Dr. Michael Eichberg

Software Engineering

Department of Computer Science

Technische Universität Darmstadt

Software Engineering

Modeling Dynamic Behavior

The following slides use material from:

Craig Larman; Applying UML and Patterns, 3rd Edition;

Prentice Hall

TECHNISCHE
UNIVERSITÄT
DARMSTADT

UML
Interaction Diagrams

Two types of diagrams can be distinguished:

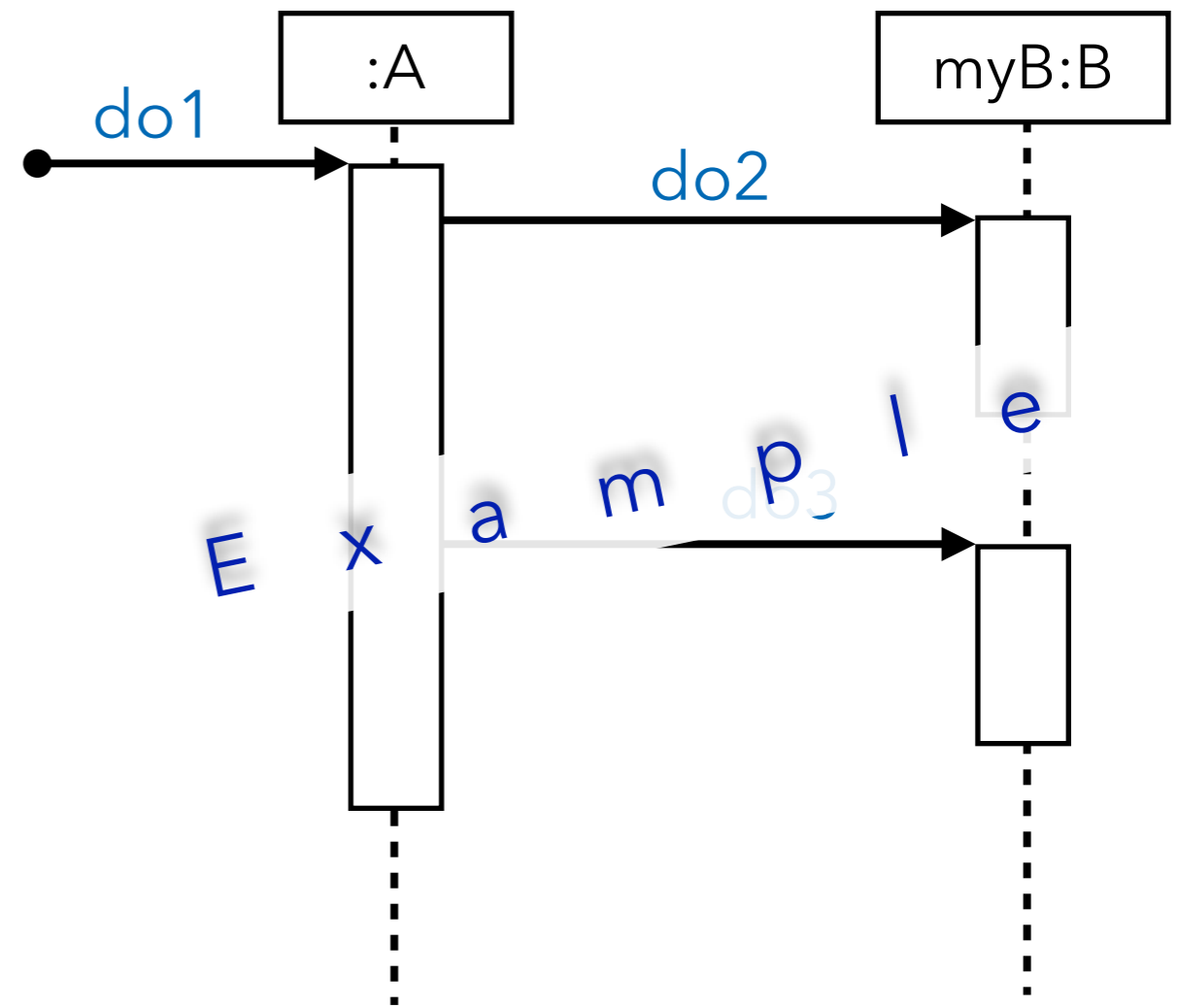- UML Sequence Diagrams
- UML Communication Diagrams

Interaction diagrams are used to **visualize the interaction via messages between objects**; they are used for *dynamic object modeling*.

Modeling the dynamic behavior is often more rewarding w.r.t. understanding the domain than modeling the static structure.
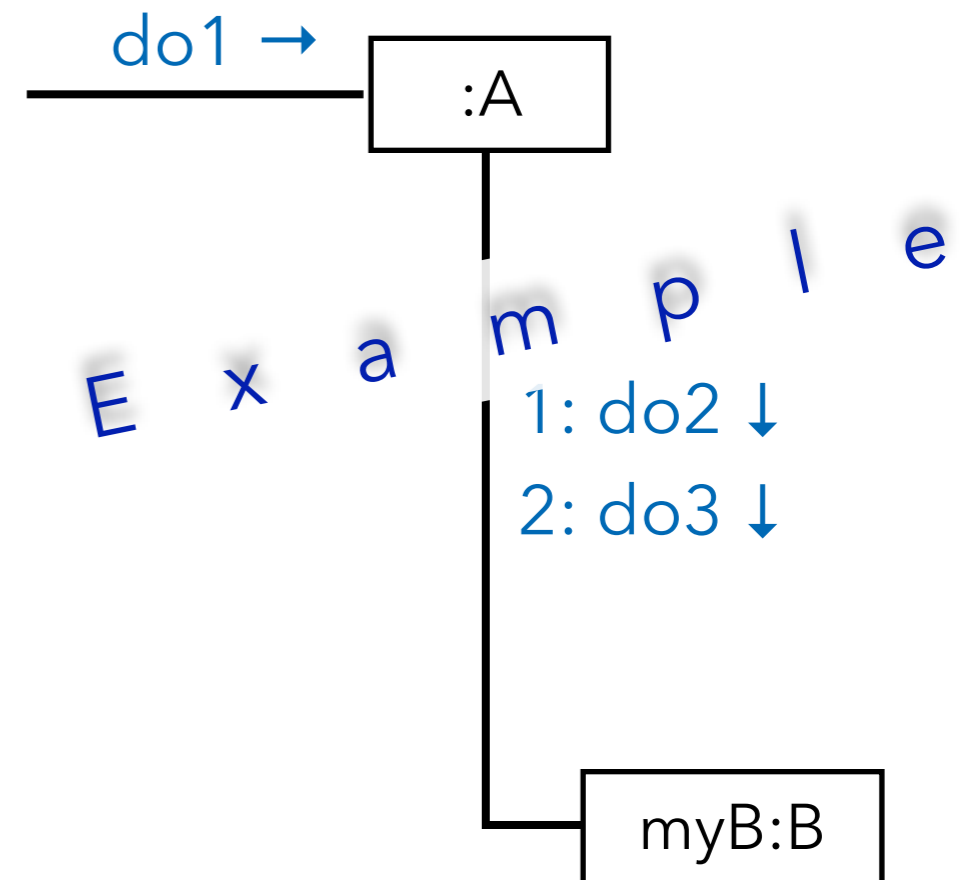
Four types of interaction diagrams are available.

- **Sequence diagrams (which use a fence format.)**
- Communication diagrams (which use a graph or network format)
- Timing diagrams (not discussed)
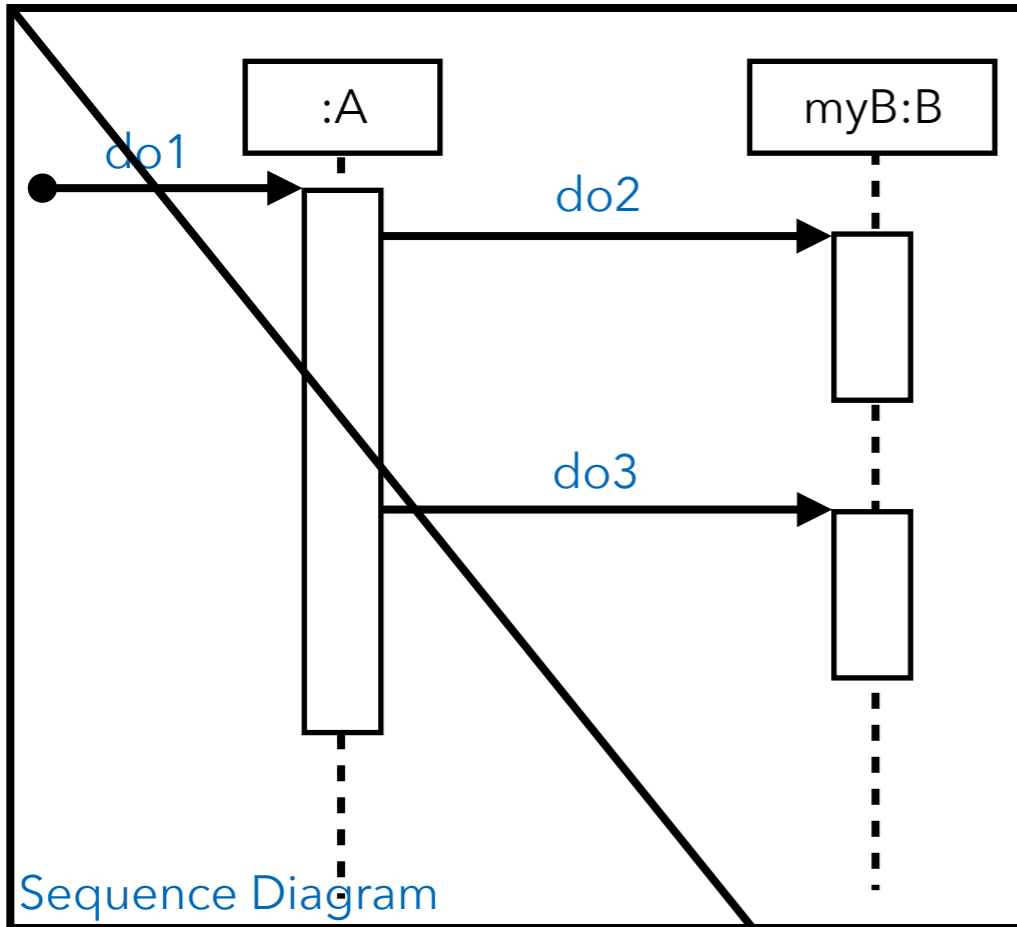- Interaction overview diagrams (not further discussed)

Four types of interaction diagrams are available.

- Sequence diagrams (which use a fence format.)

- **Communication diagrams (which use a graph or network format)**

- Timing diagrams (not further discussed)

- Interaction overview diagrams (not further discussed)

do1 →

:A

E x a m p l e

1: do2 ↓

2: do3 ↓

myB:B

Java Code for Interaction Diagrams

Sequence Diagram

E x a m p l e

```java
public class A {
    private B myB = ...;

    public void do1() {
        myB.do2();
        myB.do3();
    }
}
```

Java Code for Interaction Diagrams

E x a m p l e

do1 → :A

1: do2 ↓
2: do3 ↓

myB:B

Communication Diagram
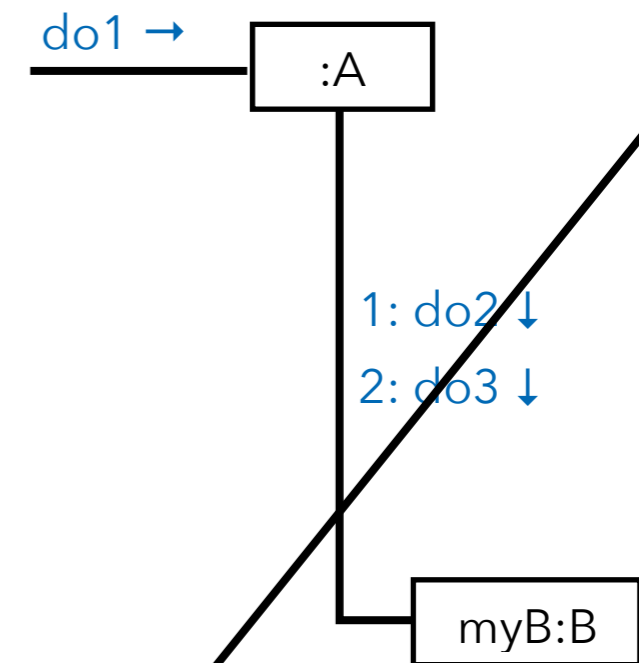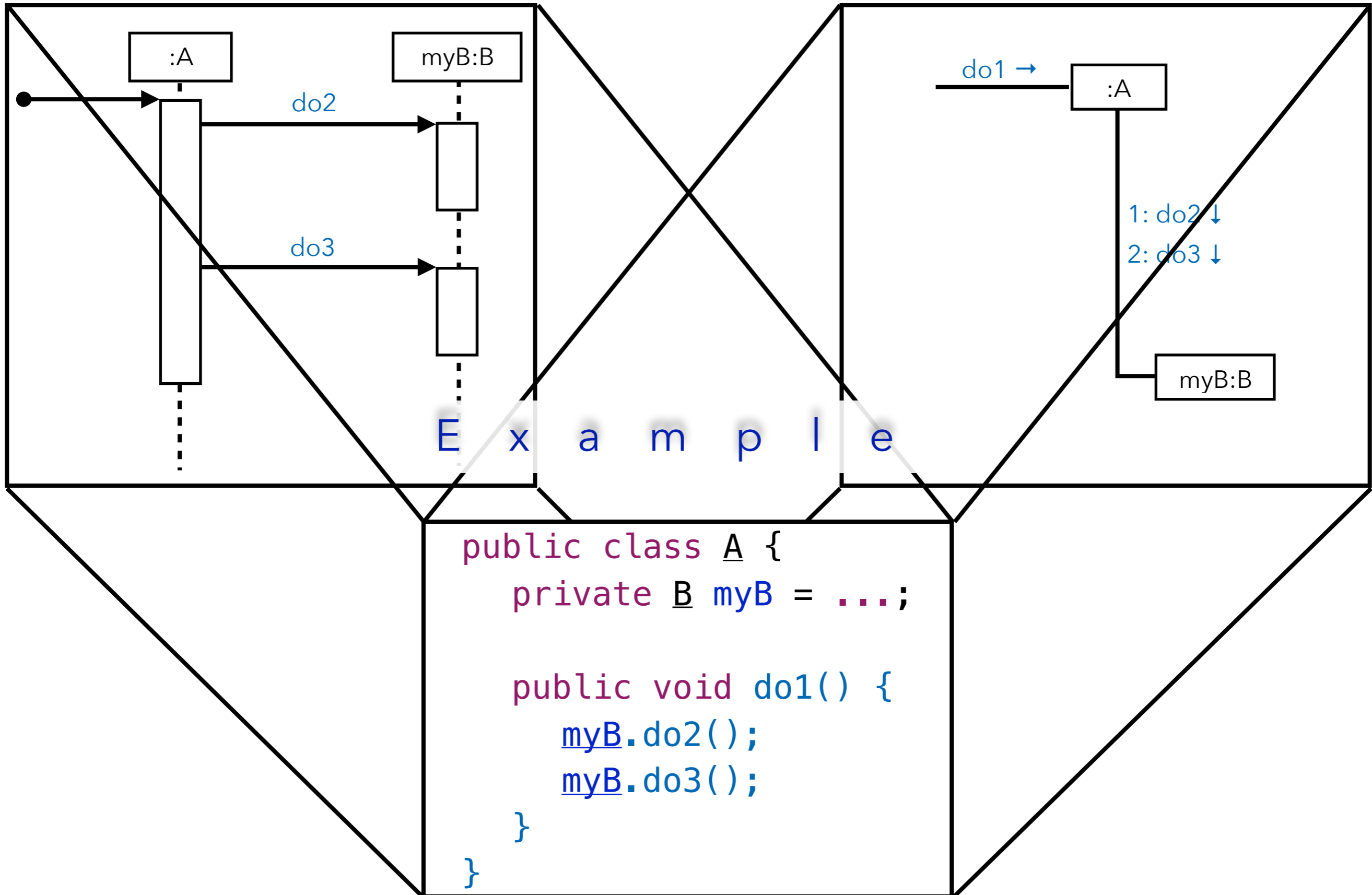
```java
public class A {
    private B myB = ...;

    public void do1() {
        myB.do2();
        myB.do3();
    }
}
```

Java Code for Interaction Diagrams

E x a m p l e

```java
public class A {
    private B myB = ...;

    public void do1() {
        myB.do2();
        myB.do3();
    }
}
```

:A   myB:B

do2

do3

do1 →   :A

1: do2 ↓
2: do3 ↓

myB:B

Common Notations for UML Interaction Diagrams

:Sale

Lifeline box representing an unnamed instance of class Sale.

Common Notations for UML Interaction Diagrams

s1:Sale

Java Code:

Sale s1 = …;

Lifeline box representing a named instance (s1) of Sale.

Common Notations for UML Interaction Diagrams

«metaclass»
Font

Java Code:

```
Class<Font> fontClass = Font.class;
```

Lifeline box representing the class Font, or more precisely, that Font is an instance of class Class - an instance of a metaclass.

Common Notations for UML Interaction Diagrams

sales:ArrayList<Sale>

```
Java Code:
ArrayList<Sale> sales = …;
```

Lifeline box representing an instance of an ArrayList class, parameterized to hold Sale objects.

Common Notations for UML Interaction Diagrams

```
sales[i]:Sale
```

Java Code:

```java
ArrayList<Sale> sales = …;
Sale sale = sales.get(i);
```

Lifeline box representing one instance of class Sale, selected from the sales ArrayList<Sale> collection.

Common Notations for UML Interaction Diagrams - Format for Interaction Messages

- "Commonly" Used Grammar:
  return = message(parameter:parameterType):returnType

- Parentheses are usually excluded if there are no parameters.

- Type information may be excluded if unimportant.

```
initialize(code)
initialize                    E  x  a  m  p  l  e  s
d = getProductDescription (id)
d = getProductDescription (id : ItemId)
d = getProductDescription (id : ItemId) : ProductDescription
```

The same syntax is, e.g., used by the Scala programming language.

UML
Sequence Diagrams

Modeling (Synchronous) Messages

a found message whose sender will not be specified

typical synchronous message shown with a filled-arrow line

:Register

:Sale

report

getDate

execution specification bar indicates focus of control (optional)

dt. Ausführungssequenz

toString

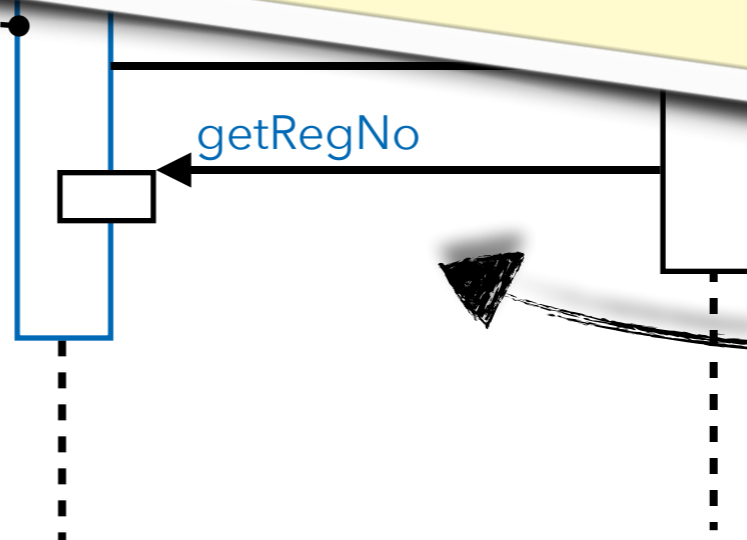getRegNo

Modeling (Synchronous) Messages
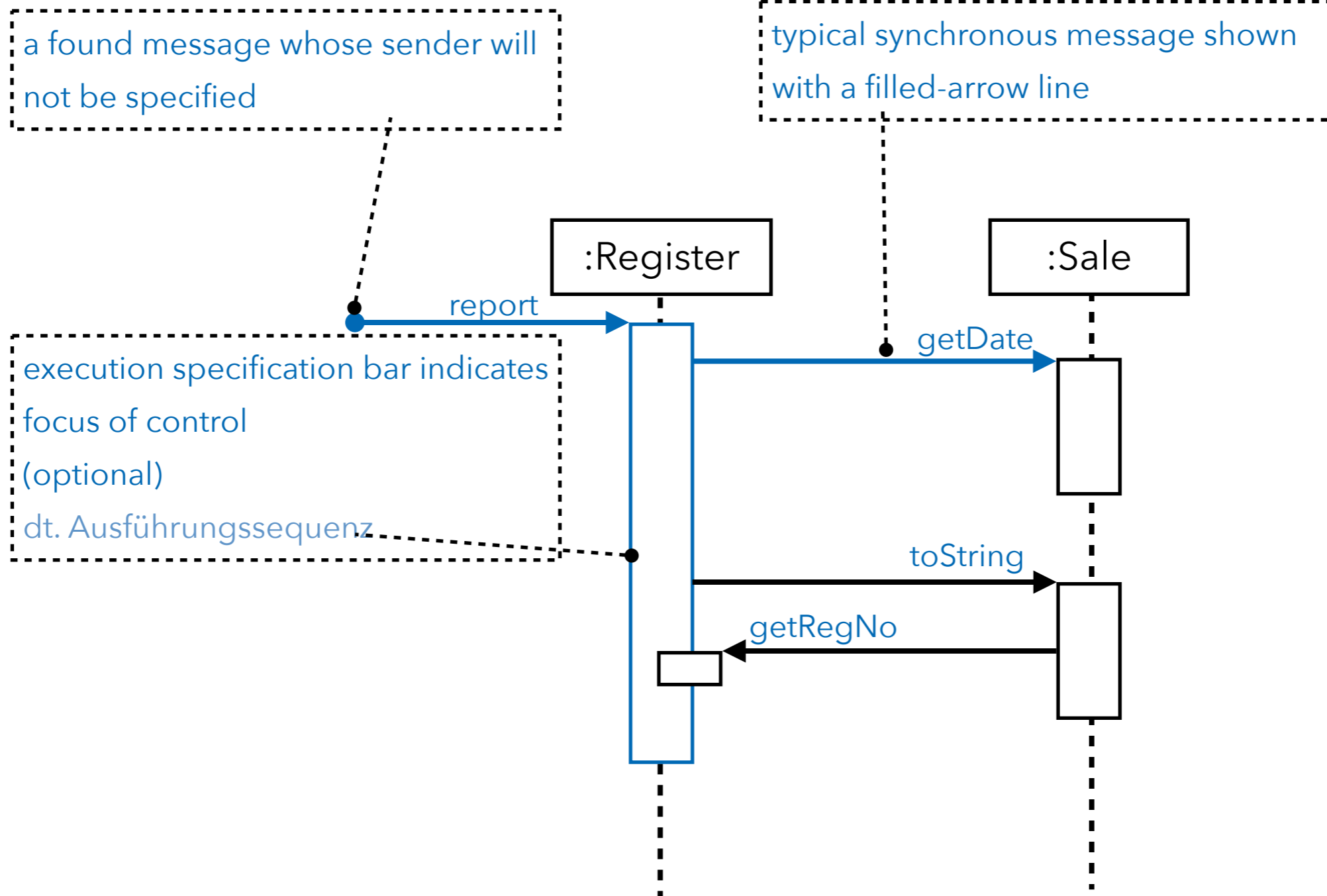
a found message wh[...]
not be specified

execution specifi[...]
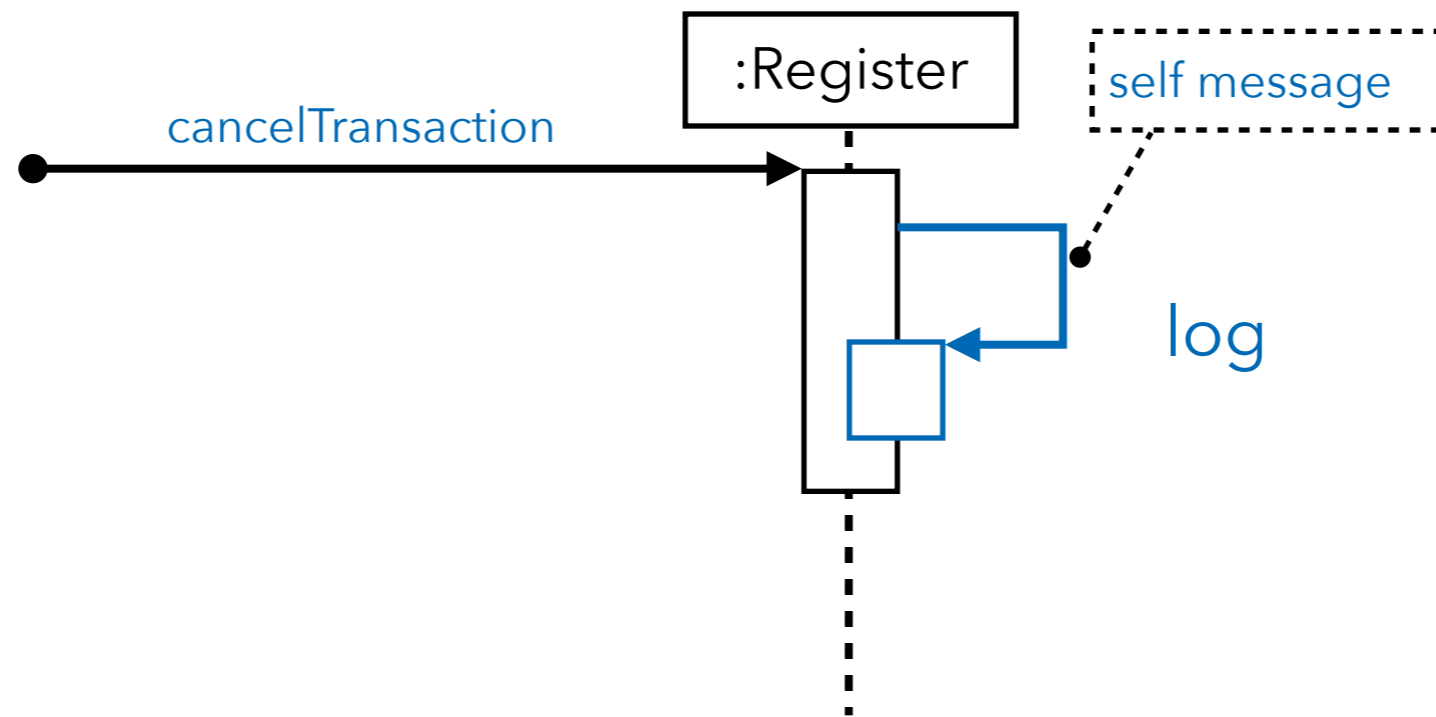focus of control
(optional)
dt. Ausführungssequenz

UML Superstructure

If the Message represents a CallAction, there will normally be a reply message from the called Lifeline back to the calling lifeline before the calling lifeline will proceed.

getRegNo

Self messages can be modeled using nested execution specification bars.

To show the return value of a message you can either use the message syntax (A) or use a message line at the end of an execution specification bar (B).

execution specification bar =dt. Ausführungssequenz
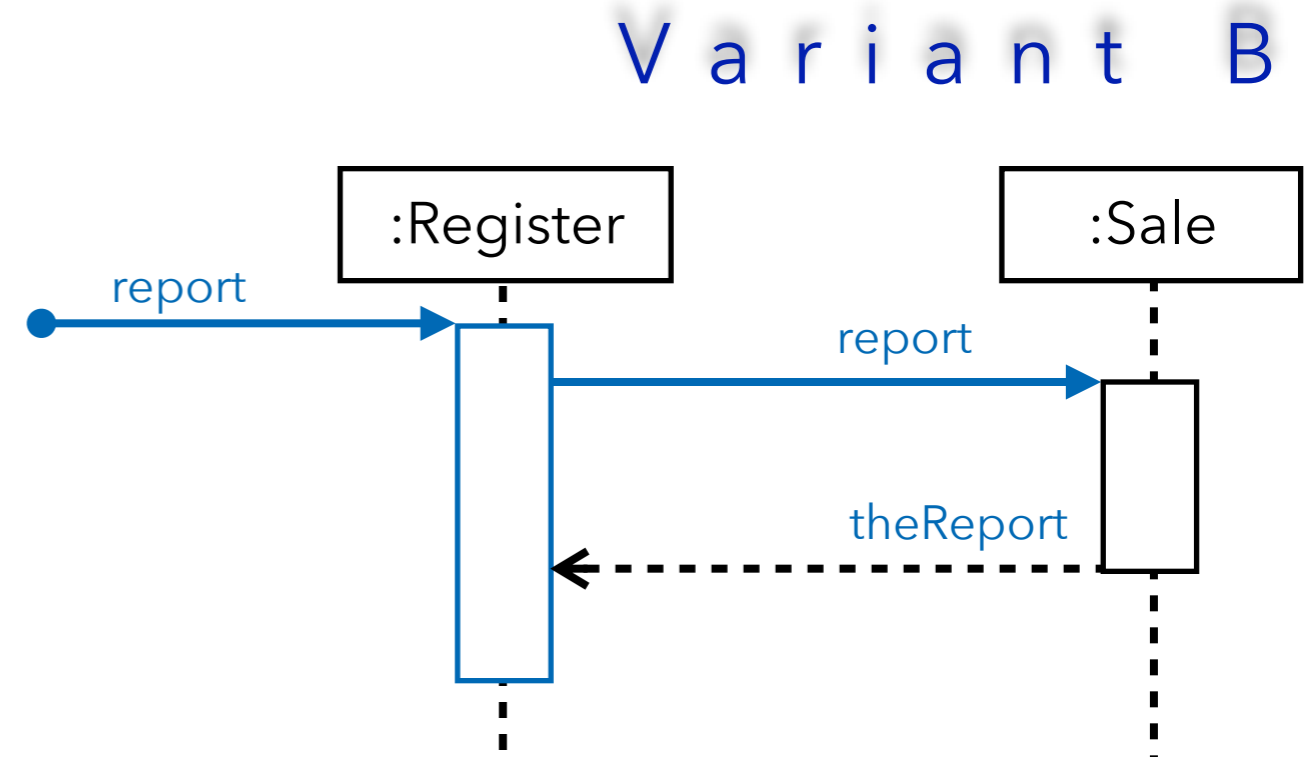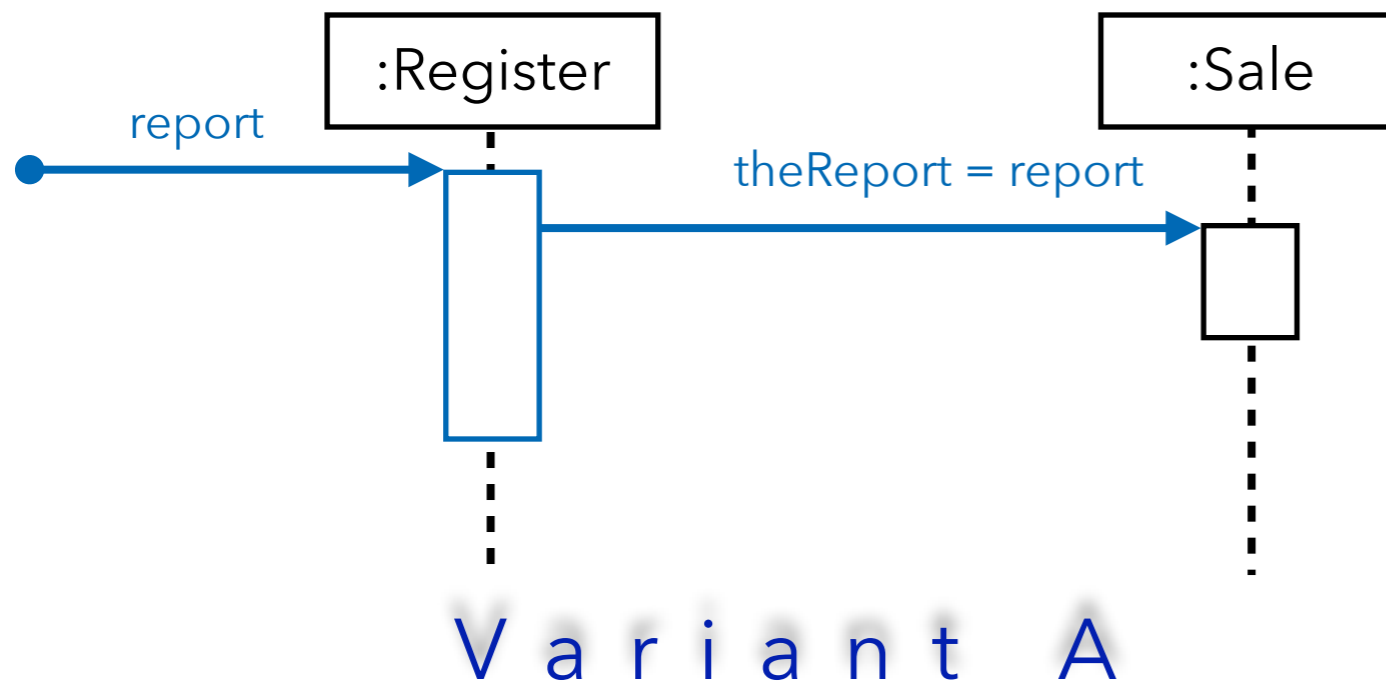
Object Instance Creation

:Register

:Sale

Newly created objects are placed at their creation "height".

makePayment(cashTendered)

create(cashTendered)

:Payment

authorize
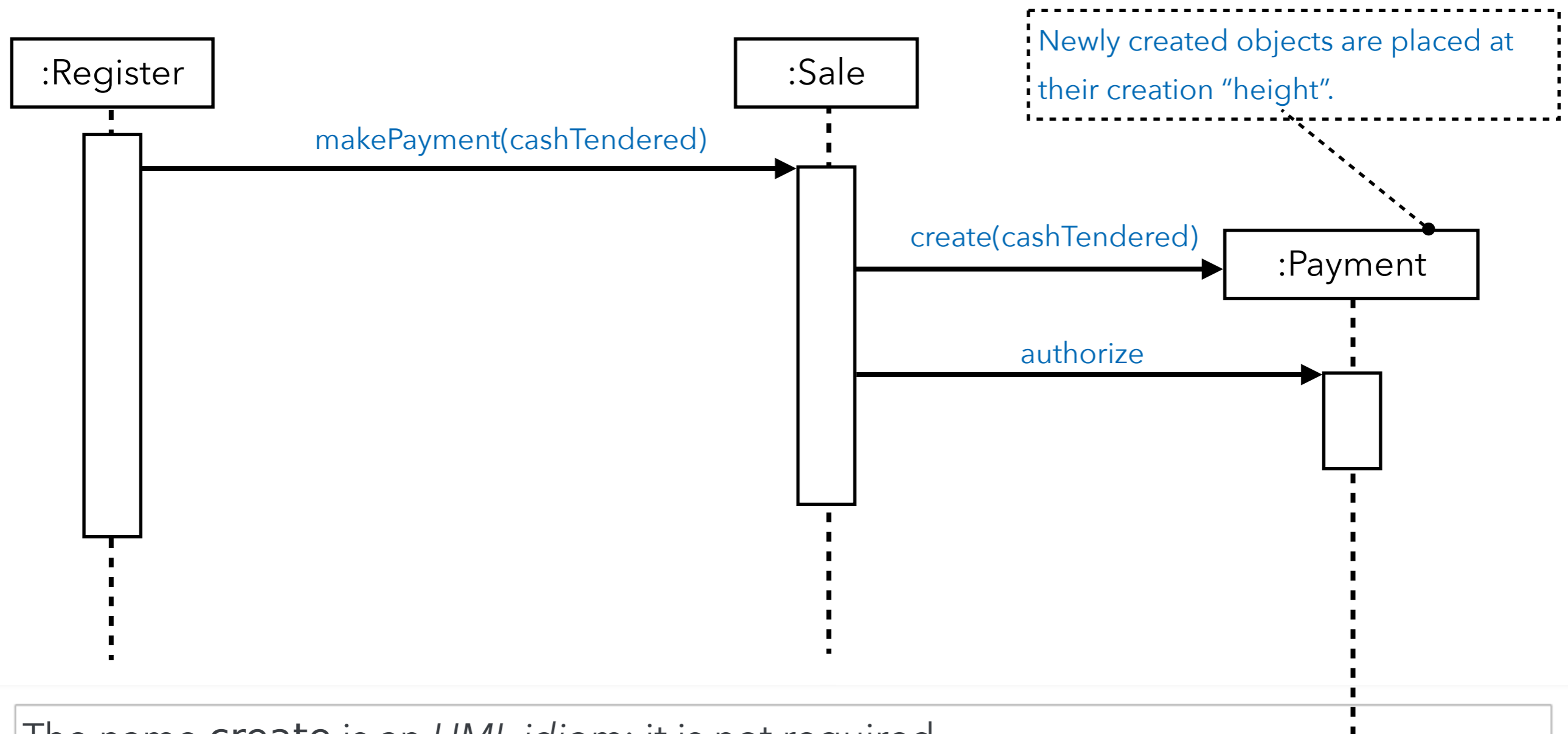
The name **create** is an *UML idiom*; it is not required.

Object Instance Destruction

:Register

:Sale

makePayment(cashTendered)

create(cashTendered)

:Payment

...

X

Not strictly required by the UML.
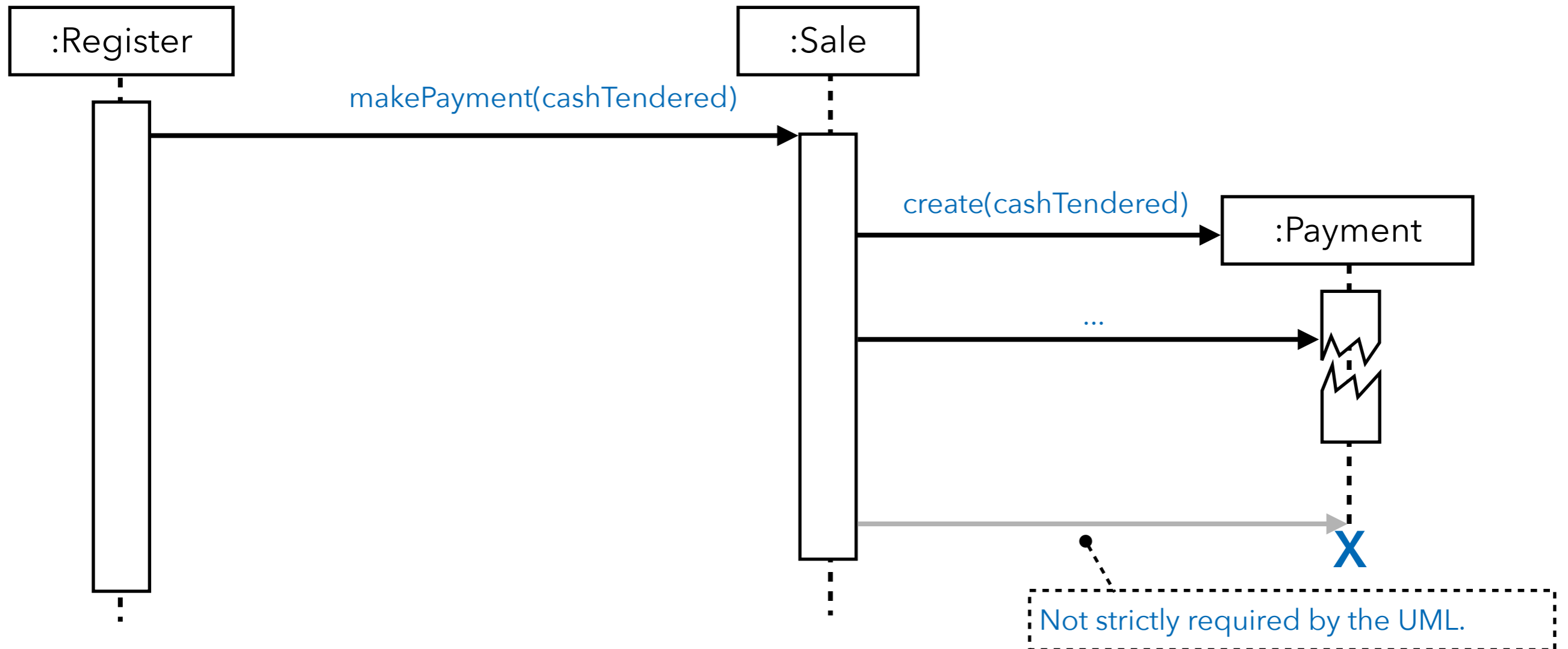
The object destruction notation is also used to mark objects that are no longer usable.

Invoking Static Methods (Class Methods)

report

:Register

locales = getAvailableLocales
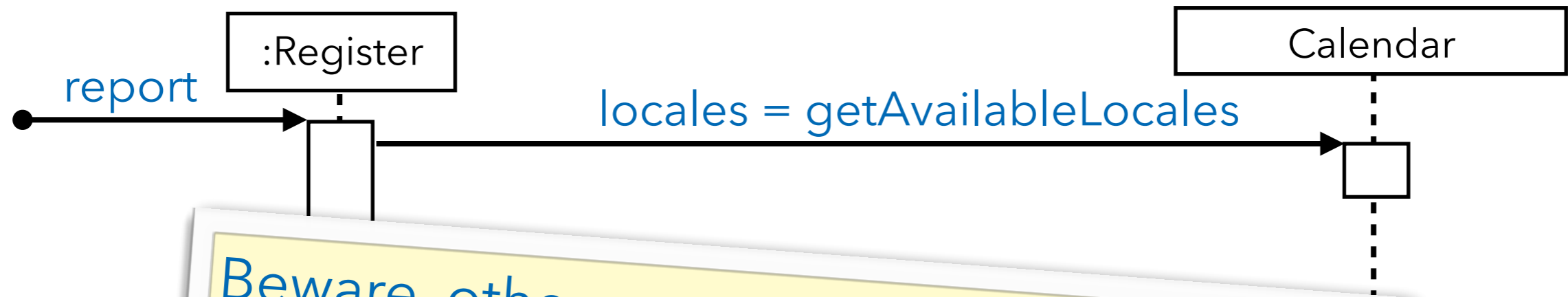
Calendar

Beware, other notations are also used (e.g. underlined method names).

Invoking Static Methods (Class Methods)
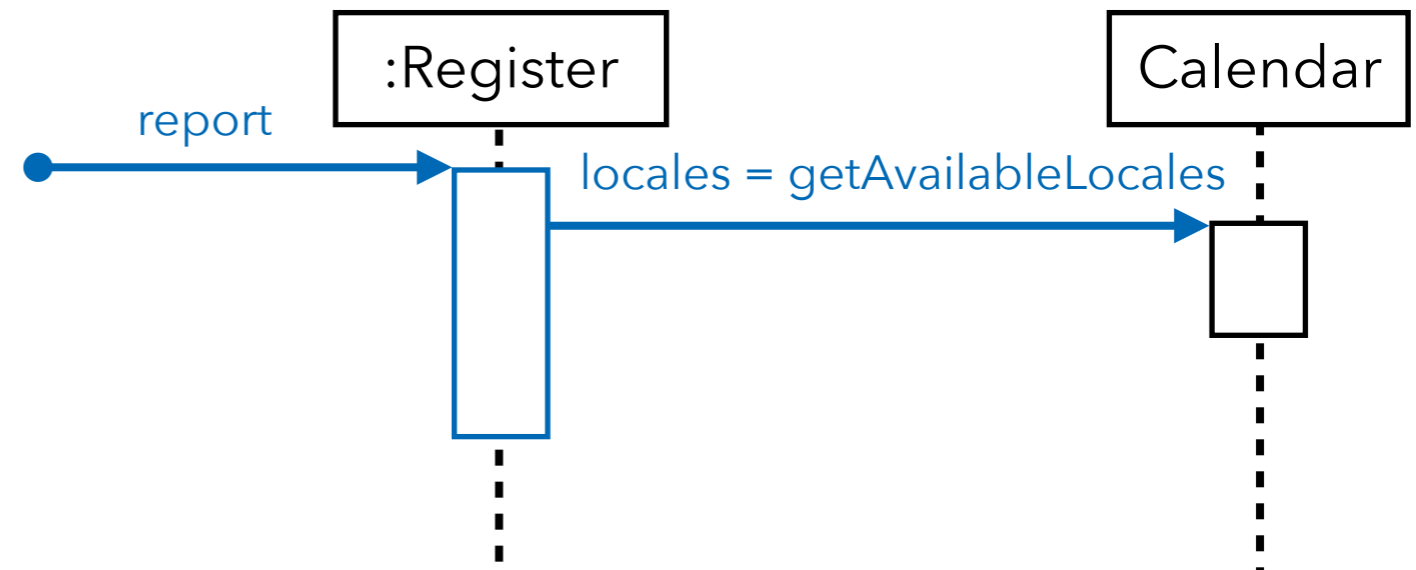
```java
public class Register {
   public void report() {
      Locale[] locales = Calendar.getAvailableLocales();
   }
}
```

Corresponding Java Code

Diagram **frames** in UML sequence diagrams are used to support - among others - conditional and looping constructs.

Diagram Frame ~dt. Fragment

```
              ┌──────────┐          ┌──────────────────┐
              │  :Sale   │          │  lineItems[i] :  │
              │          │          │   SalesLineItem  │
              └──────────┘          └──────────────────┘
                   ┆                         ┆
        t = getTotal                         ┆
   ●─────────────────▶┆                      ┆
   ┌─────────────────────────────────────────────────────┐
   │ loop ╲ [0 < i < lineItems.size]                      │
   │       ╲                                              │
   │                                                      │
   │                   st = getSubTotal                   │
   │              ┆────────────────────────▶┆             │
   │         ┌──────┐                                     │
   │         │ i++  │                                     │
   │         └──────┘                                     │
   └─────────────────────────────────────────────────────┘
```
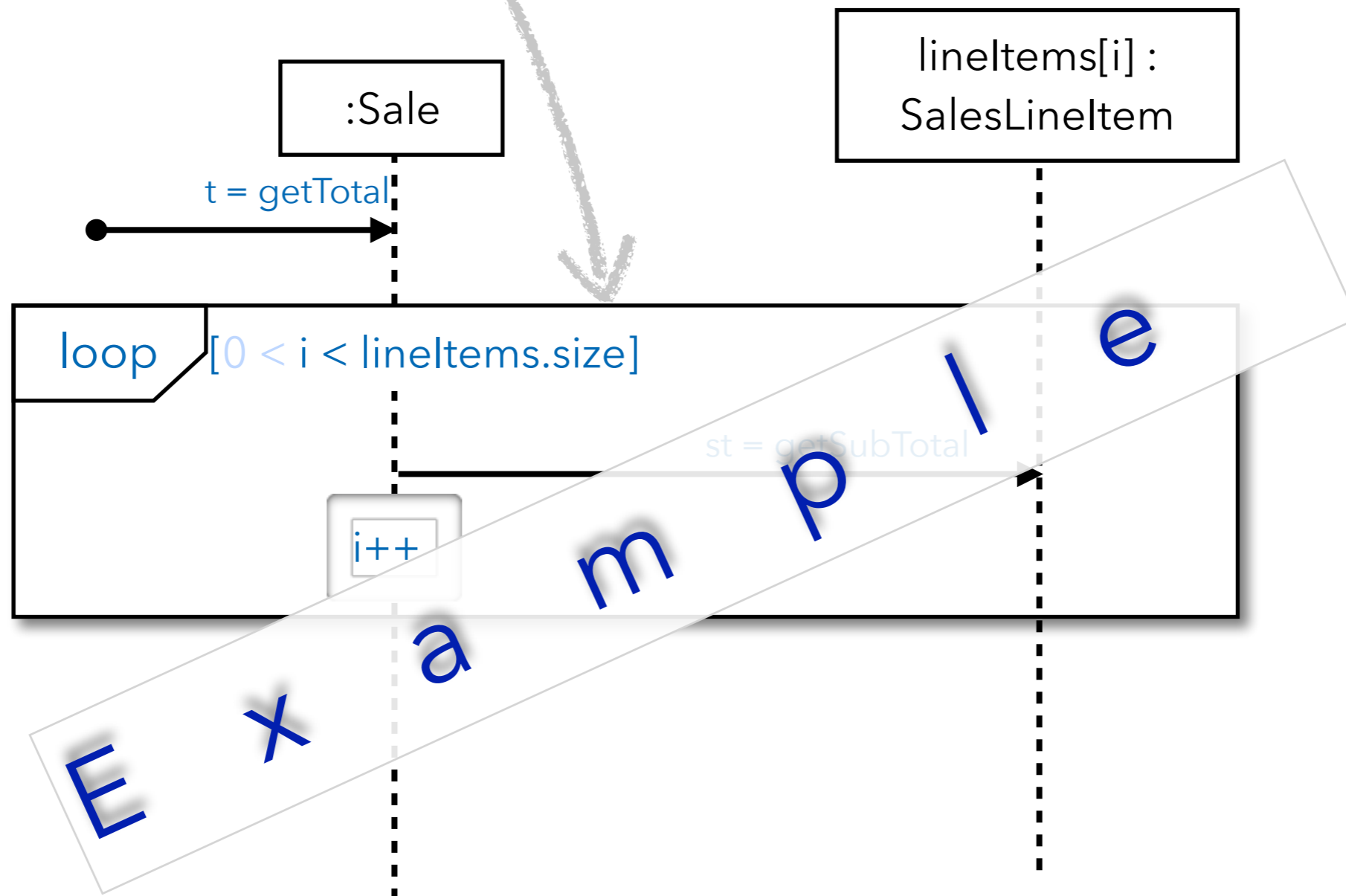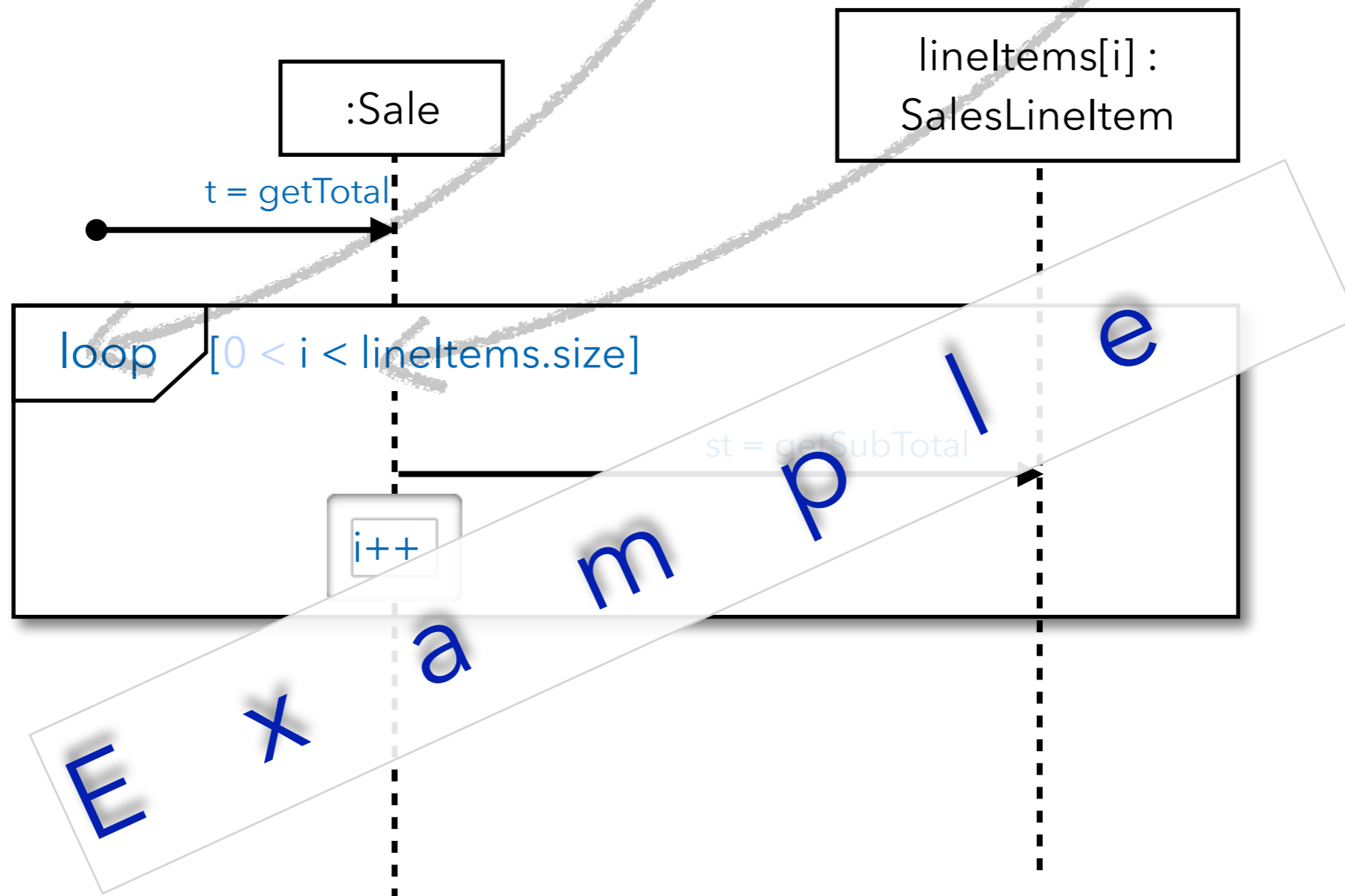
E x a m p l e

Diagram frames in UML sequence diagrams are used to support - among others - conditional and looping constructs. Frames have an operator and a guard.

Diagram Frame ~dt. Fragment

:Sale

lineItems[i] :
SalesLineItem

t = getTotal

loop   [0 < i < lineItems.size]

st = getSubTotal

i++

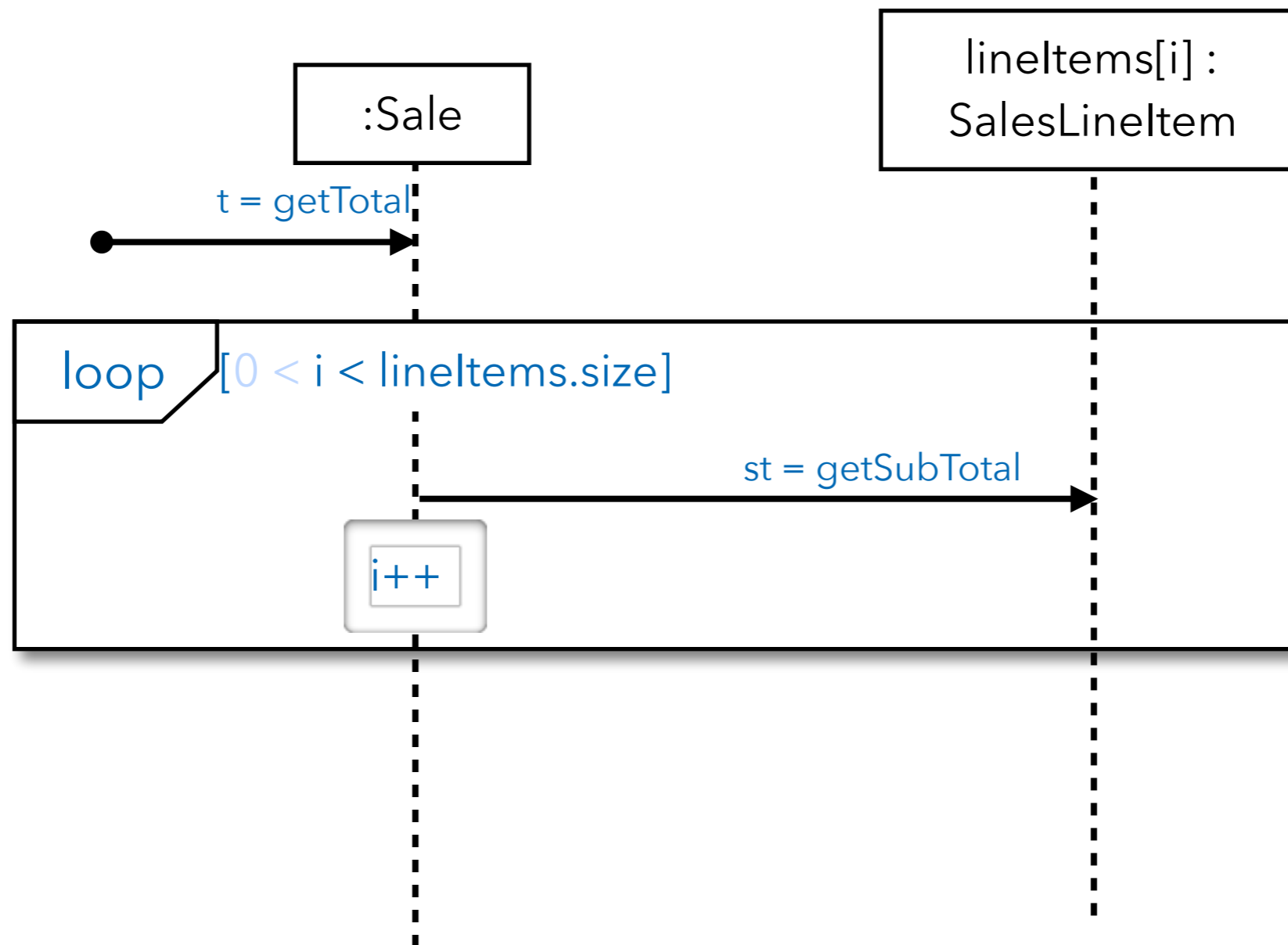Example

How to model the iteration over a collection?

```
Modeling task: Calculate the total of a sale by summing up the
sub totals for each sales line item.
```

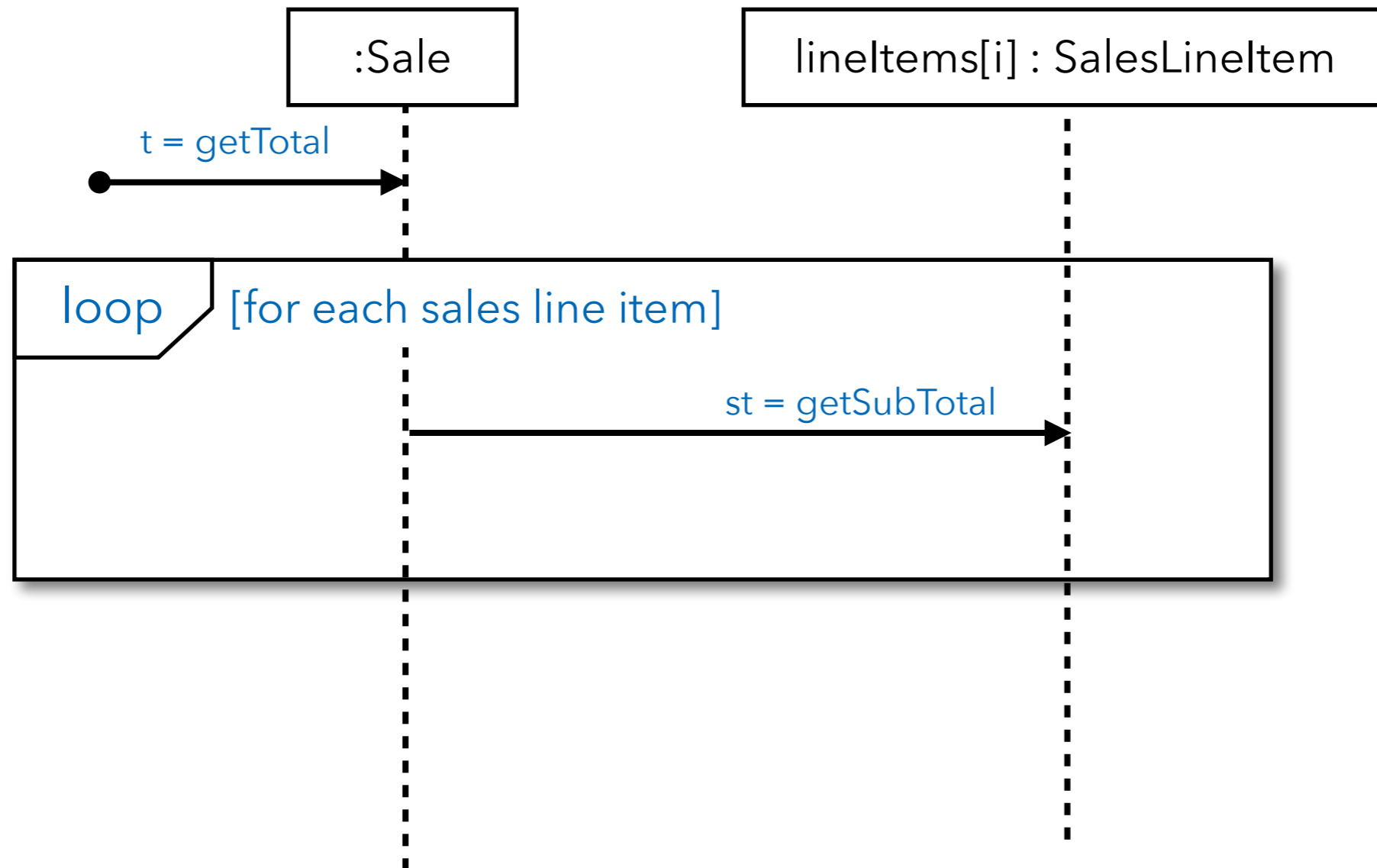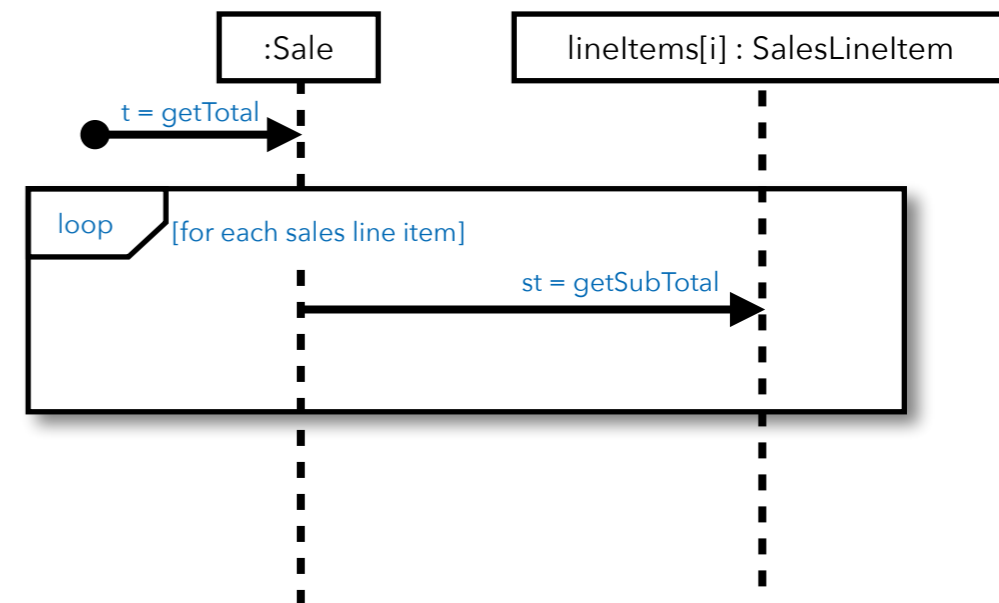# Use a **UML loop frame** to iterate over a collection.

Modeling task: Calculate the total of a sale by summing up the
sub totals for each sales line item.

Use a **UML loop frame** to iterate over a collection.

```
                        ┌──────────────┐          ┌──────────────────────────┐
                        │    :Sale     │          │ lineItems[i] : SalesLineItem │
                        └──────────────┘          └──────────────────────────┘
              t = getTotal
         ●────────────────────▶

    ┌─────────┐
    │ loop    │ [for each sales line item]
    └─────────┘
                                    st = getSubTotal
                     ──────────────────────────────────▶
```

Modeling task: Calculate the total of a sale by summing up the
sub totals for each sales line item.

Java code corresponding to a **UML loop frame**.

```java
public class Sale {

    private List<SalesLineItem> lineItems
    = new ArrayList<>();

    public Money getTotal() {
        Money t = new Money();
        Money st = null;
        for (SalesLineItem lineItem : lineItems) {
            st = lineItem.getSubtotal();
            t.add(st);
        }
        return t;
    }

}
```
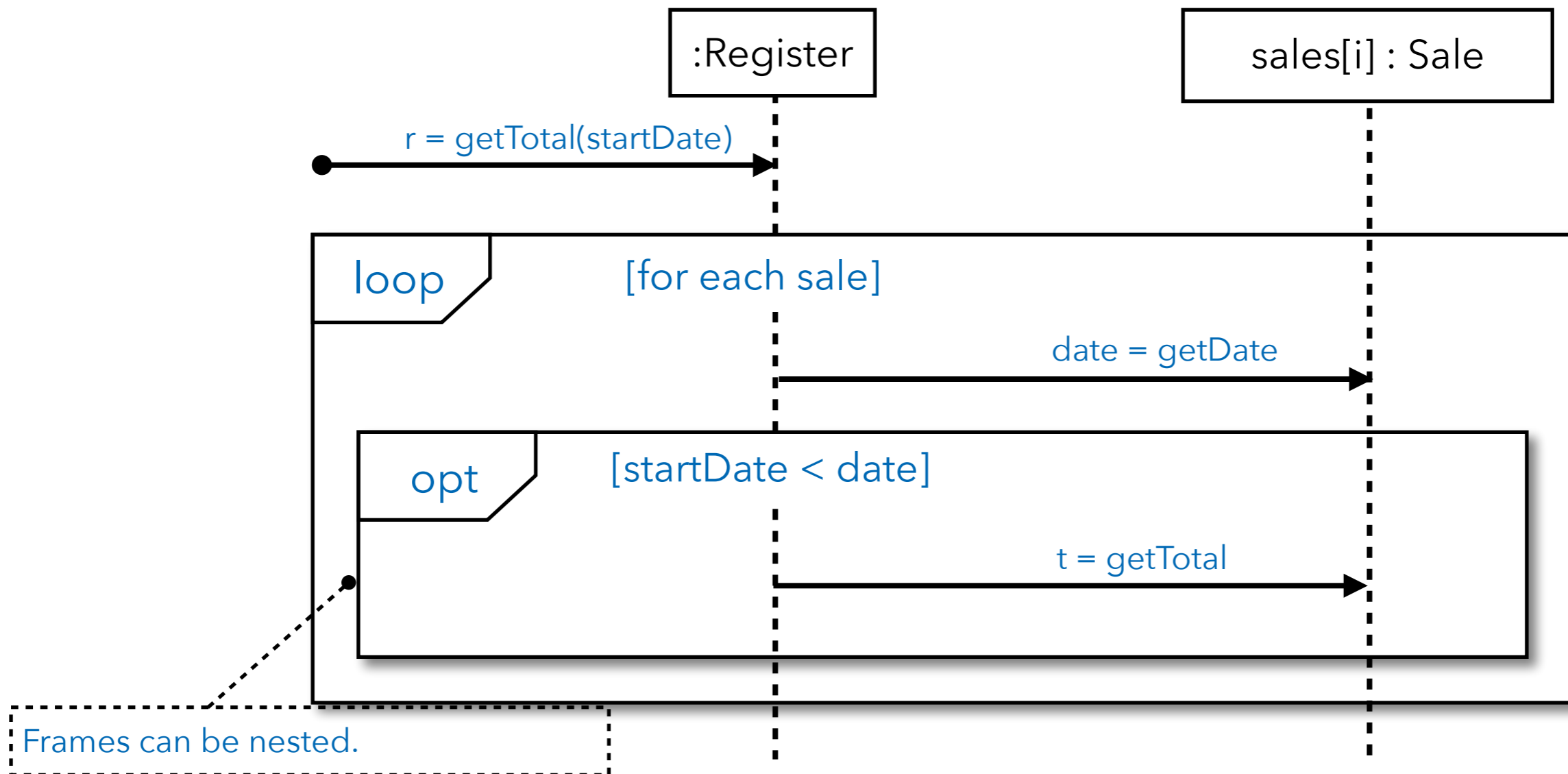
Modeling task: Calculate the total of a sale by summing up the sub totals for each sales line item.

How to model the sending of a message only if a guard condition matches?

```
Modeling task: Get the sum of all sales that happened today
after 18:00 o'clock.
```

Use a **UML opt frame** to model the sending of a message if the guard condition matches.

:Register

sales[i] : Sale

r = getTotal(startDate)

loop [for each sale]

date = getDate

opt [startDate < date]
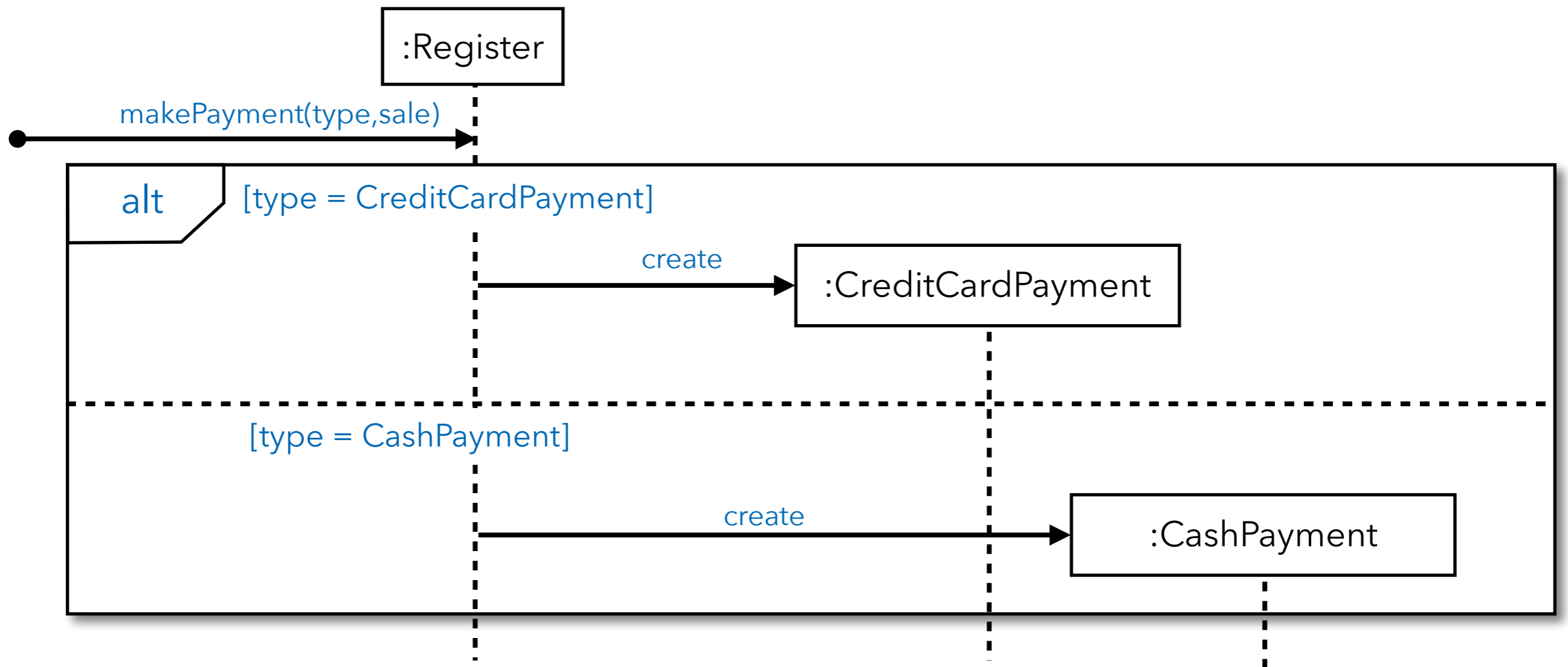
t = getTotal

Frames can be nested.

Modeling task: Get the sum of all sales that happened today after 18:00 o'clock.

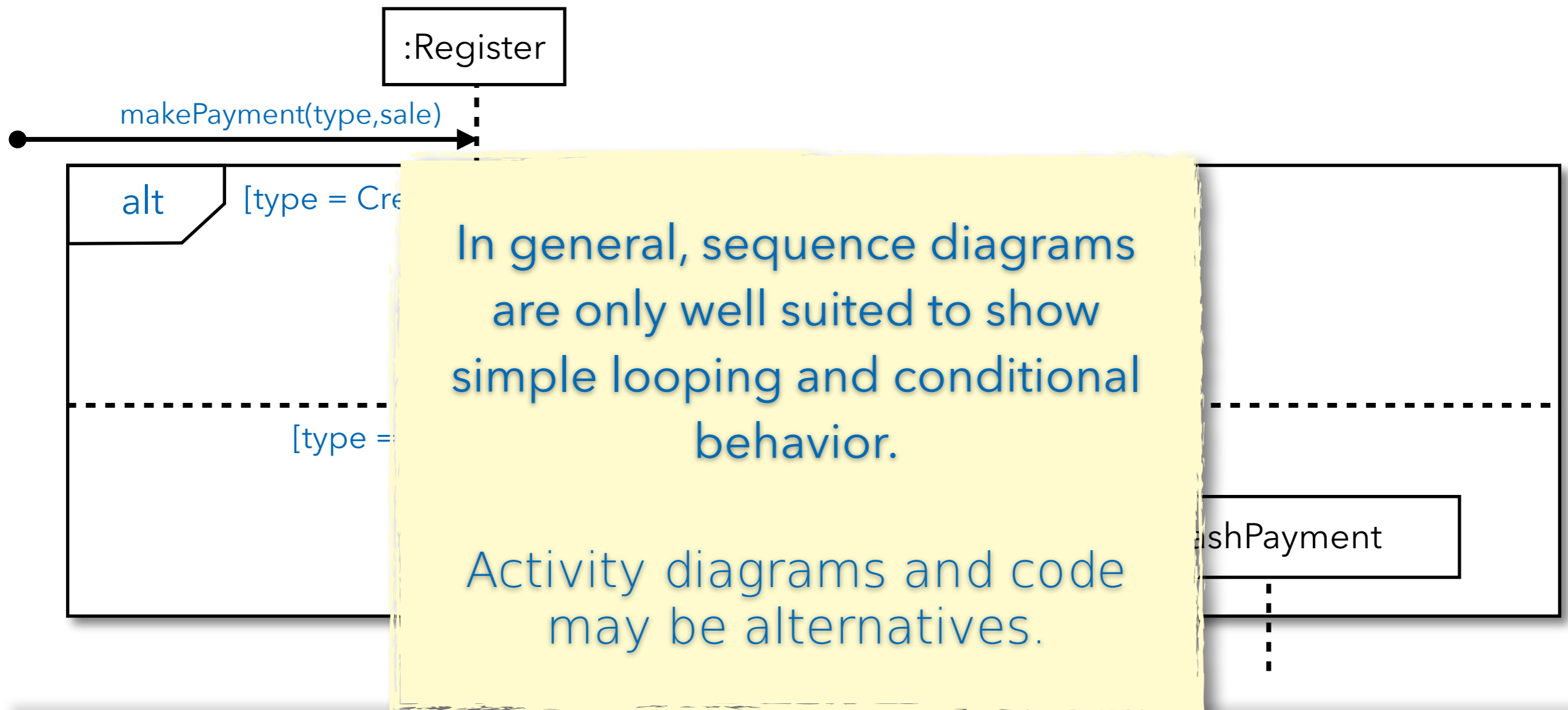# How to model mutually exclusive alternatives?

```
Modeling task: A register should be able to handle credit card
payments and cash payments.
```

Use the **UML alt frame** to model between 2 and n mutually exclusive alternatives.

Modeling task: A register should be able to handle credit card payments and cash payments.

Diagram frames in UML sequence diagrams are used to support - among others - conditional and looping constructs.

:Register

makePayment(type,sale)

alt [type = Cre...

[type = ...

In general, sequence diagrams are only well suited to show simple looping and conditional behavior.

Activity diagrams and code may be alternatives.

...ashPayment

Modeling task: A register should be able to handle credit card payments and cash payments.
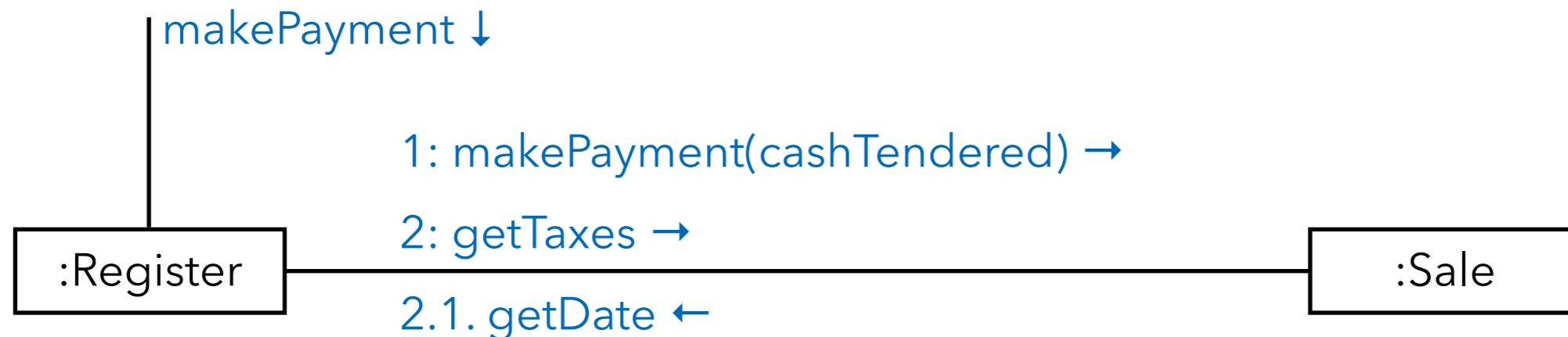
UML Communication Diagrams

Links and Messages in Communication Diagrams

- A <u>link</u> is a connection path between two objects (it is an instance of an association)

  A link indicates that some form of navigation and visibility between the objects is possible.
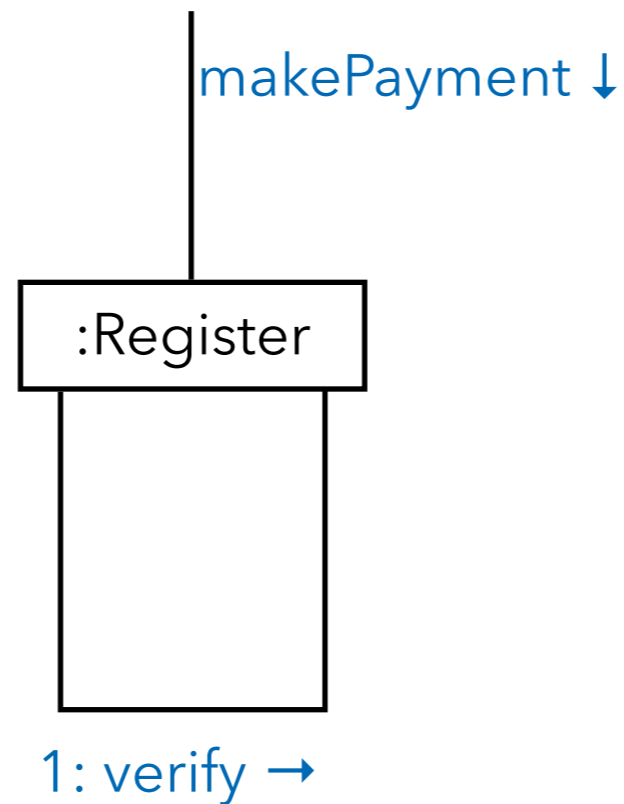
- Each **message** between objects is represented with a message expression and a small arrow indicating the direction of the message

  Sequence numbers are added to show the sequential order of messages in the current thread of control; the starting message is often not numbered.
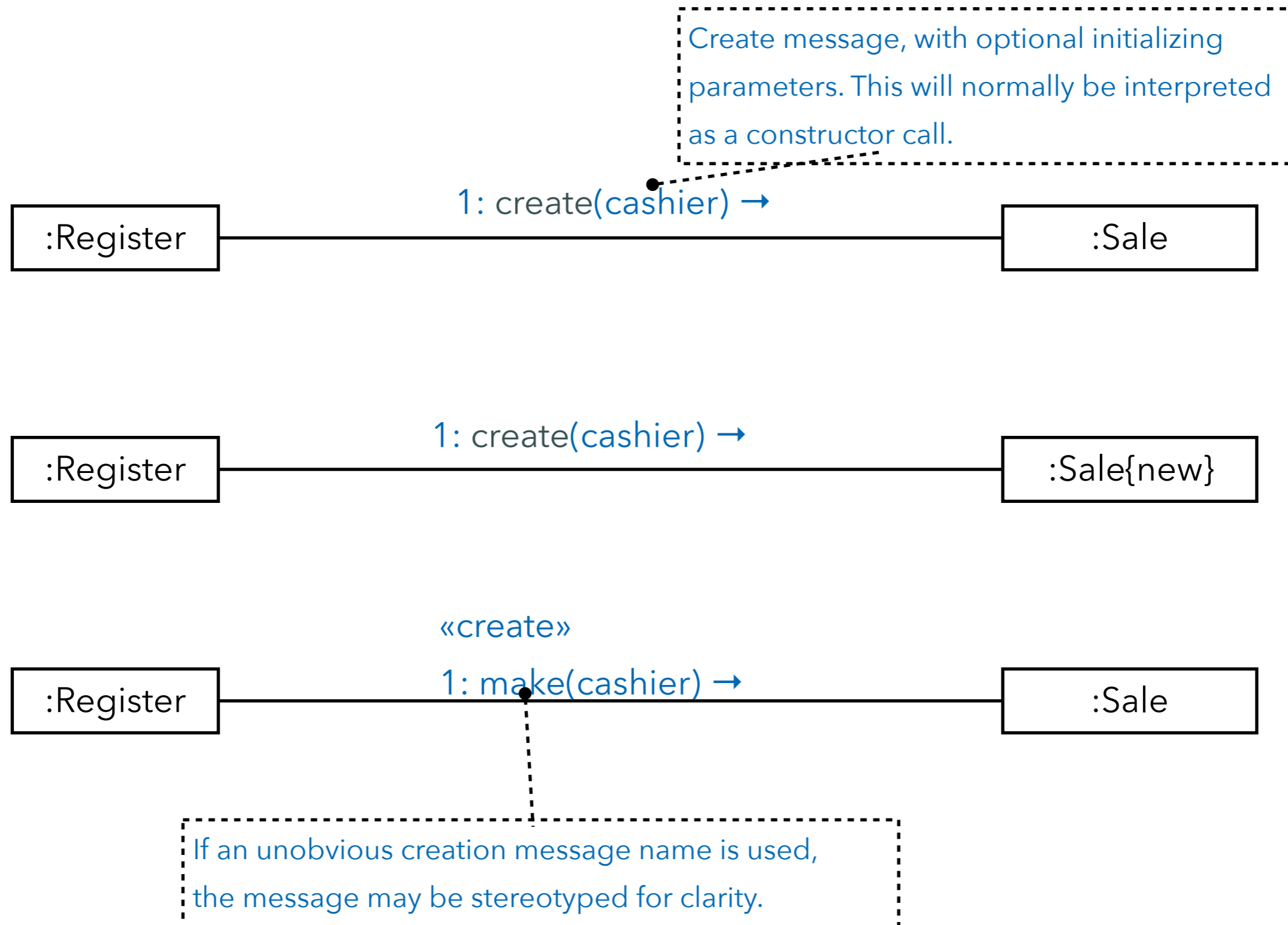
makePayment ↓

1: makePayment(cashTendered) →

2: getTaxes →

[:Register]————————————————[:Sale]

2.1. getDate ←

Links and Messages in Communication Diagrams

- Modeling self messages

Alternative Notations for Modeling Instance Creation

> Create message, with optional initializing parameters. This will normally be interpreted as a constructor call.

| :Register | —— 1: create(cashier) → —— | :Sale |

| :Register | —— 1: create(cashier) → —— | :Sale{new} |

«create»

| :Register | —— 1: make(cashier) → —— | :Sale |

> If an unobvious creation message name is used, the message may be stereotyped for clarity.
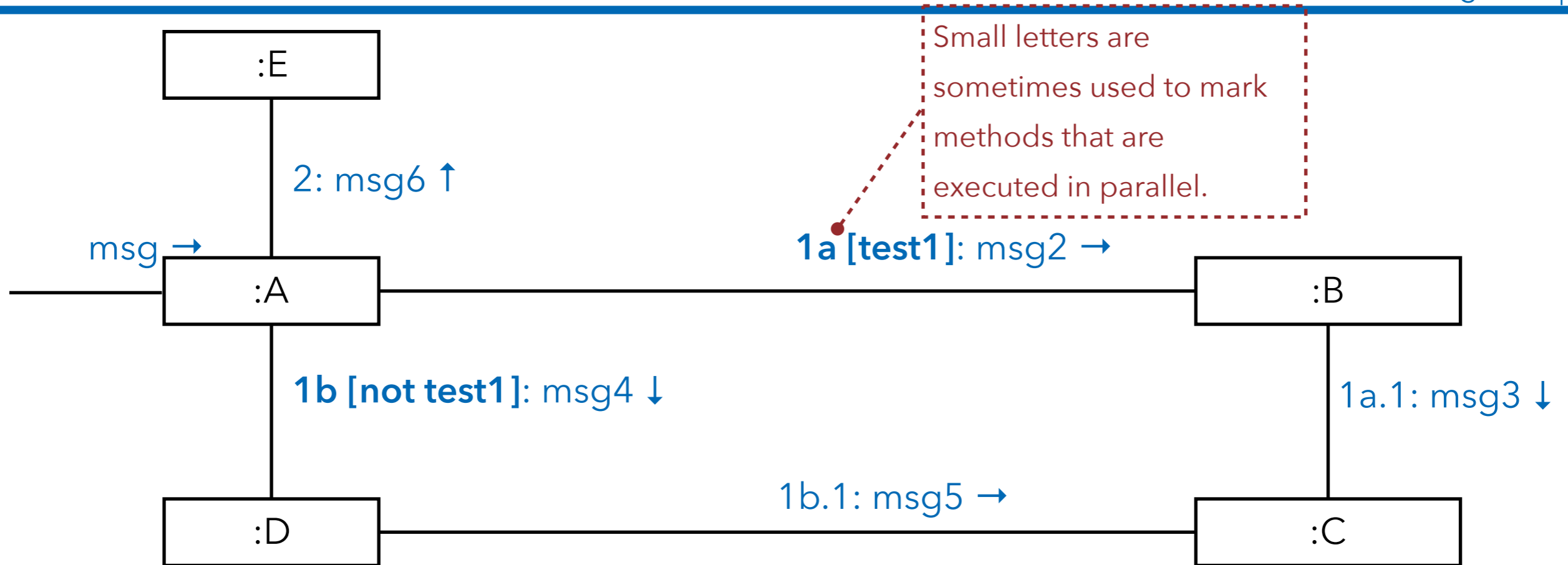
Message Number Sequencing

The initial message ist not numbered to make the numbering easier to comprehend.
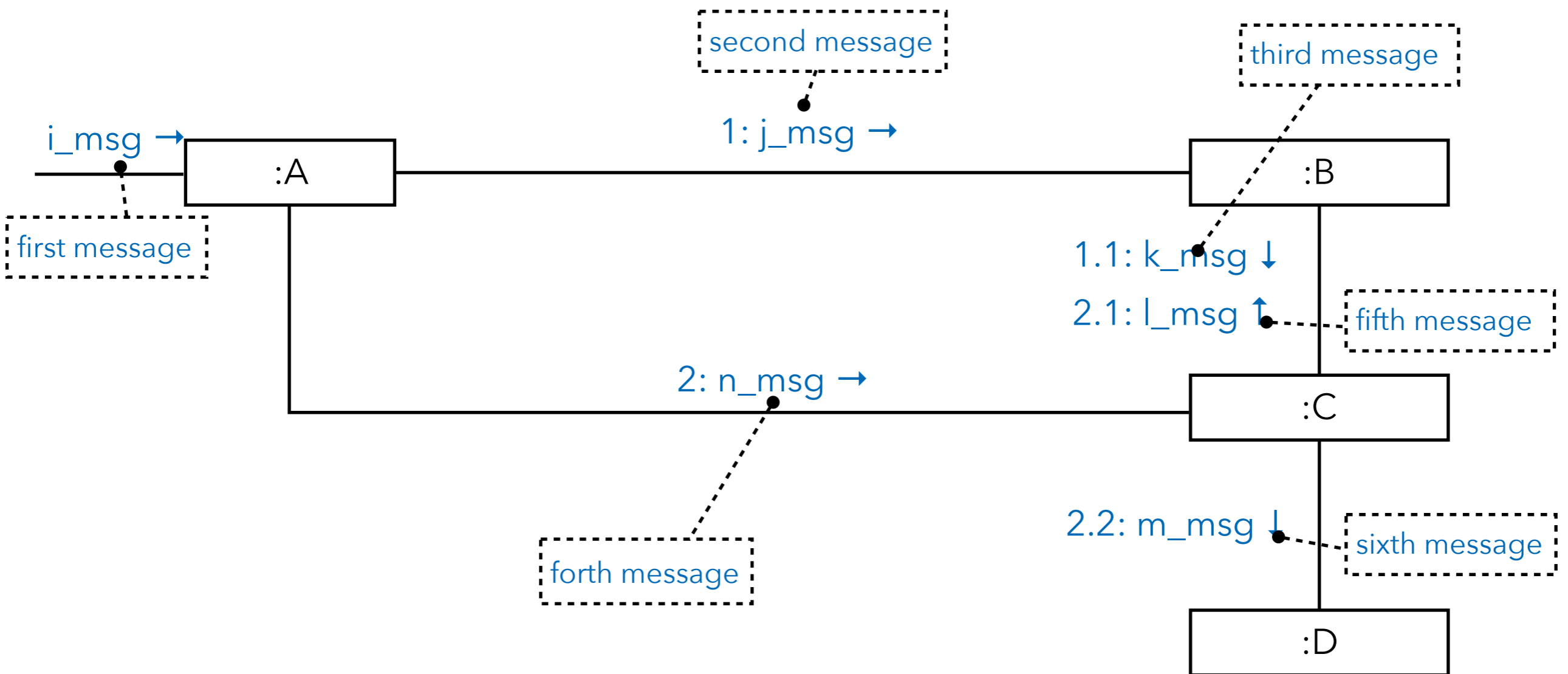
Modeling Conditional Messages

:E

2: msg6 ↑

msg →

Small letters are sometimes used to mark methods that are executed in parallel.
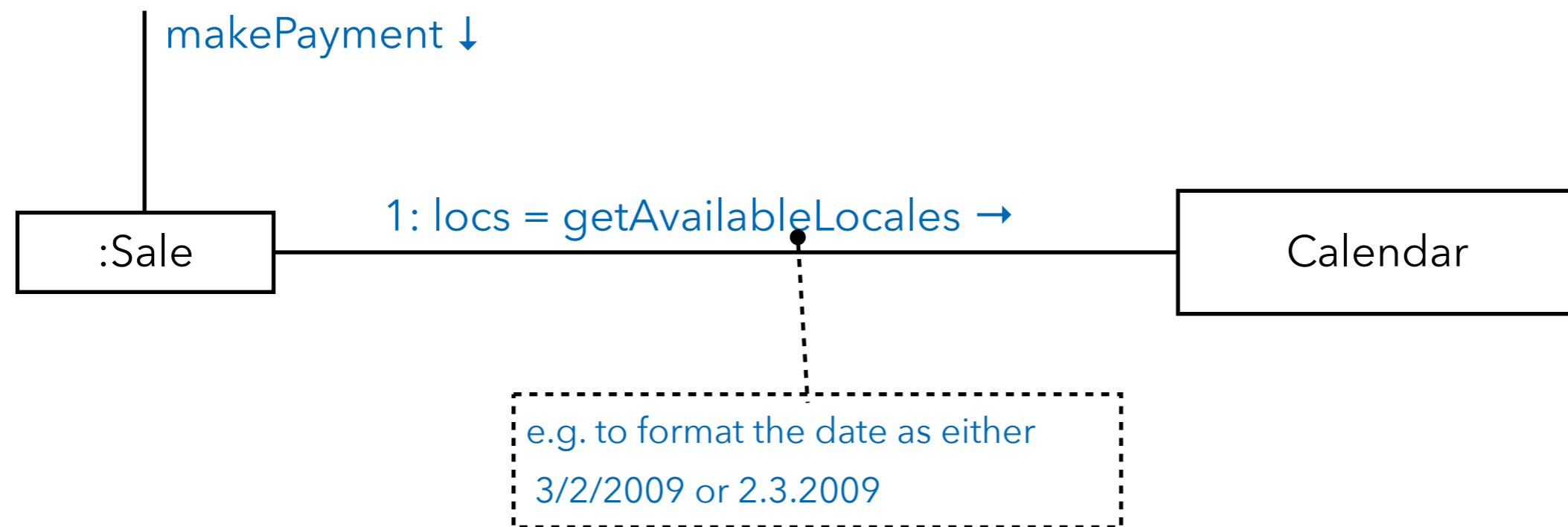
1a [test1]: msg2 →

:A

:B

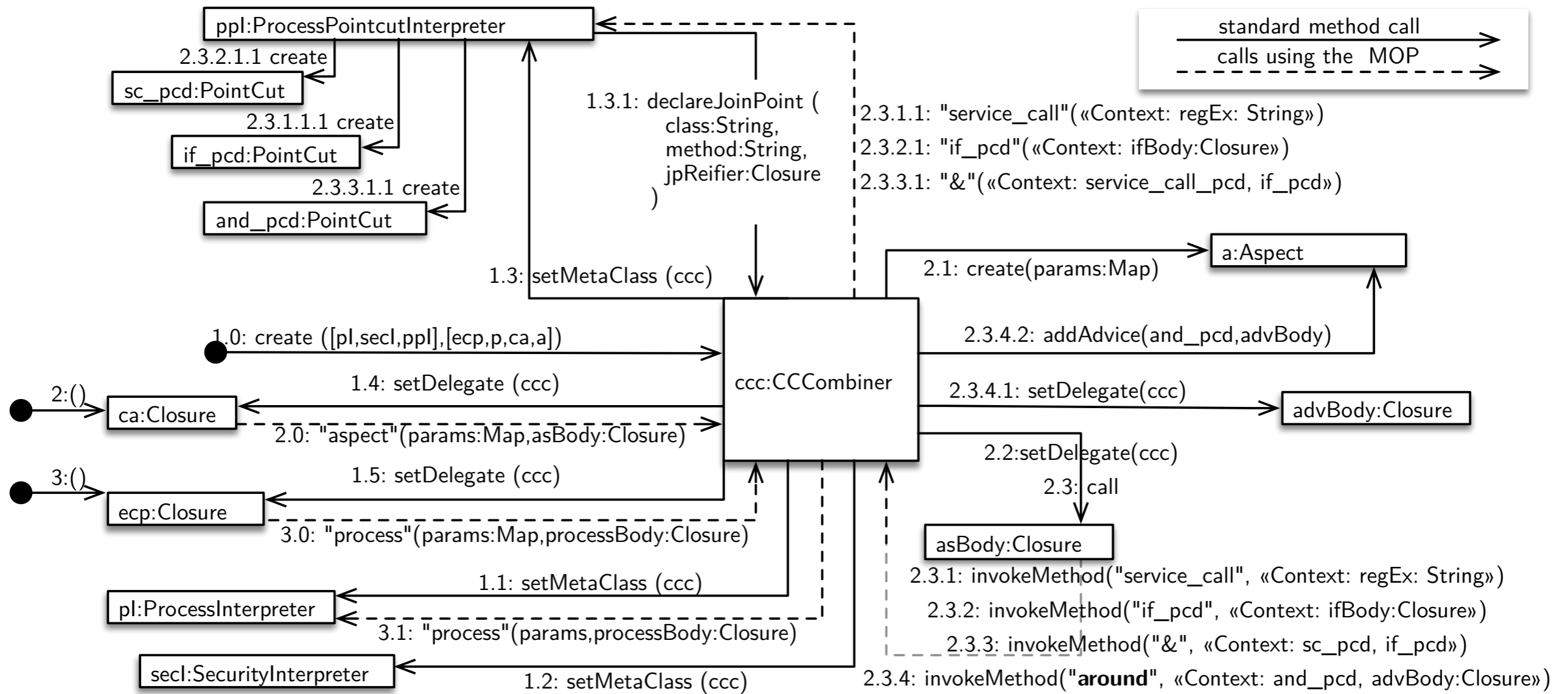1b [not test1]: msg4 ↓

1a.1: msg3 ↓

1b.1: msg5 →

:D

:C

The message is only sent if the condition evaluates to true. The condition is written in square brackets. In case of modeling mutually exclusive message conditional path letters are prepended.

Messages to Class Objects

makePayment ↓

:Sale —— 1: locs = getAvailableLocales → —— Calendar

e.g. to format the date as either
3/2/2009 or 2.3.2009

UML Communication vs. UML Sequence Diagrams

Strengths and Weaknesses Interaction Diagrams

Type	Strengths	Weaknesses
Sequence Diagram	✓ clearly shows sequence or time ordering of messages ✓ large set of detailed notation options	– forced to extend to the right when adding new objects; consumes horizontal space
Communication Diagram	✓ space economical - flexibility to add new objects in two dimensions	– more difficult to see sequence of messages – fewer notational options

Strengths and Weaknesses Interaction Diagrams

Type	Strengths	Weaknesses
Sequence Diagram	✓ clearly shows sequence or time ordering of messages	– forced to extend to the right when adding new objects; consumes
Communication Diagram	flexibility to add new objects in two dimensions	sequence of messages – fewer notational options

UML tools often emphasize sequence diagrams, because of their greater notational power.

Summary

The goal of this lecture is to enable you to systematically carry out small(er) software projects that produce quality software.

- Modeling the dynamic behavior is often more rewarding than modeling the static structure w.r.t. understanding a domain
- Modeling the dynamic behavior is often particularly useful if the control-flow is more involved; but only draw the part that is relevant to understand the problem at hand
- The UML is often used informally - this is OK if everyone interprets the diagrams in the same way

- The goal of this lecture is to enable you to systematically carry out small(er) commercial or open-source projects.

Modeling

Software Project Management

Project Start

Project End
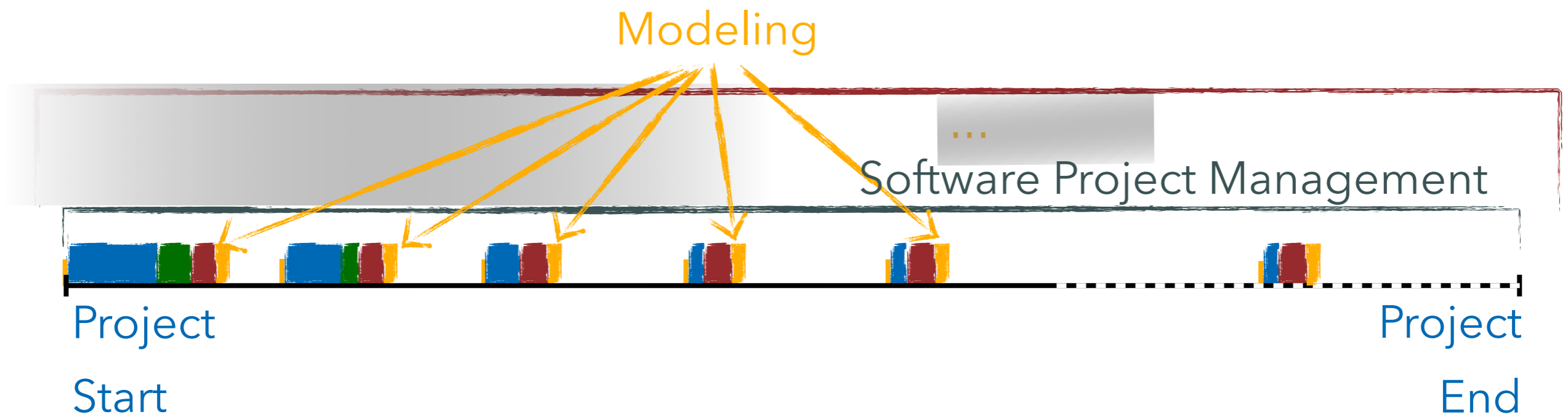
Requirements Management

Domain Modeling

Modeling

Testing