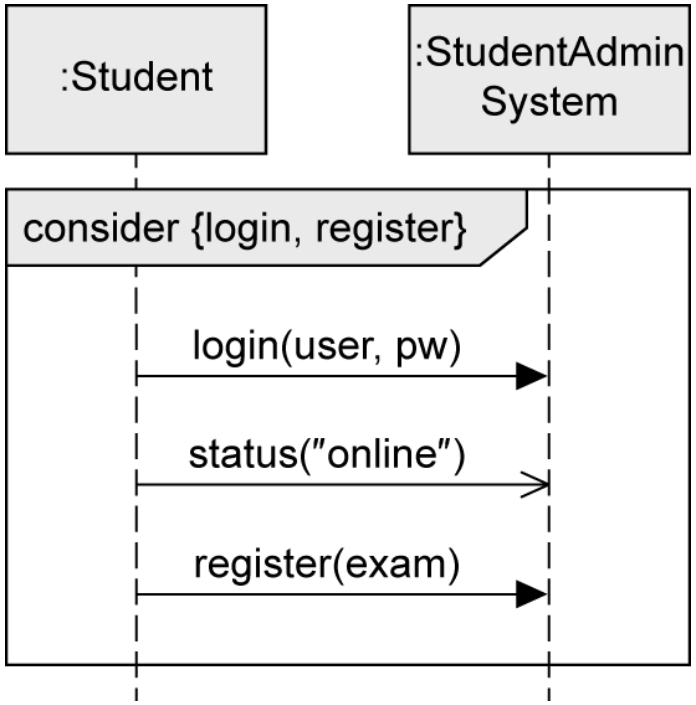




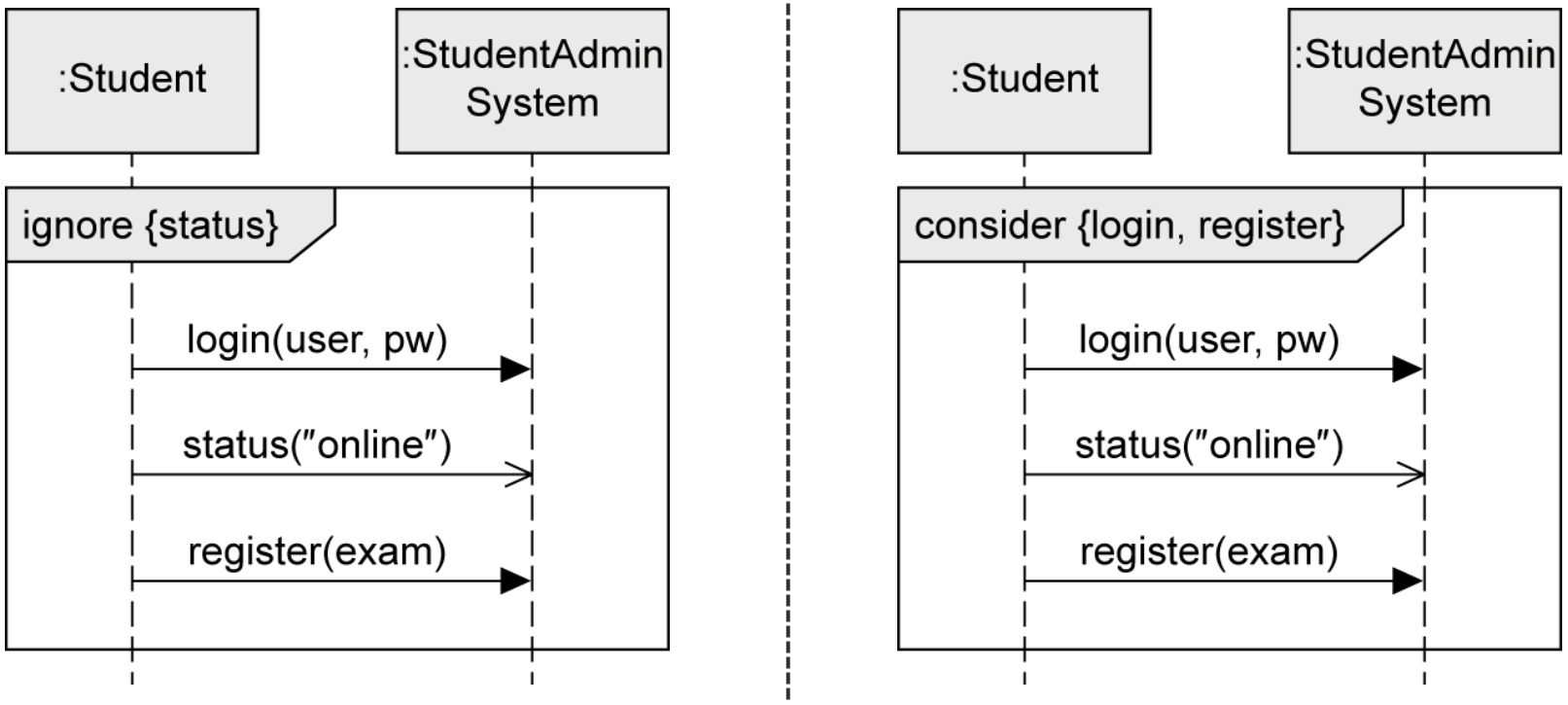
`consider` fragment

Model messages with particular importance

- Exactly one operand
- Complementary to `ignore` fragment
- Considered messages in curly brackets after the keyword



ignore vs. consider

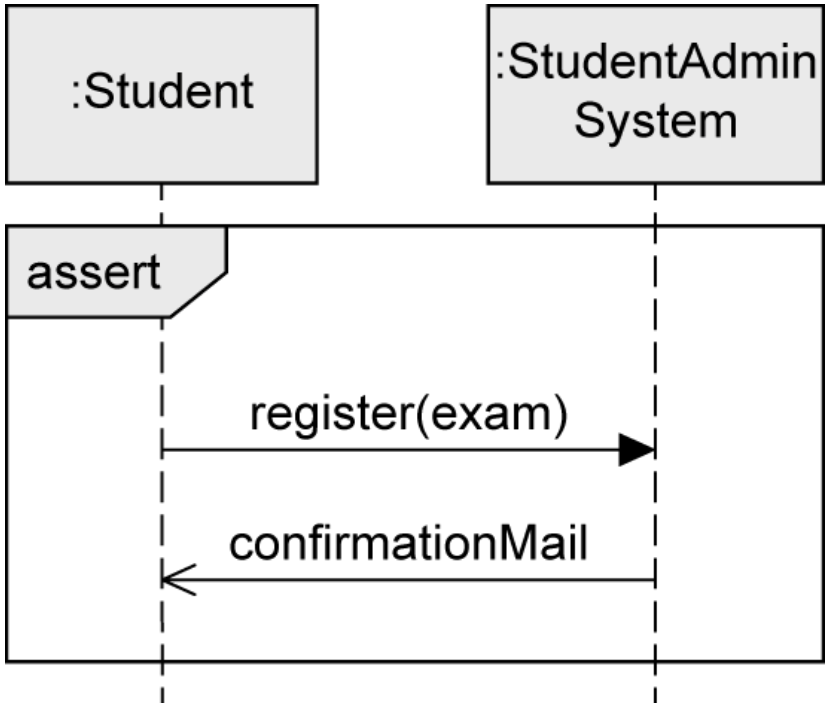


These two ways of modeling the interaction are equivalent.

`assert` fragment

Model an interaction as mandatory

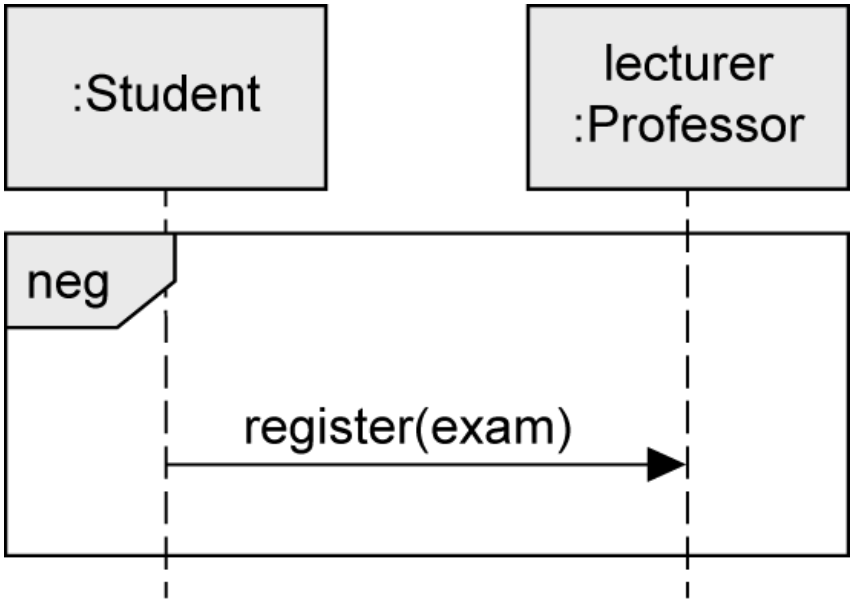
- Deviations that occur in the real world but are not included in the fragment lead to an error (not permitted)
- Reinforces the completeness of the diagram part



neg fragment

Model important invalid interactions

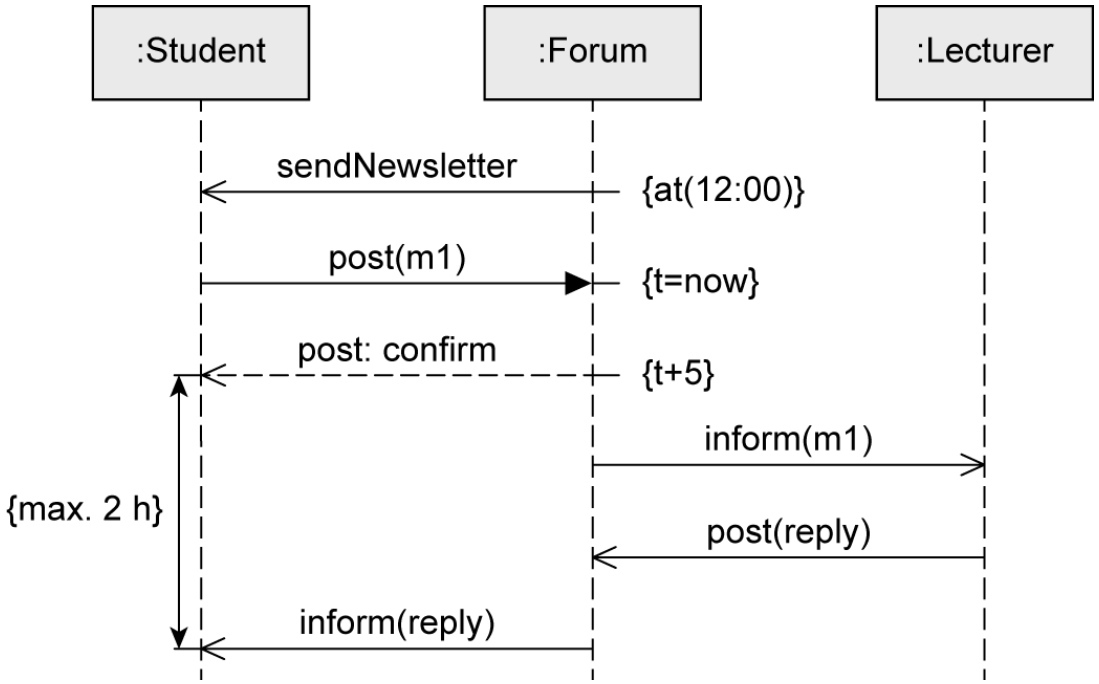
- Describing situations that must not occur
- Purpose:
  - Explicitly highlighting frequently occurring errors
  - Depicting relevant, incorrect sequences



Further Modelling Elements

Time constraints

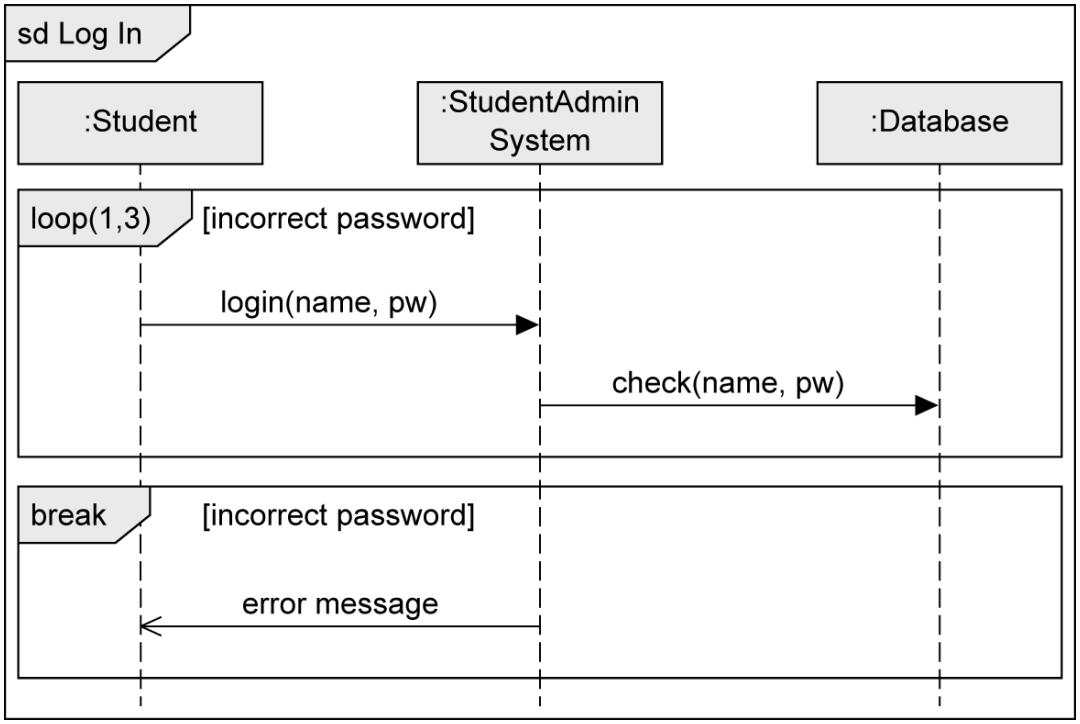
- Time for event occurrence
  - Relative: e.g., after(5sec)
  - Absolute: e.g., at(12:00)
- Time between two events: {lower..upper}, e.g., {12:00..13:00}
- Current time as `now`: assign to attribute and use in a time constraint



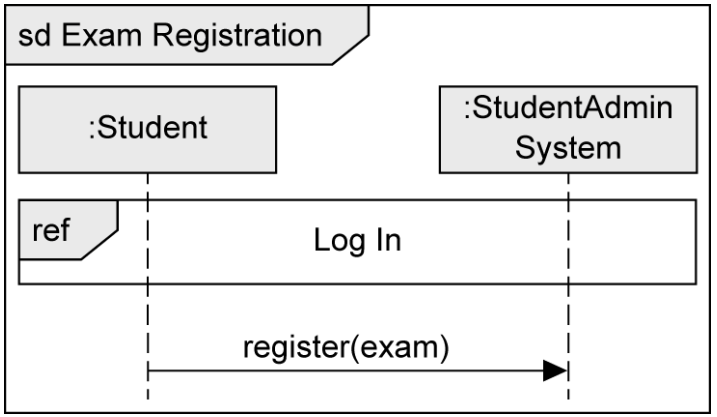
Interaction reference

- Integrates one sequence diagram into another one via `ref`

Definition

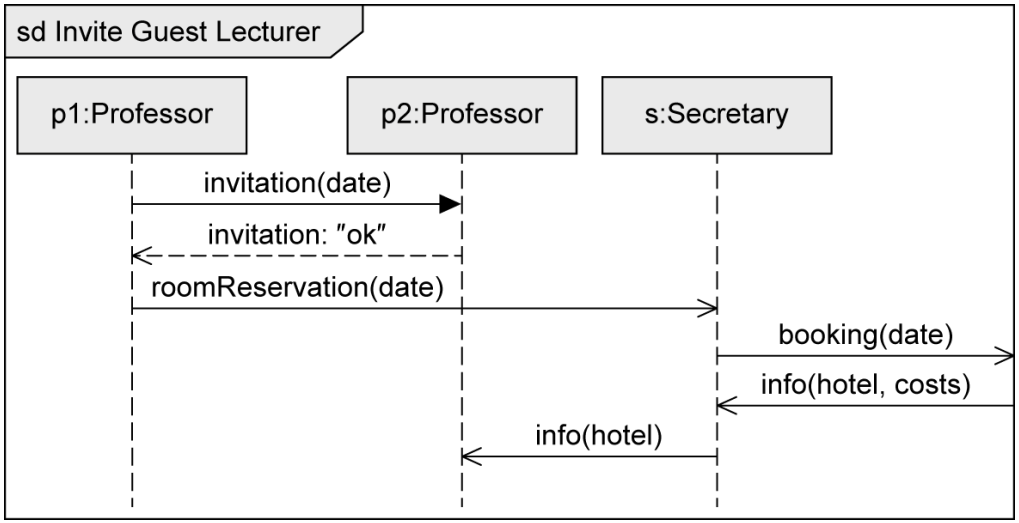


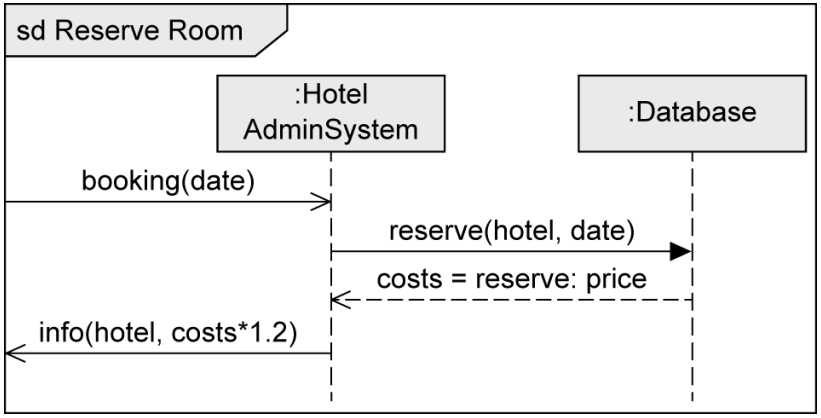
Usage



Gates

- Allow you to send and receive messages beyond the boundaries of interaction fragments or combined fragments
- Visualized by messages touching / originating from boundary



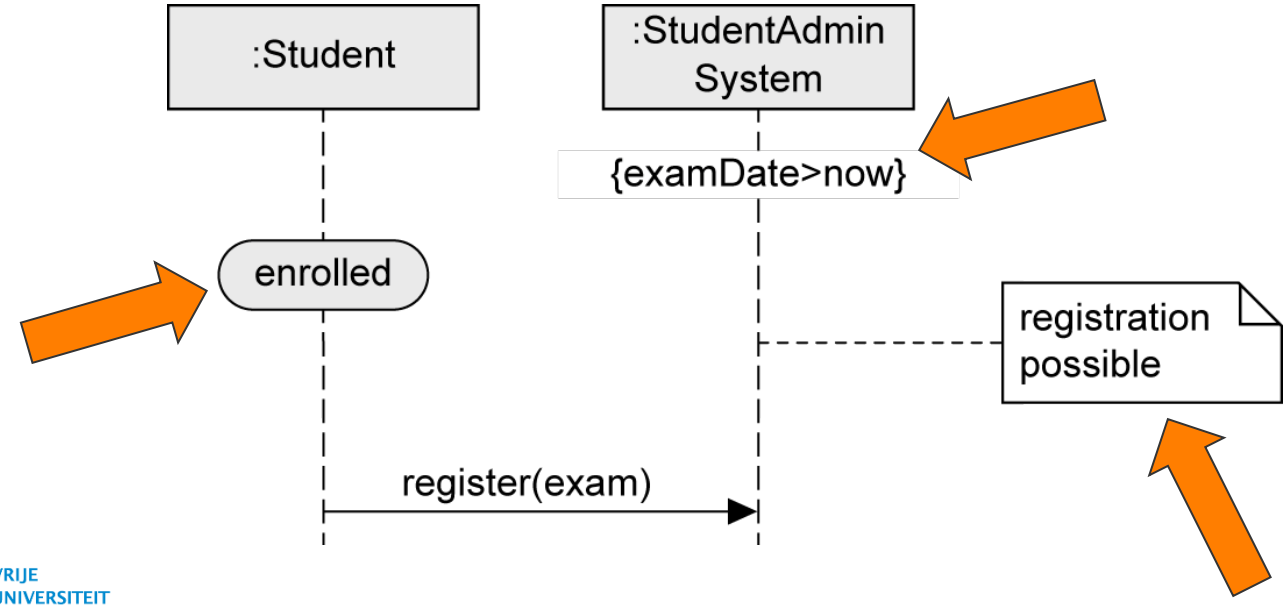


State invariants

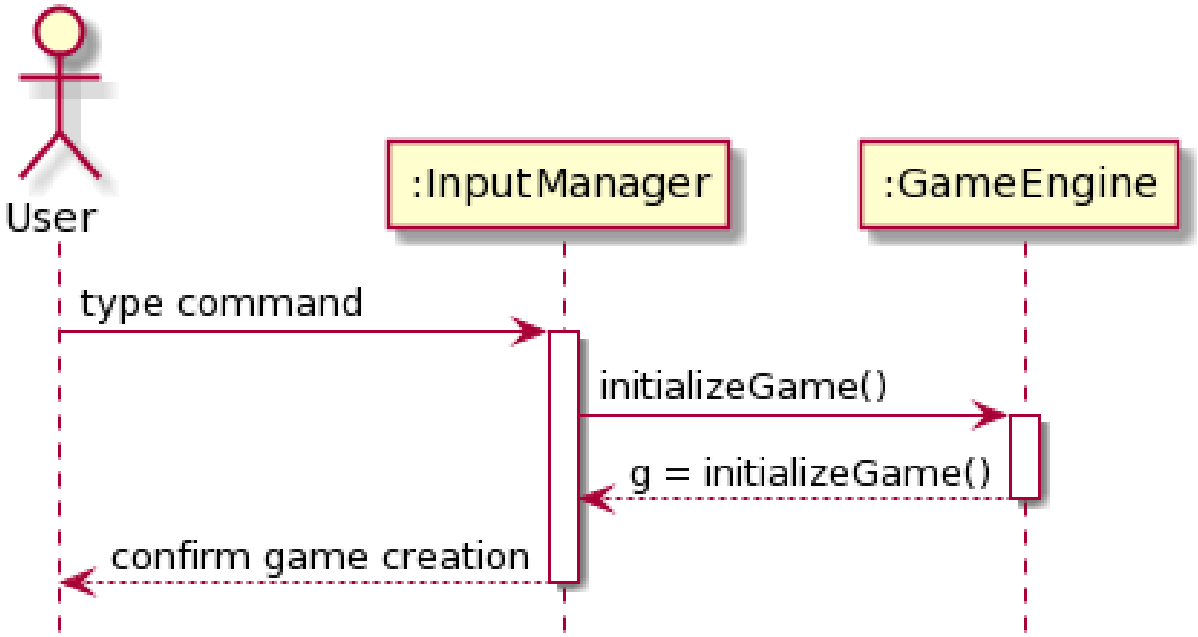
- Assert that a condition must be fulfilled at a certain time
- Evaluation before the subsequent event occurs
- If the state invariant is not `true`, either the model or the implementation is incorrect

Useful for linking your state machines and sequence diagrams

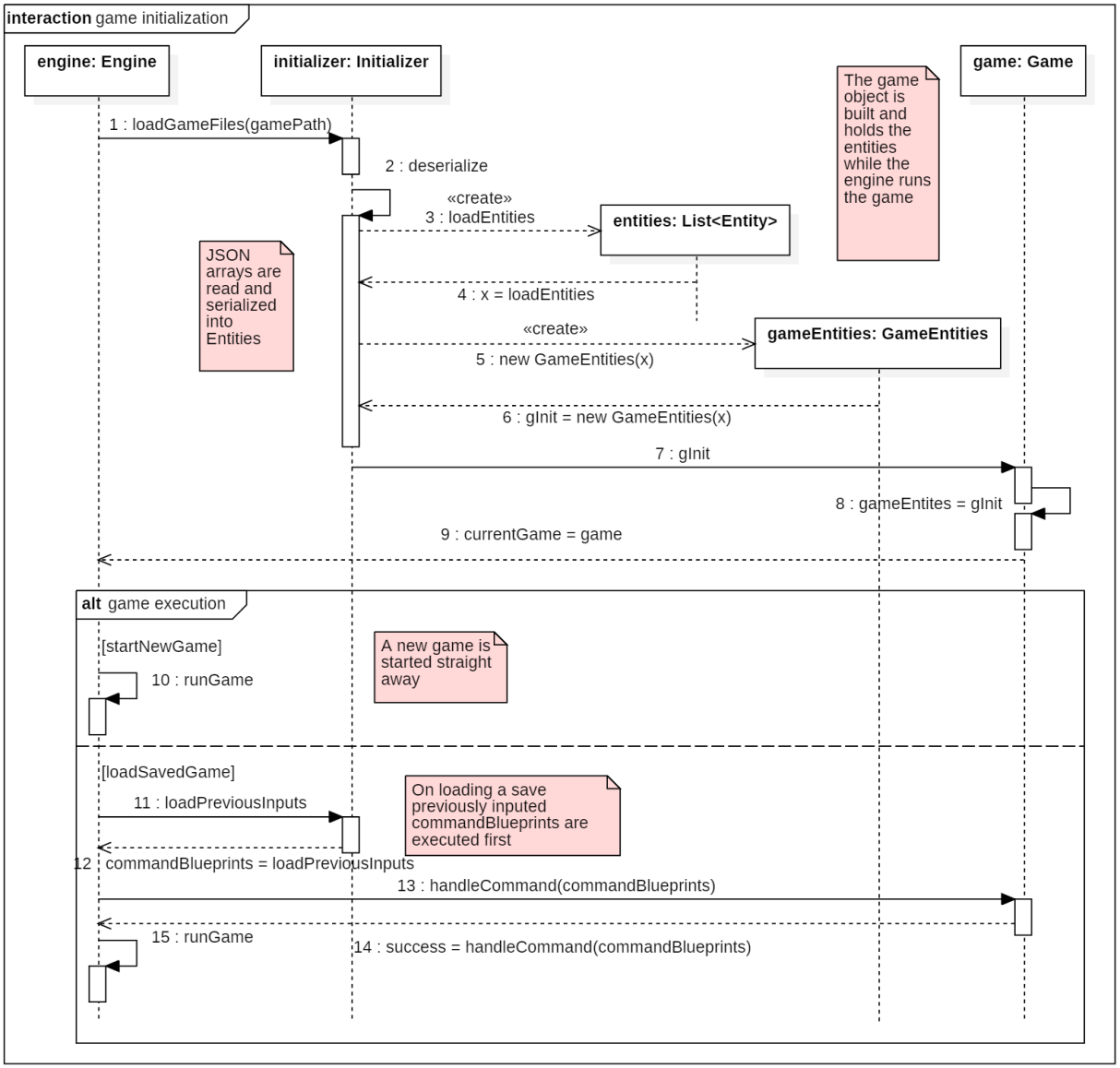
- Three alternative notations:



How to show user interaction



A final example



Key takeaways

- You can now model how objects interact with each other!

- State machine diagram
  - Internal behavior of your objects
- Sequence diagram
  - How objects interact with each other (aka external behavior)
  - Stakeholders of your system can also be involved
  - Many options to model complex control flow (**combined fragments**)
    - Branches and loops
    - Concurrency and order
    - Filters and assertions

Readings

- UML@Classroom: An Introduction to Object-Oriented Modeling – Chapter 6
- A Philosophy of Software Design, Ch. 10, 11, 12