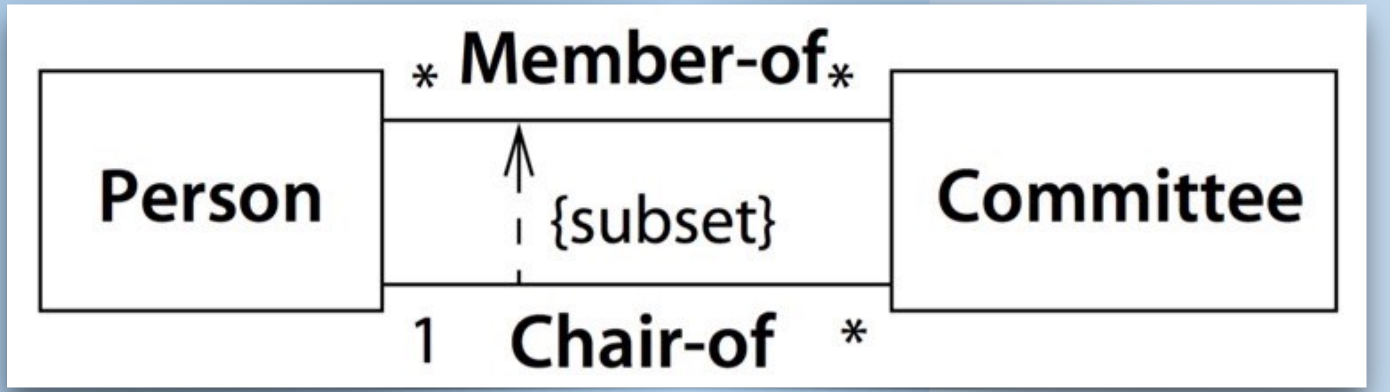
Introduction to Software Engineering

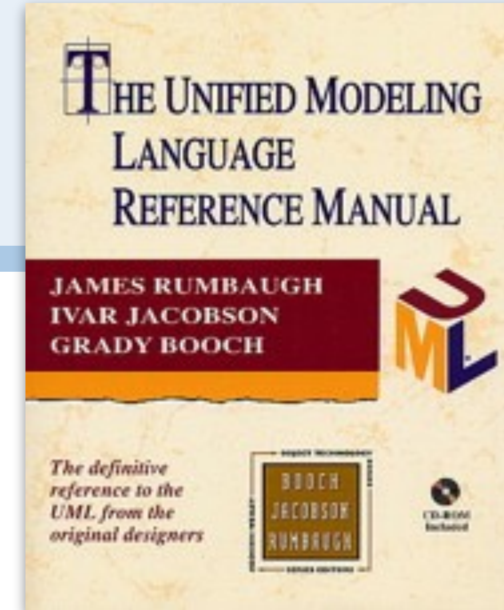
Modeling Objects and Classes



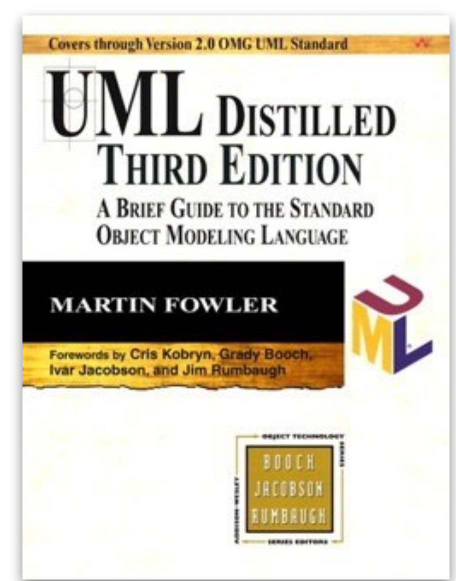
Roadmap

> UML Overview
> Classes, attributes and operations
> UML Lines and Arrows
> Parameterized Classes, Interfaces and Utilities
> Objects, Associations
> Inheritance
> Constraints, Patterns and Contracts

Sources



- *The Unified Modeling Language Reference Manual*, James Rumbaugh, Ivar Jacobson and Grady Booch, Addison Wesley, 2005, 2nd edition.
- *UML Distilled*, Martin Fowler, Kendall Scott, Addison-Wesley, Second Edition, 2003, 3rd edition.



The reference manual by the “three amigos” contains all the gritty details.

Fowler’s highly recommended book, on the other hand, is much shorter and contains much practical advice on how to apply UML.

- http://scgresources.unibe.ch/Literature/Books/Rumb05aUMLreference.pdf
- http://scgresources.unibe.ch/Literature/Books/Fowl03a-UMLDistilled.pdf

NB: these links are not accessible outside the unibe.ch domain.

Roadmap

> **UML Overview**
> Classes, attributes and operations
> UML Lines and Arrows
> Parameterized Classes, Interfaces and Utilities
> Objects, Associations
> Inheritance
> Constraints, Patterns and Contracts

UML

What is UML?

> uniform notation: Booch + OMT + Use Cases (+ state charts)
>   — UML is *not* a method or process
>   — … The *Unified Development Process* is

Why a Graphical Modeling Language?

> Software projects are carried out in *team*
> Team members need to *communicate*
>   — ... sometimes even with the end users
> "One picture conveys a thousand words"
>   — the question is only *which words*
>   — Need for *different views* on the same software artifact

The UML is a collection of notations, originally based on work by the “three amigos”, Grady Booch (who developed the Booch notation), James Rumbaugh (who developed OMT with colleagues at General Electric), and Ivar Jacobsen (who developed the Use Case driven methodology). UML was a fusion of these three notations, and later incorporated other diagramming techniques.

https://en.wikipedia.org/wiki/Unified_Modeling_Language#Before_UML_1.x

It is important note that, before UML, literally *hundreds* of different and incompatible object-oriented design notations were developed. UML was an attempt to bring order to this chaos.

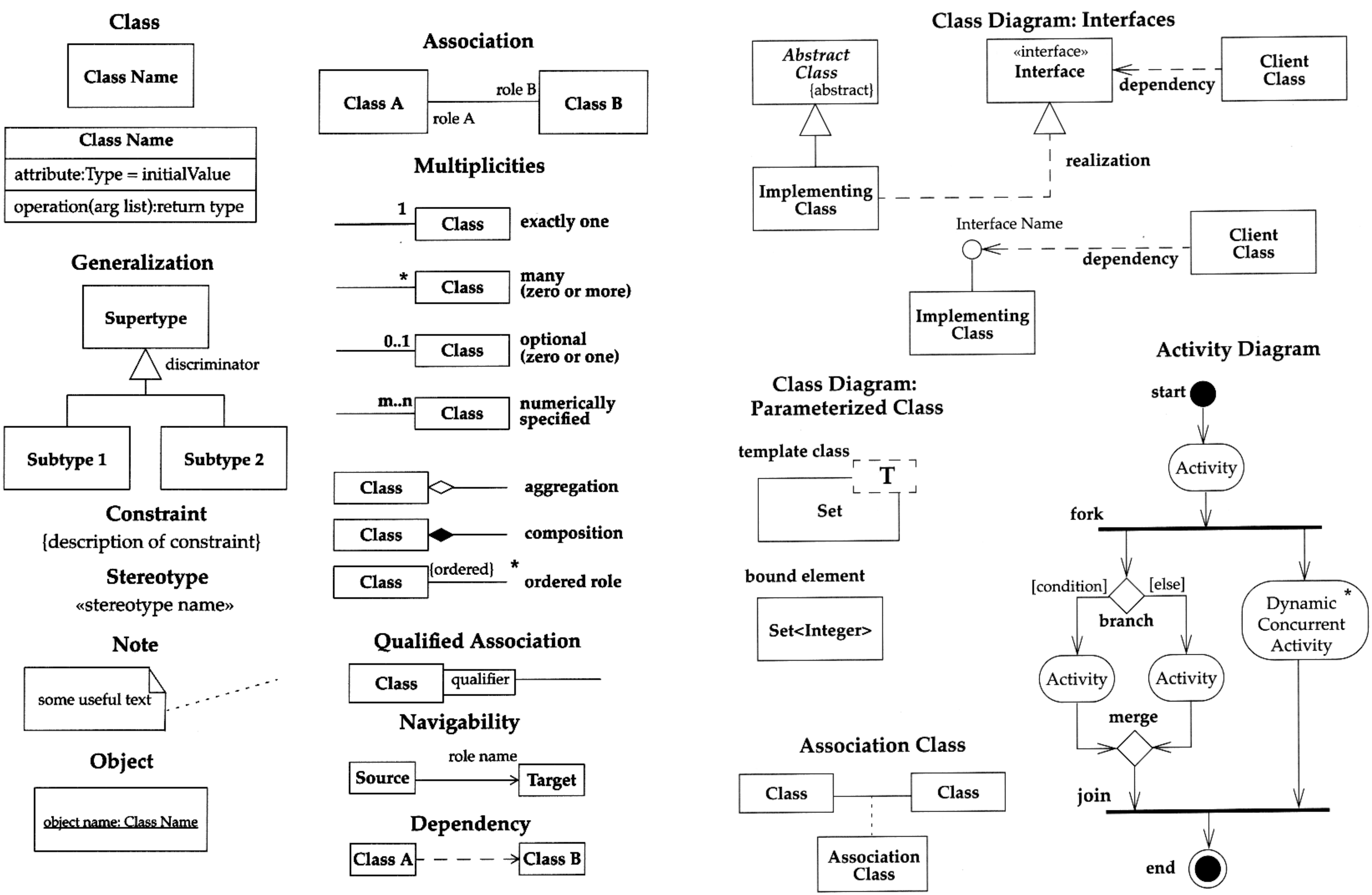
Why UML?

Why UML?

- Reduces *risks* by documenting assumptions
  - —domain models, requirements, architecture, design, implementation …
- Represents industry *standard*
  - —more tool support, more people understand your diagrams, less education
- Is reasonably *well-defined*
  - —... although there are interpretations and dialects
- Is *open*
  - —stereotypes, tags and constraints to extend basic constructs
  - —has a meta-meta-model for advanced extensions

UML History

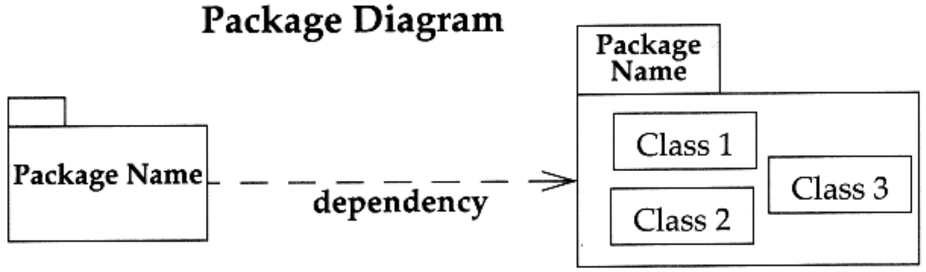
- 1994: Grady Booch (Booch method) + James Rumbaugh (OMT) at Rational
- 1994: Ivar Jacobson (OOSE, use cases) joined Rational
  - "The three amigos"
- 1996: Rational formed a consortium to support UML
- 1997: UML1.0 submitted to OMG by consortium
- 1997: UML 1.1 accepted as OMG standard
  - However, OMG names it UML1.0
- 1998-…: Revisions UML1.2 - 1.5
- 2005: Major revision to UML2.0, includes OCL



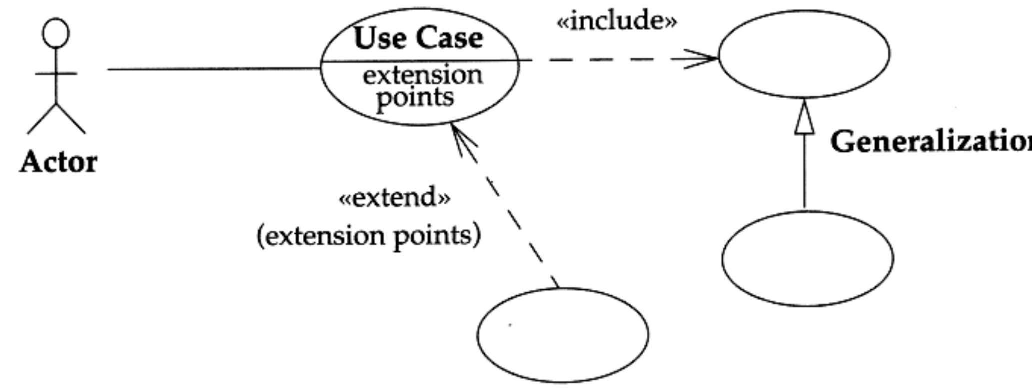
UML Distilled

These two slides are only intended to give a quick overview of the different kinds of notations supported by UML. We will look at many (not all) of them in detail over two lectures.

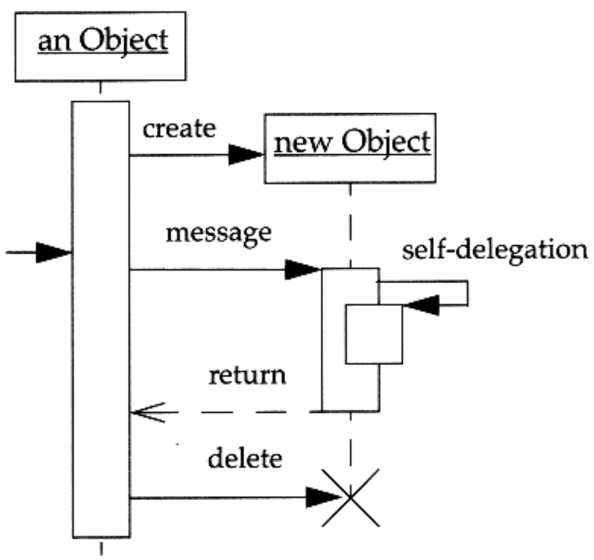
Package Diagram



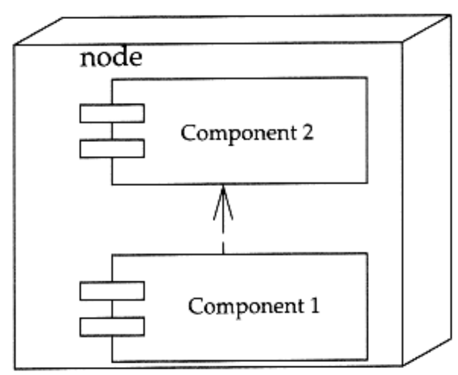
Use Case Diagram



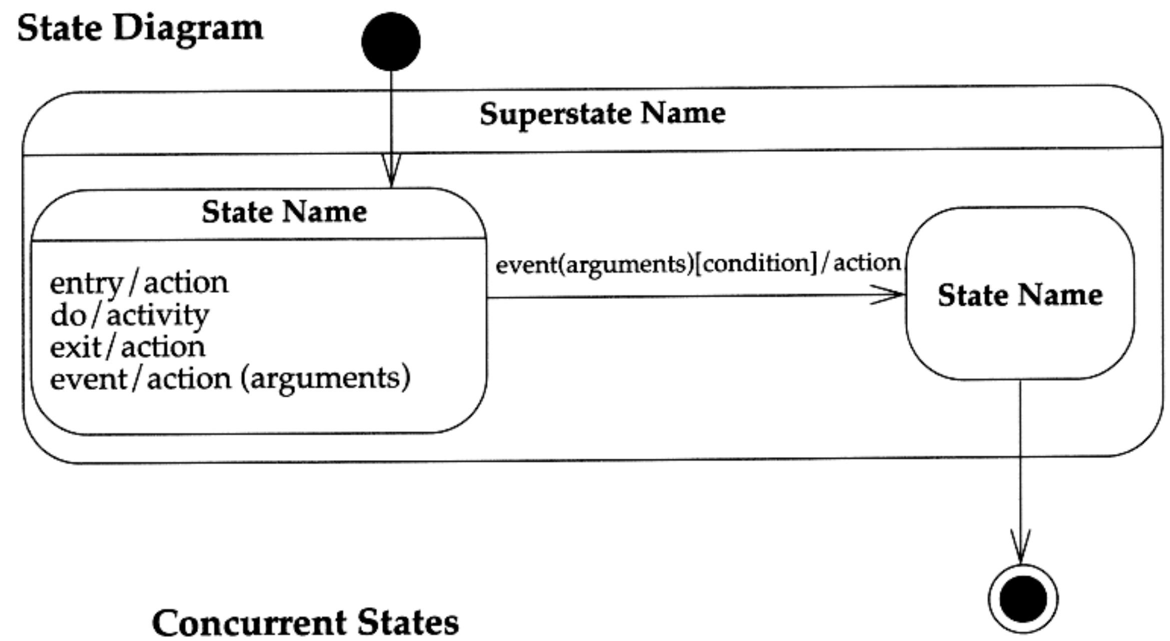
Sequence Diagram



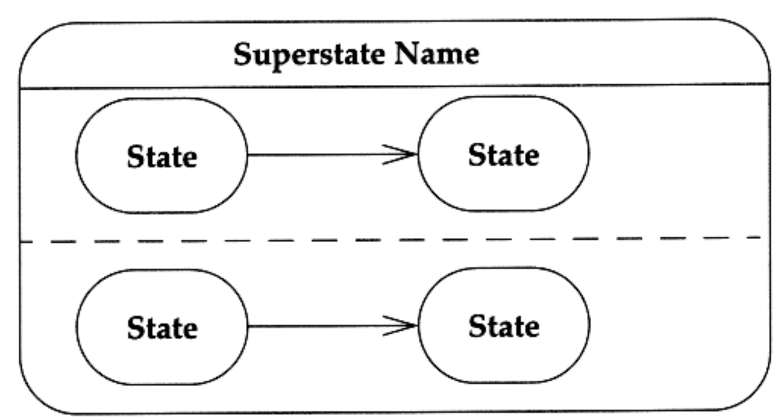
Deployment Diagram



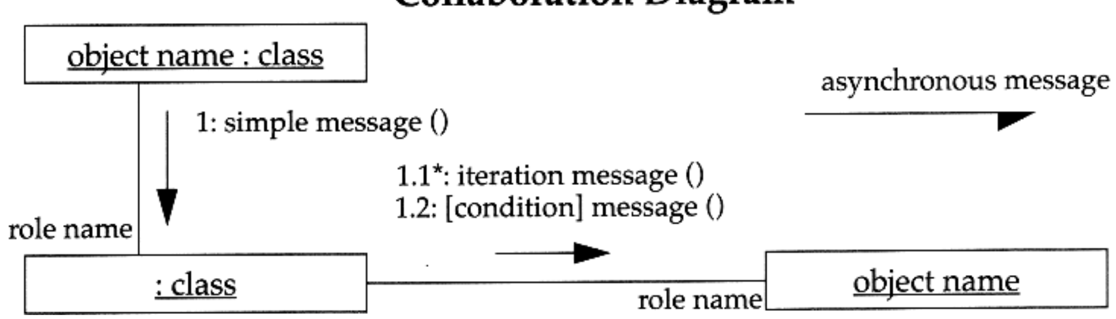
State Diagram



Concurrent States



Collaboration Diagram



Roadmap

- UML Overview
- **Classes, attributes and operations**
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- Constraints, Patterns and Contracts

Class Diagrams

"Class diagrams show generic descriptions of possible systems, and object diagrams show particular instantiations of systems and their behaviour."

Attributes and operations are also collectively called *features*.

Danger: *class diagrams risk turning into data models. Be sure to focus on behaviour*

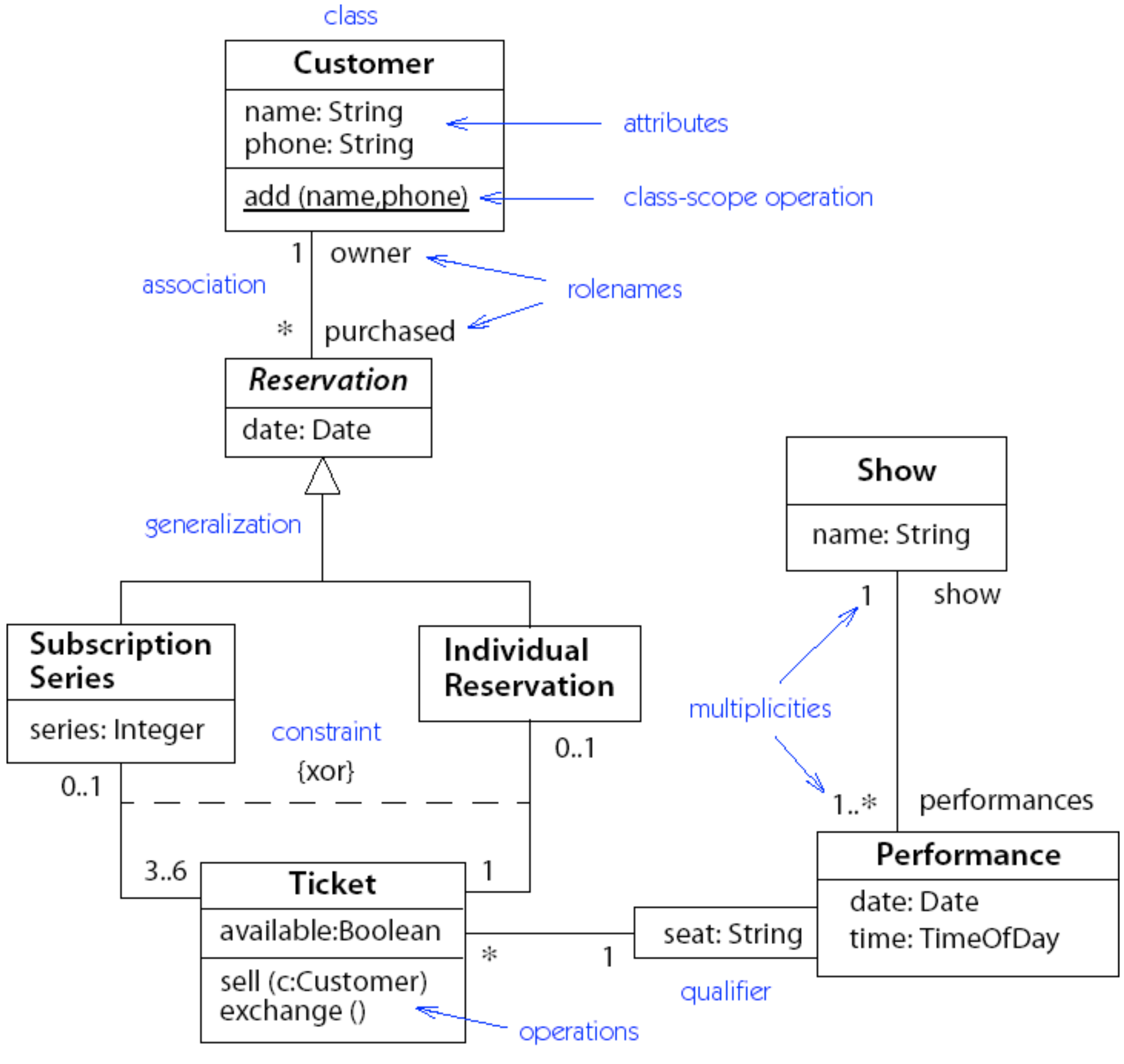


Figure 3-1. *Class diagram*

In the example diagrams (mostly from the UML reference manual), what is shown in **black** is UML; what is shown in **blue** are explanatory annotations (not UML).

Class diagrams only describe classes and their relationships. Object diagrams (seen later) show instances.

Class diagrams can be used to describe:

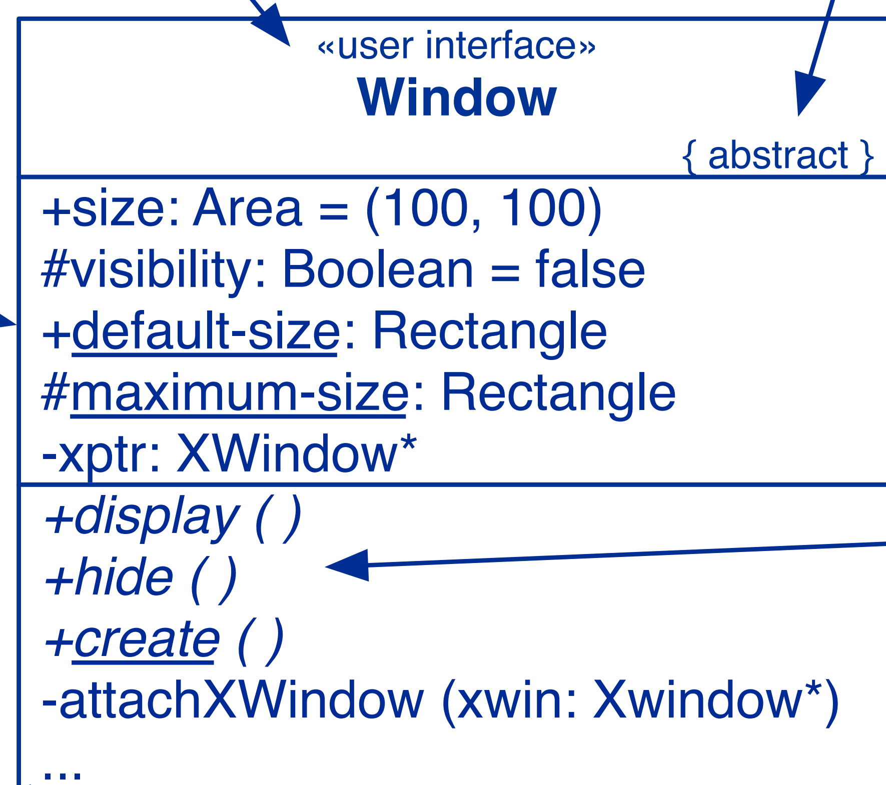
- *domain models*: classes represents concepts from an application domain
- *designs*: classes represent software entities that will be implemented in some programming language
- *implementations*: classes represent actual classes in a specific implementation

Can you read the class diagram? What does it describe?

Visibility and Scope of Features

Stereotype
(what "kind" of class is it?)

User-defined properties
(e.g., readonly, owner = "Pingu")



Don't worry about visibility too early!

underlined attributes have *class scope*

italic attributes are *abstract*

+ = "public"
# = "protected"
- = "private"

An ellipsis signals that further entries are not shown

A class is depicted as a rectangle, with up to three compartments. The top part givens the class name and (optional) «stereotype» (in guillemets) and user properties.

The other two compartments are optional, and depict the attributes of the class and its operations.

Generally in UML diagrams you can choose what you specify depending on what you want to communicate. You are not obliged to list all attributes and operations.

Attributes and Operations

Attributes are specified as:

name: type = initialValue { property string }

Operations are specified as:

name (param: type = defaultValue, ...) : resultType

You are also not obliged to indicate the visibility of features or their types.

At a minimum you can just list their names.

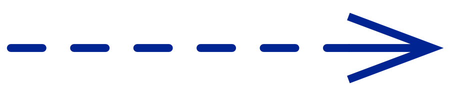
Roadmap

> UML Overview
> Classes, attributes and operations
> **UML Lines and Arrows**
> Parameterized Classes, Interfaces and Utilities
> Objects, Associations
> Inheritance
> Constraints, Patterns and Contracts

UML Lines and Arrows

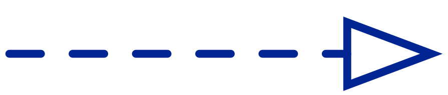
Constraint
(usually annotated)

Association
e.g., «uses»



Dependency
e.g., «requires»,
«imports» ...

Navigable
association
e.g., part-of



Realization
e.g., class/template,
class/interface

"Generalization"
i.e., specialization (!)
e.g., class/superclass,
concrete/abstract class

Aggregation
i.e., "consists of"

"Composition"
i.e., containment

UML is generally very consistent in terms of how its lines and arrows are used across the different diagram notations. A *solid line* represents some kind of *relationship*, while a *dashed line* represents a *constraint*. Arrows generally start from a *client* (tail) and point to a *supplier* (head). An “inheritance” arrow therefore goes from the subclass (client) to its superclass (supplier of inherited features) and not the other way around.

Lines and arrows may be annotated in various ways. In particular, the endpoints may be labeled with names to indicate the *role* of the entity at that end, and with cardinalities.

A simple dashed line represents a constraint (e.g., contains, owns)

A dashed arrow indicates a dependency (a special kind of constraint) from a client to a supplier.

A dashed arrow with a solid head indicates a realization or implementation: a class implements a type, an interface or a generic class.

A solid line is a regular association relationship, usually annotated.

A solid arrow indicates that the relationship can be navigated (without necessarily specifying how). An arrow from A to B means that, if you have an A, you can always get it its B.

A “generalization” arrows means that the client is subclass of its supplier. The subclass specializes its superclass, or the superclass generalizes its subclass. A Person (superclass) is more general than an Employee (subclass).

Aggregation and composition are two kinds of “part of” relationships, discussed later.

Roadmap

> UML Overview
> Classes, attributes and operations
> UML Lines and Arrows
> **Parameterized Classes, Interfaces and Utilities**
> Objects, Associations
> Inheritance
> Constraints, Patterns and Contracts

Parameterized Classes

Parameterized (aka “template” or “generic”) classes are depicted with their parameters shown in a *dashed box*.

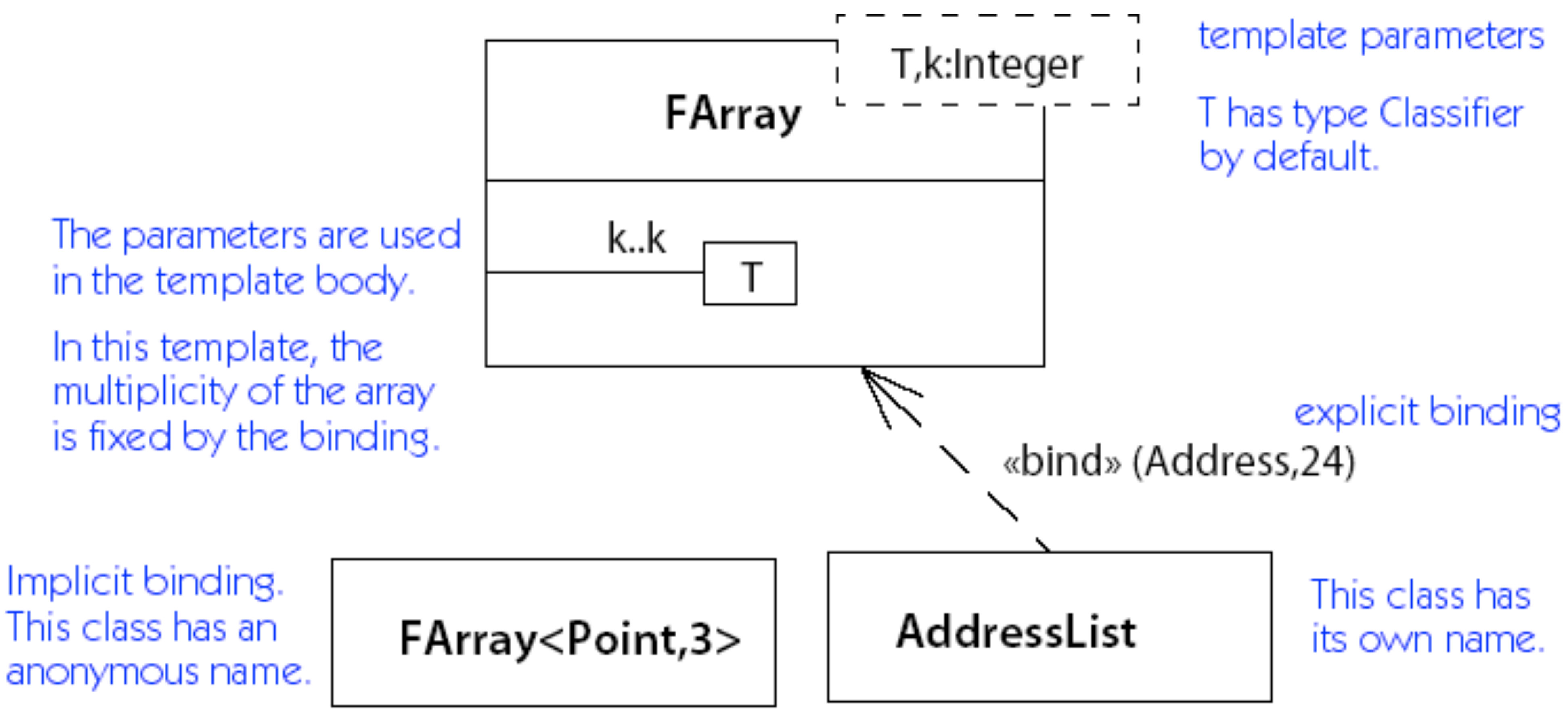


Figure 13-180. *Template notation with use of parameter as a reference*

`FArray` appears to be a fixed array of a parameter type `T`. To instantiate it both parameters `T` and `k` must be specified.

We see here two ways of specifying the parameters, either using the dashed box, or with the embedded box `T` with the cardinality `k..k` (indicating exactly `k` items).

We also see two ways of instantiating a parameterized class, either as `FArray<Point,3>`, or as the class `AddressList` with a realization arrow annotated with the binding of parameters to values.

Interfaces

Interfaces, equivalent to abstract classes with no attributes, are represented as classes with the stereotype «interface» or, alternatively, with the “Lollipop-Notation”:

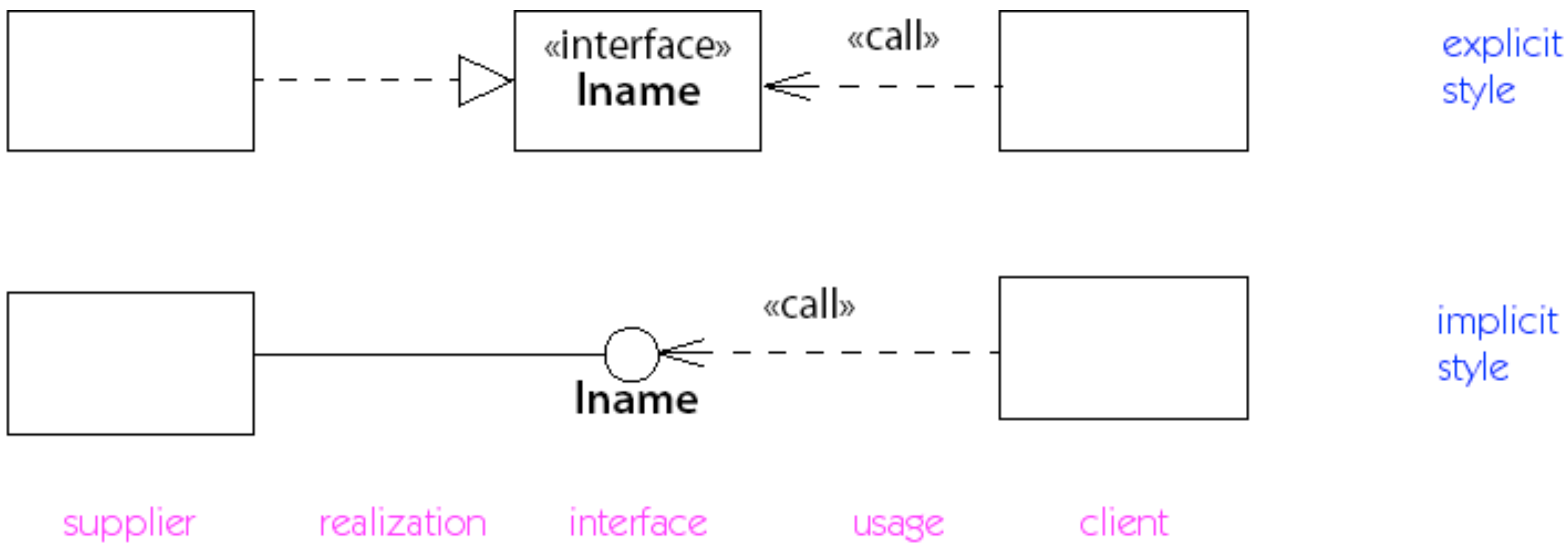
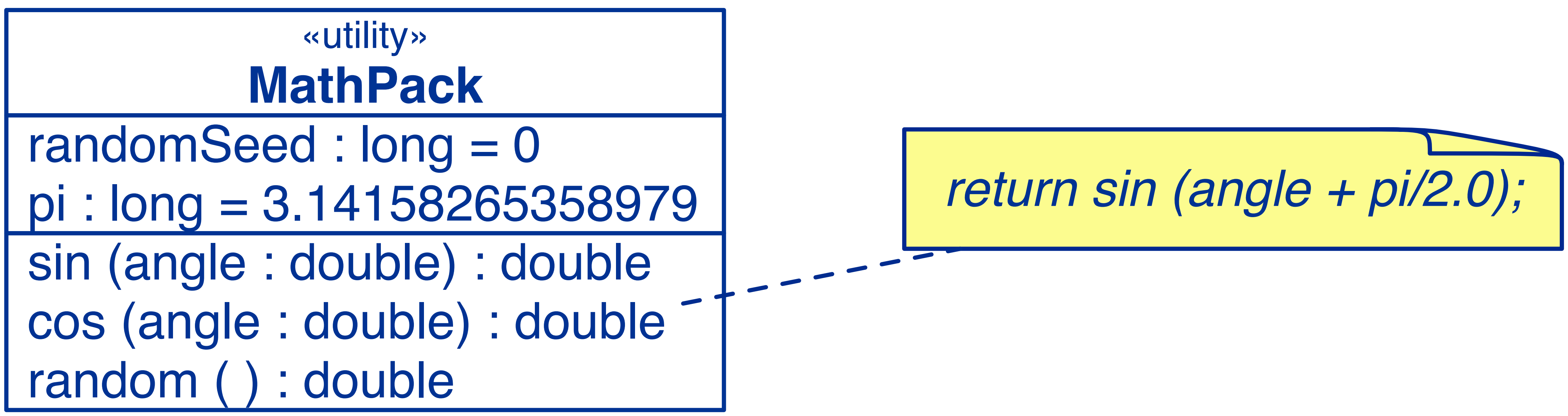


Figure B-5. *Realization of an interface*

Here we see two ways of representing interfaces, on top with a realization arrow from the class to the interface (represented as a stereotypical class), or with the newer “lollipop” notation. The latter is now the standard way to document Java interfaces.

Utilities

A utility is a grouping of global attributes and operations. It is represented as a class with the stereotype «utility». Utilities may be parameterized.



NB: A utility's attributes are already interpreted as being in class scope, so it is redundant to underline them.

A "note" is a text comment associated with a view, and represented as box with the top right corner folded over.

Roadmap

> UML Overview
> Classes, attributes and operations
> UML Lines and Arrows
> Parameterized Classes, Interfaces and Utilities
> **Objects, Associations**
> Inheritance
> Constraints, Patterns and Contracts

Objects

Objects are shown as rectangles with their name and type underlined in one compartment, and attribute values, optionally, in a second compartment.

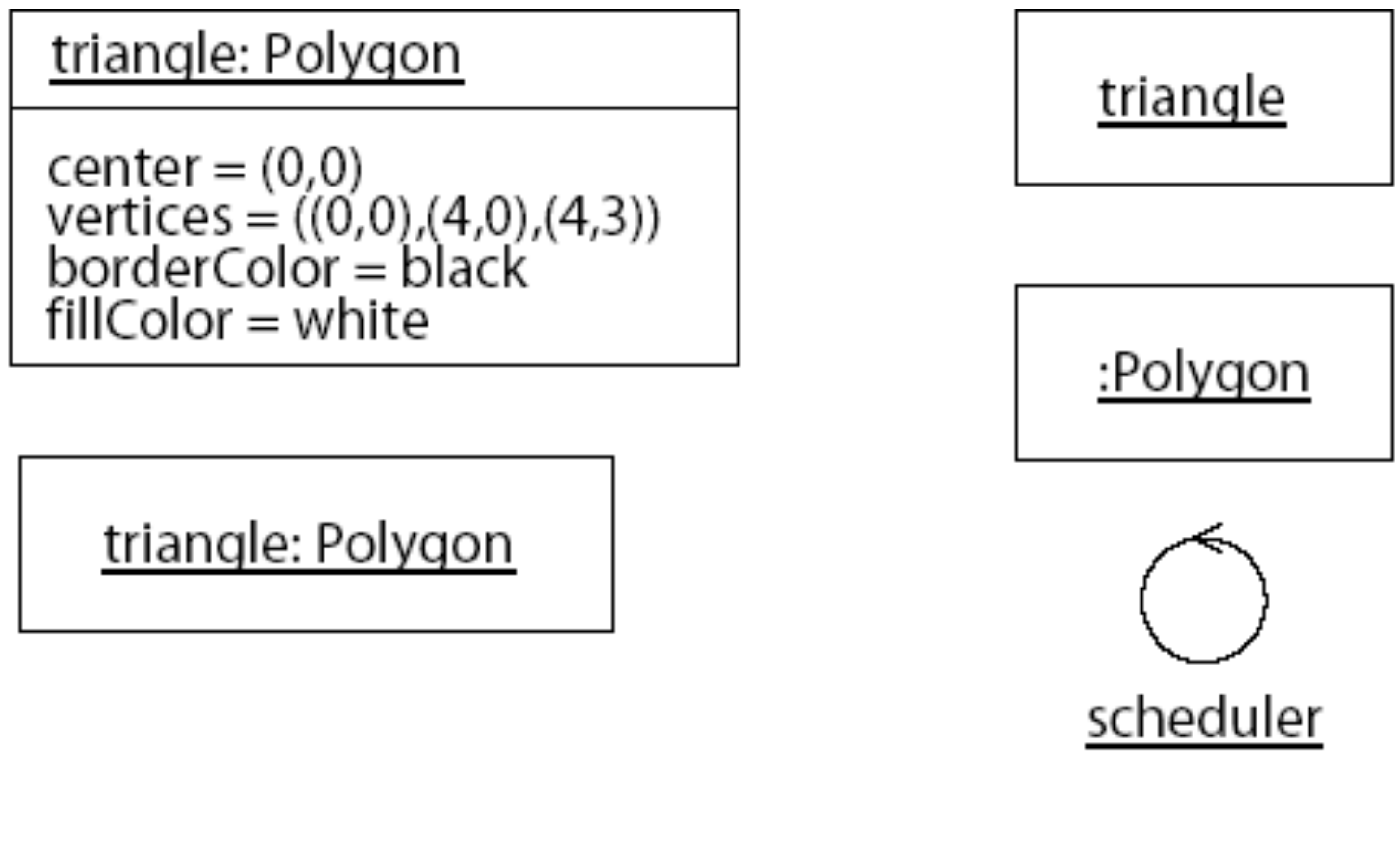


Figure 13-134. *Object notation*

At least one of the name or the type must be present.

In UML, underlined names are either *static* (such as static attributes or operations of a class, that can be accessed without having an instance of that class), or they represent *objects*.

An object is referenced by its (role) name and its class, separated by a colon, as in "`triangle : Polygon`" (the object "`triangle`" is an instance of the class `Polygon`).

It is also possible to leave out either the name or the class, if it is not relevant for the diagram. If the role name is missing, however, then the class name must still be preceded by a colon.

The *cycle icon* represents an instance of a control class (i.e., an "active object" that is constantly performing some action or service).

Associations

Associations represent *structural relationships* between objects

- usually *binary* (but may be ternary etc.)
- optional *name* and *direction*
- (unique) *role names* and *multiplicities* at end-points

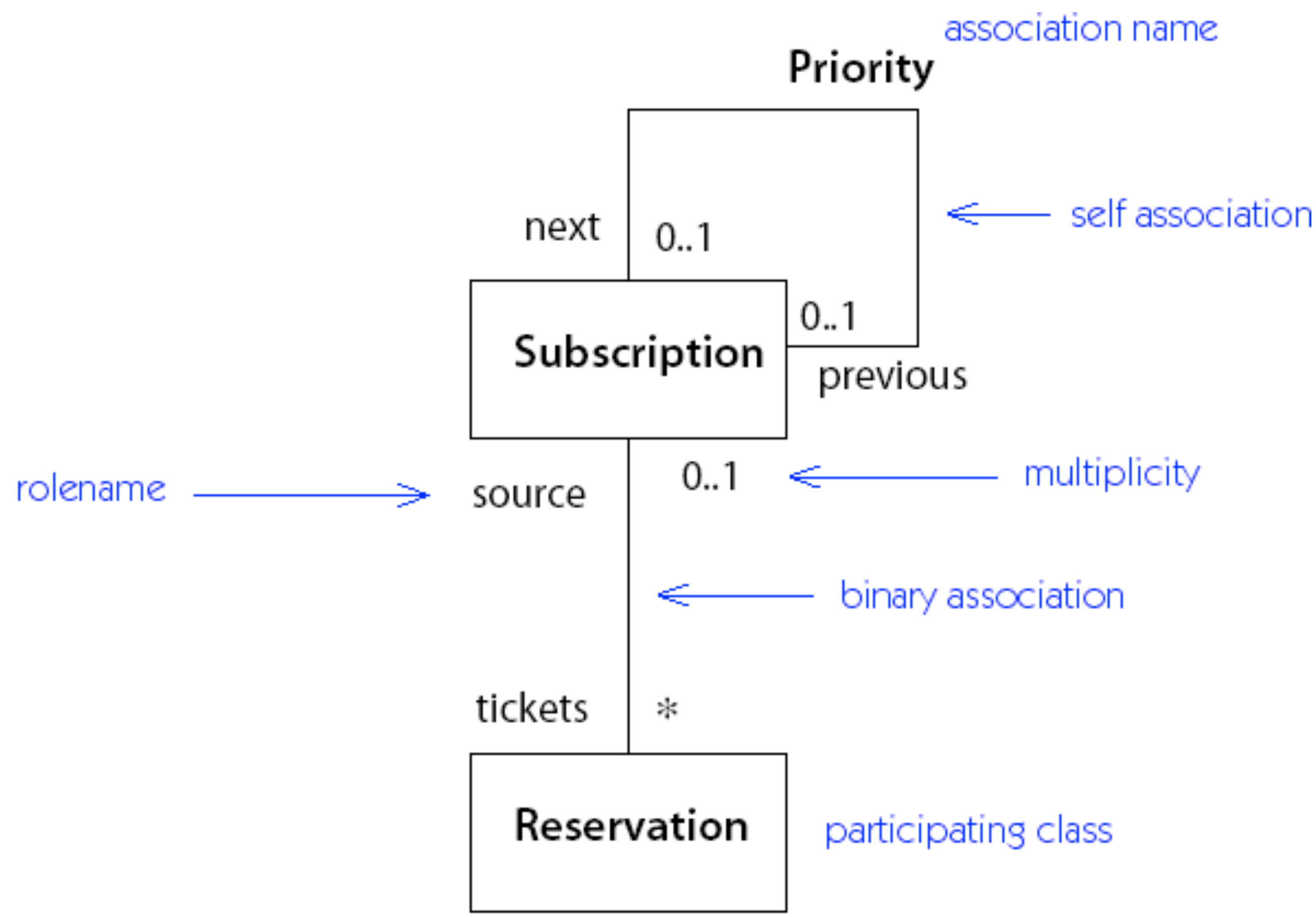


Figure 4-2. *Association notation*

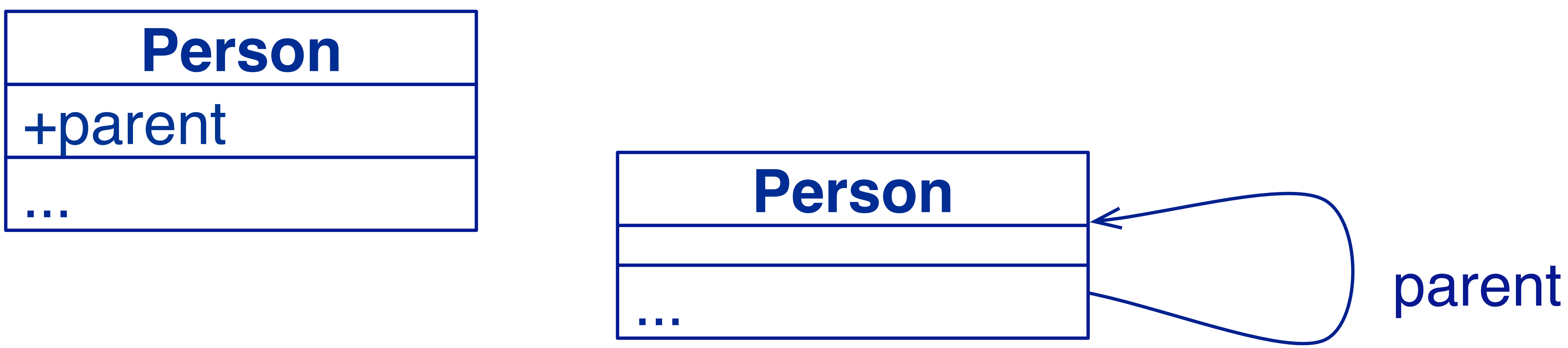
Multiplicity

- The *multiplicity* of an association constrains how many entities one may be associated with
  - Examples:

0..1	Zero or one entity
1	Exactly one entity
*	Any number of entities
1..*	One or more entities
1..n	One to n entities
	And so on …

Associations and Attributes

> Associations may be implemented as attributes
>
> —But need not be …



The first diagram states that parent is (and must be) an attribute of `Person`.

The second diagram merely states that there is a parent relationship between persons, and that one may navigate from a `Person` to that person's parent, but it says nothing about how that relationship is represented. It could be an attribute of `Person`, it could be stored in a third object, or it could be computed from other attributes.

Aggregation and Composition

Aggregation is denoted by a *diamond* and indicates a *part-whole dependency*:

A *hollow diamond* indicates a *reference*; a *solid diamond* an *implementation* (i.e., ownership).

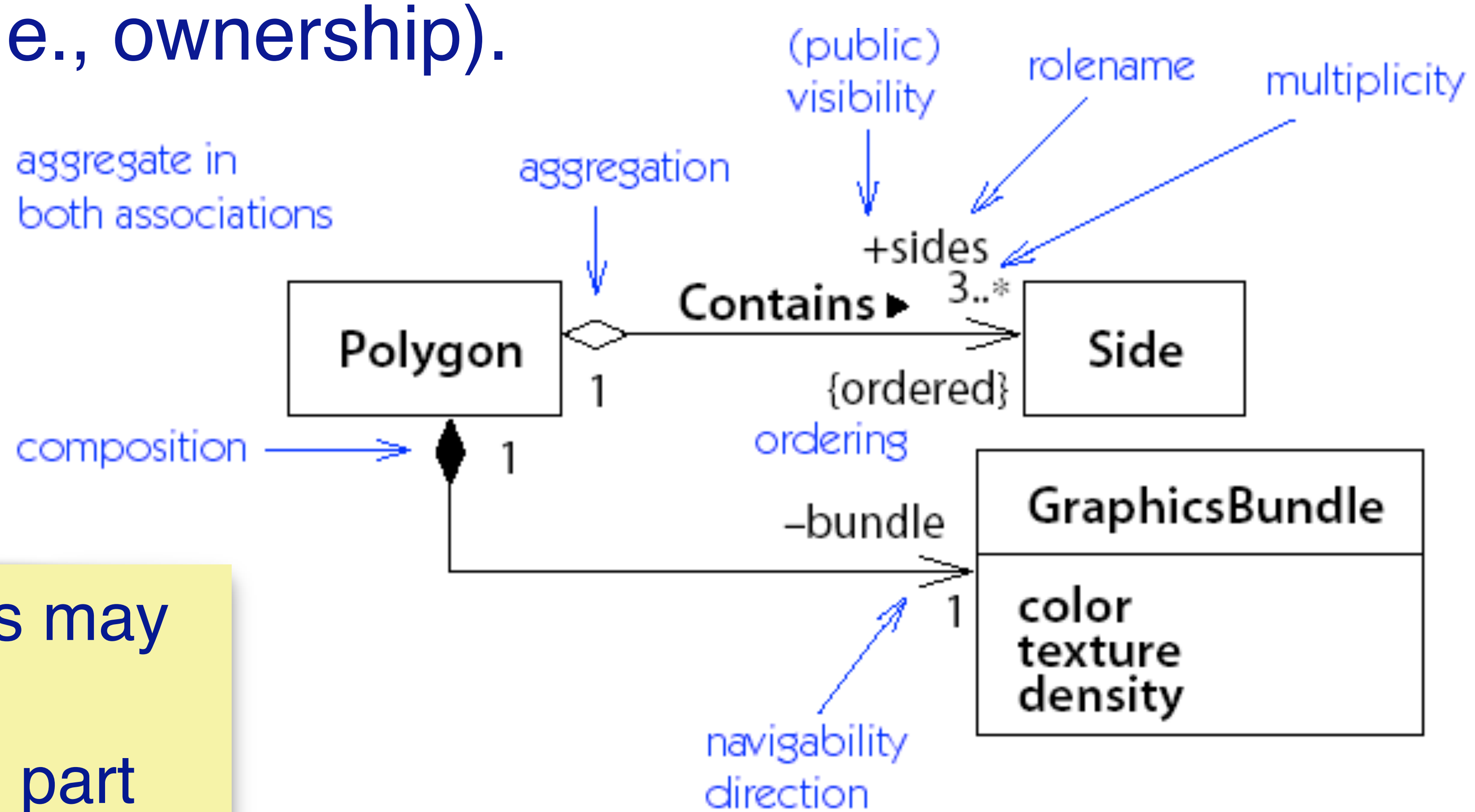


Figure 13-29. *Various adornments on association ends*

Aggregation: parts may be shared.
Composition: one part belongs to one whole.

There are two part-whole relationships in UML. *Aggregation* simply states that one object may contain others, but this relationship may be temporal or even shared. *Composition*, on the other hand, indicated an existential dependency: the part cannot exist without its whole.

In the example a Polygon has several sides, but these may be shared with other polygons. A GraphicsBundle on the other hand is unique to a Polygon and is not shared.

For most purposes the distinction is not critical. Unless you need to make a special point about the existence of parts depending on the whole, *stick to aggregation*.

https://en.wikipedia.org/wiki/Object_composition#UML_notation

Association Classes

An association may be an instance of an association class:

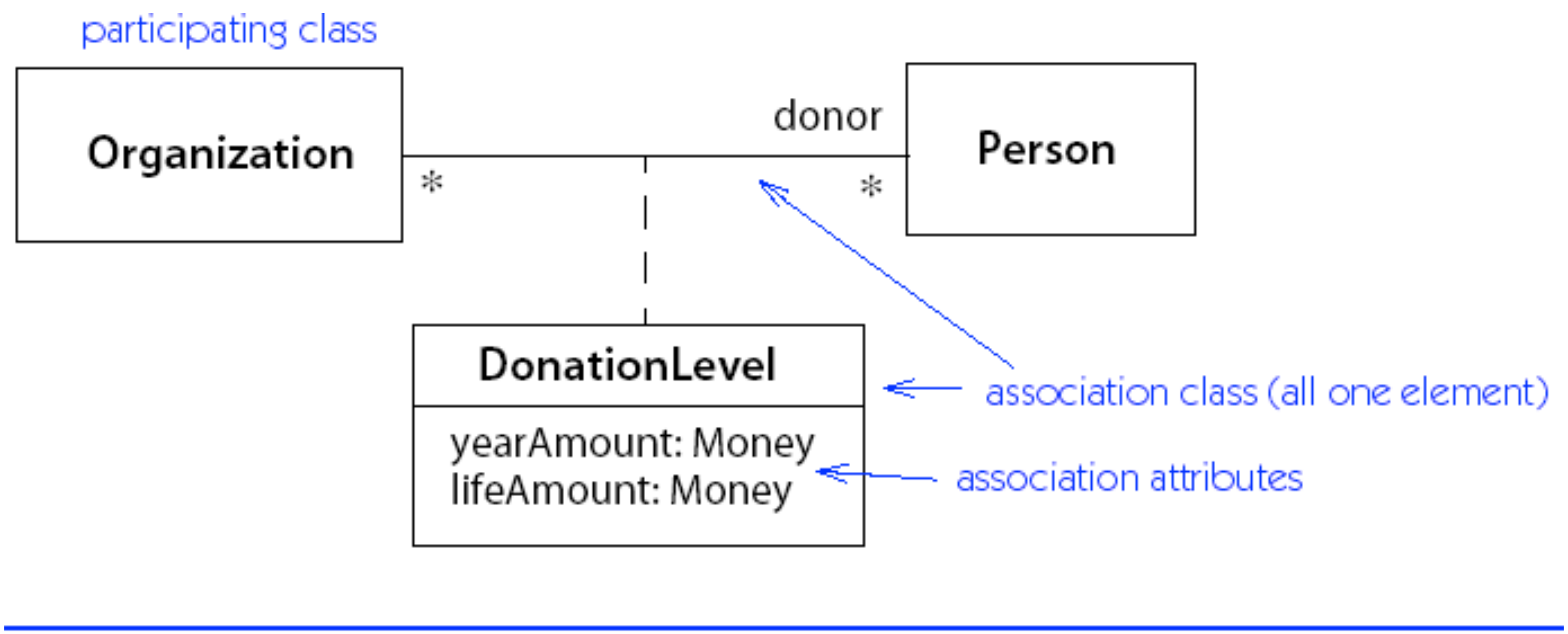


Figure 4-3. *Association class*

In many cases the association class only stores attributes, and its name can be left out.

Sometimes associations between classes may entail additional information or constraints. In such cases the association can be modeled as belonging to an association class.

Qualified Associations

A qualified association uses a special *qualifier value* to identify the object at the other end of the association.

NB: Qualifiers are part of the association, not the class

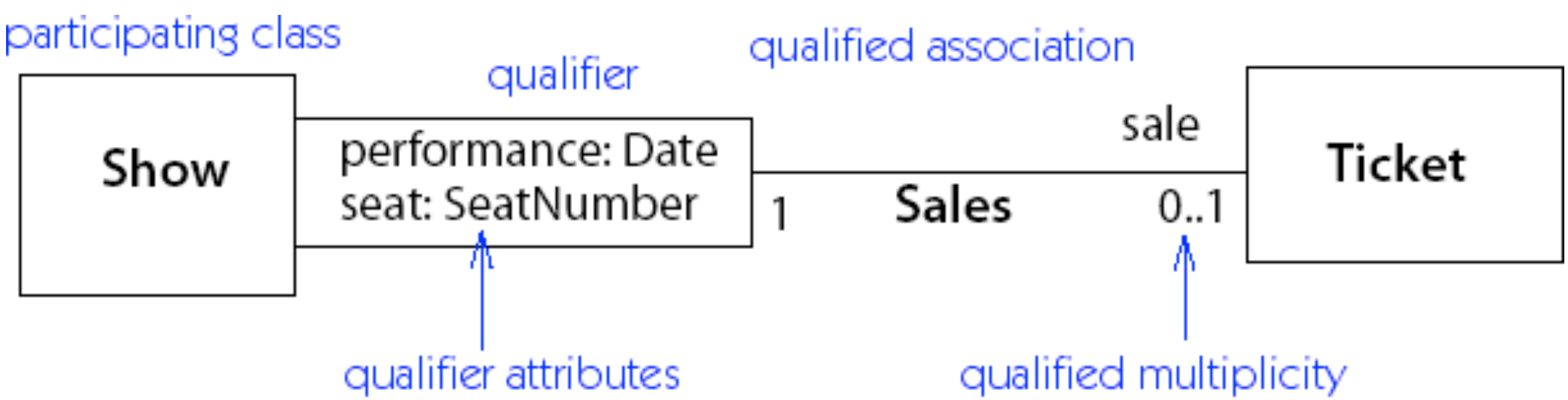


Figure 4-4. *Qualified association*

Roadmap

> UML Overview
> Classes, attributes and operations
> UML Lines and Arrows
> Parameterized Classes, Interfaces and Utilities
> Objects, Associations
> **Inheritance**
> Constraints, Patterns and Contracts

Generalization

A superclass <u>generalizes</u> its subclass.
The subclass <u>specializes</u> its superclass.

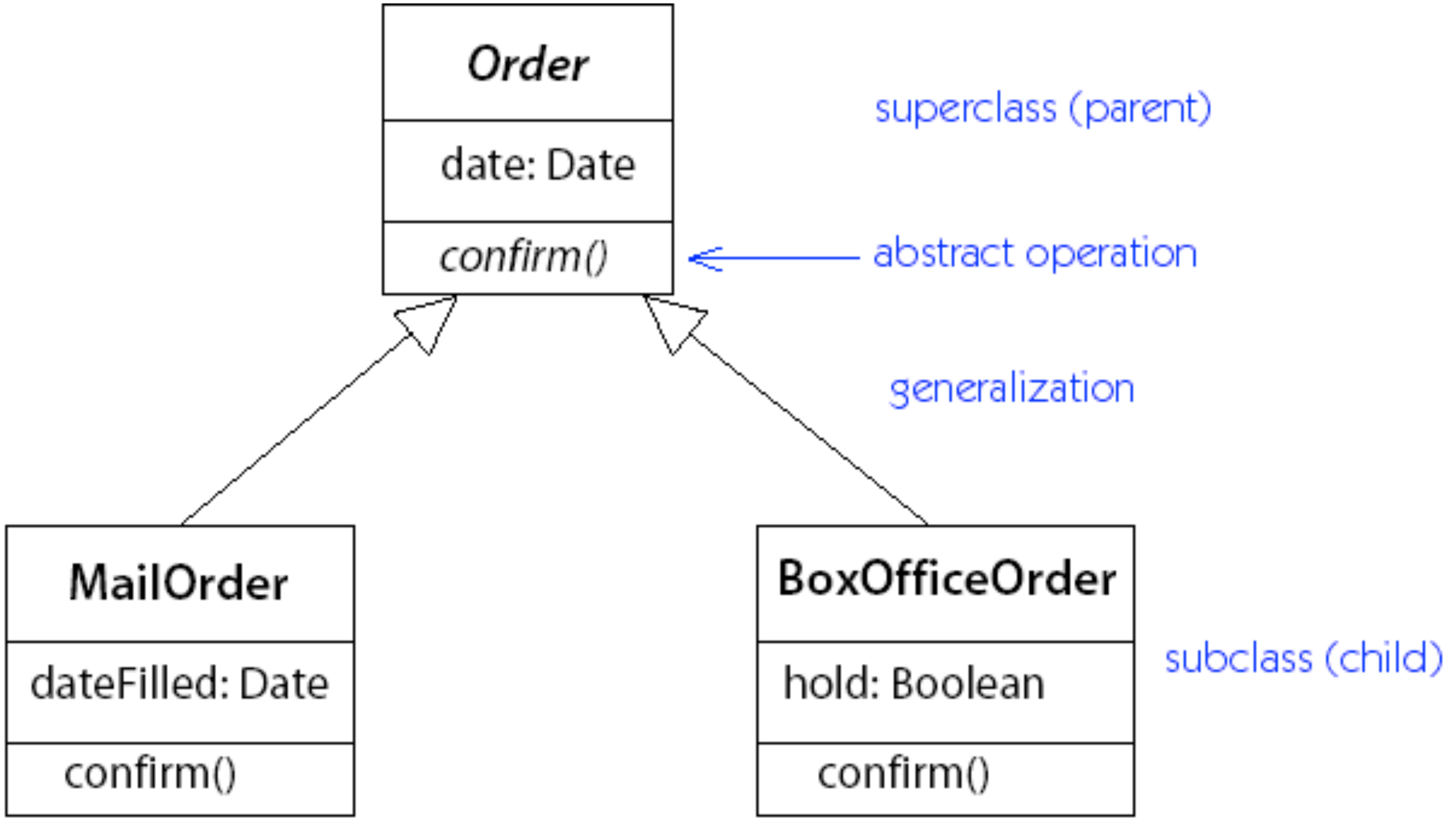


Figure 4-7. *Generalization notation*

Generalization and *specialization* are *dual* concepts. A superclass is more general than its subclasses, and encompasses all its subclasses. Every `MailOrder` is an Order, so `Order` is more general. On the other hand, subclasses specialize their superclasses. A `MailOrder` is a special kind of `Order`, as is a `BoxOfficeOrder`.

Of course we can use UML's notion of generalisation to *model inheritance* in OO languages, but we can also use it as a more general modeling tool that has *nothing to do with inheritance*. (An `Employee` in the real world does not "inherit" anything from `Person`, though it is a more specialized role.)

What is Inheritance For?

- New software often builds on old software by *imitation*, *refinement* or *combination*.

- Similarly, classes may be *extensions*, *specializations* or *combinations* of existing classes.

Generalization expresses ...

Conceptual hierarchy:

> conceptually related classes can be organized into a *specialization* hierarchy
  — people, employees, managers
  — geometric objects ...

Polymorphism:

> objects of distinct, but related classes may be *uniformly treated* by clients
  — array of geometric objects

Software reuse:

> related classes may *share* interfaces, data structures or behaviour
  — geometric objects ...

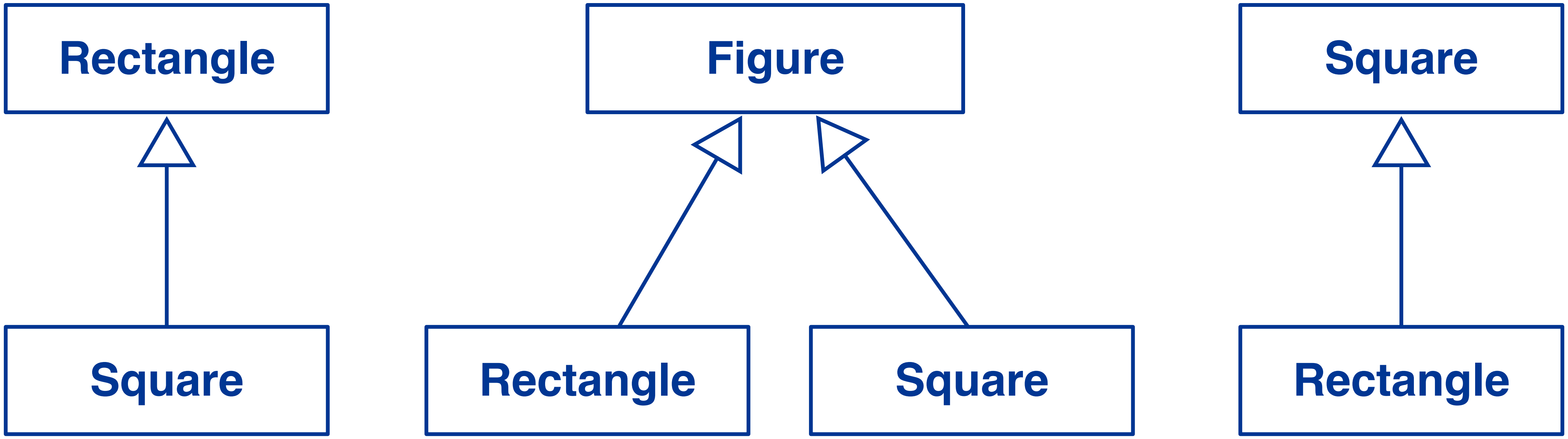
This slide is review from the Inheritance lecture in P2:

http://scg.unibe.ch/teaching/p2

Inheritance is used for three different purposes, and these reflect also the different way generalization is used as a modeling tool in UML:

1. We can use classes to *model domain concepts*, some of which are more general than others
2. We can design software classes into a type hierarchy where more refined types can be *polymorphically substituted for more general ones*
3. We can inherit implementation from superclasses, achieving *a form of software reuse*

The different faces of inheritance



Is-a

Polymorphism

Reuse

Usually these three uses of inheritance coincide, but they may not.

- A `Square` is a specialized form of `Rectangle` (every `Square` is a `Rectangle`, where the height and width are equal).
- `Rectangles` and `Squares` can be used wherever we expect a geometric Figure
- A `Rectangle` can inherit its `width` from a `Square`, and add a new `height` attribute

Exercise: turn this into a UML class diagram …

The Faculty of Science of the University of Bern forms various committees to make decisions on various issues throughout the year (budgets, hiring of professors, teaching evaluations, etc.).
Each committee is composed of Faculty members (i.e., professors), assistants, and also some students.
The chair of a committee is always a Faculty member.
Committees meet on various dates and may deliver reports to the Dean or to the Faculty.
Committee members can be contacted by email or phone.

Exercise: turn this into a UML class diagram.

Figure out what are the *domain concepts* that should be modeled as classes. (Perhaps not everything is important.)

What are the *relationships* between the classes? Are the relationships *inheritance* (is-kind-of), *composition* (part-of) or simple *associations*? Are the associations navigable?

Is every concept a first-class concept, or are some things simple *attributes* of classes?

Compare your solutions with others and *discuss*.

Roadmap

- UML Overview
- Classes, attributes and operations
- UML Lines and Arrows
- Parameterized Classes, Interfaces and Utilities
- Objects, Associations
- Inheritance
- **Constraints, Patterns and Contracts**

OCL — Object Constraint Language

> Used to express queries and constraints over UML diagrams
  - Navigate associations:
    - *Person.boss.employer*
  - Select subsets:
    - *Company.employee->select(title="Manager")*
  - Boolean and arithmetic operators:
    - *Person.salary < Person.boss.salary*

OCL is a standard of the Object Management Group, which is also responsible for UML. All the standard documents can be accessed online:

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

Constraints

Constraints are *restrictions* on values attached to classes or associations.

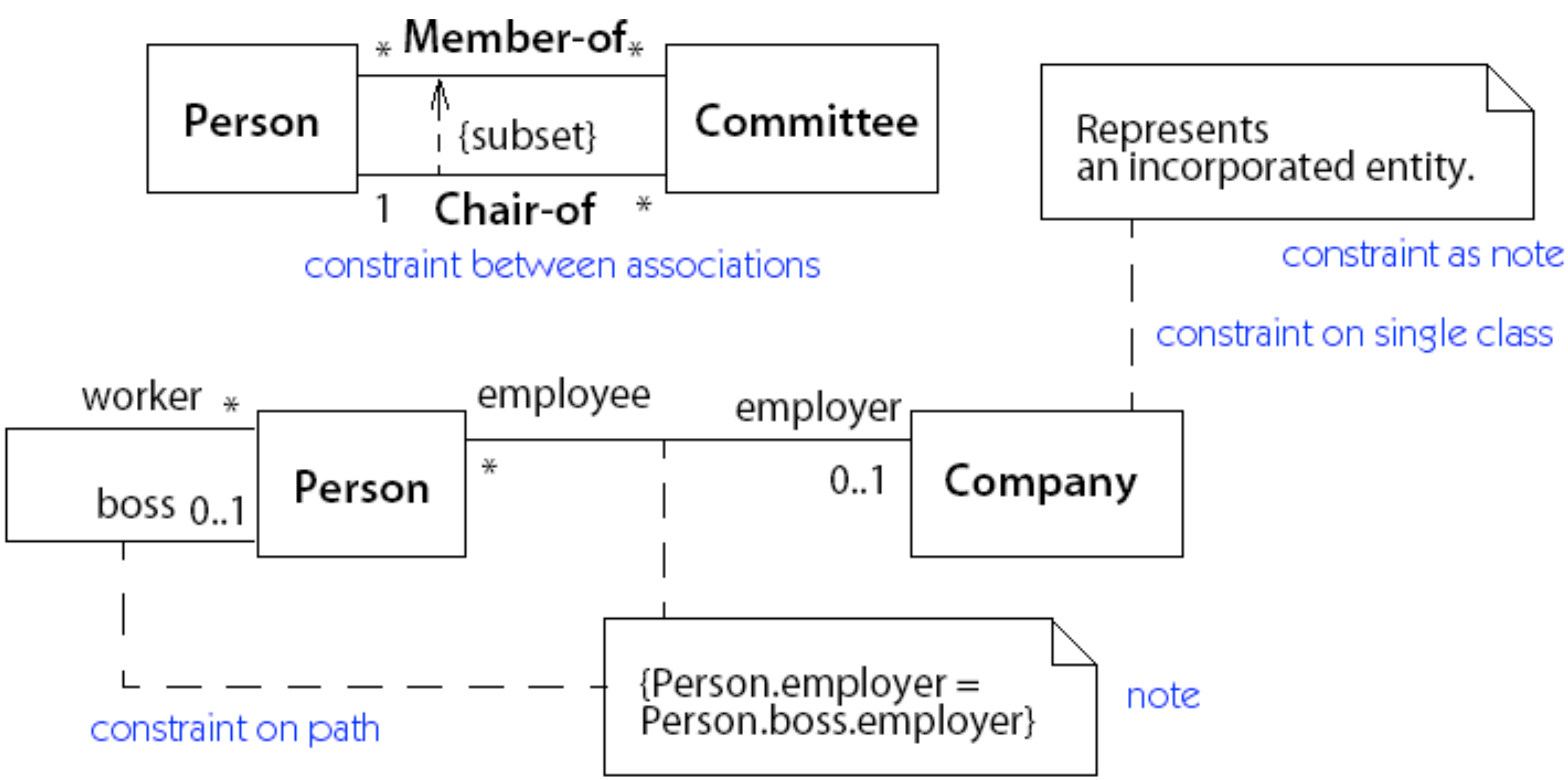


Figure 4-12. *Constraints*

Constraints are indicated using dashed lines or arrows, and may be annotated with an OCL expression or a natural language note describing the constraint. Examples:

- The chair of a committee must be a member of that committee. This is a simple OCL *subset constraint* over these two relations.
- A Company is an incorporated entity. Since these concepts are not modeled in the diagram, we express this constraint in *natural language*.
- An employee and the employee's boss must work for the same company. We express this in OCL by *navigating the class diagram*.

Design Patterns as Collaborations

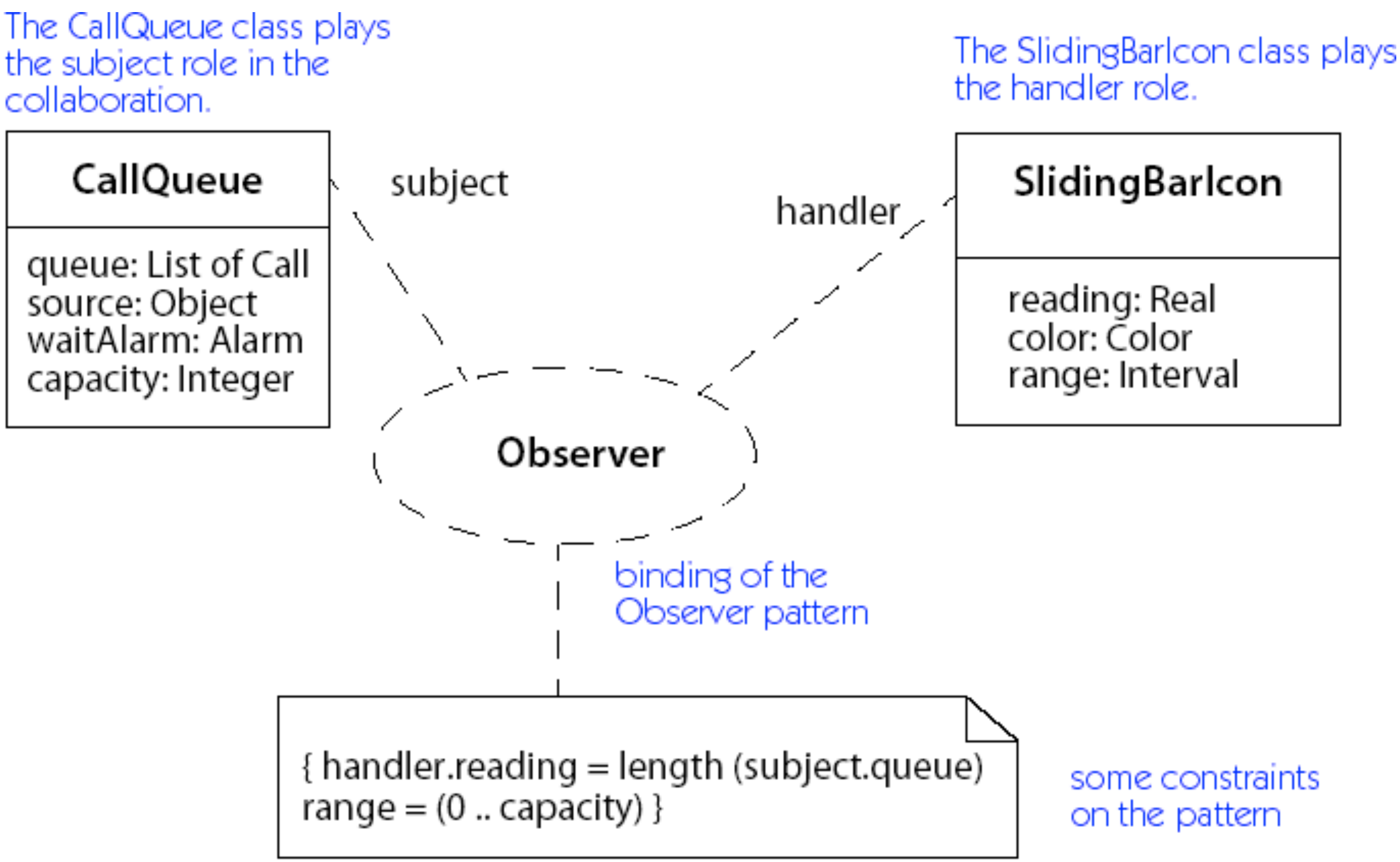


Figure 13-144. *Binding of a pattern to make a collaboration*

A design pattern can be modeled as a dashed ellipse (here we have the `Observer` pattern), with dashed lines linking *roles* associated to the design pattern with the *classes* playing those roles. In the example, the `CallQueue` is the *subject* being observed, and the `SlidingBarIcon` is the *handler* observing the subject.

We furthermore have an OCL constraint specifying that the handler's reading attribute should reflect the current length of the queue.

Design by Contract in UML

Combine constraints with stereotypes:

NB: «invariant», «precondition», and «postcondition» are predefined in UML.

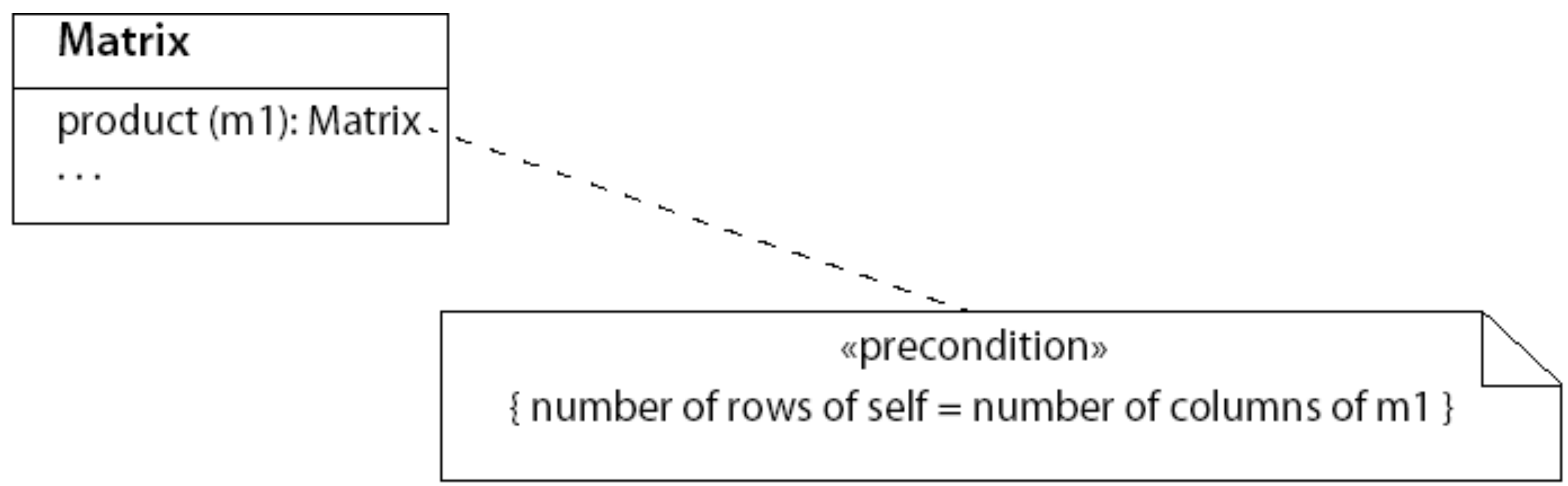


Figure 13-147. *Precondition*

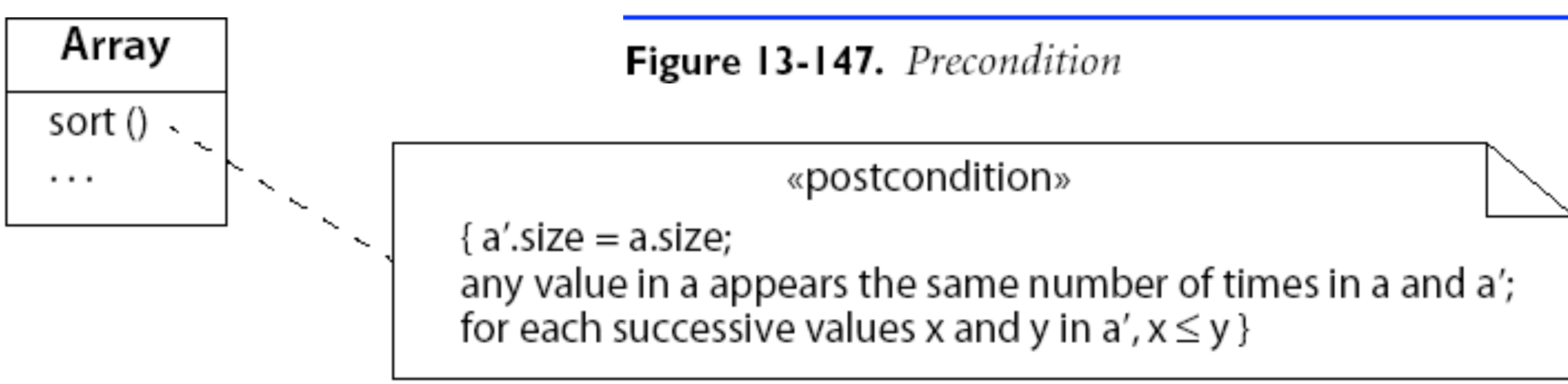


Figure 13-145. *Postcondition*

Preconditions, postconditions, invariance and other assertions can also be expressed as UML constraints, preferably in OCL or in natural language.

Using the Notation

During Analysis:

- —Capture classes visible to *users*
- —Document *attributes and responsibilities*
- —Identify *associations and collaborations*
- —Identify *conceptual hierarchies*
- —Capture all *visible features*

During Design:

- —Specify *contracts and operations*
- —*Decompose* complex objects
- —Factor out *common interfaces* and functionalities

NB: The graphical notation is only *one part* of the analysis or design document. For example, a data dictionary cataloguing and describing all names of classes, roles, associations, etc. must be maintained throughout the project.

What you should know!

- How do you represent classes, objects and associations?
- How do you specify the visibility of attributes and operations to clients?
- How is a utility different from a class? How is it similar?
- Why do we need both named associations and roles?
- Why is inheritance useful in analysis? In design?
- How are constraints specified?

Can you answer the following questions?

- Why would you want a feature to have class scope?
- Why don’t you need to show operations when depicting an object?
- Why aren’t associations drawn with arrowheads?
- How is aggregation different from any other kind of association?
- How are associations realized in an implementation language?