Software Engineering

Lesson 04
UML / Behavioral Diagrams
v1.4

Uwe Gühl

Fall 2007/ 2008

Contents

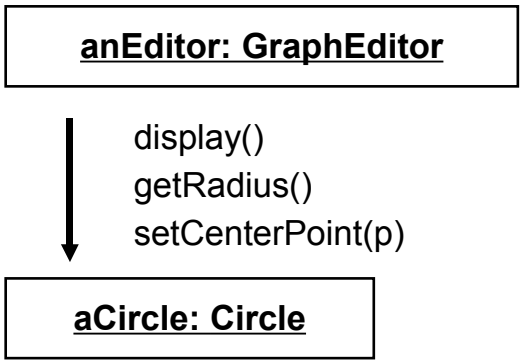
- Messages and Methods
- Behavioral Diagrams
  - Activity Diagrams
  - Status Machine Diagrams
  - Interaction Diagrams
    - Sequence Diagrams
      - Fragments
      - Proceeding
    - Communication Diagrams

Messages and Methods

- Objects communicate with messages
  - A sender Object sends a message to a receiver object.
  - A message has a name, parameters, if needed, and provides a result.
  - The receiver object must be able to understand the message
  - The receiver object determines a confirming method and executes the message
  - An object could send a message to itself
- **Please don't mistake message and method!**

Messages and Methods

- Example



Circle
radius {radius>0} CenterPoint: Point = (0, 0)
display() delete() setCenterPoint(newCenterPoint: Point) setRadius (newRadius) int getRadius()

Receiver

Java notation:

Message

```
aCircle.display();
int i = aCircle.getRadius();
aCircle.setCenterPoint(p);
```

Smalltalk notation:

```
aCircle display.
i := aCircle getRadius.
aCircle setCenterPoint: p.
```

Procedural:

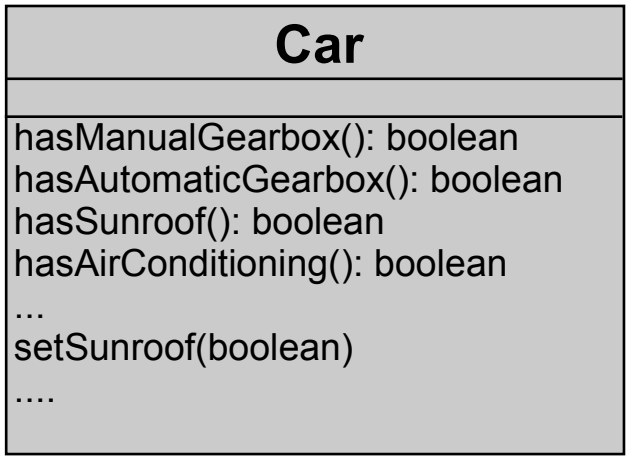
```
display(aCircle);
i = getRadius(aCircle);
setCenterPoint(aCircle, p);
```

Messages and Methods Access

- The public methods of a class (the interface) could hide different implementations
- The internal implementation could change over time (e. g. concerning maintenance, tuning, ...), but the clients of this class don't have to be changed
- Encapsulation principle: The object offers with its methods a service – how this service is implemented is not important for the user

Messages and Methods Access

- Example of a class Car



- How could the access to this class be implemented?

Messages and Methods Access

- Alternative 1: Variables

```
public class Car {
  public Engine engine = new Engine("V6");
  public boolean hasManualGearbox = false;
  public boolean hasAutomaticGearbox = false;
  public boolean hasSunroof = false;
  public boolean hasAirConditioning = false;
  public Integer performanceHP;
  public Integer performanceKW;
  ...
}
```

+ Fewest code
– Implementation not encapsulated.
  If the internal representation changes, all requests have to change as well

Example

```
myCar = new Car();
myCar.hasManualGearbox = true;
myCar.engine = new Engine("V12");
```

Messages and Methods Access

- Alternative 2: Methods

```
public class Car {
  private boolean hasManualGearbox = false;
  private boolean hasAutomaticGearbox = false;
  ...

  public boolean getAutomaticGearbox() {
    return hasAutomaticGearbox;
  }

  public void setAutomaticGearbox(boolean automaticGearbox) {
    hasAutomaticGearbox = automaticGearbox;
  }
  ...
}
```

Messages and Methods Access

- Alternative 2: Methods
  - \+ Access only with get and set methods. If the implementation is going to be changed, the client could still access like before
  - – More code (but typically support by IDE with automated code generation)

Example

```
myCar = new Car();
myCar.setManualGearbox(true);
if (myCar.getManualGearbox()) { ... }
```

Messages and Methods Access

- More alternatives
  - lazy initialization
  - calculating of results
  - “tricky storage”

```
public class Fahrzeug {

  static final int SUNROOF = 1;
  static final int AIRCONDITIONING = 2;
  static final int ESP = 4;
  static final int NAVIGATIONSYSTEM = 8;

  public boolean hasAutomaticGearbox;
  int optionalEquipment = 0;

  public Engine getEngine() { // lazy initialization
    if (engine == null) {
      engine = new Engine();
    }
    return engine;
  }

  public Integer getPerformanceHP() { // calculated result
    return getPerformanceKW() / 0.745699871;
  }

  // complete different internal storage, e. g. as bit vector
  public boolean hasSunroof() {
    return ((optionalEquipment & SUNROOF) == SUNROOF);
  }

  public void setSunroof(boolean hasSunroof) {
    if (hasSunroof) {
       optionalEquipment = optionalEquipment | SUNROOF;
    } else {
       optionalEquipment = optionalEquipment ^ SUNROOF;
    }
  }
}
```

Worth to consider, if initialization is time consuming and the attribute is not always needed (immediately)

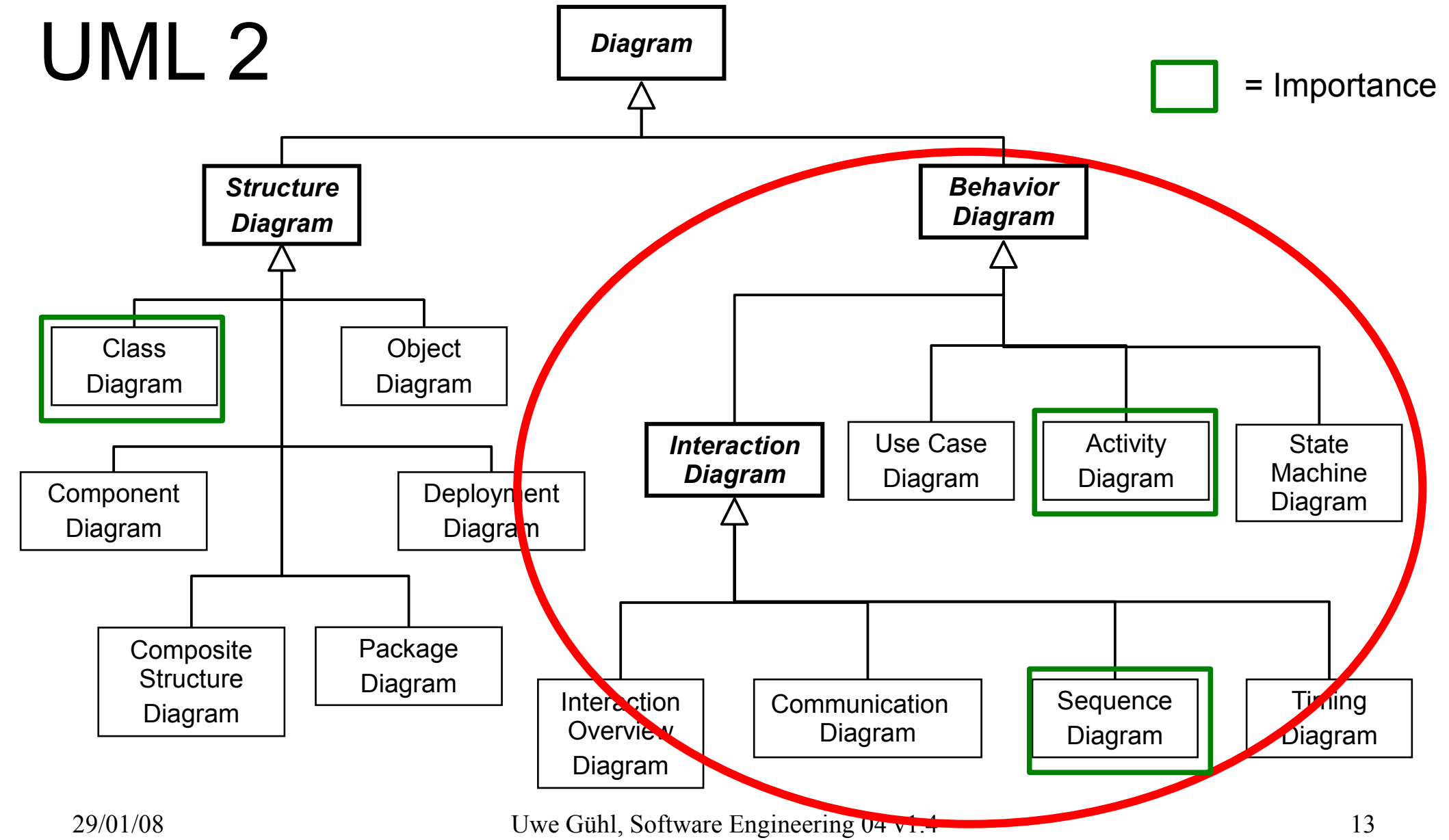
Saving variables (disk space), but more computing time needed

e. g. because of disk space or reorganization

Behavioral Diagrams Context

- UML Structural and Behavioral Diagrams
  - Structural Diagrams show basic information of a class, or a complete organization of an architecture or whole system. They support a structural view focusing on the static structure of the system using objects, attributes, operations, and relationships.
  - Behavioral Diagrams describe the dynamic behavior view focusing on object activities and interactions as well as internal changes of states. Use Case Diagrams show functional requirements from the user's point of view.

Behavioral Diagrams Overview



Behavioral Diagrams Overview

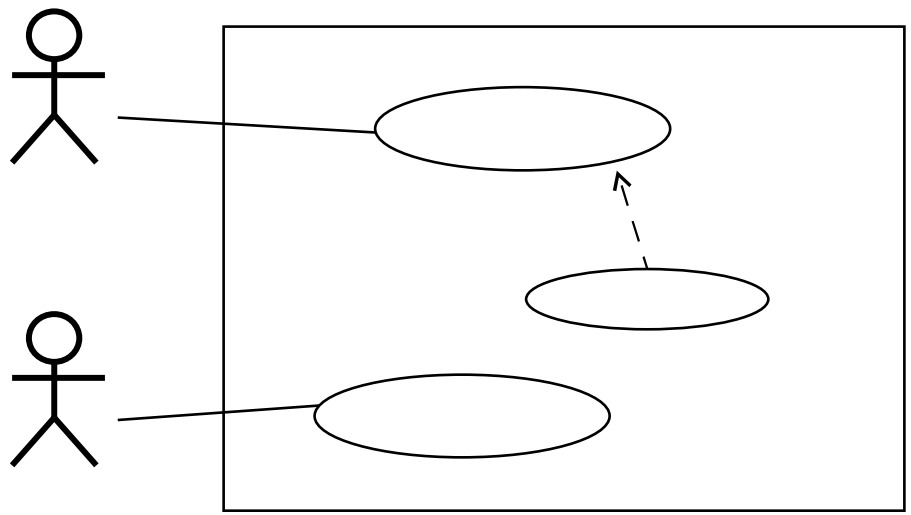
- **Use Case Diagrams** mainly show Actors, Use Cases, and their relations
- **Activity Diagrams** illustrate business processes, data flows, or complex logic within a system
- **State Machine Diagrams** depict the states of objects or interactions may be in, as well as the transitions between states because of events

Behavioral Diagrams Overview

- ***Interaction Diagrams***
  - **Sequence Diagrams** model the sequential logic, in effect the time ordering of messages between classifiers
  - **Communication Diagrams** show instances of classes, their relations, and the message exchange. Formerly called a Collaboration Diagram
  - **Timing Diagrams** depict the change in state or condition of a classifier instance or role over time. Typically used to show the change in state because of external events.
  - **Interaction Overview Diagram** overviews the control flow within a system or business process. Each node / activity within the diagram can show another interaction diagram

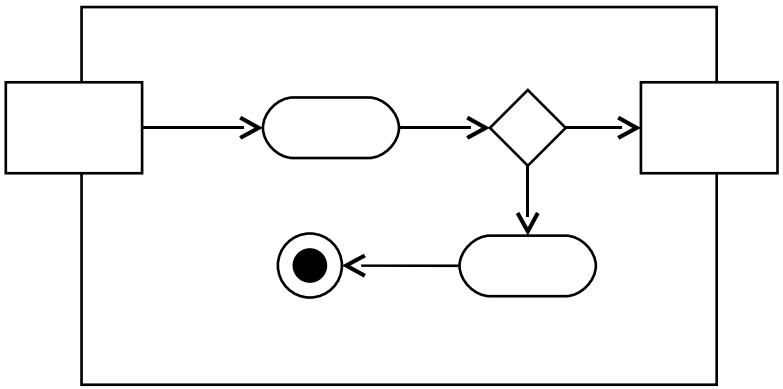
Behavioral Diagrams
Use Case Diagrams

- Questions to be answered
  - What provides my system for the environment?
- Strengths
  - shows the external point of view
  - outlines the context
  - high abstraction level
  - easy notation



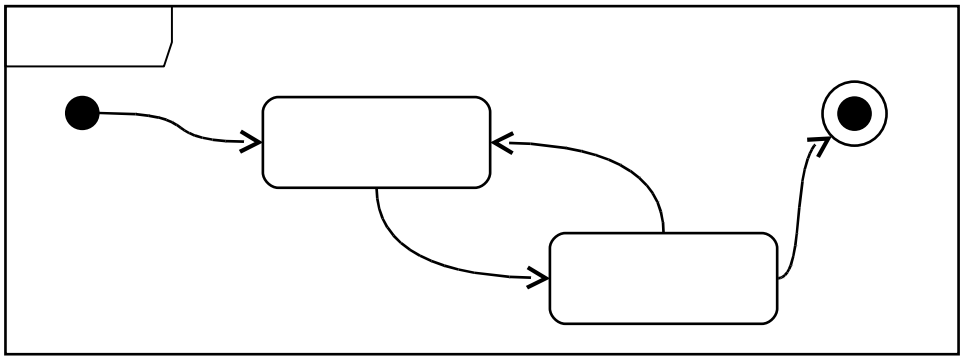
Behavioral Diagrams
Activity Diagrams

- Questions to be answered
  - What is the data flow in a process or algorithm?
- Strengths
  - Detailed description with
    - conditions
    - loops
    - junctions
  - Parallelization and synchronization possible
  - Depicting of data flows



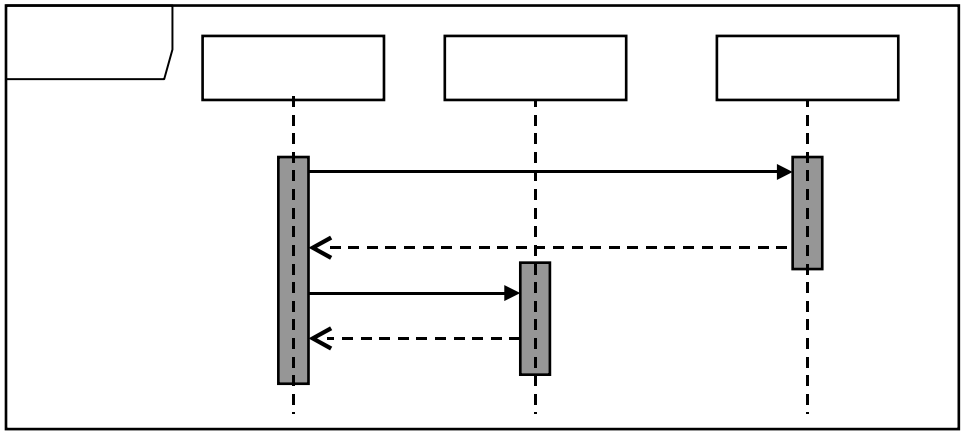
Behavioral Diagrams
State Machine Diagrams

- Questions to be answered
  - Which state could an object, an interface, or a Use Case get concerning specific events?
- Strengths
  - Precise illustration of a state model with
    - states
    - events
    - concurrencies
    - conditions
    - enter / exit activities



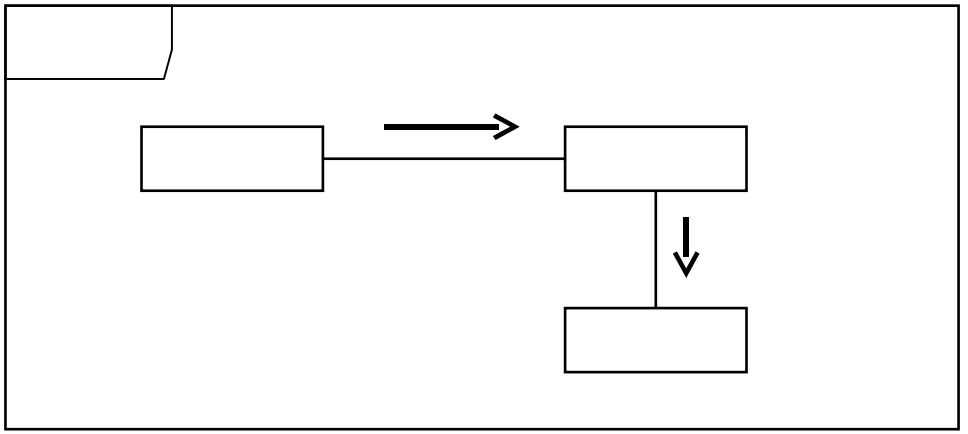
Behavioral Diagrams
Sequence Diagrams

- Questions to be answered
  - Who is exchanging with whom when what information?
- Strengths
  - shows in detail information exchange between communication partners
  - Precise presentation of chronological sequences with concurrencies



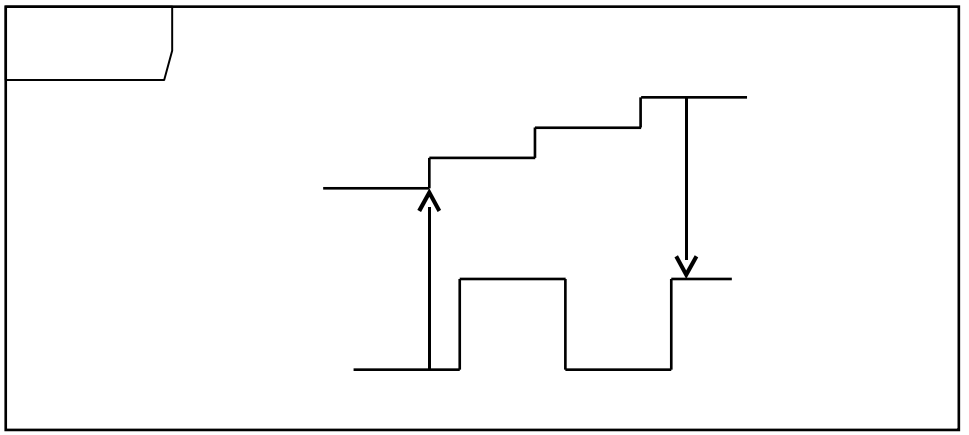
Behavioral Diagrams
Communication Diagrams

- Questions to be answered
  - Who communicates with whom?
  - Who collaborates?
- Strengths
  - displays information exchange between communication partners
  - offers an overview



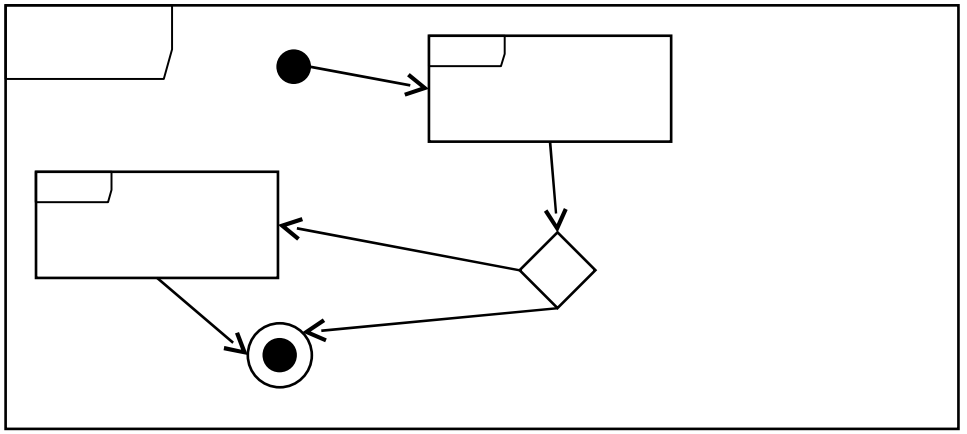
Behavioral Diagrams
Timing Diagrams

- Questions to be answered
  - When are which interaction partner in which state?
- Strengths
  - Visualization of time behavior of classes and interfaces
  - Ideal for studying time-critical details



Behavioral Diagrams
Interaction Overview Diagrams

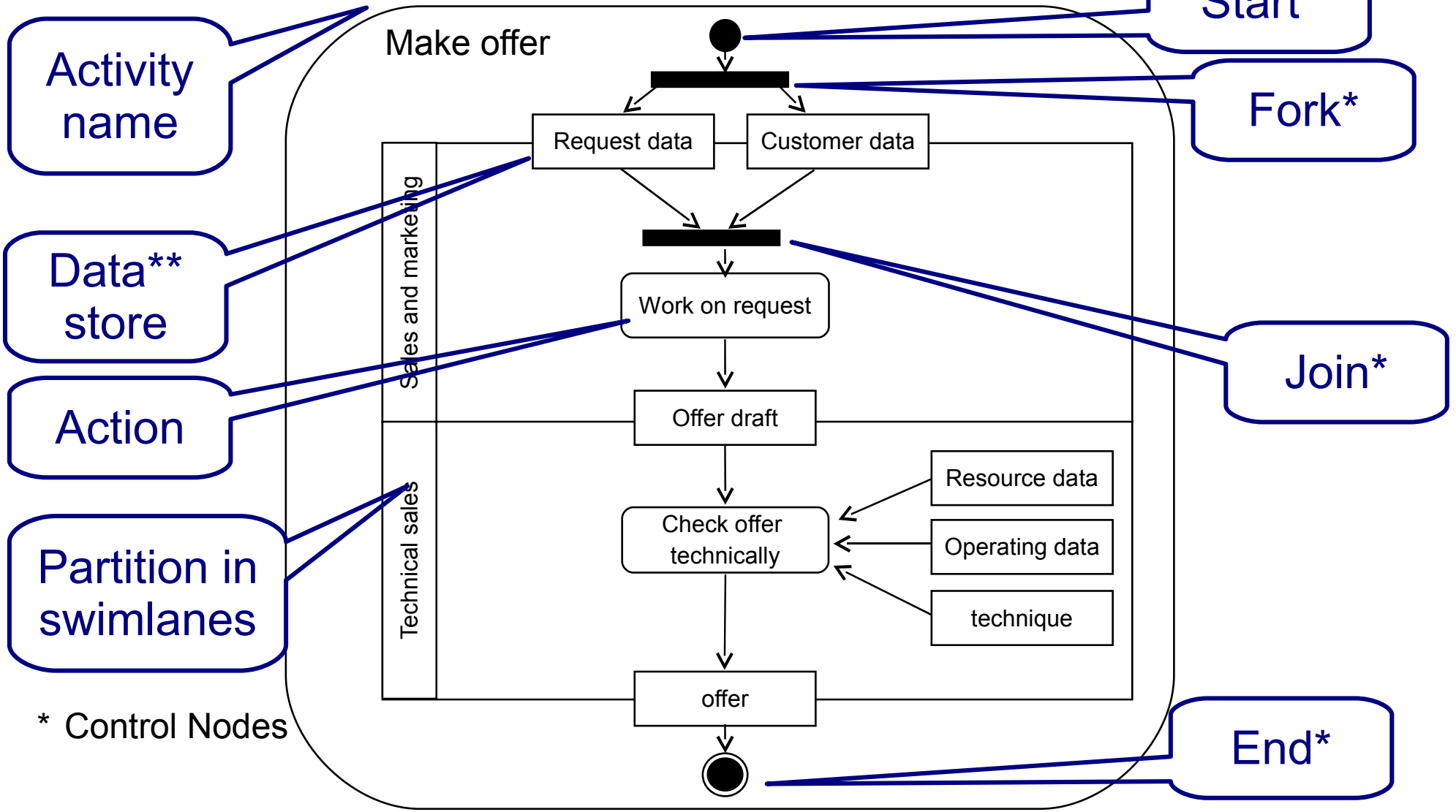
- Questions to be answered
  - When proceeds which action?
- Strengths
  - Connects Interaction Diagrams (Sequence / Communication / and Timing Diagrams) on top level
  - High abstraction level



Activity Diagrams

- Activity Diagrams
  - depict, how a system realizes a specific behavior
  - describe possible activities of a systems with different nodes, which are connected through object and control flows
  - are good for modelling of activity oriented classes (Example: Activities of Business Use Cases)
  - follow since UML 2.0 the Petri Net semantics
  - Also Nassi Shneiderman Struktrograms could be transferred to Activity Diagrams

Activity Diagrams



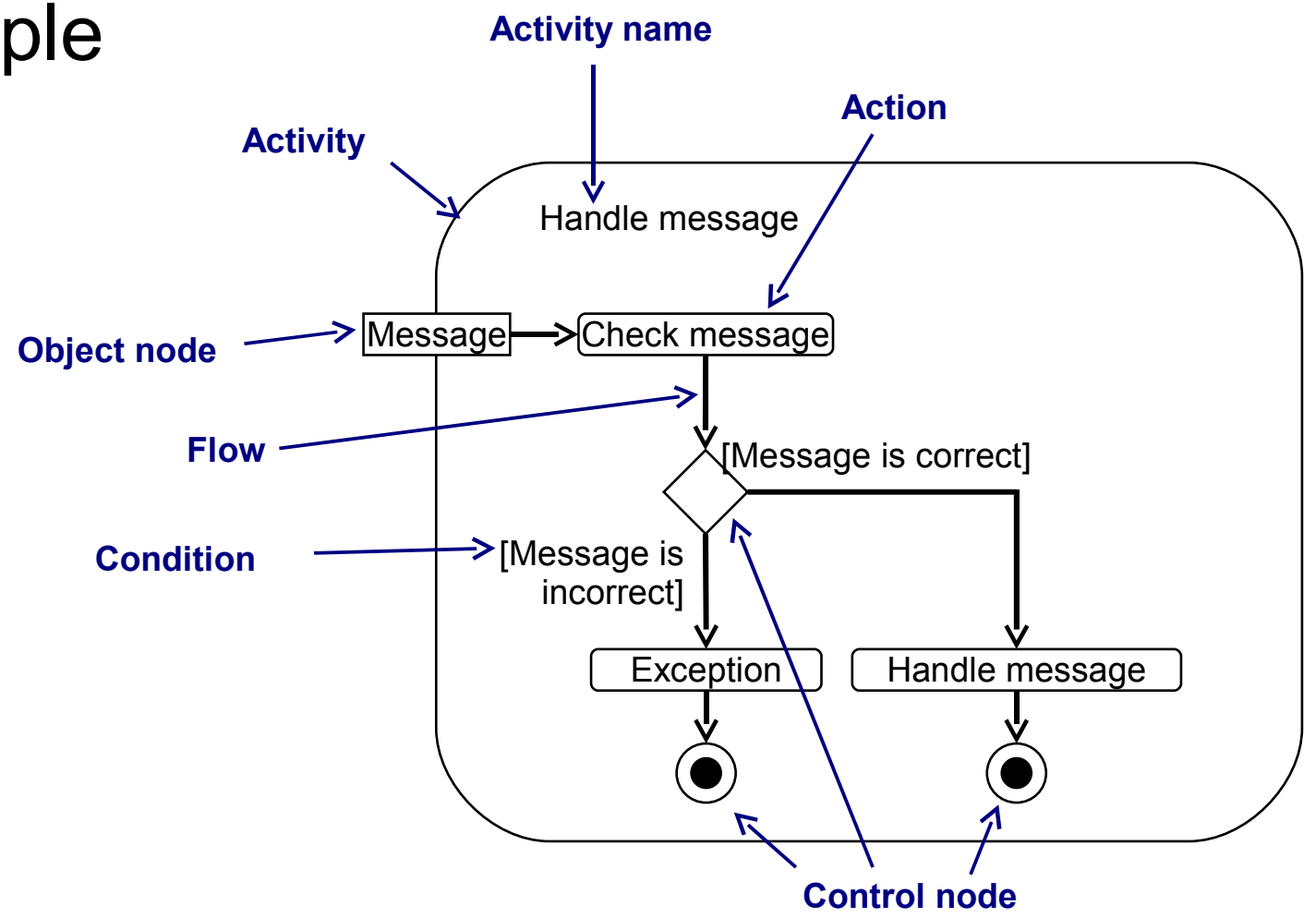
* Control Nodes

** Object Nodes

Example using an activity diagram

Activity Diagrams

- Example



State Machine Diagrams

- State models help in illustrating condition based behavior
- State Machine Diagrams show that
  - an object has different behavior in different states
  - an object reacts in different states only concerning defined events
  - defined events change the status of an object
- If such statements are not needed in your system you don't need State Machine Diagrams

State Machine Diagrams

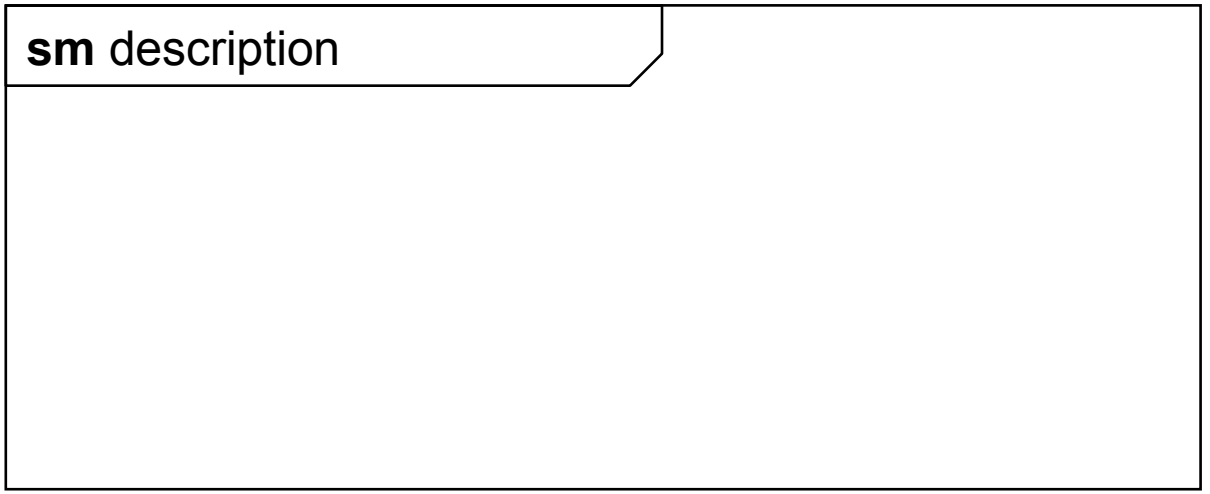
- State Machine Diagrams in the UML are an extension of „Final machines“ (work of D. Harel Mid of the 1980s)
- Simplified assumptions:
  - At a defined point in time a system has exactly one state
  - The transition from one state to another happens without time lag

State Machine Diagrams

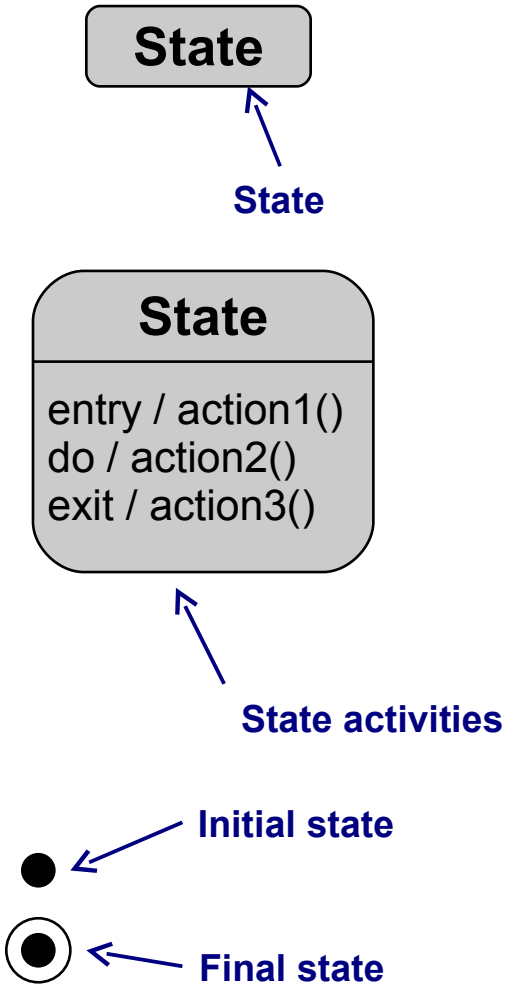
- When to use State Machine Diagrams?
  - To model system behavior in a defined state if defined events occur
  - System behavior should be fractioned in smaller and more simple parts …
    - … to develop and to code easier
    - … to test easier
  - To model parallel running state machines
  - to describe distributed systems

State Machine Diagrams

- A state machine is described as a frame, contenting a pentagon in the left top
- In the pentagon the token „**sm**“ stands for State Machine

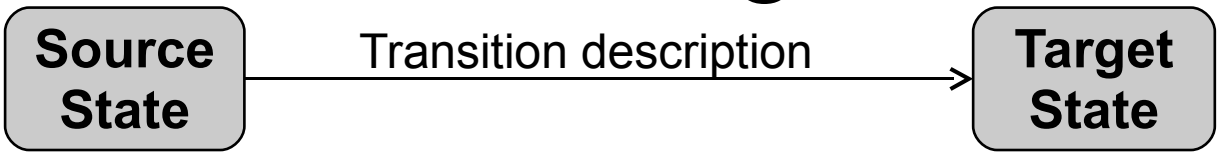


State Machine Diagrams



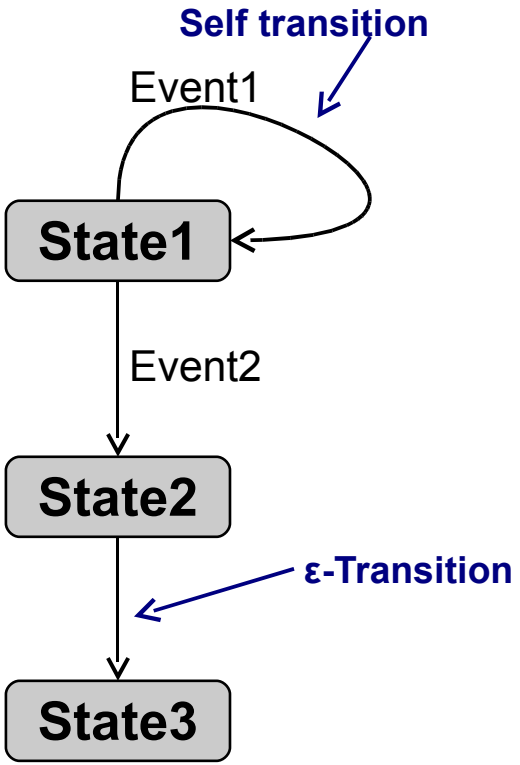
- States are symbolized with rounded rectangles, where the name of the state is written
- Events cause actions in the states:
  - **entry** (action by entering the state)
  - **exit** (action by leaving the state)
  - **do** (as long as the state is active and not left)
- An initial state is a special state without an entry transition
- A final state is a special state without an exit transition

State Machine Diagrams



- From one state to another are transitions possible
- The transition is described in following syntax:
  `event(arguments)`
  `[condition]`
  `/operation(arguments)`
- alternative description
  `Trigger`
  `[Guard]`
  `/Effect`
- Trigger: Activator for the transition, several triggers are separated by comma
- Guard: Condition, that must be true, to execute the transition
- Activity: This gets executed in the transition

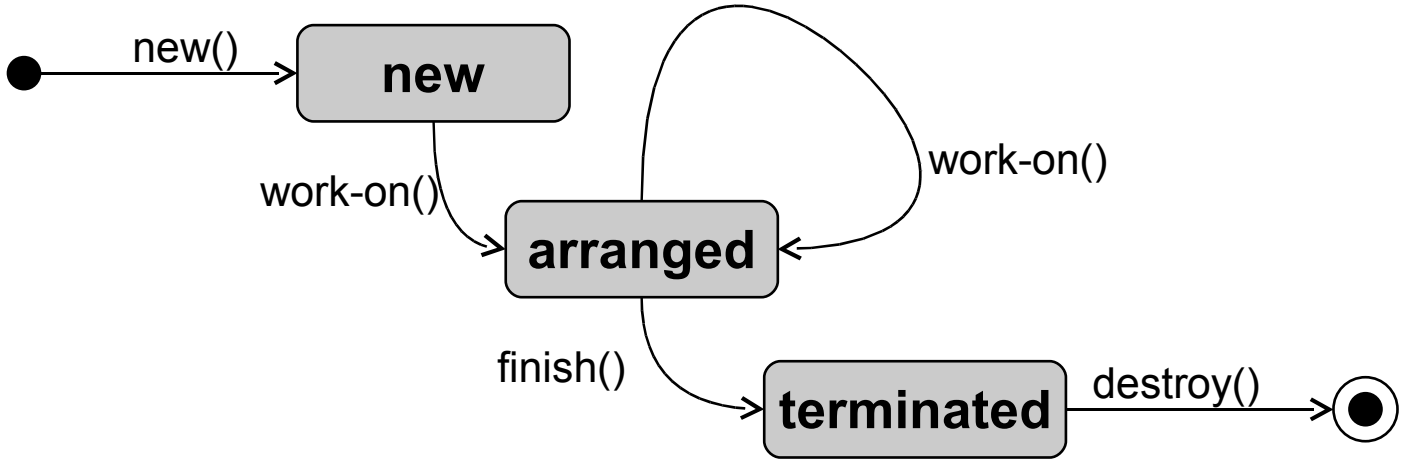
State Machine Diagrams



- If source state and target state are the same, we talk about self transition
- Transitions without label get executed automatically, as soon as the actions of the source state are finished
- Hint: Epsilon transitions are transitions a machine can make without consuming any input symbol

State Machine Diagrams

- Example for states of a document

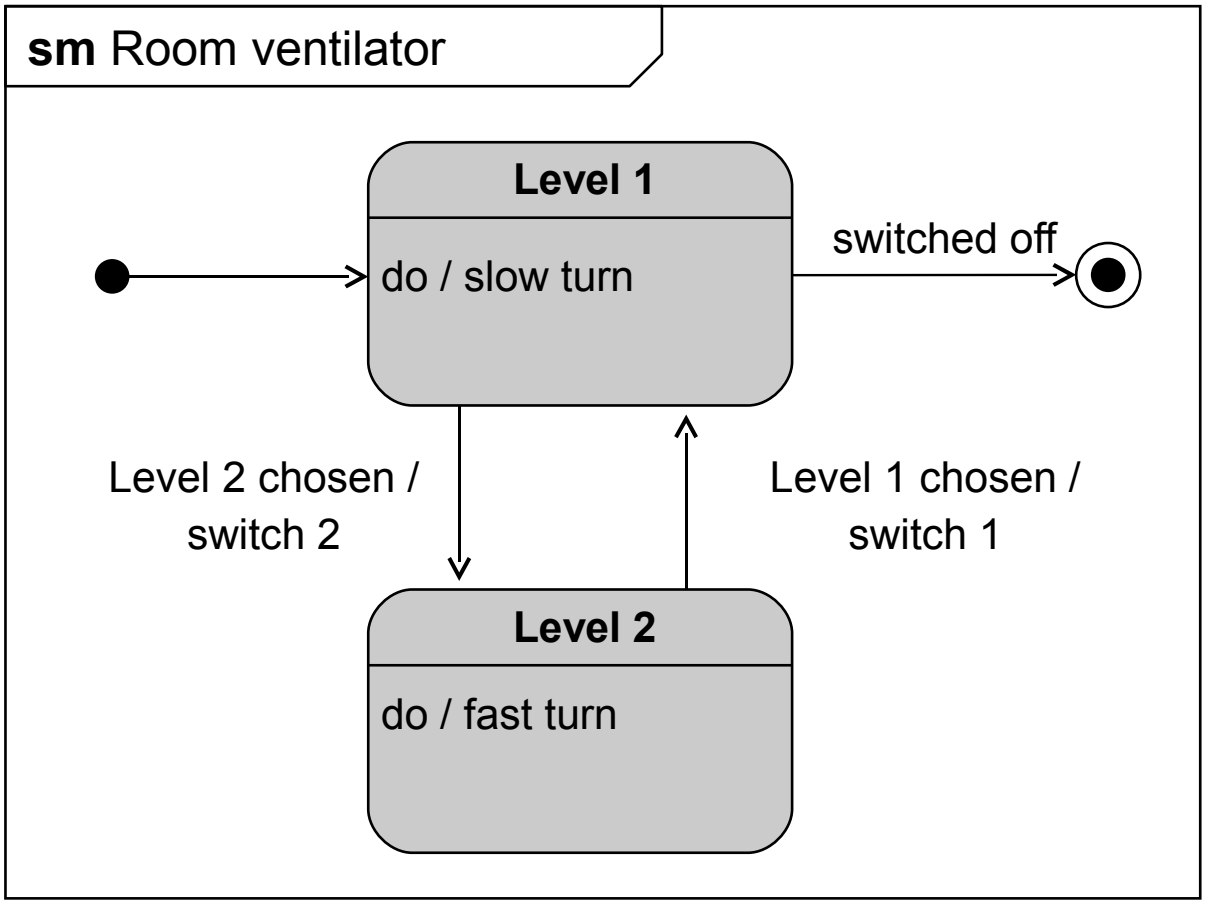


State Machine Diagrams

- Example for states of a document
  - A document is after creation in the state `new`
  - The activity `work-on()` changes it into the state `arranged`
  - Proceeding with `work-on()` does not change the state.
  - The transition `finish()` moves the document to the status `terminated`
  - Only from the state `terminated` it is possible to remove the document out of the system

State Machine Diagrams

- Example for a State Machine Diagram

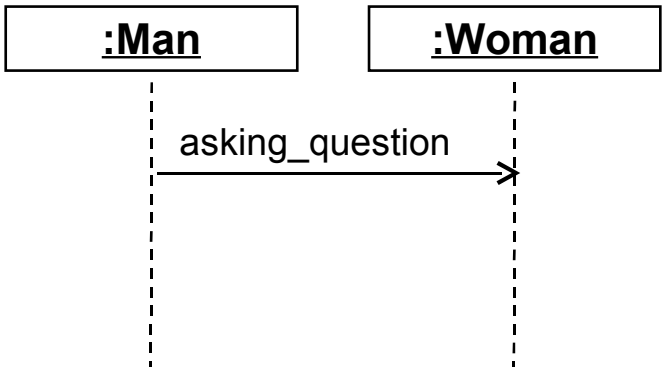


Interaction Diagrams

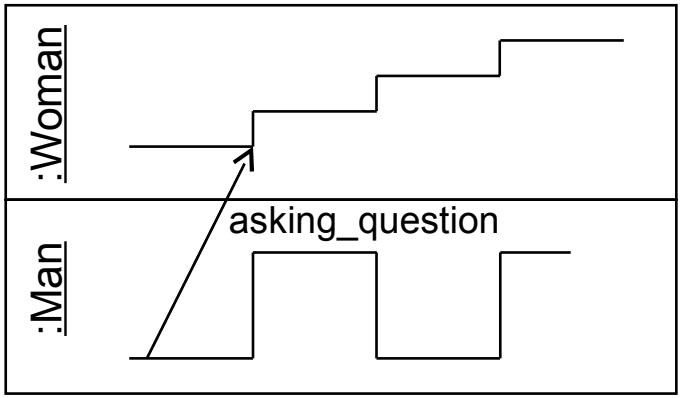
- Basic principle of UML:
  A UML Diagram is only one possible view, describing a specific modeling aspect
- This basic principle becomes apparent in the modelling of interactions

Interaction Diagrams

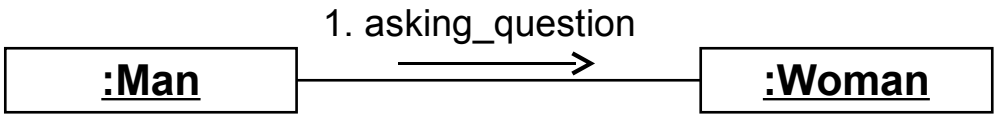
- Examples



Sequence Diagram



Timing Diagram



Communication Diagram

Interaction Diagrams

- In the Interaction diagrams we focus on
  - **Sequence Diagrams** show the communication in a system
  - **Communication Diagrams** show, how parts of a structure work together to fulfill a specified function
- Additional information
  - **Timing Diagrams** show time behavior of a system, e. g. time behavior of digital circuits
  - **Interaction Overview Diagrams** illustrate the collaboration of different interactions with a variant of the Activity Diagram

Sequence Diagrams

- Sequence Diagrams ...
  - are the most used Interaction Diagrams
  - show the information exchange
    - between communication partners in a system
    - between systems
  - model
    - fix sequences
    - chronological and logical activity conditions
    - loops
    - concurrencies

Sequence Diagrams

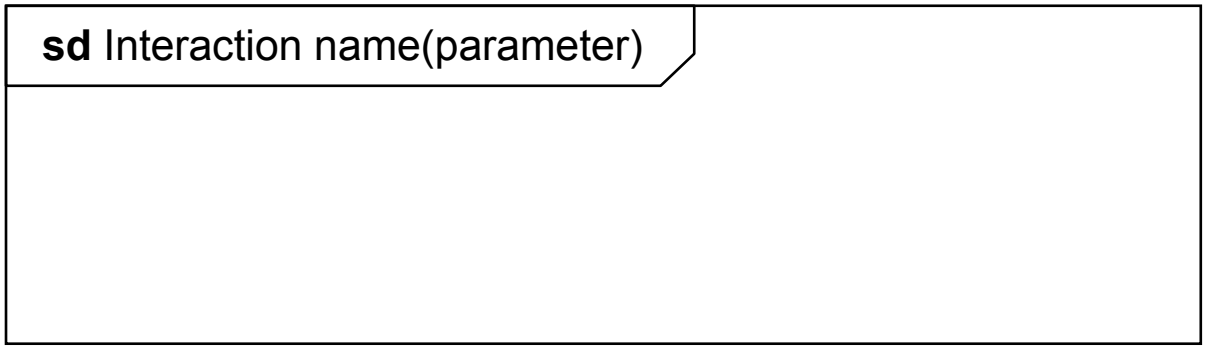
- A sequence shows a number of messages, exchanged by a defined set of objects in a temporal limited situation
- A Sequence Diagram highlights the chronological way of the communication, where the time flows top down

Sequence Diagrams

- In principle a Sequence Diagram could be transferred in a Communication Diagram, as is shows the same facts, only from another perspective
- Following exceptions could not be displayed in Communication Diagrams
  - Interaction references („`ref`“)
  - Combined fragments
  - Event order

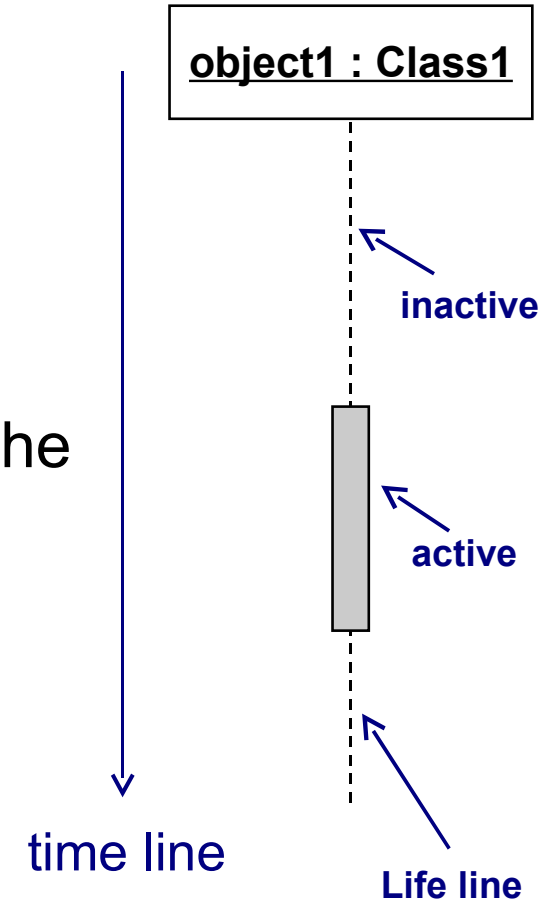
Sequence Diagrams

- Notation (1)
  - **sd** (for Sequence Diagram) with the name of the interaction and optional parameter
  - **sd** is used as well in
    - Communication Diagrams
    - Timing Diagrams
    - Interaction Overview Diagrams



Sequence Diagrams

- Notation (2)
  - Time flows top down
  - A dashed perpendicular (time) line shows the life time of a communication partner (objects)
  - Above the line the object name is placed in a box.
  - A gray bar, overlaying the lifeline, represents the control flow, that is the area, where the communication partner are active
  - Modeling of the activity sequence is optional
  - At the border explanations and conditions like time constraints could be added)



Sequence Diagrams

- Notation (3)
  - Messages can be complete, lost or found, synchronous or asynchronous, call or signal
  - Messages are depicted as arrows between the lifelines of the objects. The message is displayed on them following the form **`message(arguments)`**
  - Solid arrowheads represent **synchronous messages** (answer expected)

  - Line arrowheads represent **asynchronous messages** (no answer expected)

  - Asynchronous messages may intersect, as they could arrive in another order as they were sent

Sequence Diagrams

- Notation (4)
  - A creation of a new entity is symbolized with a dashed line, pointing to the created object

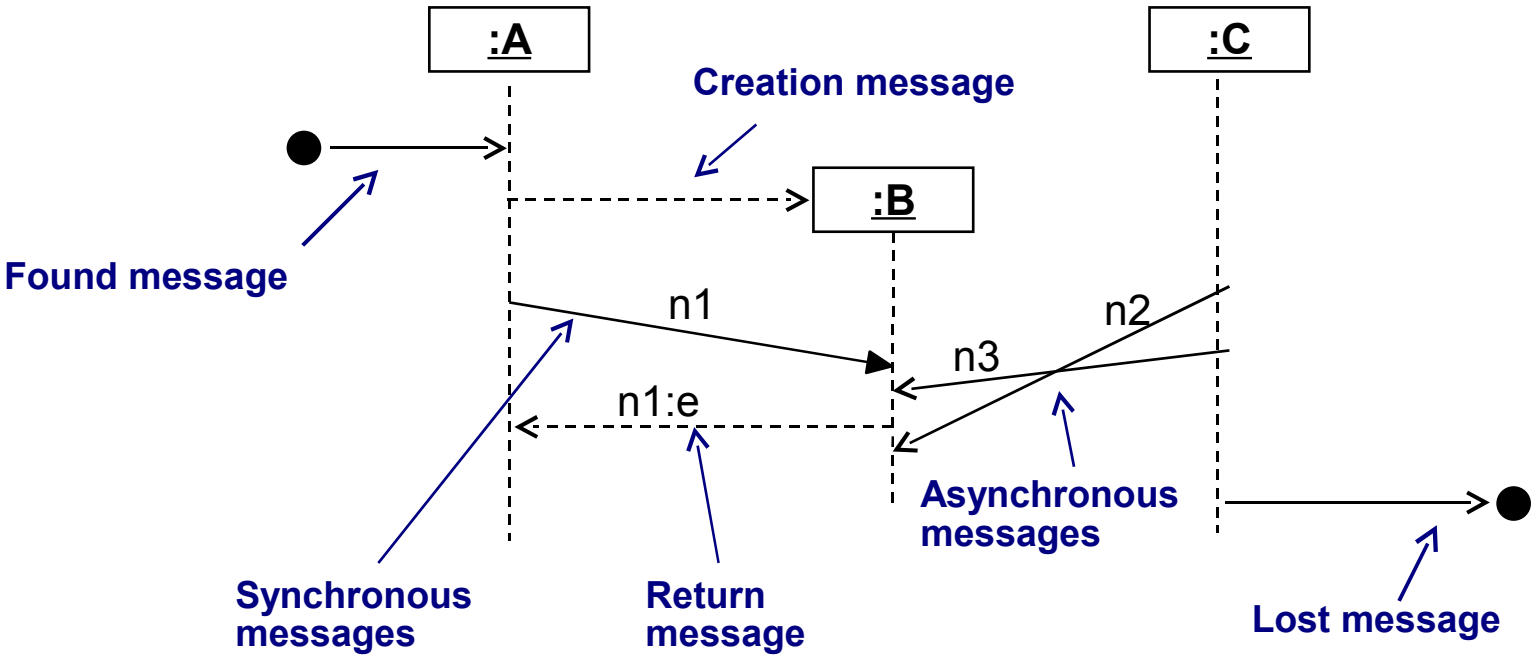
  ---------->

  - Return messages are displayed either as text in the form (**`return := message()`**), or as separate, dashed arrow with solid arrowhead

  <----------

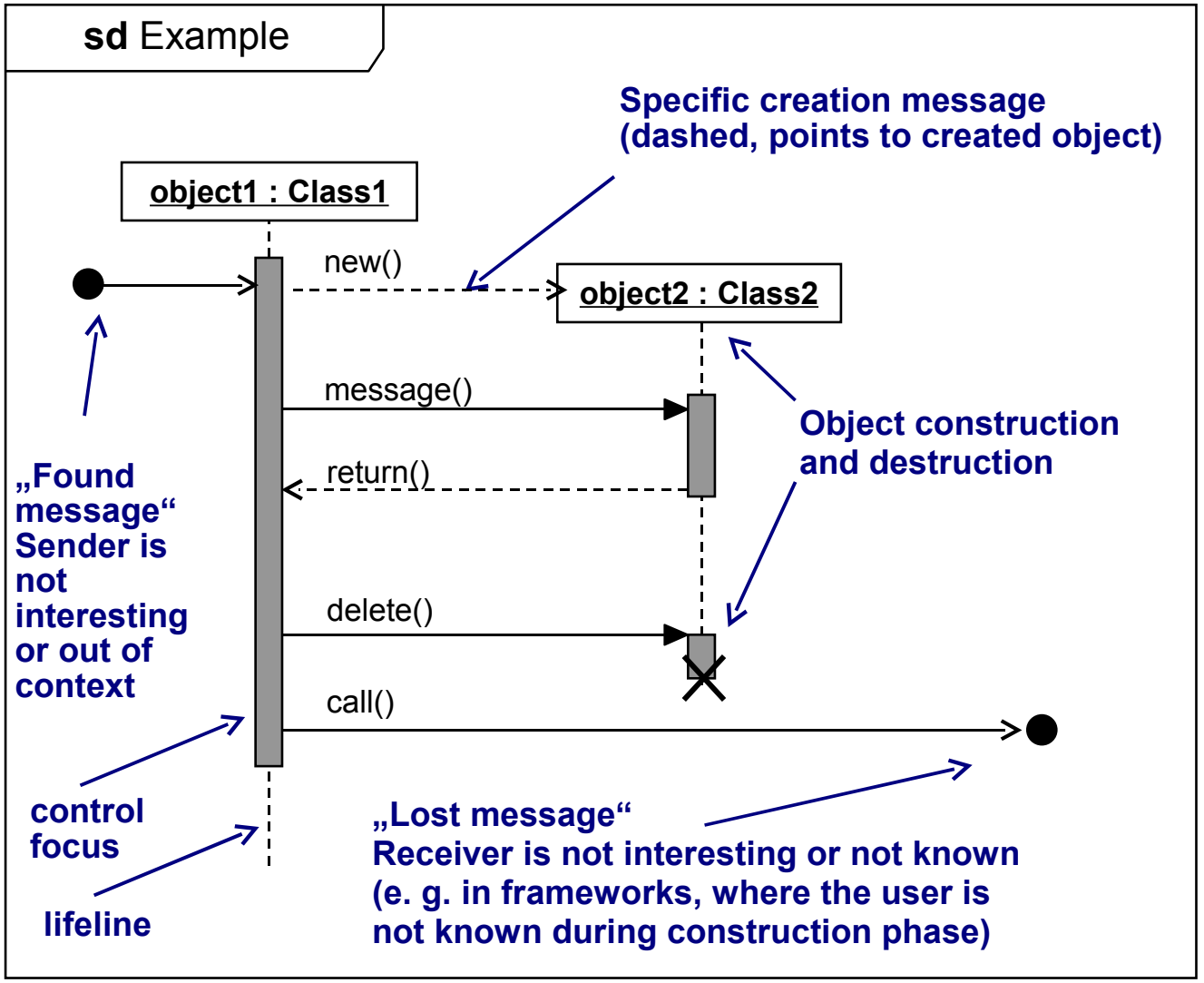
Sequence Diagrams

- Notation (5)
  - Example



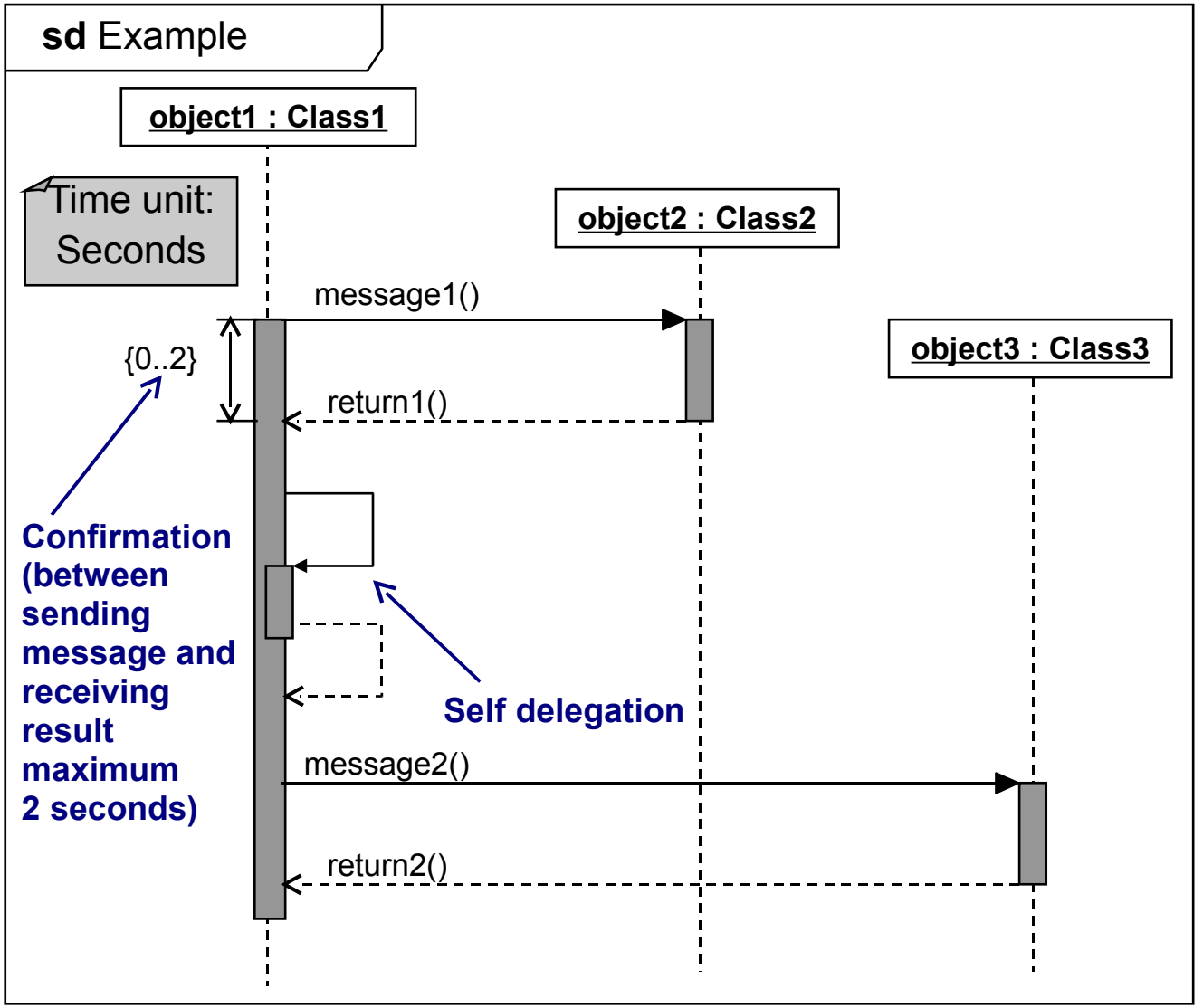
Sequence Diagrams

- Notation (6)



Sequence Diagrams

- Notation (7)



Sequence Diagrams Combined fragments (1)

- Combined (interaction) fragment help to model a set of possible application flows in Sequence Diagrams
- A combined fragment indicates an interaction part, where specific rules are valid, influencing the choices, order, and frequency of sending and receiving events in the fragment
- Combined fragments represent all fundamental control structures of programming languages like loops and conditions

Sequence Diagrams Combined fragments (2)

- Overview Interaction operators

`alt`	**Alternative** fragment E. g.: Models „IF/THEN/ELSE“ constructs
`opt`	**Optional** fragment E. g.: Models „SWITCH/CASE“ constructs
`break`	**Break** controls exceptions and alternatives E. g.: Exception handling
`neg`	**Negative** fragment encloses an invalid series of messages E. g.: Negative Test sequences
`loop`	**Loop** fragment encloses a series of messages which are repeated E. g. : modelling of loops (minint, maxint)
`par`	**Parallel** fragment – models concurrent processing E. g.: Interactions in any order
`seq`	**Weak sequencing** fragment – encloses a number of sequences for which all the messages must be processed in a life line in a defined order
`strict`	**Strict sequencing** fragment - encloses a series of messages which must be processed in the given order
`critical`	**Critical region** encloses a critical atomic section. E. g.: Fragments, which should not be interrupted
`ignore`	**Ignore** is a filter for *unimportant* messages E. g.: Messages of no interest in specific context, a timer
`consider`	**Consider** is a filter for *important* messages E. g..: Emphasizing of notable messages
`assert`	**Assertion** describes indispensable Interactions An implementation has to follow the model exactly

= important

Sequence Diagrams Combined fragments (3)

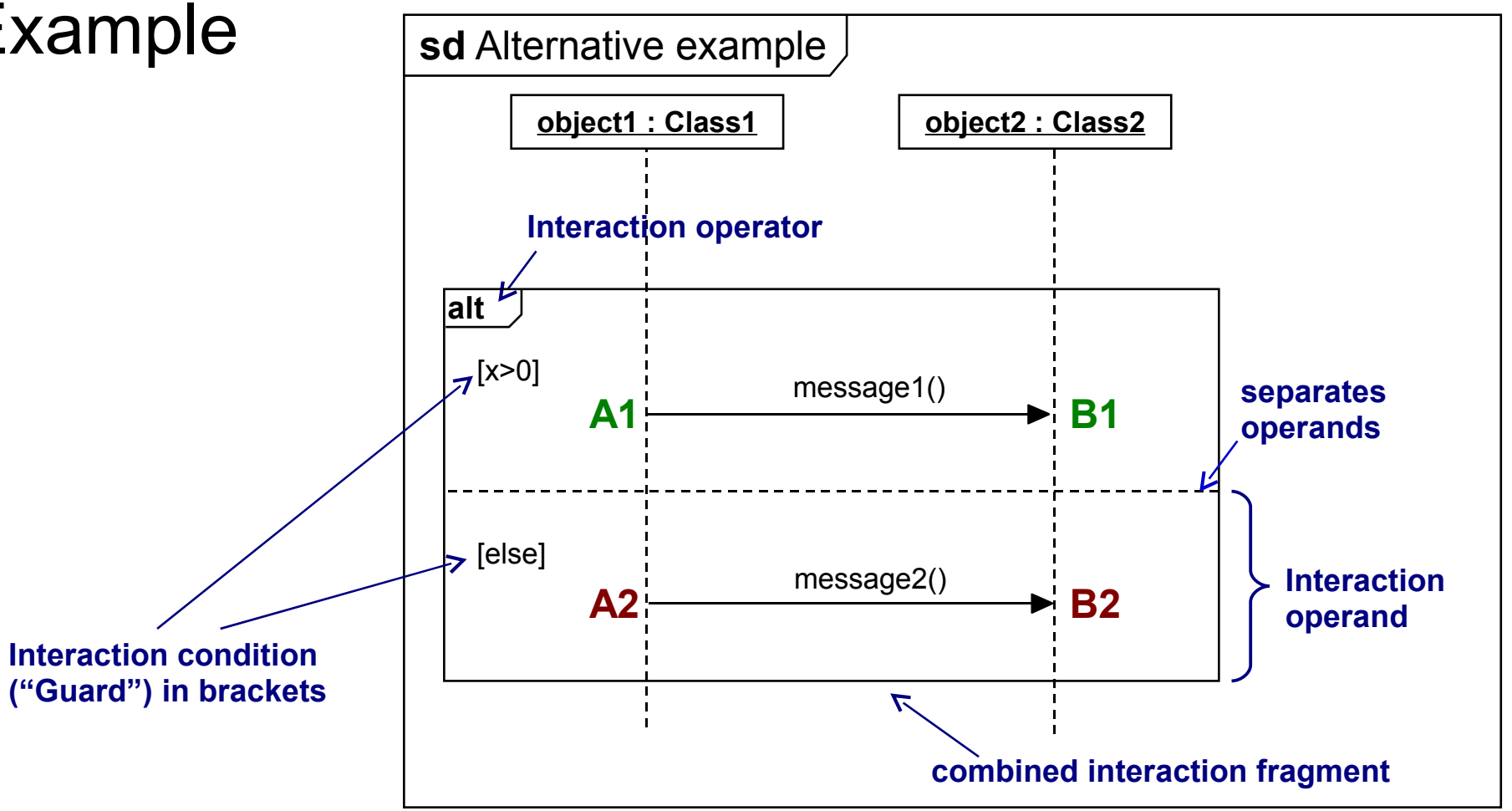
- Alternative fragment (`alt`)
  - Alternative fragments model two or more alternative activities, executed depending on conditions
  - The conditions have to be disjunctive – exactly one alternative has to be executed
  - Example: Two possible sets of activities concerning sending and receiving events

    A) Sending event message1() concerning object1 (**A1**)
    Receiving event message1() concerning object2 (**B1**)

    B) Sending event message2() concerning object1 (**A2**)
    Receiving event message2() concerning object2 (**B2**)

Sequence Diagrams Combined fragments (4)

- Alternative fragment (`alt`) Example

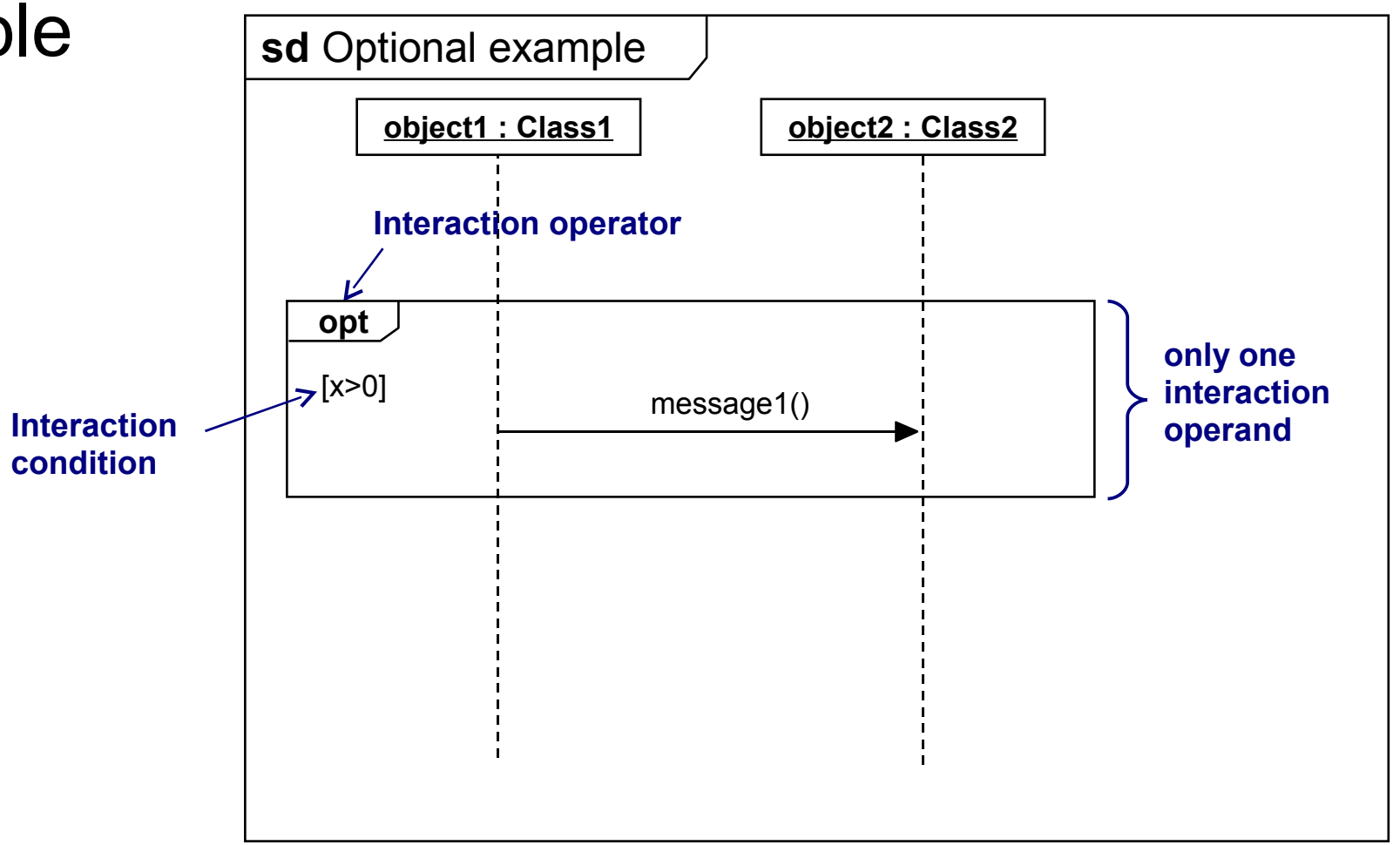


Sequence Diagrams Combined fragments (5)

- Optional fragment (`opt`)
  - The optional fragment models an optional activity, executed depending on a condition
  - The optional fragment is a simplified presentation of two alternative fragments, where the second fragment is empty

Sequence Diagrams Combined fragments (6)

- Optional fragment (`opt`) Example

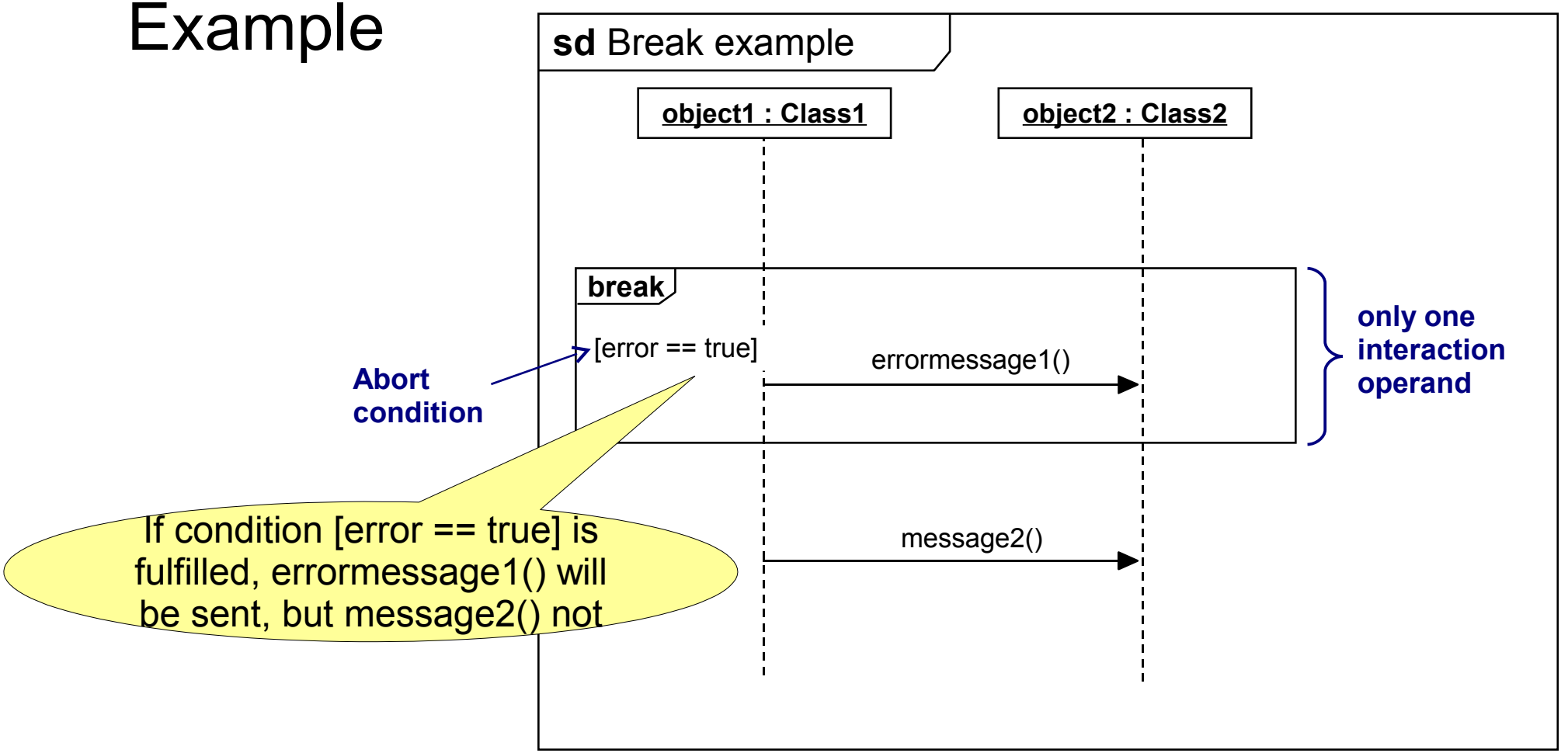


Sequence Diagrams Combined fragments (7)

- Break fragment (`break`)
  - The break fragment models an alternative sequence of events that is processed instead of the whole of the rest of the diagram
  - The break fragment models the handling of exceptions
  - If a condition is fulfilled, the break fragment gets executed, and then the direct environmental interaction operator gets terminated – e. g. a loop fragment

Sequence Diagrams Combined fragments (8)

- Break fragment (`break`) Example

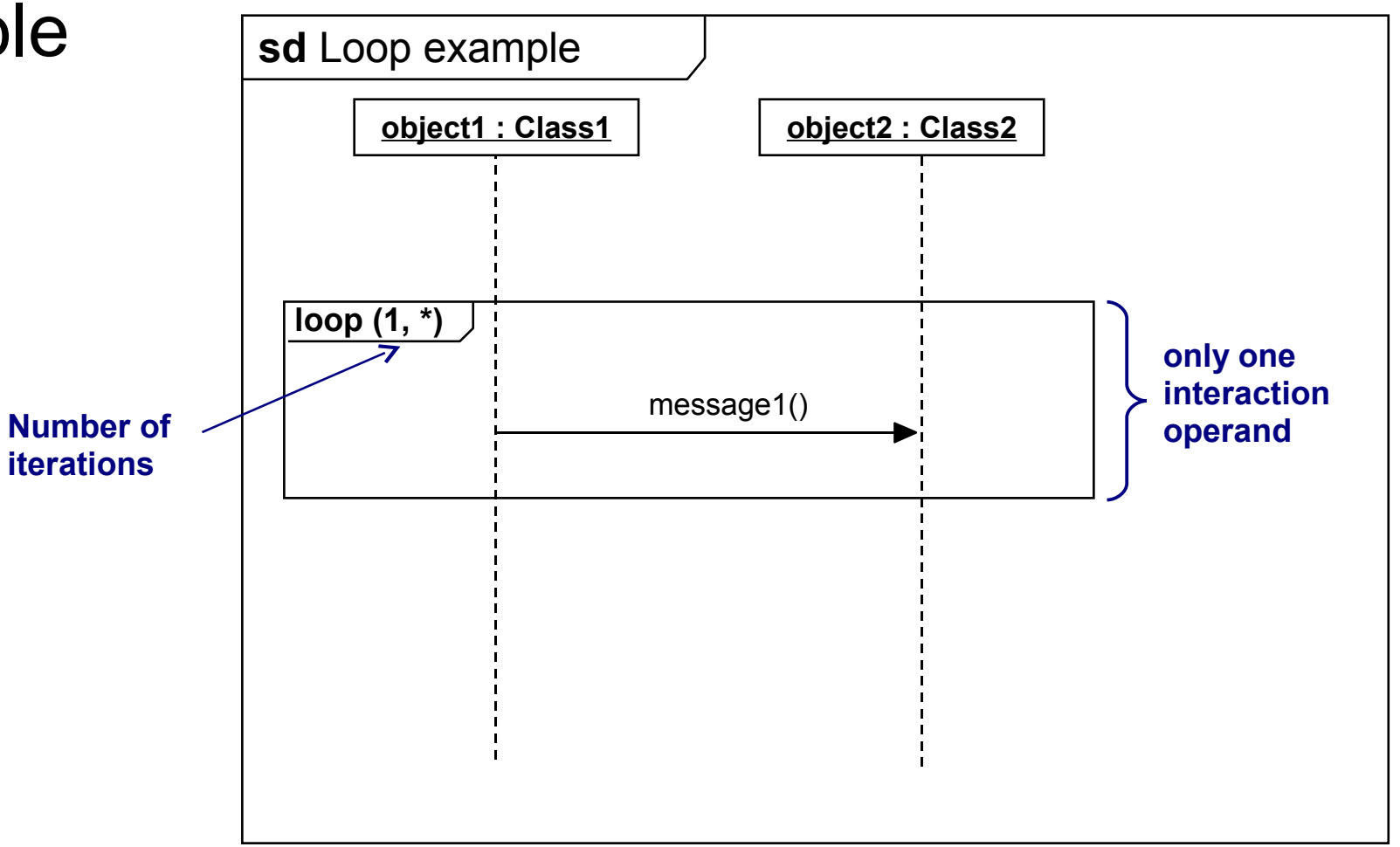


Sequence Diagrams Combined fragments (9)

- Loop fragment (`loop`)
  - The loop models an iteration, possible notations:
    - `loop(minint, maxint)`
      The loop traverses minimal `minint` and maximal `maxint` times
    - `loop(minint, *)`
      The loop traverses minimal `minint` times
    - `loop(minint)`
      The loop traverses exact `minint` times
    - `loop(minint)` means `loop(0, *)`
    - `loop(<boolean expression>)`
      The loop gets repeated, until the boolean operator is evaluated as `false`

Sequence Diagrams Combined fragments (10)

- Loop fragment (`loop`) Example



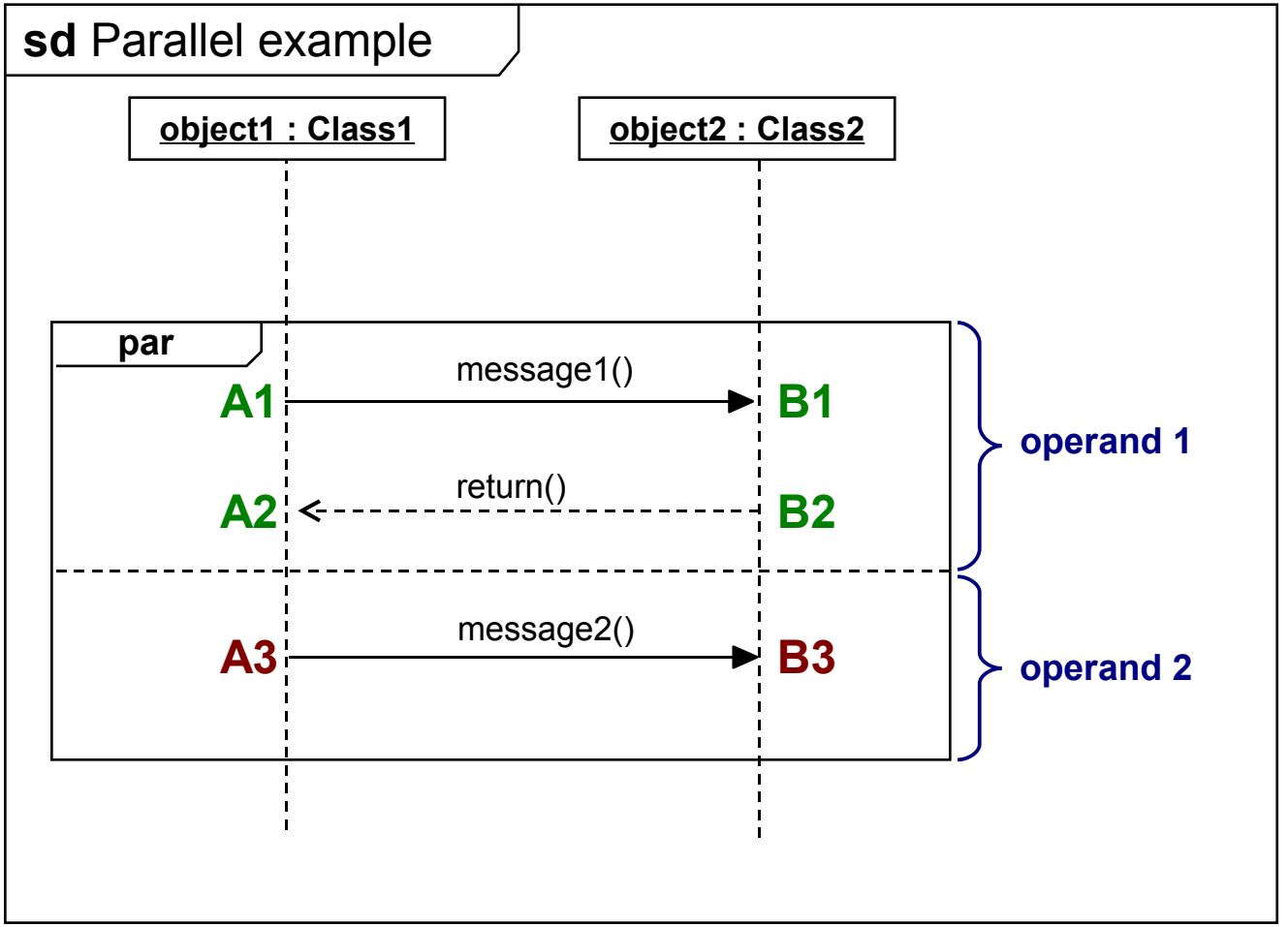
Sequence Diagrams Combined fragments (11)

- Parallel fragment (`par`)
  - The parallel fragment models operands, that could be executed in any order
  - In the operands the order is sequential

Sequence Diagrams Combined fragments (12)

- Parallel fragment (`par`) Example

Order in operands fix
A1, B1, B2, A2
A3, B3
Possible combinations
A3, A1, B1, B3, B2, A2
A1, A3, B1, B2, A2, B3
....



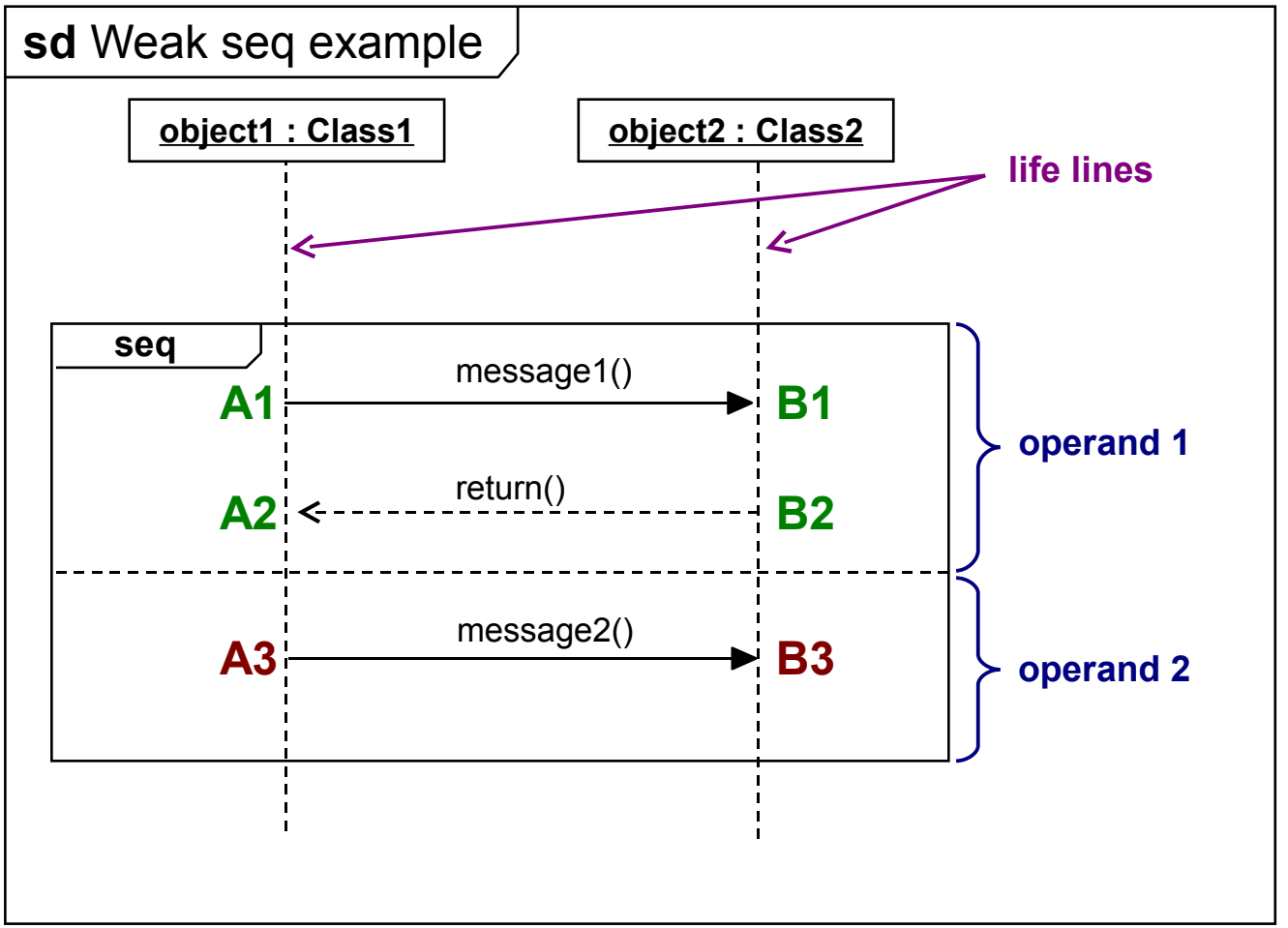
Sequence Diagrams Combined fragments (13)

- Weak sequencing fragment (`seq`)
  - The weak sequencing fragment acts like `par`, but additionally it has to be considered, that the order on a life line stays remained
  - A Sequence Diagram without interaction operators has the same logic like a `seq` fragment

Sequence Diagrams Combined fragments (14)

- Weak sequencing fragment (`seq`) Example

Order in operands fix
A1, B1, B2, A2
A3, B3
Order in life lines fix
A1, A2, A3
B1, B2, B3
Here only one possible combination
A1, B1, B2, A2, A3, B3



Sequence Diagrams Combined fragments (15)

- Weak sequencing fragment (`seq`) Example

Order in operands fix

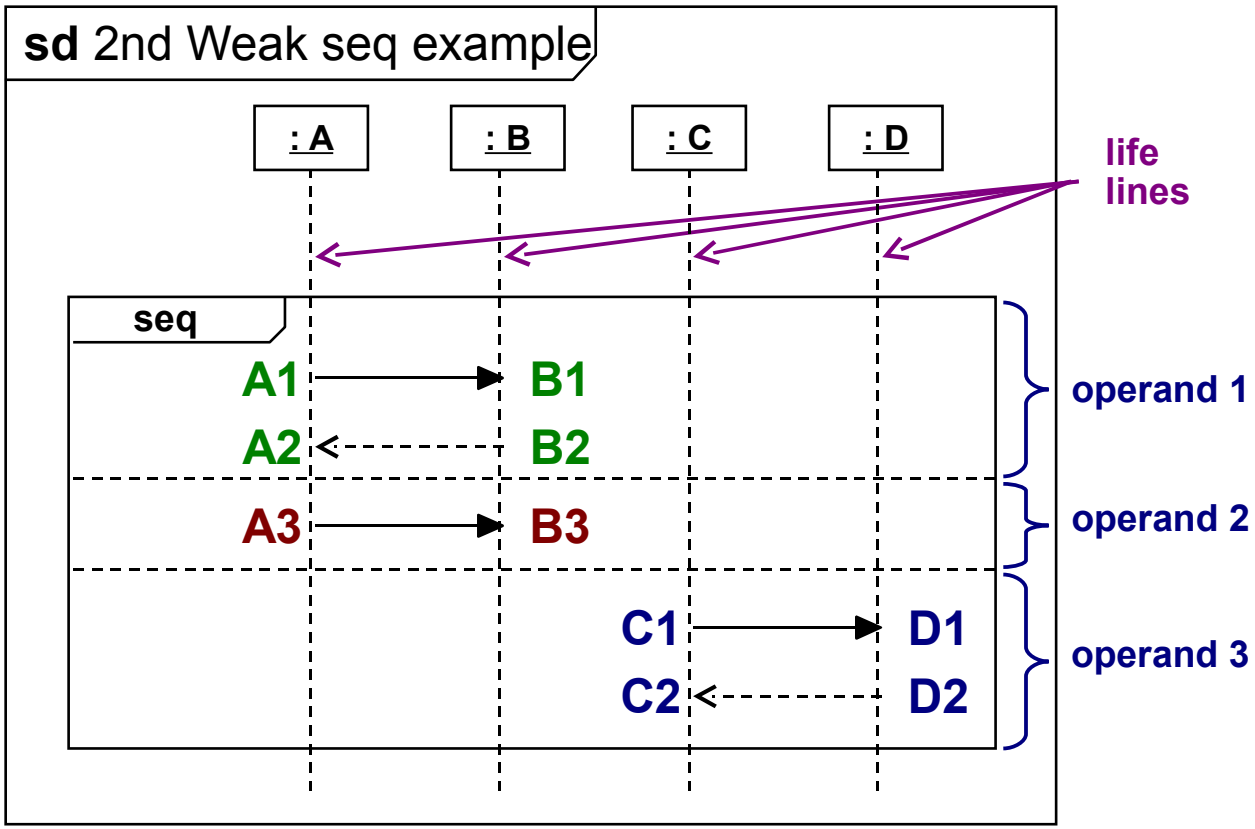
A1, B1, B2, A2
A3, B3
C1, D1, D2, C2

Order in life lines fix

A1, A2, A3
B1, B2, B3
C1, C2
D1, D2

Possible combinations

A1, B1, B2, A2, A3, B3, C1, D1, D2, C2
A1, B1, C1, D1, B2, A2, A3, B3, D2, C2



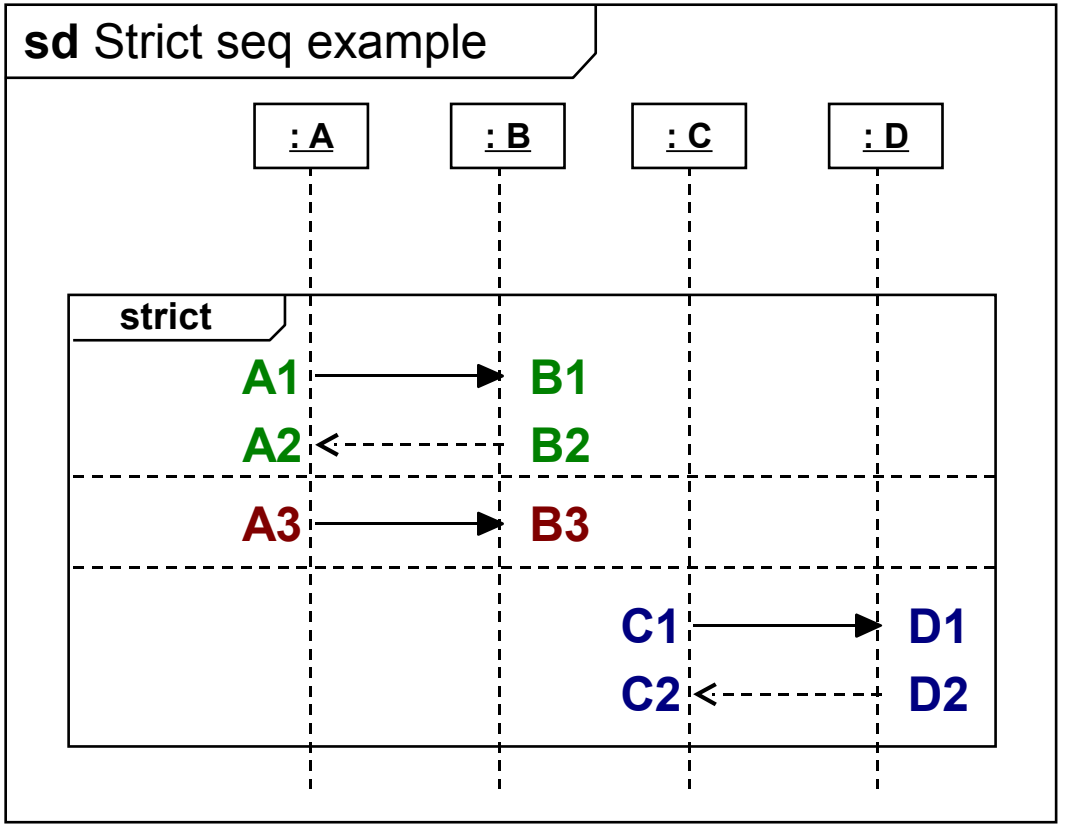
Even if **C1** is noticed after **B2**, it could be executed before **B2**

Sequence Diagrams Combined fragments (16)

- Strict sequencing fragment (`strict`)
  - `strict` is acting like a `seq` fragment, but the order has to be noted exactly like in the diagram
  - Several operands are not needed, when using `strict`
  - `strict` affects only operands on its level. Additional nested operands have their one rules
  -

Sequence Diagrams Combined fragments (17)

- Weak sequencing fragment (`strict`) Example



Only possible combination

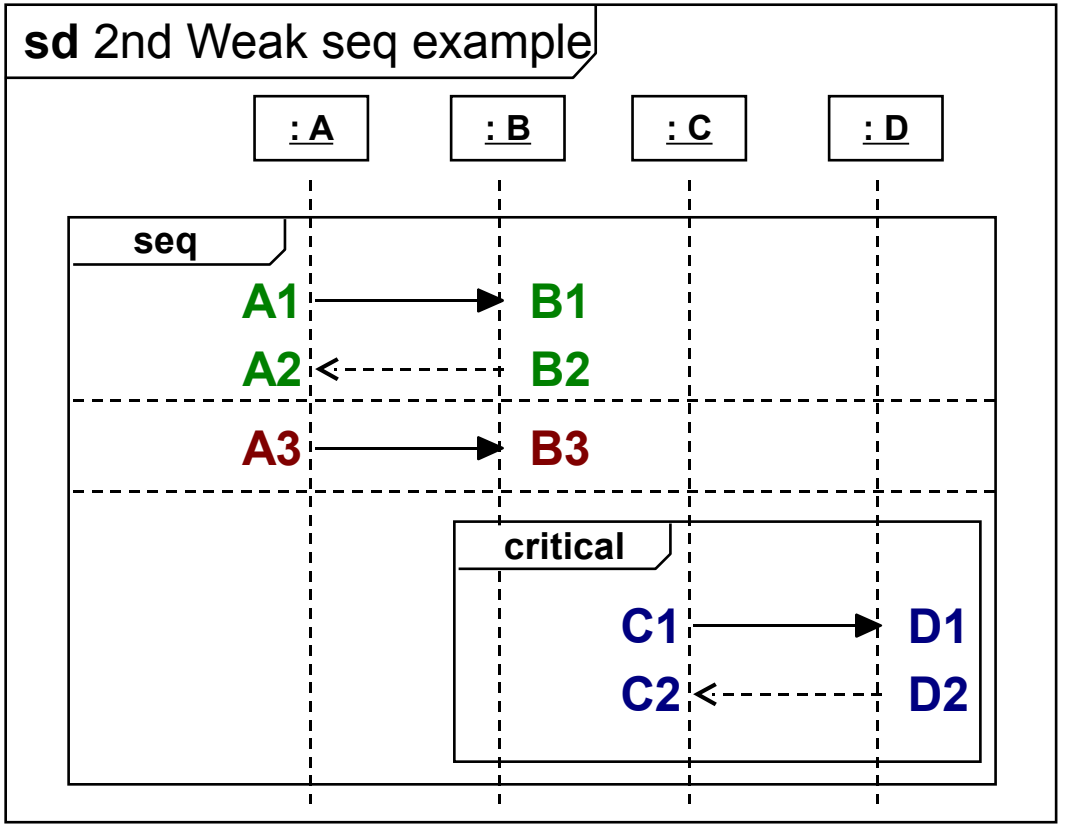
A1, B1, B2, A2, A3, B3, C1, D1, D2, C2

Sequence Diagrams Combined fragments (18)

- Critical fragment (`critical`)
  - Critical fragments model an atomic area
  - During the execution of a `critical` area no other sending and requesting activities take place beyond
  - `critical` is important for concurrent systems

Sequence Diagrams Combined fragments (19)

- Critical fragment (`critical`) Example



Possible combinations

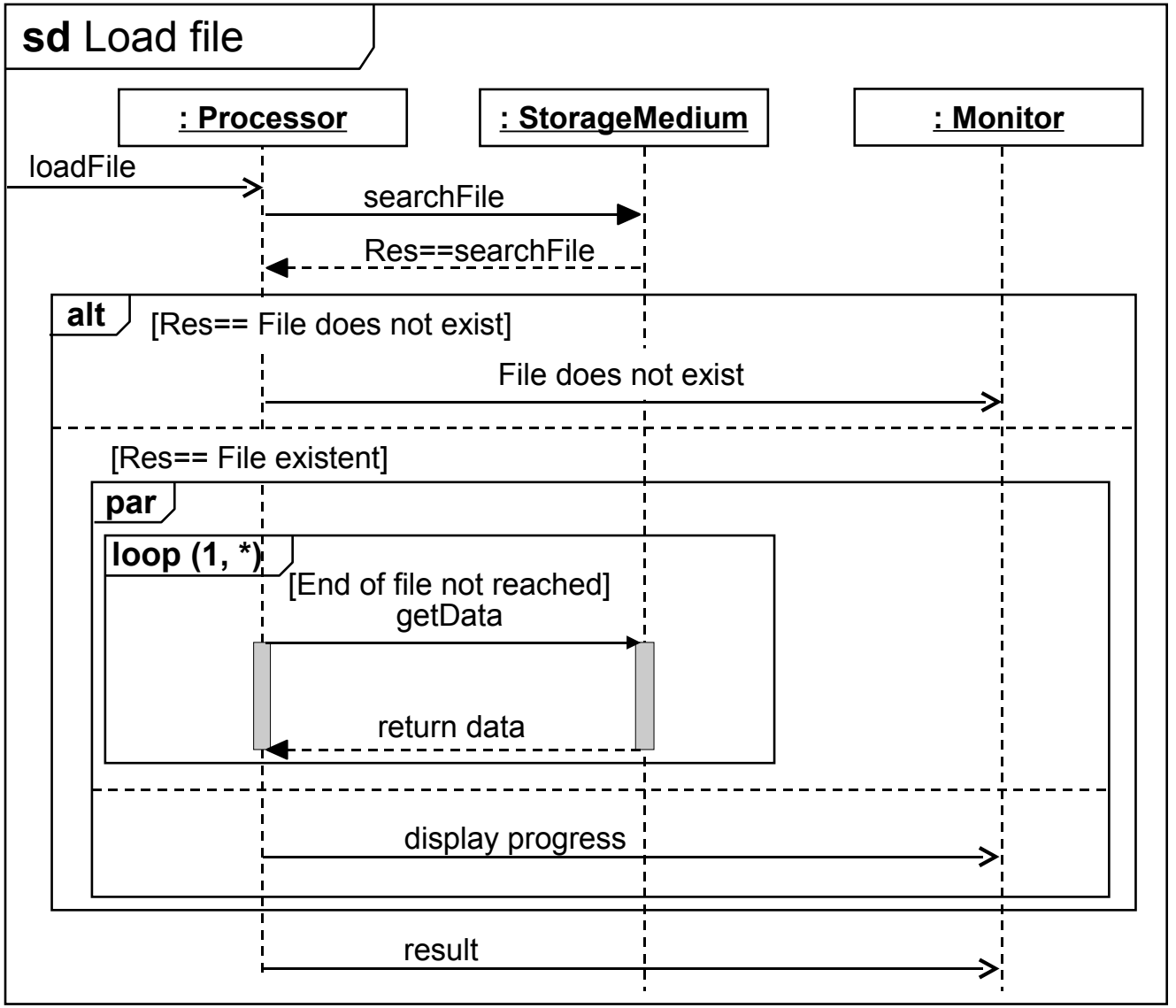
A1, B1, B2, A2, A3, B3, C1, D1, D2, C2

A1, B1, C1, D1, D2, C2, B2, A2, A3, B3

C1, D1, D2, C2 must always be handled together

Sequence Diagrams
Combined fragments (20)

Example:



Sequence Diagrams Proceeding

- When to use Sequence Diagrams
  - Object Oriented Analysis
    - to analyze information flows during the Business Process Modeling
    - to determine interactions in a Use Case
  - Object Oriented Design
    - Sequence of user interactions
    - Interaction of specific system parts
  - Implementation
    - Documentation of critical algorithms

Sequence Diagrams Proceeding

- Fragments should be applied to visualize important details
- Readability is important, so may be not all details should be expressed – depending on context
  - fundamental representation – e. g. for communication in a project team
  - detailed representation – to describe complex algorithms

Communication Diagrams

- A Communication Diagram shows a set of interactions between specified objects in a context
- In the foreground are the objects and their relations
- A Communication Diagram shows similar facts as a Sequence Diagram, but with another view
- The presentation volume is a subset of the Sequence Diagram

Communication Diagrams

- Communication diagrams typically focus on the structural organization of objects that send and receive messages

Communication Diagrams

- Notation
  - Between the objects are association lines, where messages are noted. A small arrow points from the sender to the receiver
  - Messages are written as
    `returnValue := message(arguments)`

Communication Diagrams

- Notation
  - Before a message you find a sequence expression, finalized with a colon, to model
    - the *order* of messages with unique numbers, where parallel messages get the same number but additionally characters, e. g. `1a`, `1b`
    - the send conditions could be expressed as a guard statement in square brackets, e. g.
      `2 [file loaded] print`
    - Iterations are labeled with a star `*`, in square brackets a condition of the iteration could be described, e. g.
      `3.1* [data in cache] write`

Communication Diagrams

- Example

```
reserve (o: Order)
{
  OrderItem oitem;
  Article article;
  int quantity;

  for (int i = 1; i <= o.quantity(); i++)
  {
    oitem = o.getOrderItem(i);
    article = oitem.getOrderItem();
    quantity = oitem.getQuantity();
    articleStock.reserve (article, quantity);
  }
}
```