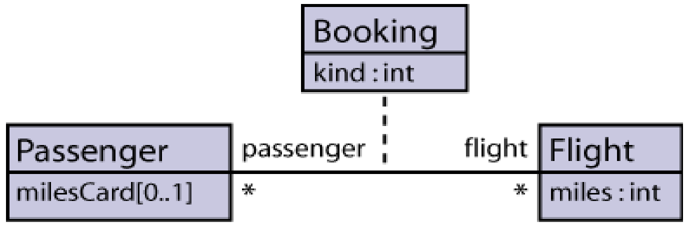
Connection to UML

- Import of classifiers and enumerations as types
- Properties accessible in OCL
  - Attributes
    - p.milesCard (with p : Passenger)
  - Association ends
    - p.flight, p.booking, p.booking[flight]
  - { query } operations
  - Access to stereotypes via v.`stereotype`



- **Representation of multiplicities**

$a[1] : T$	$a : T$
$a[0..1] : T$	a : `Set`(T) or T
$a[m..n] : T$	a : `Set`(T)
$a[*] : T$ { unordered }	a : `Set`(T)
$a[*] : T$ { ordered }	a : `OrderedSet`(T)
$a[*] : T$ { bag }	a : `Bag`(T)

Invariants

context classifier

boolean expression

```
context Passenger
inv: ma.statusMiles > 10000 implies
     status = Status::Albatros
```

Notational variants

explicit self (refers to instance of discourse)

```
context Passenger
inv statusLimit: self.ma.statusMiles > 10000 implies
                 self.status = Status::Albatros
```

optional name

```
context p : Passenger
inv statusLimit: p.ma.statusMiles > 10000 implies
                 p.status = Status::Albatros
```

replacement for self

Semantics of invariants

- Restriction of valid states of classifier instances
  - when observed from outside
- Invariants (as all constraints) are inherited via generalizations
  - but how they are combined is not predefined
- One possibility: Combination of several invariants by **conjunction**

context C
inv: I_1

context C
inv: I_2

$\leadsto$

context C
inv: I_1 **and** I_2

Pre-/post-conditions

- In UML models, pre- and post-conditions are defined separately
  - not necessarily as pairs
  - «precondition» and «postcondition» as constraint stereotypes

```
context Passenger::consumeMiles(b : Booking) : Boolean
pre: ma->notEmpty() and
     ma.flightMiles >= b.flight.miles

context Passenger::consumeMiles(b : Booking) : Boolean
post: ma.flightMiles = ma.flightMiles@pre-b.flight.miles and
      result = true
```

- Some constructs only available in post-conditions
  - values at pre-condition time — *p*`@pre`
  - result of operation call — `result`
  - whether an object has been newly created — *o*`.oclIsNew()`
  - messages sent — *o*`^`*op*`()`, *o*`^^`*op*`()`

Semantics of pre-/post-conditions

- Standard interpretation
  - A pre-/post-condition pair (P, Q) defines a relation R on system states such that $(\sigma, \sigma') \in R$, if $\sigma \models P$ and $(\sigma, \sigma') \models Q$.
    - system state σ on operation invocation
    - system state σ' on operation termination (Q may refer to σ by `@pre`).
  - Thus (P, Q) equivalent to (`true`, P`@pre` **and** Q).

- **Meyer's contract view**
  - A pre-/post-condition pair (P, Q) induces benefits and obligations.
  - benefits and obligations differ for implementer and user

	obligation	**benefit**
user	satisfy P	Q established
implementer	if P satisfied, establish Q	P established

Combining pre-/post-conditions

- Standard interpretation
  - joining pre- and post-conditions conjunctively

context C::op()
pre: P_1 **post:** Q_1

context C::op()
pre: P_2 **post:** Q_2

$\leadsto$

context C::op()
pre: P_1 **and** P_2
post: Q_1 **and** Q_2

- Alternative interpretation
  - **case distinction** (like in protocol state machines)
  - only useful for pre-/post-condition pairs

context C::op()
pre: P_1 **post:** Q_1

context C::op()
pre: P_2 **post:** Q_2

$\leadsto$

context C::op()
pre: P_1 **or** P_2
post: (P_1@pre **implies** Q_1)
and (P_2@pre **implies** Q_2)

Messages

```
context Subject::hasChanged()
post: observer^update(self)
```

- - - - in calls on hasChanged, some update message with argument `self` will have been sent to observer

```
context Subject::hasChanged()
post: observer^update(? : Subject)
```

- - - - the actual argument does not matter

```
context Subject::hasChanged()
post: let messages : Set(OclMessage) =
            observer^^update(? : Subject)
      in messages->notEmpty() and
         messages->forAll(m |
            m.result().oclIsUndefined() and
            m.hasReturned() and
            m.subject = self)
```

- observer^^update(? : Subject) - - - - all those messages
- m.result() - - - result of message call
- m.hasReturned() - - - whether it has finished
- m.subject - - - its actual parameter value

Initial values and derived properties

- Initial values
  - fix the initial value of a property of a classifier

```
package Booking                          -- which package
  context Passenger::status              -- which property
  init: Status::Swallow                  -- initial value
endpackage
```

- { derived } properties
  - define how the value of a property is derived from other information

```
context Passenger::currentFlights : Sequence(Flight)
derive: self->collect(booking)
            ->select(date = today()).flight->asSequence()
```

Query bodies and model features

- Bodies of { query } operations
  - define the value returned by a query operation
  - can be combined with a precondition

```
context TravelHandling::delay() : Minutes
body: tsh.delay->sum()
```

- Definition of additional model features
  - defined for the context classifier

```
context TravelStageHandling
def: isEarly() : Boolean = self.delay < 0

context TravelHandling
def: someEarly() : Boolean = tsh->exists(isEarly())
```

Wrap up

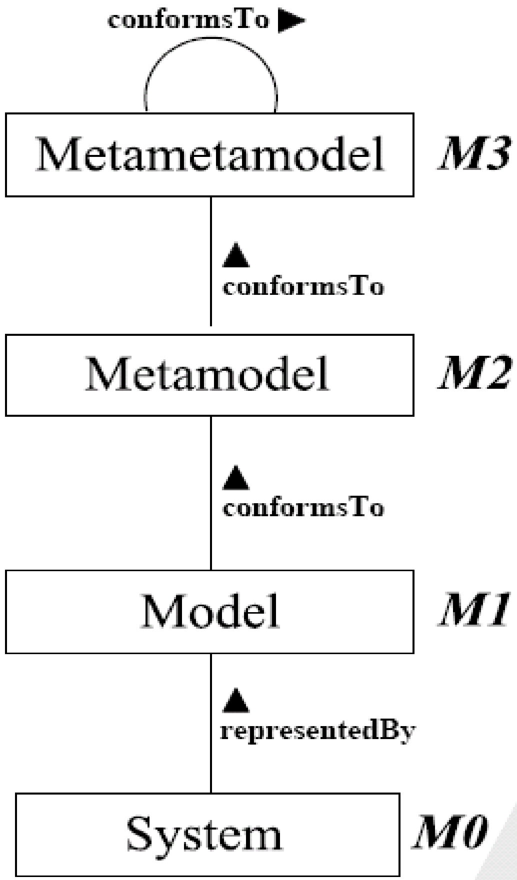
- Formal language for specifying
  - invariants — **context** C **inv:** I
  - pre-/post-conditions — **context** C::$op()$: T **pre:** P **post:** Q
  - query operation bodies — **context** C::$op()$: T **body:** e
  - initial values — **context** C::p : T **init:** e
  - derived attributes — **context** C::p : T **derive:** e
  - modelling attributes and operations — **context** C **def:** p : T = e

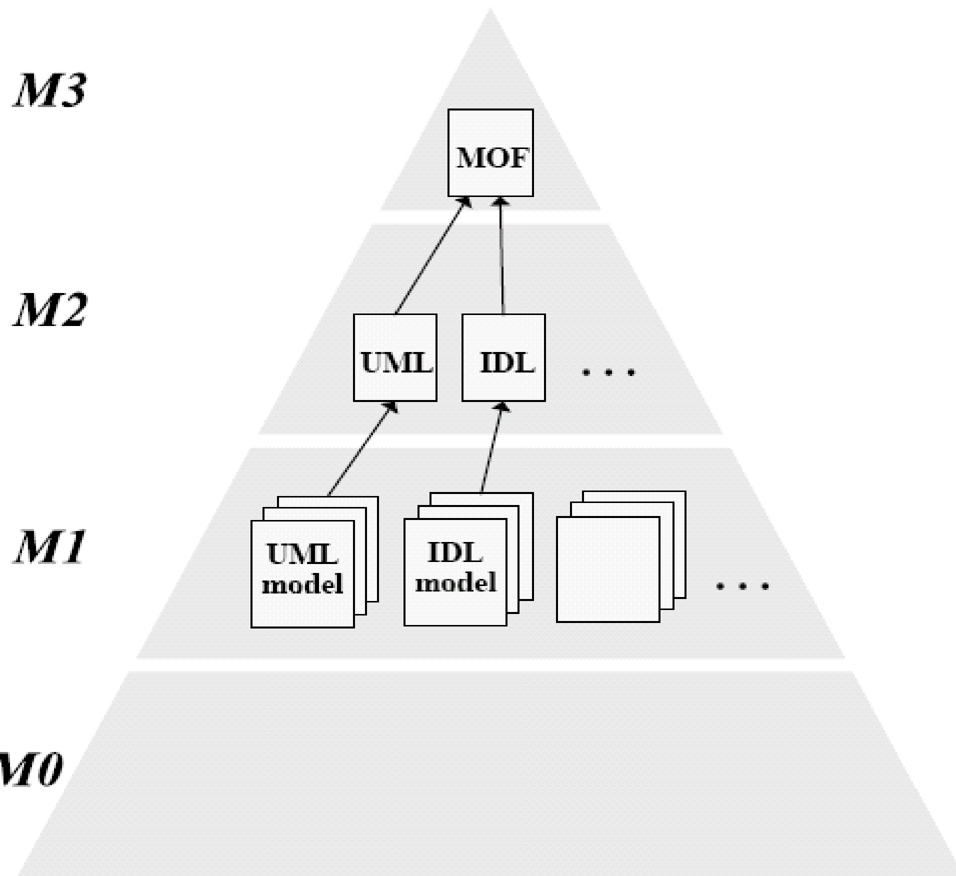
- Side-effect free
- Typed language

- OCL specifications provide
  - verification conditions
  - assertions for implementations

Meta-Object Facility 2

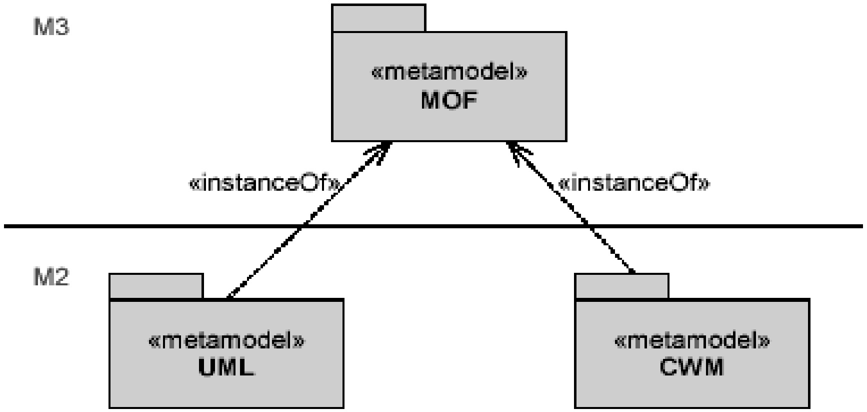
OMG's standards UML and MOF

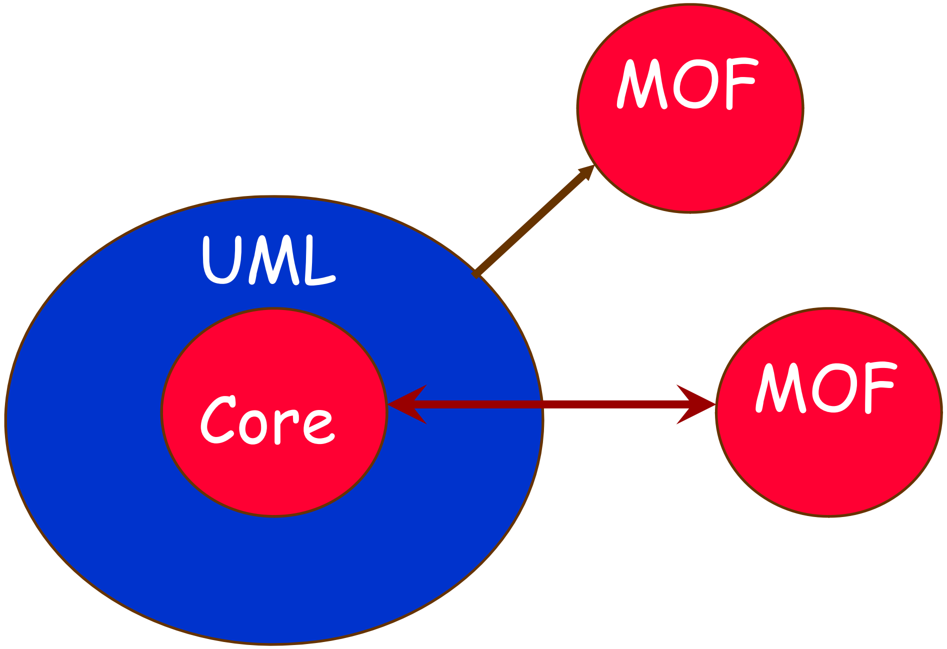




Relations between UML 2 and MOF 2

- MOF meta-meta-model of UML 2
- MOF is (based on) the core of UML 2
- UML 2 is a drawing tool of the MOF 2
- Definition synchronization





Meta-Object Facility (MOF)

- A **meta-data management framework**
- A language to be used for defining languages
  - i.e., it is an OMG-standard meta-modelling language.
  - The UML metamodel is defined in MOF.
- **MOF 2.0 shares a common core with UML 2.0**
  - Simpler rules for modelling metadata
  - Easier to map from/to MOF
  - Broader tool support for metamodeling (i.e., any UML 2.0 tool can be used)

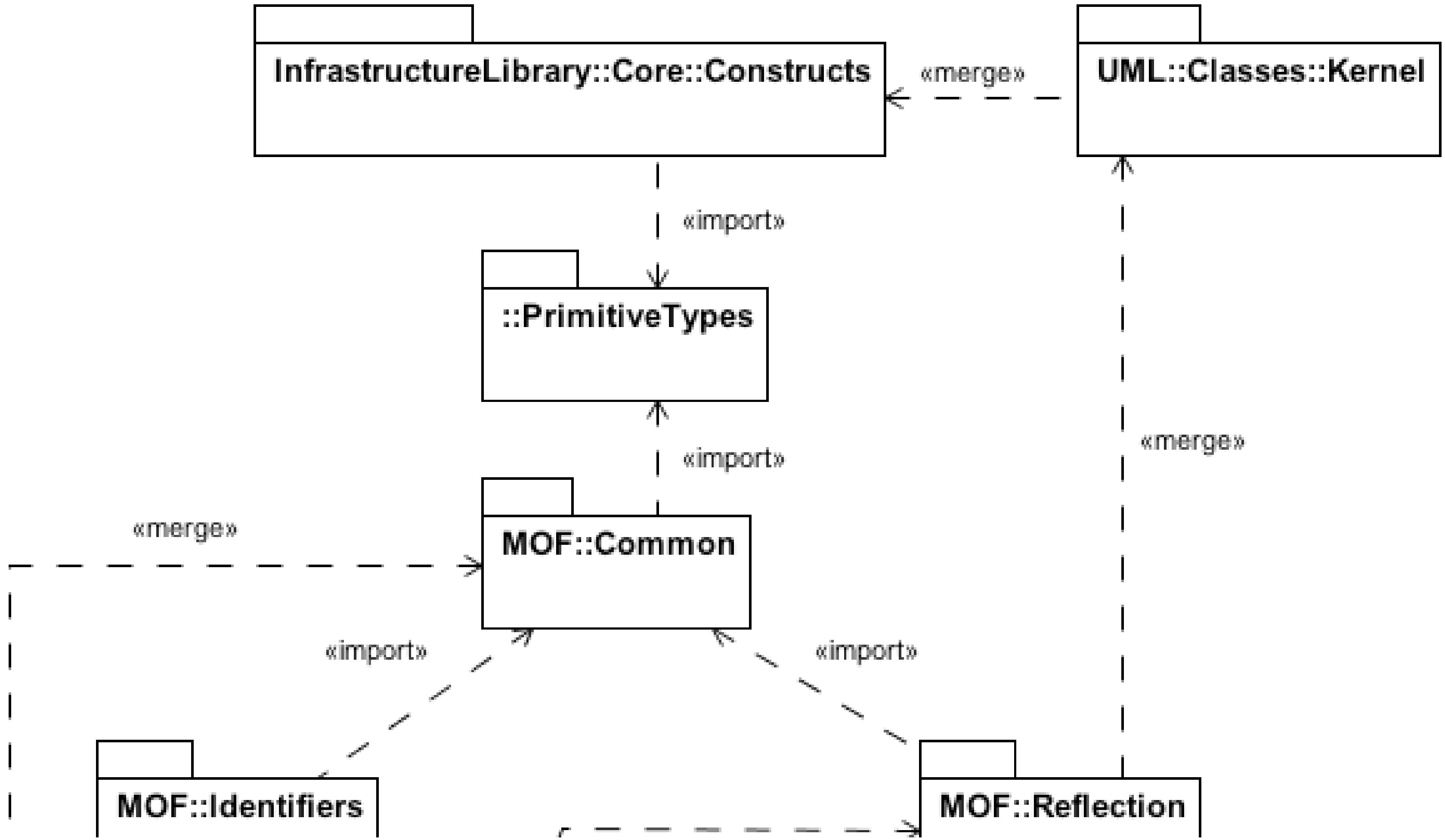
- MOF has **evolved** through several versions
  - MOF 1.x is the one most widely supported by tools
  - MOF 2.0 is the current standard, and it has been substantially influenced by UML 2.0
  - MOF 2.0 is also critical in supporting transformations, e.g., QVT and Model-to-text

http://www.omg.org/spec/MOF/2.0

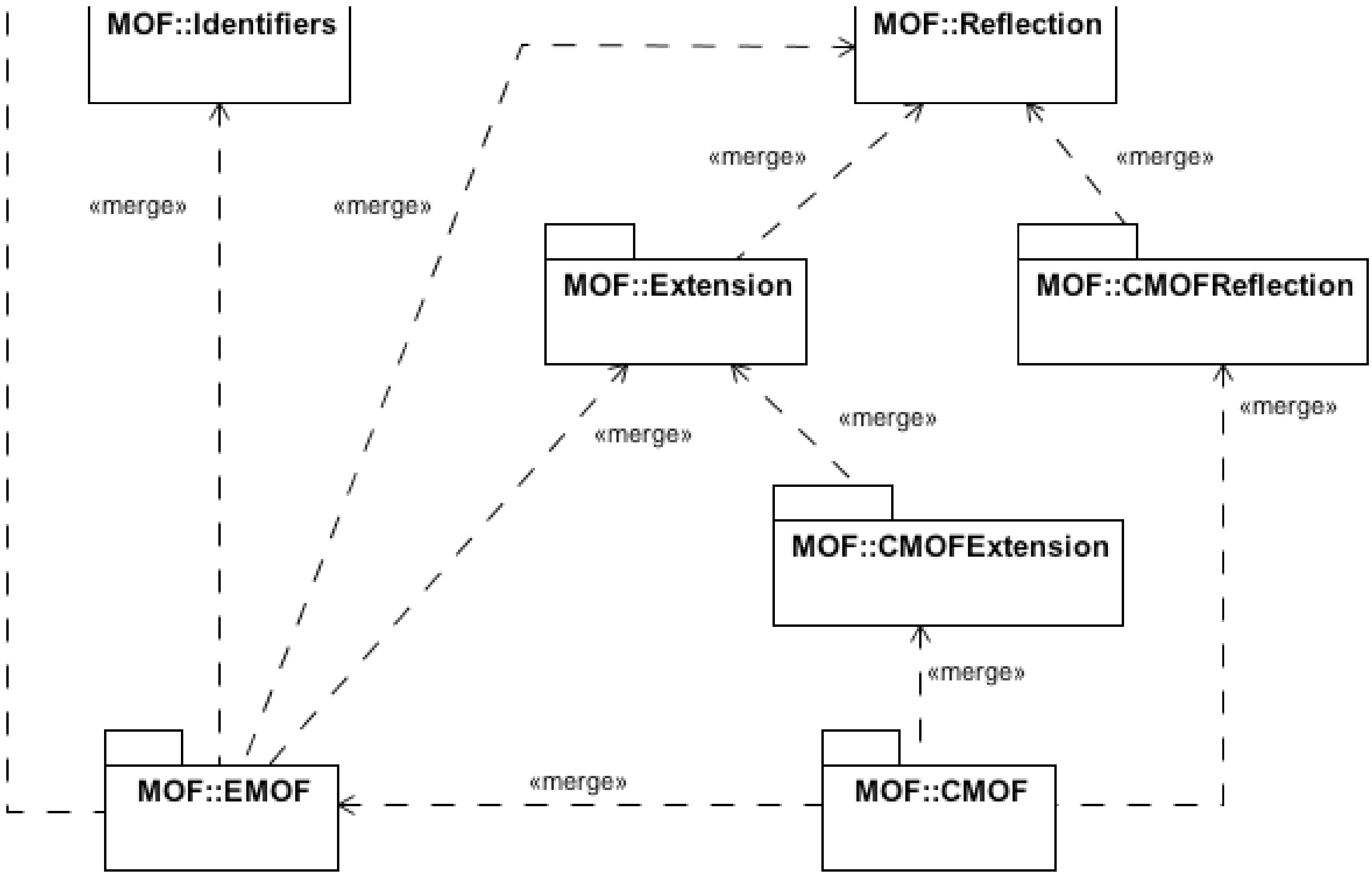
MOF 2.0 Structure

- MOF is separated into **Essential MOF** (**EMOF**) and **Complete MOF** (**CMOF**)
- EMOF corresponds to facilities found in OOP and XML.
  - Easy to map EMOF models to JMI, XMI, etc.
- CMOF is what is used to specify metamodels for languages such as UML 2.
  - It is built from EMOF and the core constructs of UML 2.
  - Both EMOF and CMOF are based on variants of UML 2.

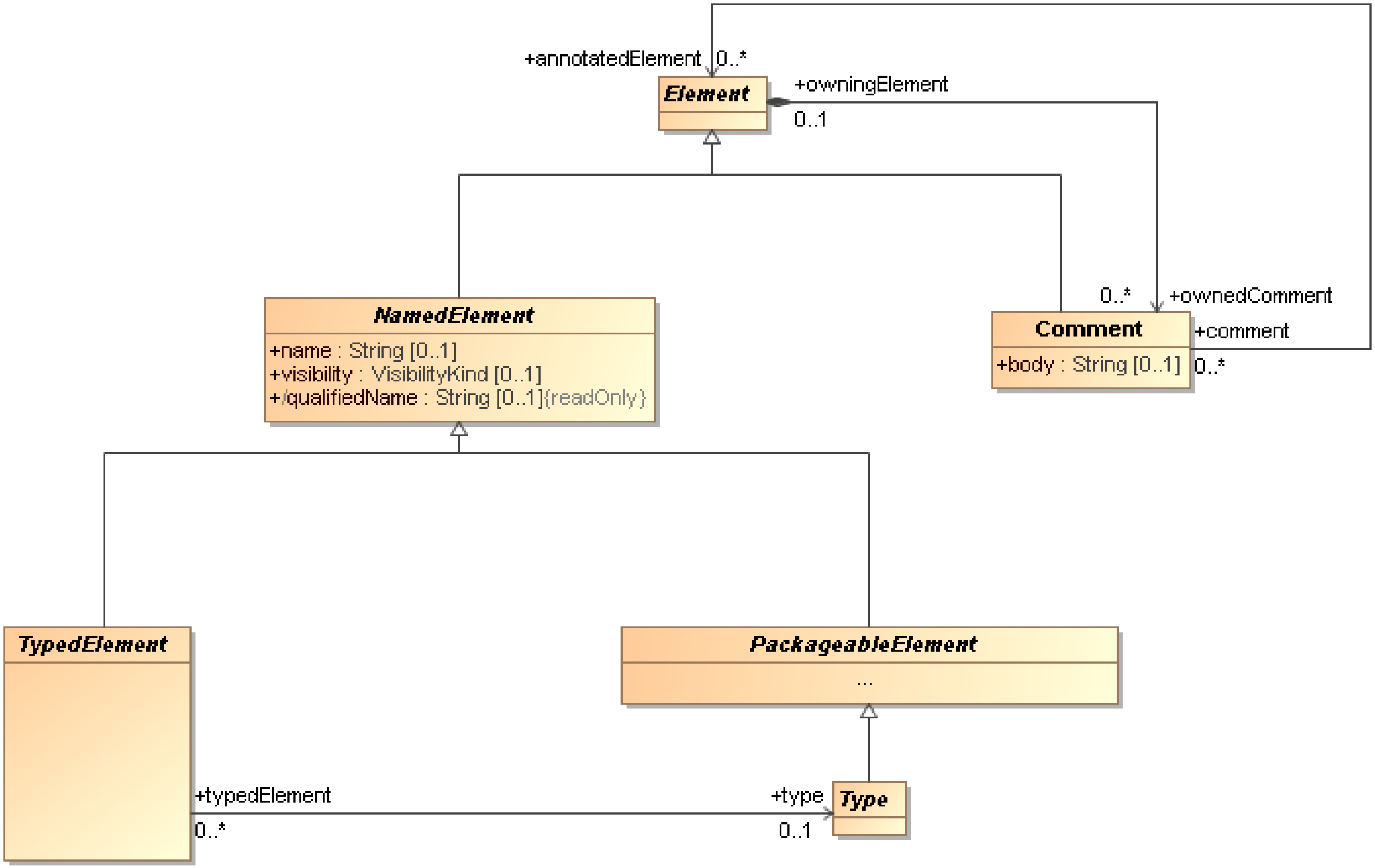
MOF 2.0 Relationships (1)



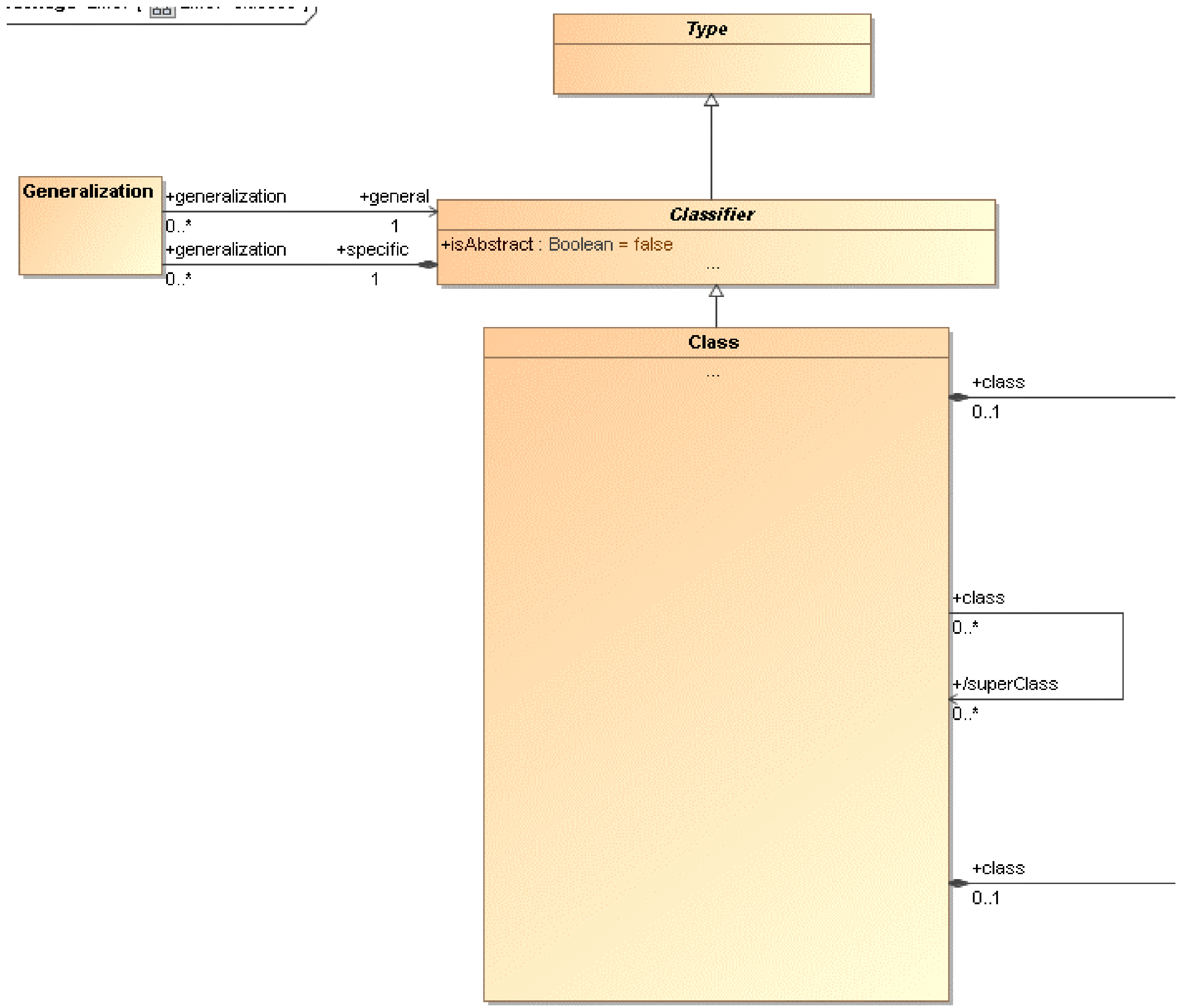
MOF 2.0 Relationships (2)



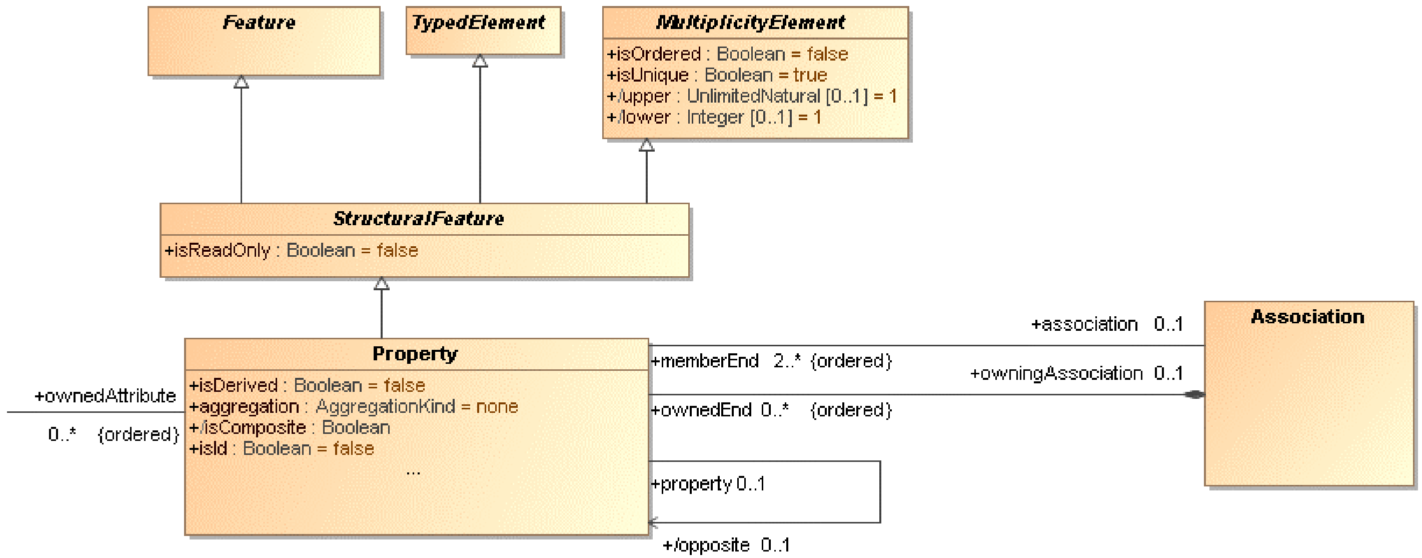
EMOF Types — merged from UML Infrastructure

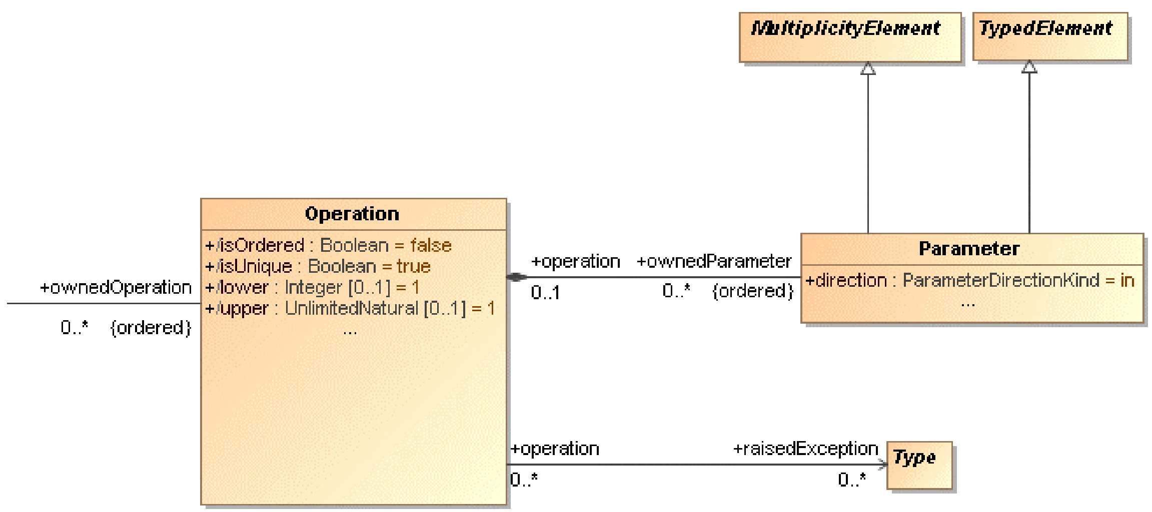


EMOF Classes — merged from UML Infrastructure (1)

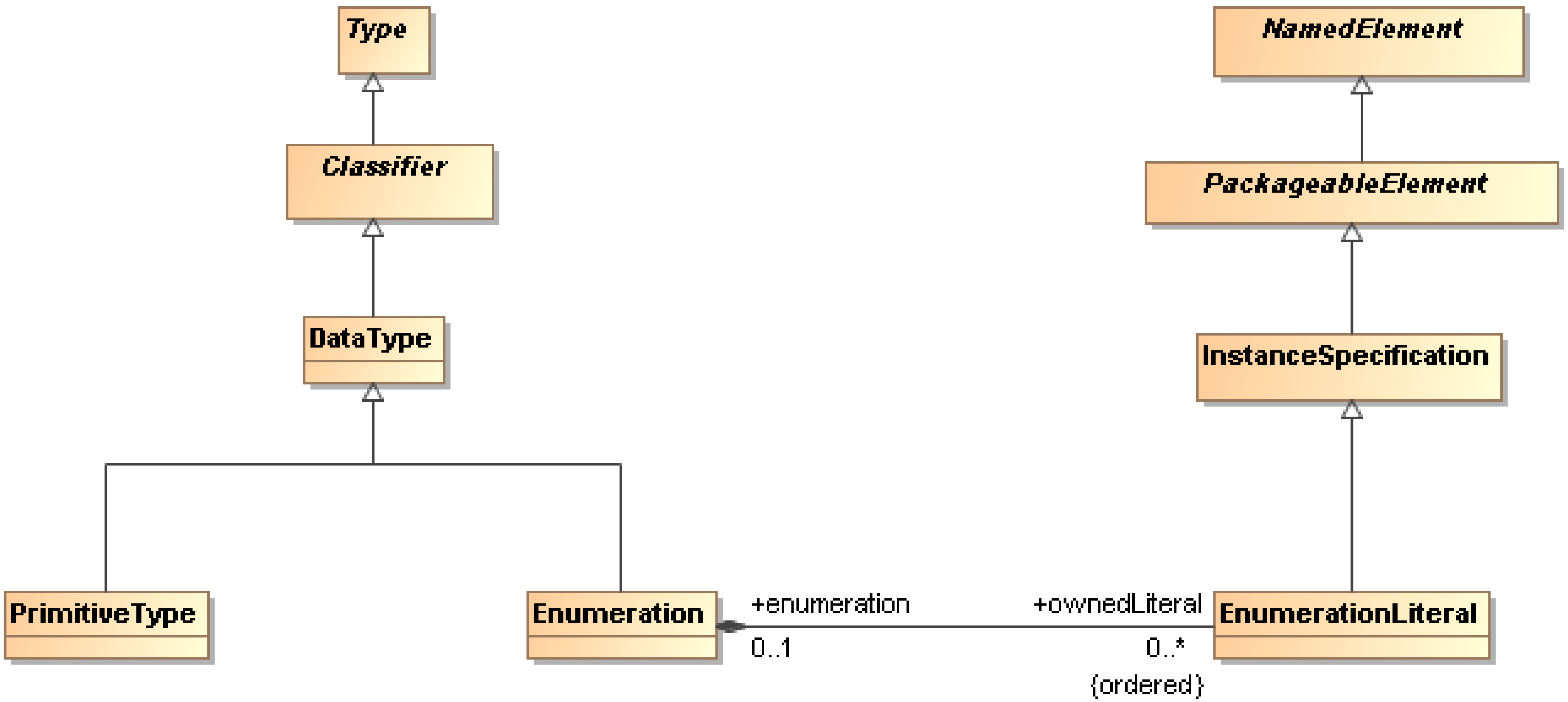


EMOF Classes — merged from UML Infrastructure (2)

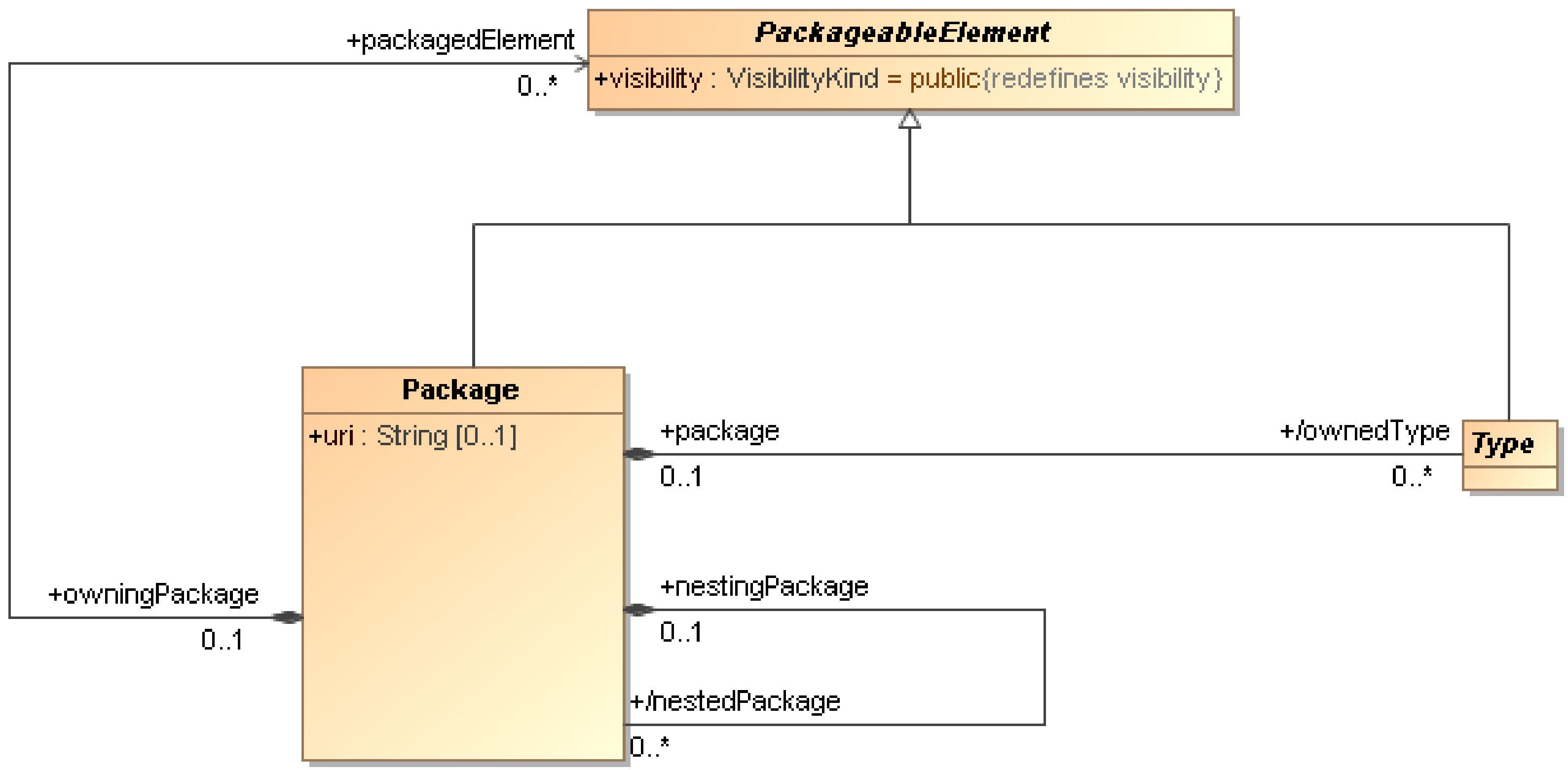




EMOF Data Types — merged from UML Infrastructure



EMOF Packages — merged from UML Core:Basic



XML Metadata Interchange (XMI)

- XMI is a **standard** (and a trademark) from the OMG.
- XMI is a **framework** for
  - defining, interchanging, manipulating and integrating XML data and objects.
- Used for **integration**
  - tools, applications, repositories, data warehouses
  - typically used as interchange format for UML tools

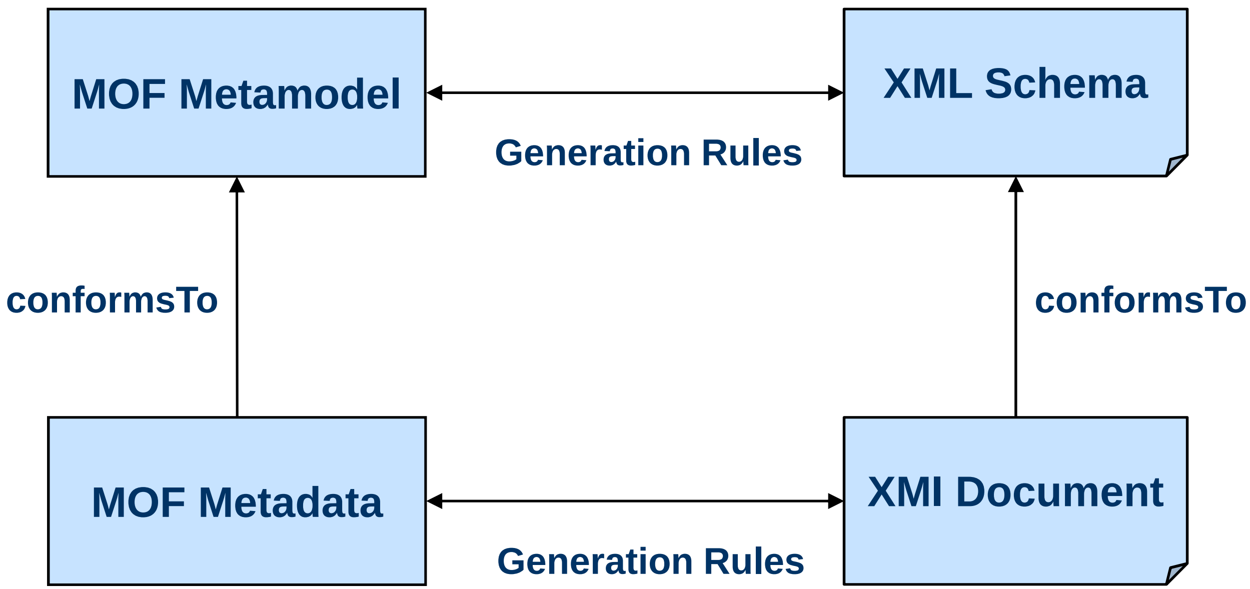
- XMI defines **rules for schema definition**
  - schema production — how is a metamodel mapped onto a grammar?
  - definition of schema from any valid Meta Object Facility (MOF) model
- XMI defines **rules for metadata generation**
  - document production — how is a model mapped onto text?
  - Metadata according to a MOF metamodel is generated into XML according to the generated XML schema.

http://www.omg.org/spec/XMI/2.4.1/

XMI versions and MOF versions

- XMI 1.1 corresponds to MOF 1.3
- XMI 1.2 corresponds to MOF 1.4
- XMI 1.3 (added schema support) corresponds to MOF 1.4
- XMI 2.0 (adds schema support and changes document format) corresponds to MOF 1.4
- **XMI 2.1** corresponds to **MOF 2.0**
- XMI 2.4.1 corresponds to MOF 2.4.1

MOF and XMI


MOF Metamodel
XML Schema
Generation Rules
conformsTo
conformsTo
MOF Metadata
XMI Document
Generation Rules