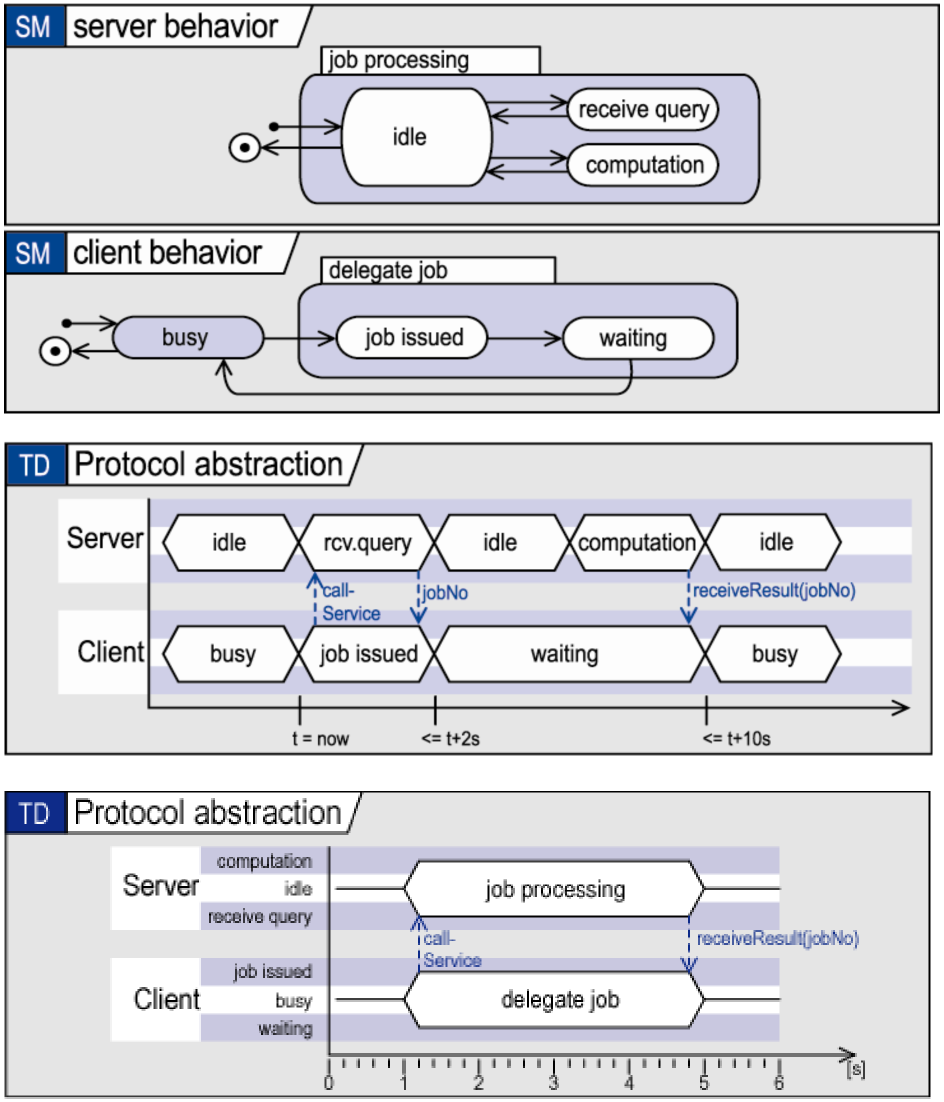
# Abstraction in timing diagram

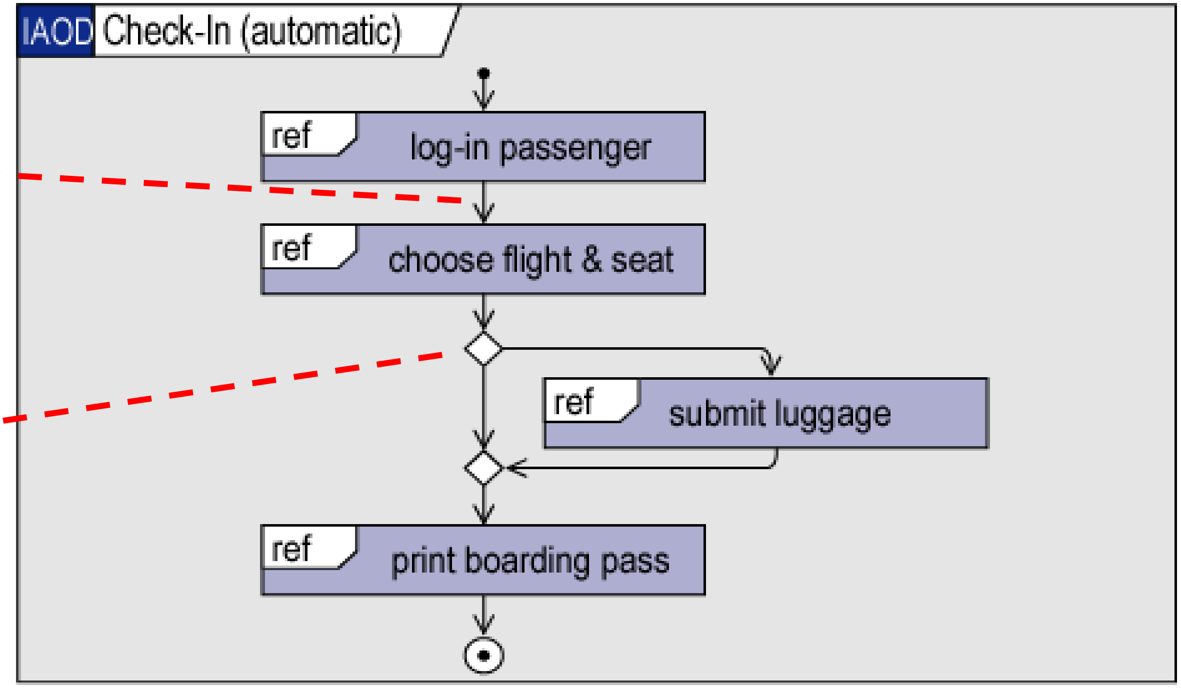
- An alternative syntax presents states not on the vertical axis but as hexagons on the lifeline.

- Timing diagrams present the coordination of (the states of) several objects over (real) time.



# Usage: Interaction overview

- Organize large number of interactions in a more visual style
- Defined as equivalent to using interaction operators



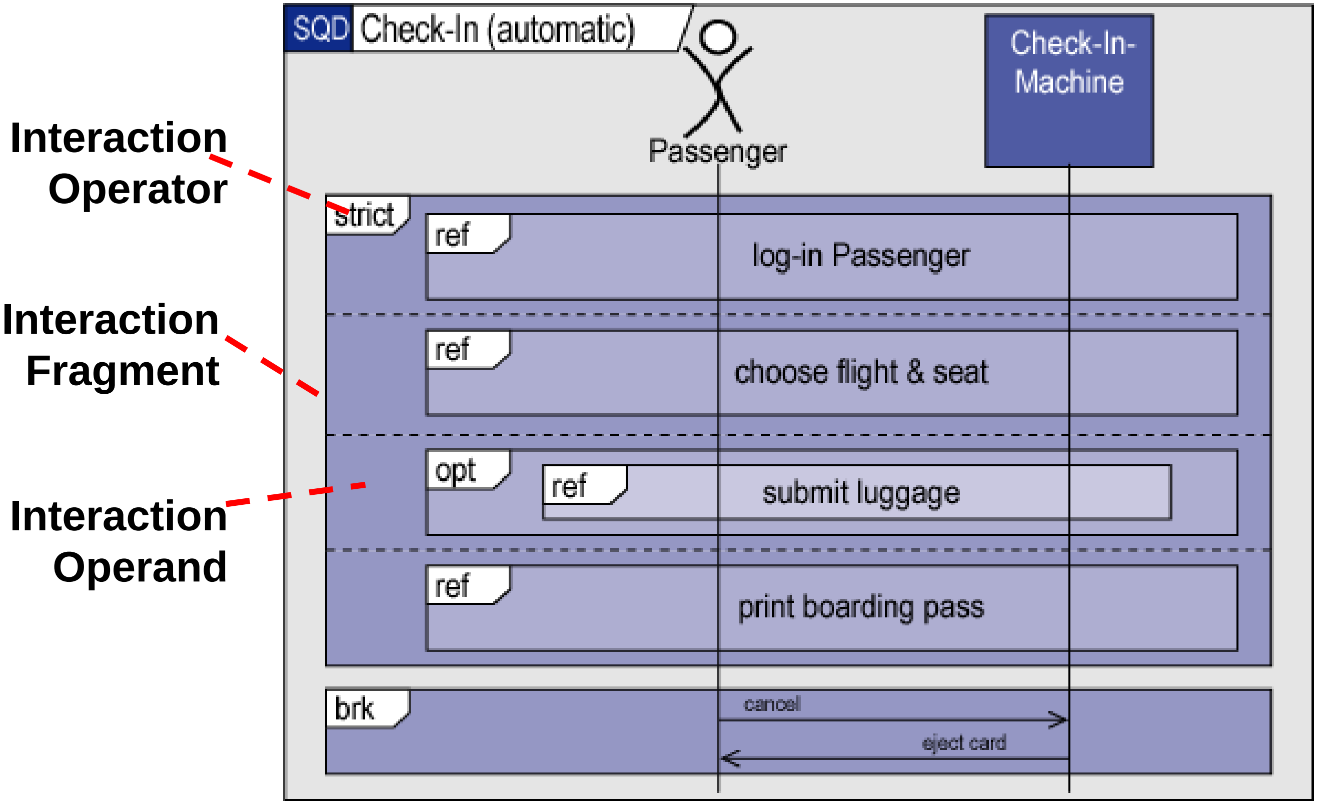
sequence equivalent to seq

choice/merge
equivalent to alt/opt

also allowed: fork/join
(said to be equivalent to par, but …)

# Complex interactions

- A complex interaction is like a functional expression:
  - an InteractionOperator,
  - one or several InteractionOperands (separated by dashed lines),
  - (and sometimes also numbers or sets of signals).



# Interaction operators (overview)

- strict
  - operand-wise sequencing
- seq
  - lifeline-wise sequencing
- loop
  - repeated seq

- par
  - interleaving of events

- region (aka. “critical”)
  - suspending interleaving

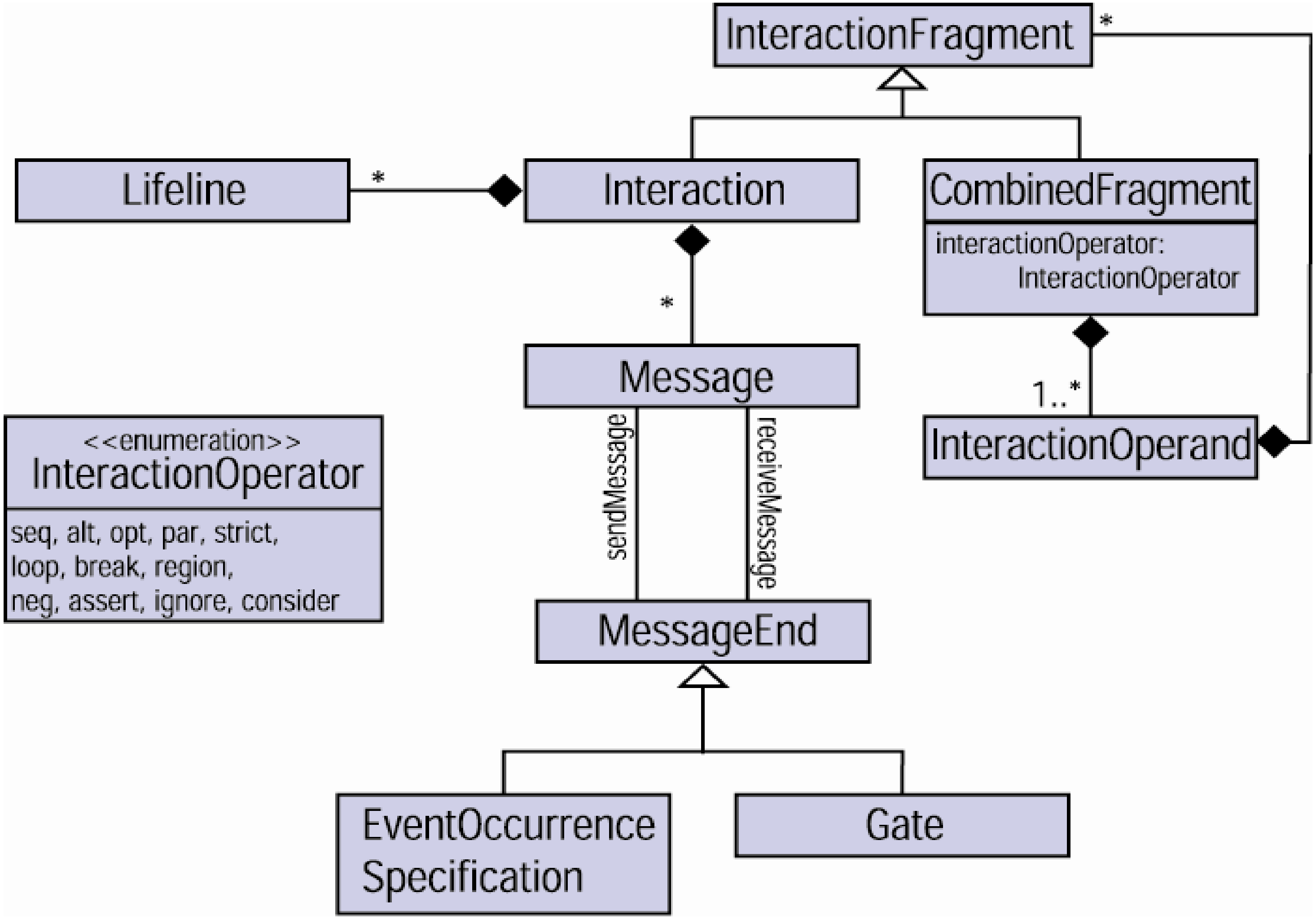
- consider
  - restrict model to specific messages
  - i.e. allow anything else anywhere
- ignore
  - dual to consider

- ref
  - macro-expansion of fragment

- alt
  - alternative execution
- opt
  - optional execution
  - syntactic sugar for alt
- break
  - abort execution
  - sometimes written as “brk”

- assert
  - remove uncertainty in specification
  - i.e. declare all traces as valid
- neg
  - declare all traces as invalid
    ( → three-valued semantics)

# Main concepts (metamodel)

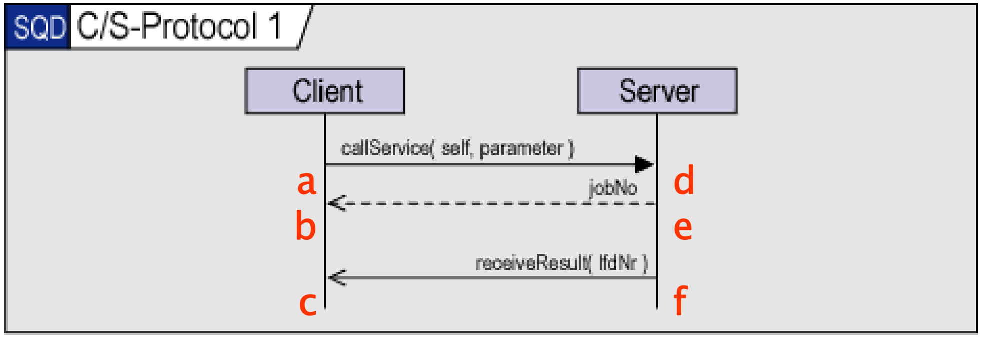


# Semantics

- The meaning of an interaction is
  - a set of valid traces, plus
  - a set of invalid traces.
- Traces are made up of occurrences of events such as
  - sending/receiving a message,
  - instantiating/terminating an object, or
  - time/state change events.
- Two types of constraints determine the valid traces:

  1) send occurs before receive,

  2) order on lifelines is definite.



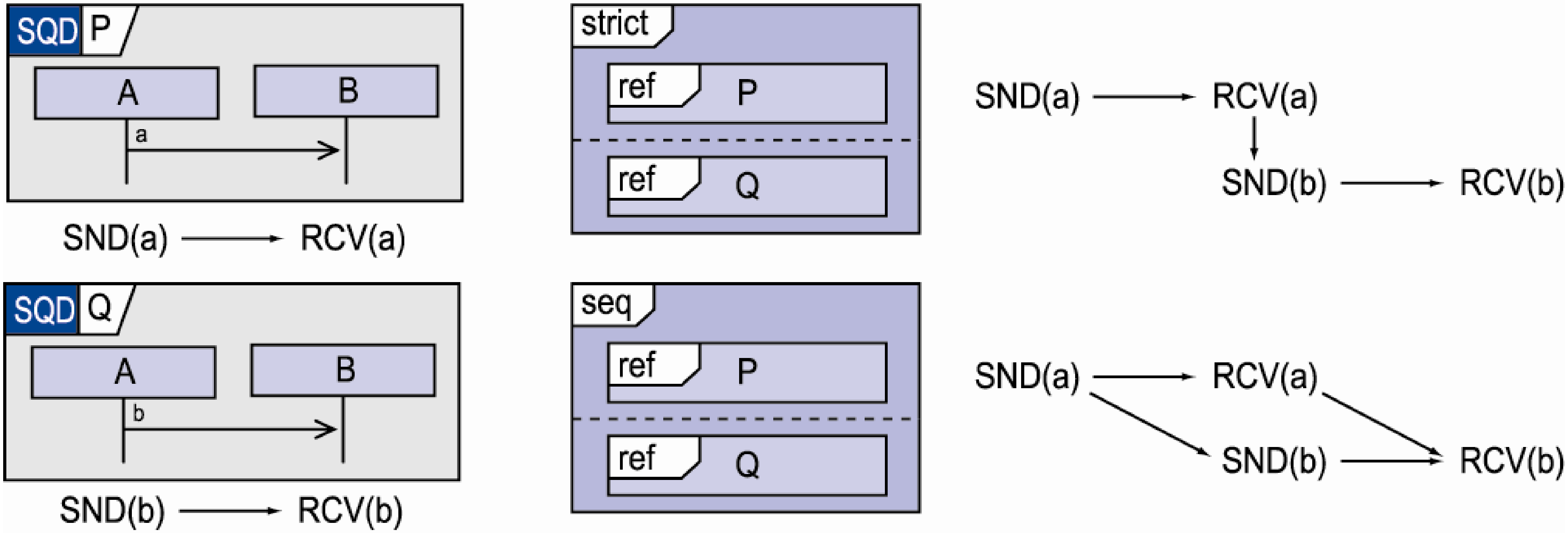
This diagram contains the following seven constraints:

1) a→d, e→b, f→c
2) a→b, b→c, d→e, e→f

The set of resulting traces is:
{ a.d.e.b.f.c, a.d.e.f.b.c }.

# Interaction operators seq & strict

- **seq**
  - compose two interactions sequentially lifeline-wise (default!)
- **strict**
  - compose two interactions sequentially diagram-wise



# Interaction operator loop

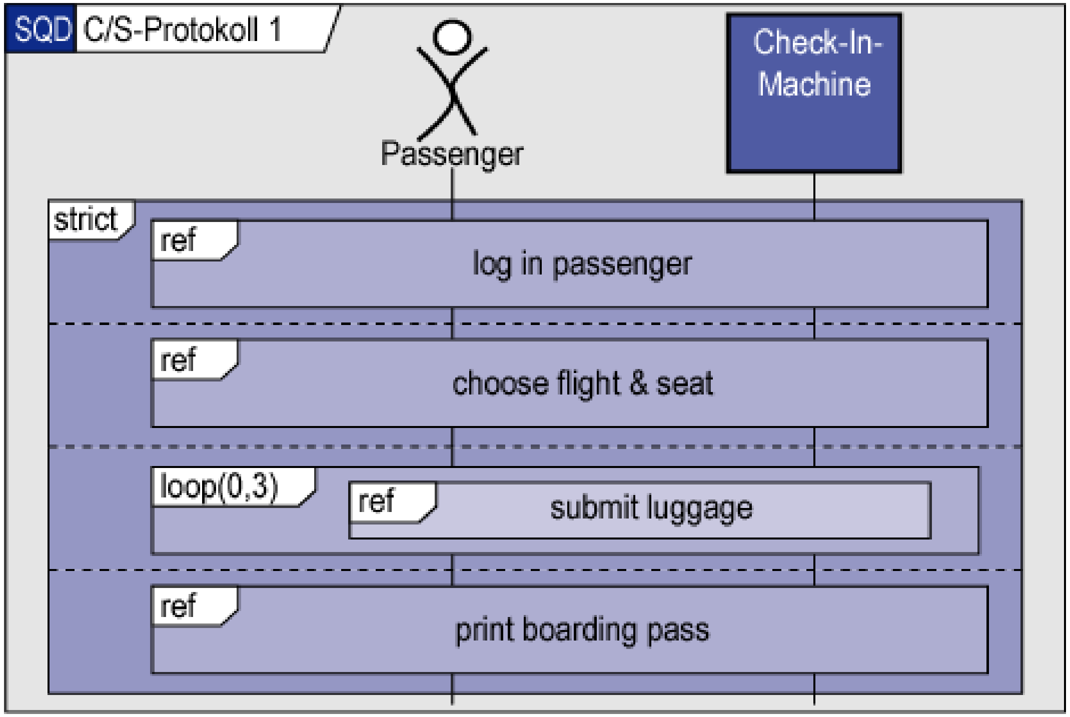
- **loop**
  - repeated application of seq

    loop(P, min, max) = seq(P, loop(P, min-1, max-1))

    loop(P, 0, max) = seq(opt(P), loop(P, 0, max-1))

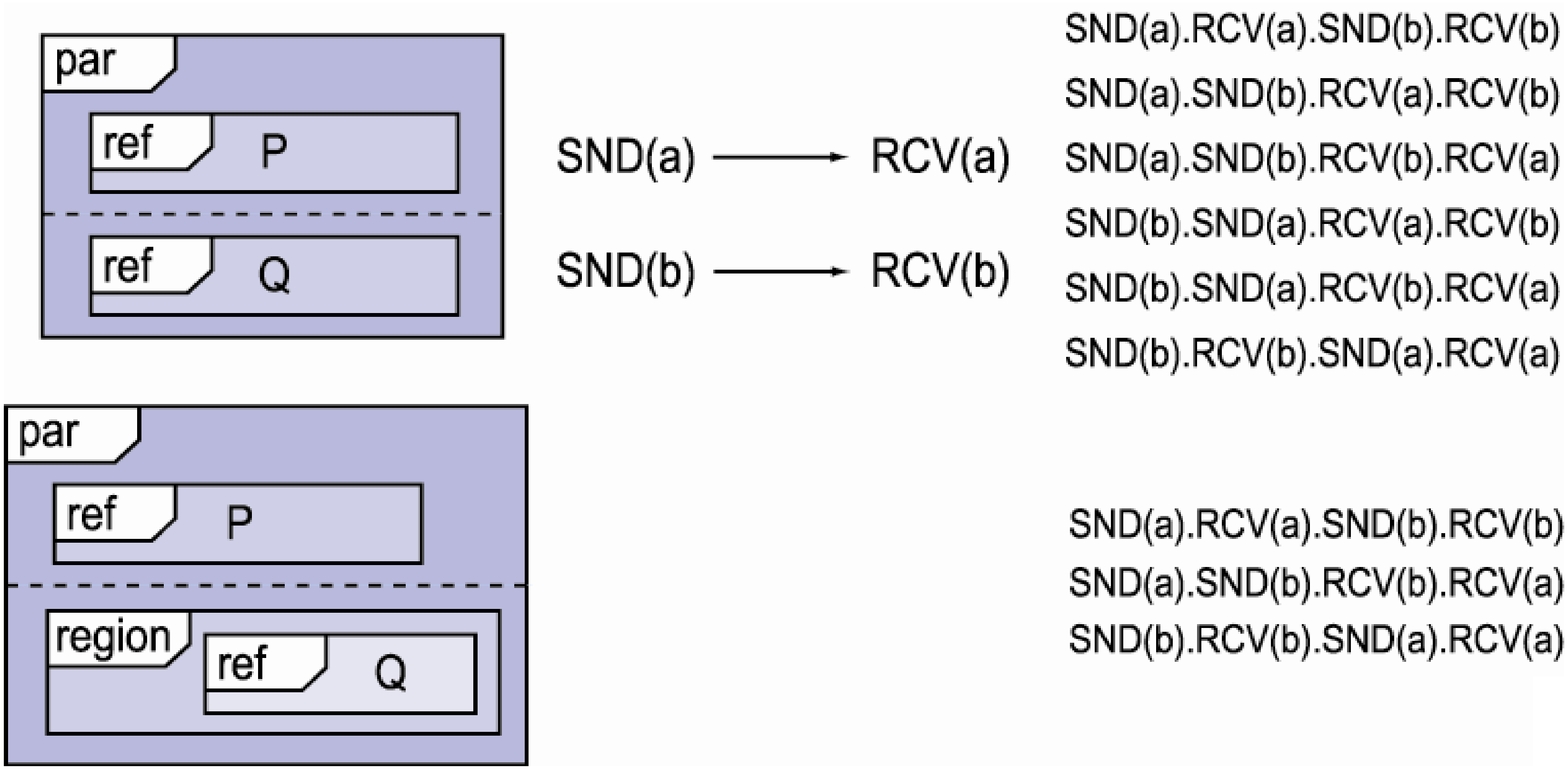
    loop(P, *) = seq(opt(P), loop(P, *))

    for some interaction fragment P



# Interaction operators: interleaving

- **par**
  - shuffle arguments
- **region**
  - execute argument atomically, i.e. disallow interleaving



SND(a).RCV(a).SND(b).RCV(b)
SND(a).SND(b).RCV(a).RCV(b)
SND(a).SND(b).RCV(b).RCV(a)
SND(b).SND(a).RCV(a).RCV(b)
SND(b).SND(a).RCV(b).RCV(a)
SND(b).RCV(b).SND(a).RCV(a)

SND(a).RCV(a).SND(b).RCV(b)
SND(a).SND(b).RCV(b).RCV(a)
SND(b).RCV(b).SND(a).RCV(a)

# Interaction operators alt, opt, brk: choice

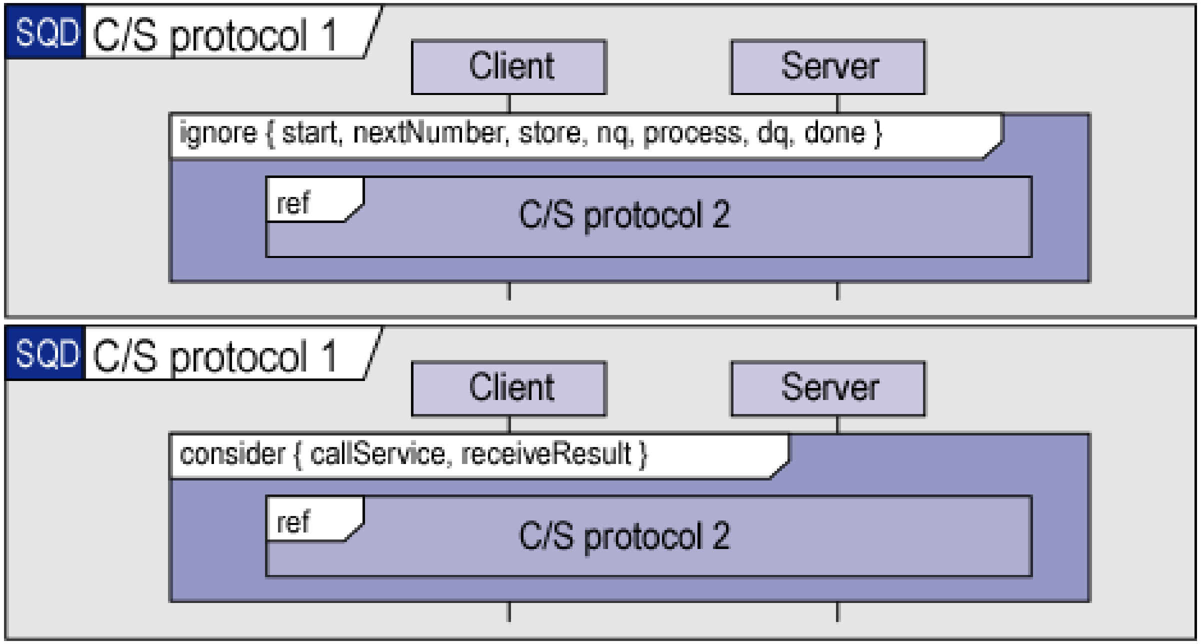
- **alt**
  - alternative complete execution of one of two interaction fragments

- **opt**
  - optional complete execution of interaction fragment:
    opt(P) = alt(P, nop)

- **break**
  - execute interaction fragment partially, skip rest, and jump to surrounding fragment

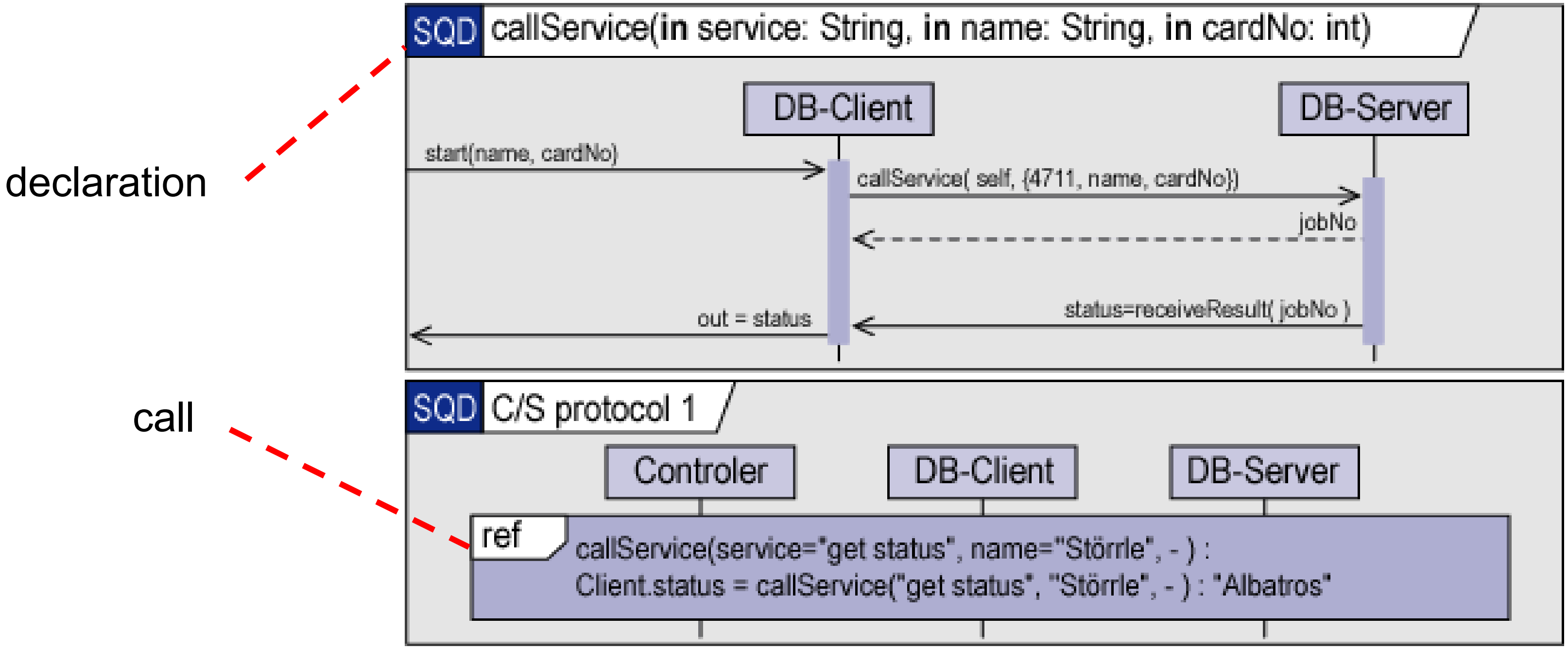
# Interaction operators: abstraction

- **ignore**, **consider**
  - dual way of expressing:
    - allow the ignorable messages (!) anywhere
    - present only those messages that are to be considered
    - $[\![ \mathrm{ignore}(P,Z) ]\!] = \mathrm{shuffle}( [\![P]\!] , Z^*)$



# Interaction operator ref & parameters

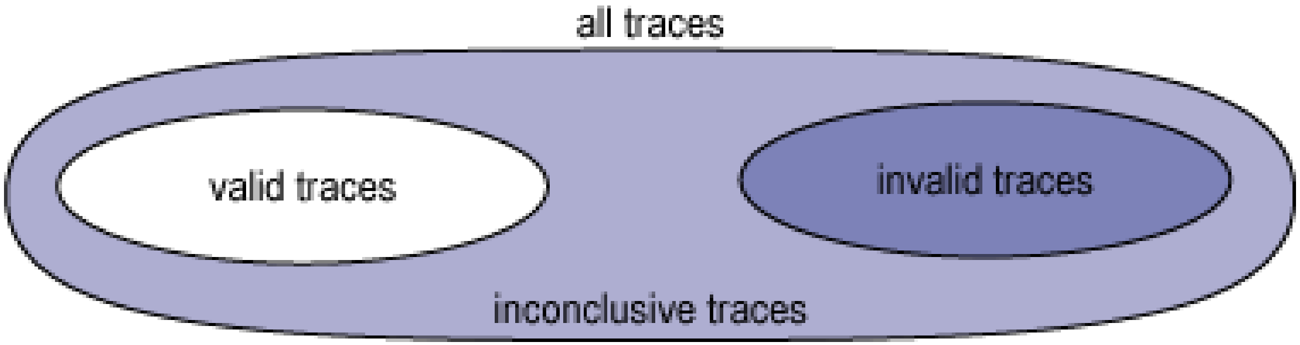
- **ref**
  - refers to a fragment defined elsewhere (macro-expansion)
  - Formal and actual parameters (bindings) are declared in the diagram head.



- Signals to the containing classifier appear as arrows form the diagram border.

# Interaction operators: negation

- The semantics of neg and assert is unclear.
- In contrast to that the other operators, they refer not just to the positive traces, but to invalid and inconclusive traces as well.



- **neg**
  - declare all valid traces as invalid
  - inconclusive traces: unknown

- **assert**
  - remove uncertainty by declaring all inconclusive traces as invalid

# Wrap up

- Complex interactions **like high-level MSCs** added.

- New diagram types:
  - timing diagrams (like oscilloscope), and
  - interaction overview (similar to restricted activity diagram)
  - renamed collaboration diagram to communication diagram

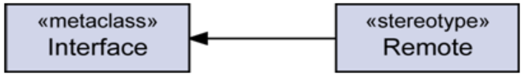
- Completely **new metamodel**.

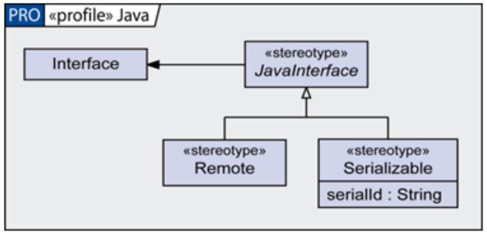
- Almost formal three-valued semantics of valid, invalid and inconclusive interleaving traces of events.

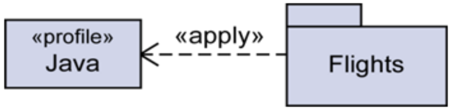
- Some semantical problems are yet to be solved.

# Unified Modeling Language 2

## *Profiles*


«metaclass»
Interface
«stereotype»
Remote

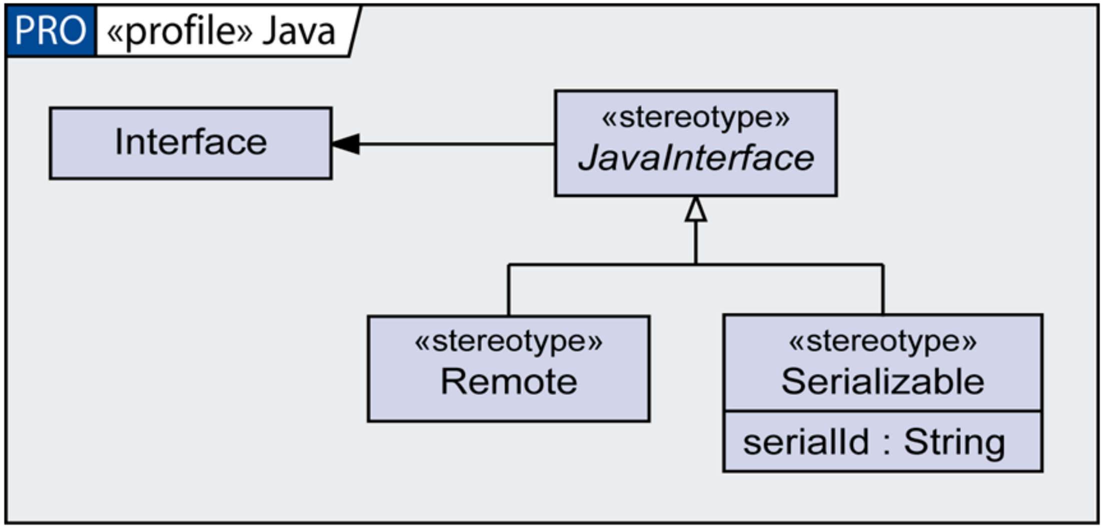

PRO «profile» Java
Interface
«stereotype»
*JavaInterface*
«stereotype»
Remote
«stereotype»
Serializable
serialId : String


«profile»
Java
«apply»
Flights

# Usage scenarios

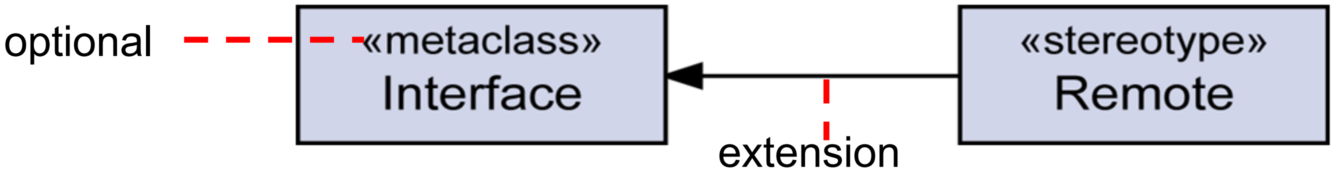
- **Metamodel customization** for
  - adapting terminology to a specific platform or domain
  - adding (visual) notation
  - adding and specializing semantics
  - adding constraints
  - transformation information

- **Profiling**
  - packaging domain-specific extensions
  - "domain-specific language" engineering

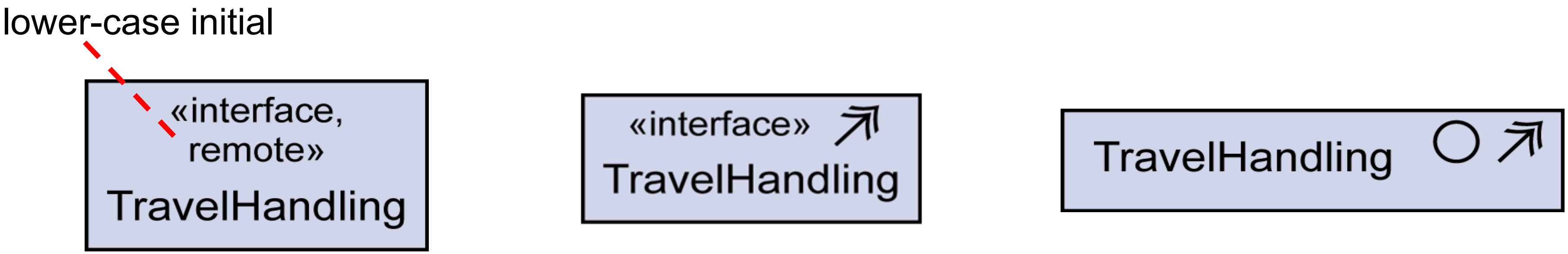


# Stereotypes (1)

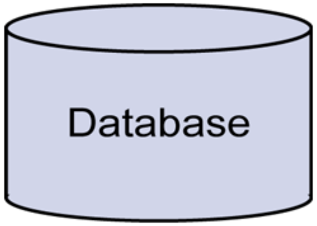
- Stereotypes define how an existing (UML) metaclass may be extended.



- Stereotypes may be applied **textually** or **graphically**.

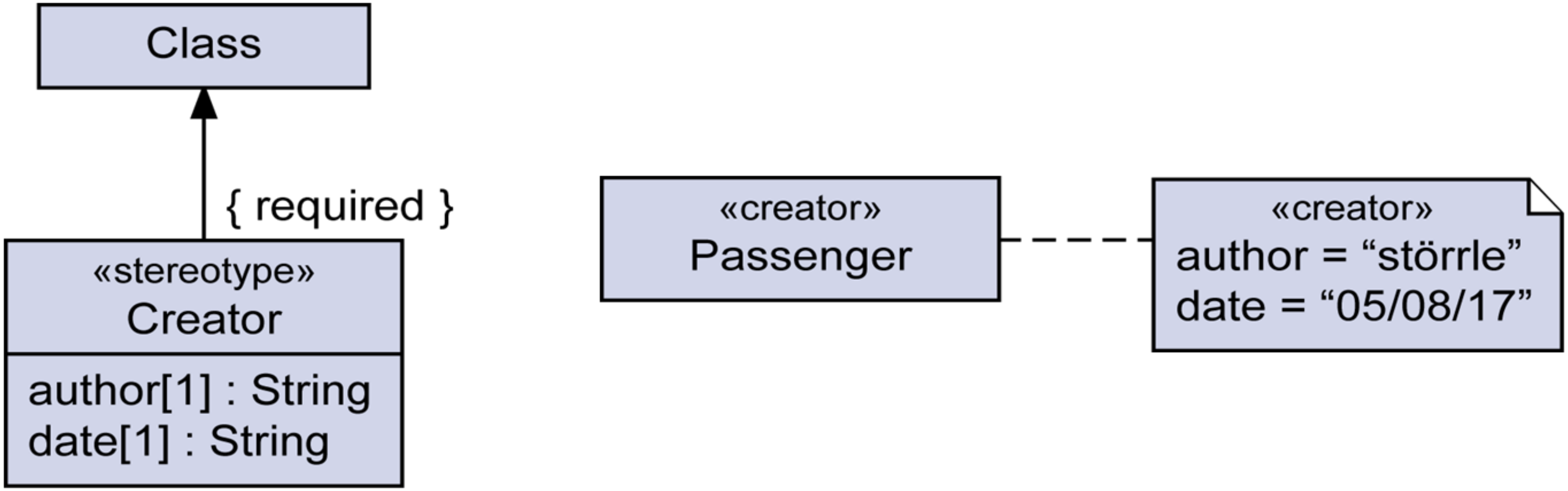


- Visual stereotypes may replace original notation.
  - But the element name should appear below the icon…



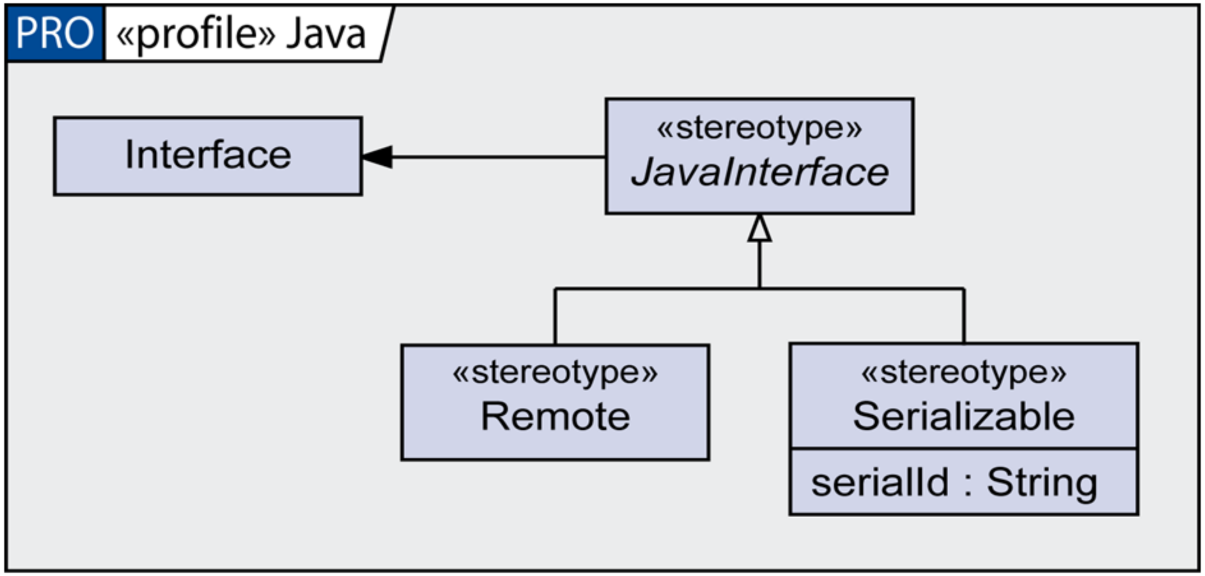
# Stereotypes (2)

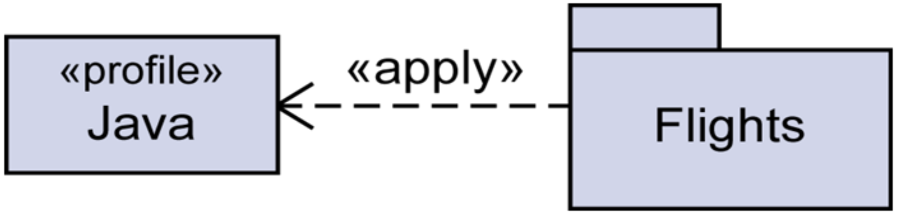
- Stereotypes may define **meta-properties**.
  - commonly known as “tagged values”
- Stereotypes can be defined to be **required**.
  - Every instance of the extended metaclass has to be extended.
  - If a required extension is clear from the context it need not be visualized.



# Profiling

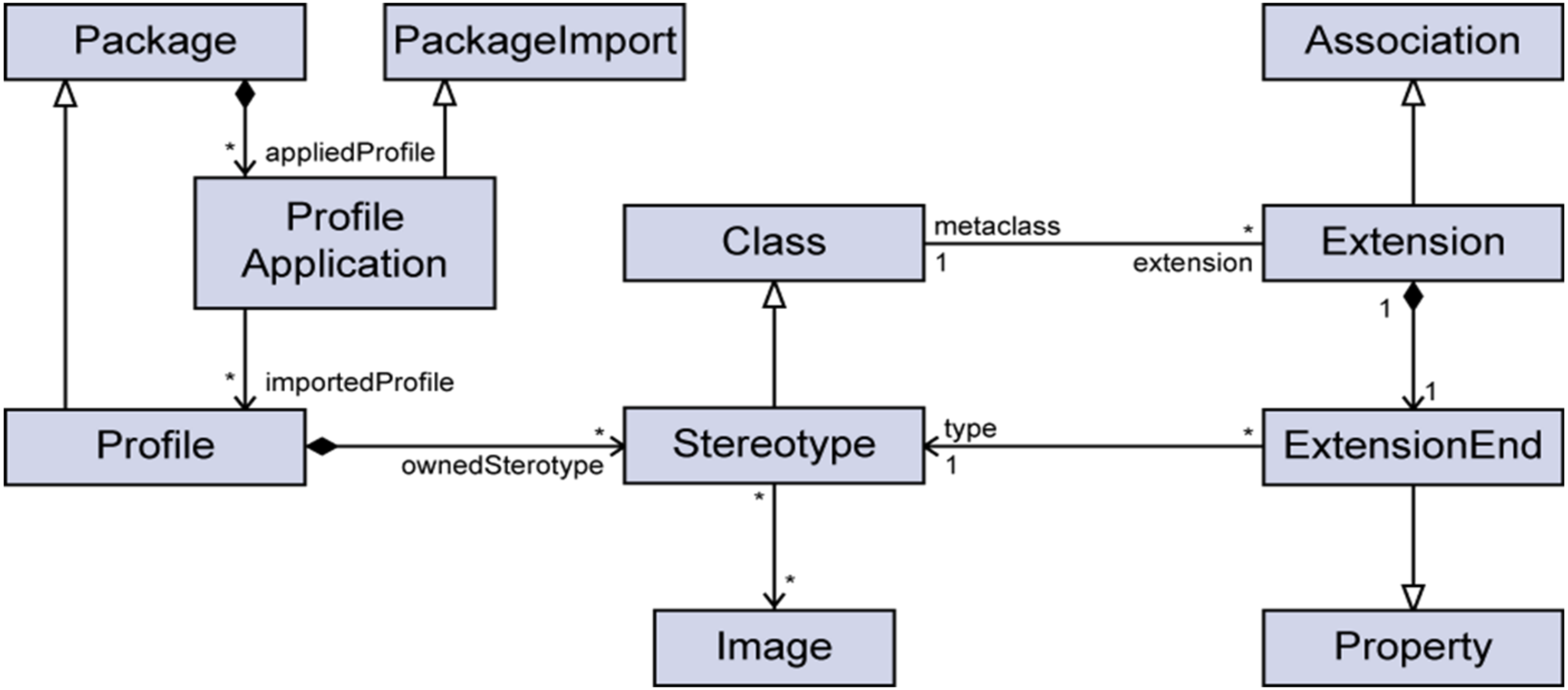
- Profiles **package** extensions.





# Metamodel

- Based on **infrastructure library** constructs
    - Class, Association, Property, Package, PackageImport



# Metamodeling with Profiles

- Profile extension mechanism imposes **restrictions** on how the UML metamodel can be modified.
  - UML metamodel considered as “read only”.
  - No intermediate metaclasses

- Stereotypes metaclasses below UML metaclasses.

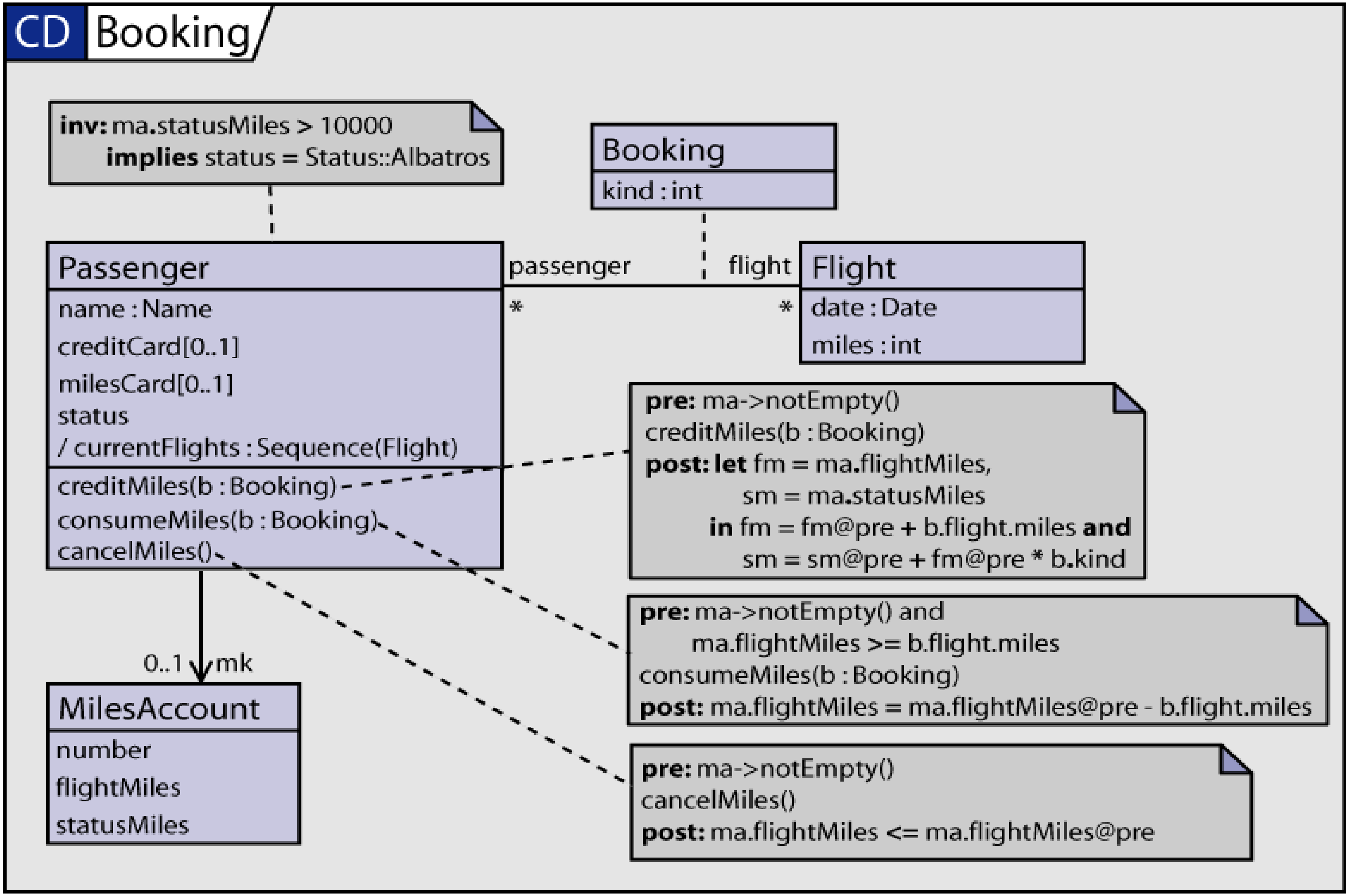
# Wrap up

- Metamodel extensions
  - with stereotypes and meta-properties
  - for restricting metamodel semantics
  - for extending notation

- Packaging of extensions into profiles
  - for declaring applicable extensions
  - “domain-specific language” engineering

# Object Constraint Language 2

# A first glimpse



CD Booking

**inv:** ma.statusMiles > 10000
**implies** status = Status::Albatros

Booking
kind : int

Passenger
name : Name
creditCard[0..1]
milesCard[0..1]
status
/ currentFlights : Sequence(Flight)
creditMiles(b : Booking)
consumeMiles(b : Booking)
cancelMiles()

passenger
*
flight
*

Flight
date : Date
miles : int

**pre:** ma->notEmpty()
creditMiles(b : Booking)
**post: let** fm = ma.flightMiles,
sm = ma.statusMiles
**in** fm = fm@pre + b.flight.miles **and**
sm = sm@pre + fm@pre * b.kind

**pre:** ma->notEmpty() and
ma.flightMiles >= b.flight.miles
consumeMiles(b : Booking)
**post:** ma.flightMiles = ma.flightMiles@pre - b.flight.miles

0..1
mk

MilesAccount
number
flightMiles
statusMiles

**pre:** ma->notEmpty()
cancelMiles()
**post:** ma.flightMiles <= ma.flightMiles@pre

# History and predecessors

- **Predecessors**
  - Model-based specification languages, like
    - Z, VDM, and their object-oriented variants; B
  - Algebraic specification languages, like
    - OBJ3, Maude, Larch

- Similar approaches in programming languages
  - ESC, JML

- **History**
  - developed by IBM as an easy-to-use formal annotation language
  - used in UML metamodel specification since UML 1.1
  - current version: OCL 2.3.1
    - specification: formal/2012-01-01

# Usage scenarios

- Constraints on implementations of a model
  - invariants on classes
  - pre-/post-conditions for operations
    - cf. protocol state machines
  - body of operations
  - restrictions on associations, template parameters, …
- Formalization of side conditions
  - derived attributes
- Guards
  - in state machines, activity diagrams
- Queries
  - query operations
- **Model-driven architecture (MDA)/query-view-transformation (QVT)**

# Language characteristics

- Integration with UML
  - access to classifiers, attributes, states, …
  - navigation through attributes, associations, …
  - limited reflective capabilities
  - model extensions by derived attributes
- **Side-effect free**
  - *not* an action language
  - only possibly describing effects
- **Statically typed**
  - inherits and extends type hierarchy from UML model
- Abstract and concrete syntax
  - precise definition new in OCL 2

# Simple types

- Predefined primitive types
  - `Boolean` `true`, `false`
  - `Integer` `-17, 0, 3`
  - `Real` `-17.89, 0.0, 3.14`
  - `String` `“Hello”`
- Types induced by UML model
  - Classifier types, like
    - Passenger no denotation of objects, only in `context`
  - Enumeration types, like
    - Status `Status::Albatros, #Albatros`
  - Model element types
    - `OclModelElement`, `OclType`, `OclState`
- Additional types
  - `OclInvalid` invalid (`OclUndefined`)
  - `OclVoid` null
  - `OclAny` top type of primitives and classifiers

# Parameterized types

- Collection types
  - Set($T$) sets
  - OrderedSet($T$) like `Sequence` without duplicates
  - Bag($T$) multi-sets
  - Sequence($T$) lists
  - Collection($T$) abstract
- Tuple types (records)
  - Tuple($a_1$ : $T_1$, …, $a_n$ : $T_n$)
- Message type
  - `OclMessage` for operations and signals

Examples

- `Set{Set{ 1 }, Set{ 2, 3 }} : Set(Set(Integer))`
- `Bag{1, 2.0, 2, 3.0, 3.0, 3} : Bag(Real)`
- `Tuple{x = 5, y = false} : Tuple(x : Integer, y : Boolean)`

# Type hierarchy

- Type conformance (reflexive, transitive relation ≤)
  - `OclVoid` $\leq T$ for all types $T$ but `OclInvalid`
  - `OclInvalid` $\leq T$ for all types $T$
  - `Integer` ≤ `Real`
  - $T \leq T' \Rightarrow C(T) \leq C(T')$ for collection type $C$
  - $C$`(`$T$`)` ≤ `Collection(`$T$`)` for collection type $C$
  - generalization hierarchy from UML model
  - $B \leq$ `OclAny` for all primitives and classifiers $B$

**Counterexample**

  - ¬(`Set(OclAny)` ≤ `OclAny`)

- Casting
  - $v$`.oclAsType(`$T$`)` if $v : T'$ and ($T \leq T'$ or $T' \leq T$)
  - upcast necessary for accessing overridden properties

# Expressions

- Local variable bindings

  **`let`** `x = 1` **`in`** `x+2`

- Iteration

  $c$`->iterate(`$i$ : $T$; $a$ : $T'$ = $e'$ | $e$`)`

source collection

iteration variable
(bound to current value in *c*)

accumulator with initial value *e’*
(gathers result, returned after iteration)

iteration expression
(using variables *i* and *a*)

Example:

```
Set{1, 2}->iterate(i : Integer; a : Integer = 0 | a+i) = 3
```

- Many operations on collections are **reduced** to `iterate`

# Expressions: Standard library (1)

- Operations on primitive types (written: $v$.$op$(…))
  - operations without ( ) are mixfix

| | |
|---|---|
| OclAny | =, <>, oclIsTypeOf($T$), oclIsKindOf($T$), … |
| Boolean | **and**, **or**, **xor**, **implies**, **not** |
| Integer | +, -, *, /, div($i$), mod($i$), … |
| Real | +, -, *, /, floor(), round(), … |
| String | size(), concat($s$), substring($l$, $u$), … |

- Operations on collection types (written: $v$->$op$(…))

| | |
|---|---|
| Collection | size(), includes($v$), isEmpty(), … |
| Set | union($s$), including($v$), flatten(), asBag(), … |
| OrderedSet | append($s$), first(), at($i$), … |
| Bag | union($b$), including($v$), flatten(), asSet(), … |
| Sequence | append($s$), first(), at($i$), asOrderedSet(), … |