2 UML for OOP

2.1 What is UML?
2.2 Classes in UML
2.3 Relations in UML
2.4 Static and Dynamic Design with UML
2.5 Use case modelling (tbd)

2.1 UML Background

"The Unified Modelling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a systems blueprints, including conceptual things like business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components."

Grady Booch, Ivar Jacobsen, Jim Rumbaugh
Rational Software Corp.

[The Unified Modelling Language User Guide, Addison-Wesley 2003]

2.1 Brief UML History

- Around 1980
  - first OO modelling languages
  - other techniques, e.g. SA/SD
- Around 1990
  - “OO method wars”
  - many modelling languages
- End of 90's
  - UML appears as combination of best practices

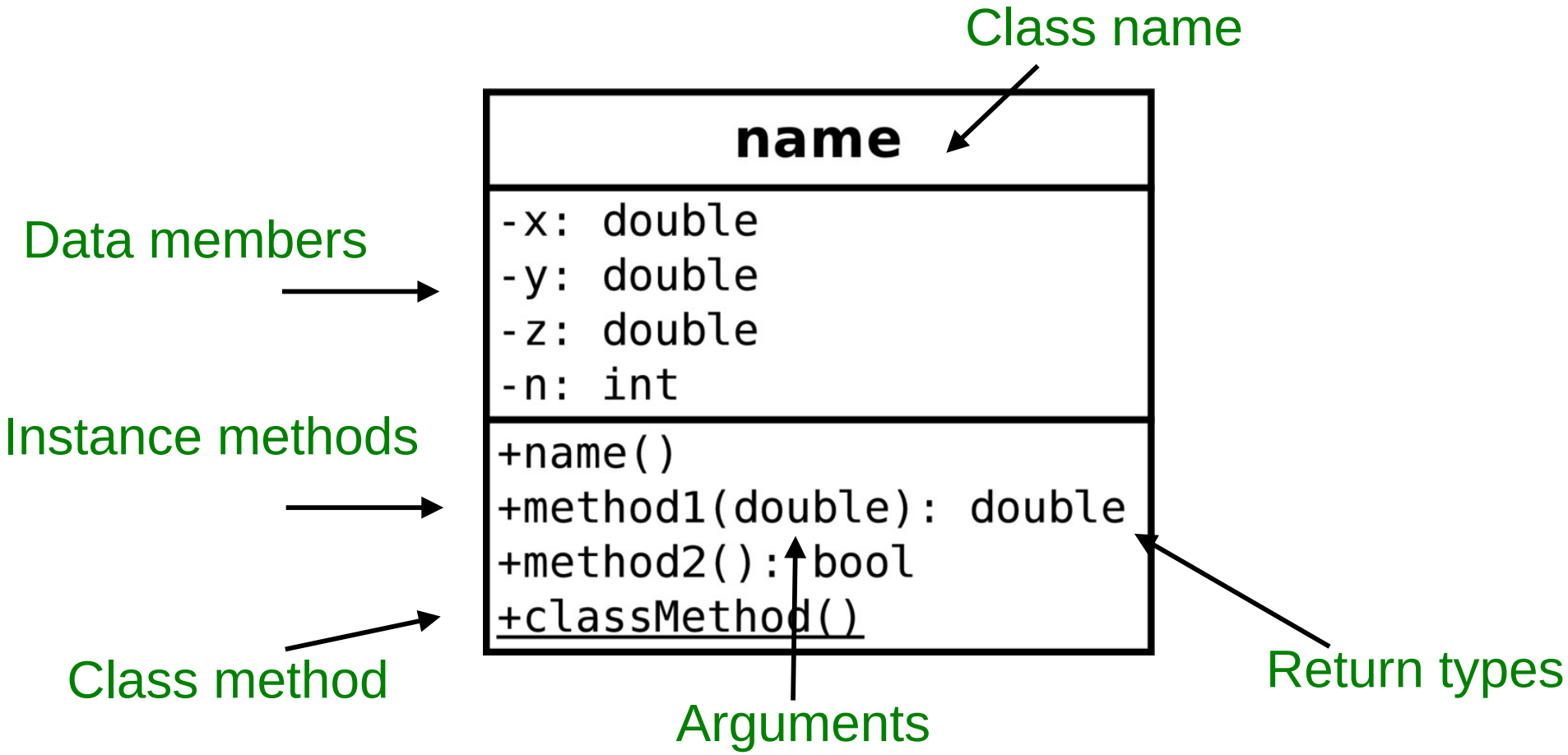
2.1 Why UML?

- Physicists know formal graphical modelling
  - Mathematics to describe nature
  - Feynman graphs to calculate amplitudes
- We need a common language
  - discuss software at a black- (white-) board
  - Document software systems
  - UML is an important part of that language
  - UML provides the “words and grammar”

2.2 Classes in UML

- Classes describe objects
  - Interface (member function signature)
  - Behaviour (member function implementation)
  - State bookkeeping (values of data members)
  - Creation and destruction
- Objects described by classes collaborate
  - Class relations ⇒ object relations
  - Dependencies between classes

2.2 UML Class



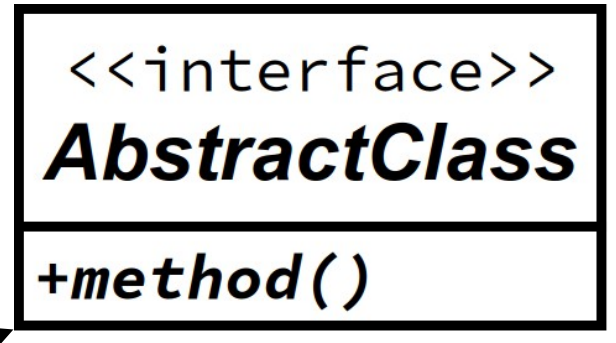
Data members, arguments and methods are specified by
visibility name : type

2.2 Class Name

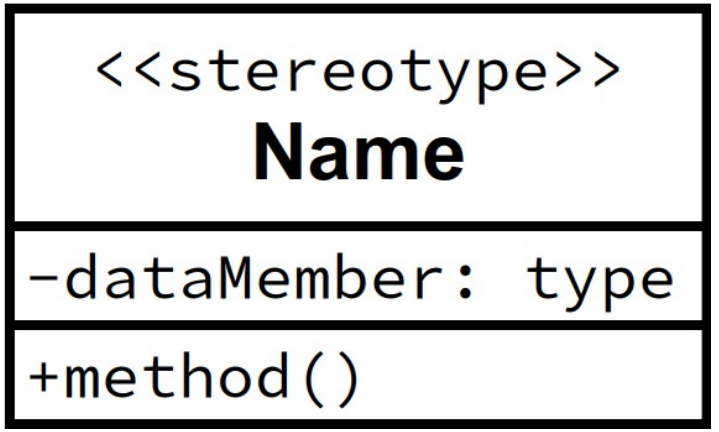
The top compartment contains the class name

Abstract classes have italicised names

Abstract methods also have italicised names



Stereotypes are used to identify groups of classes, e.g. interfaces or persistent (storeable) classes



2.2 Class Attributes

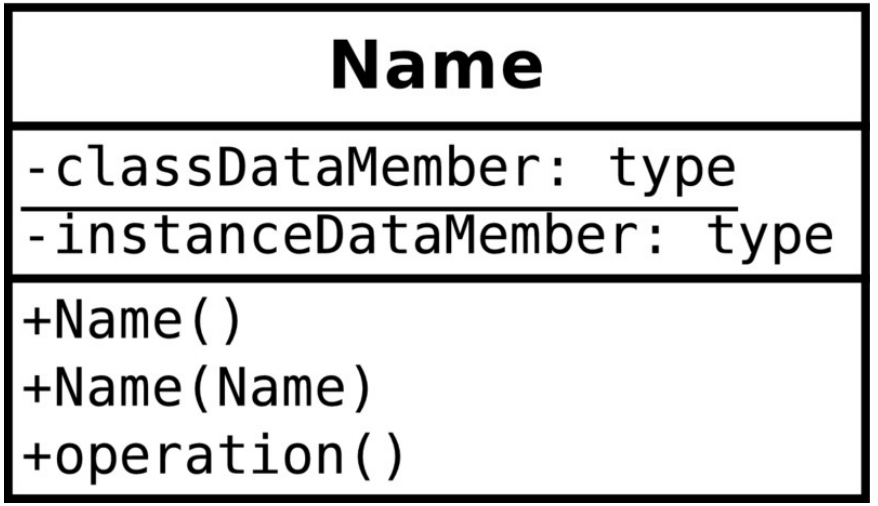
Attributes are the instance and class data members

Class data members (underlined) are shared between all instances (objects) of a given class

Data types shown after ":"

Visibility shown as
- + public
- – private
- # protected

Attribute compartment



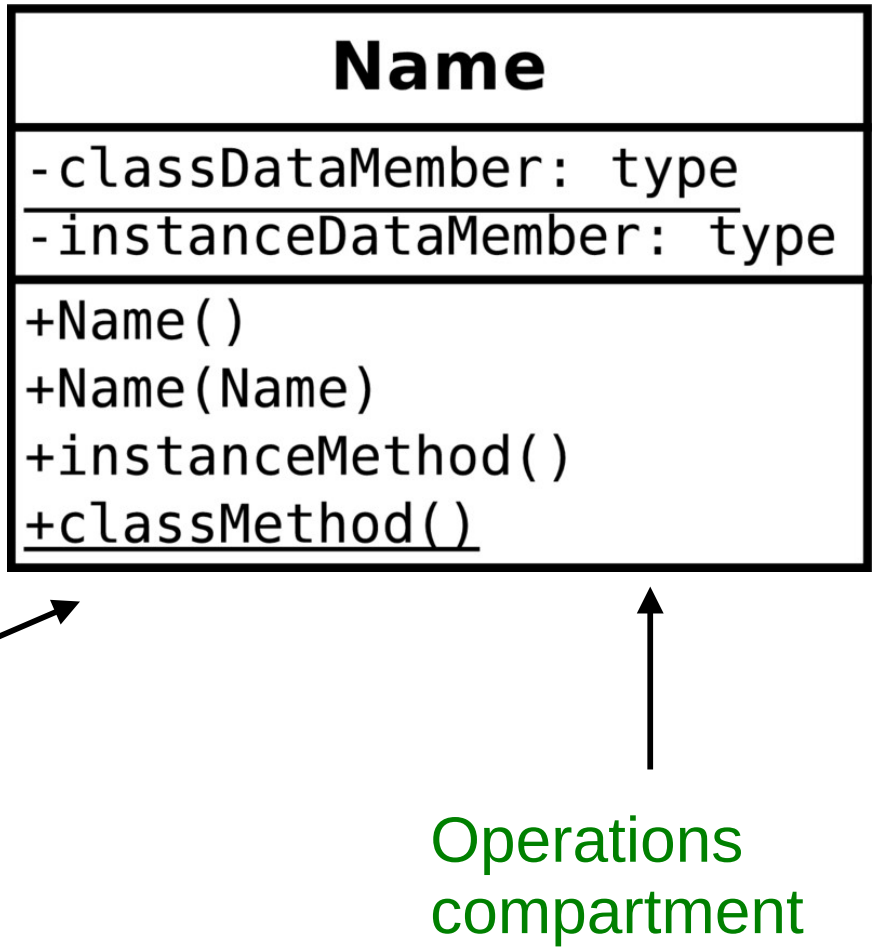
visibility name : type

2.2 Class Operations (Interface)

Operations are the class methods with their argument and return types

Public (+) operations define the class interface

Class methods (underlined) have only access to class data members, no need for a class instance (object)



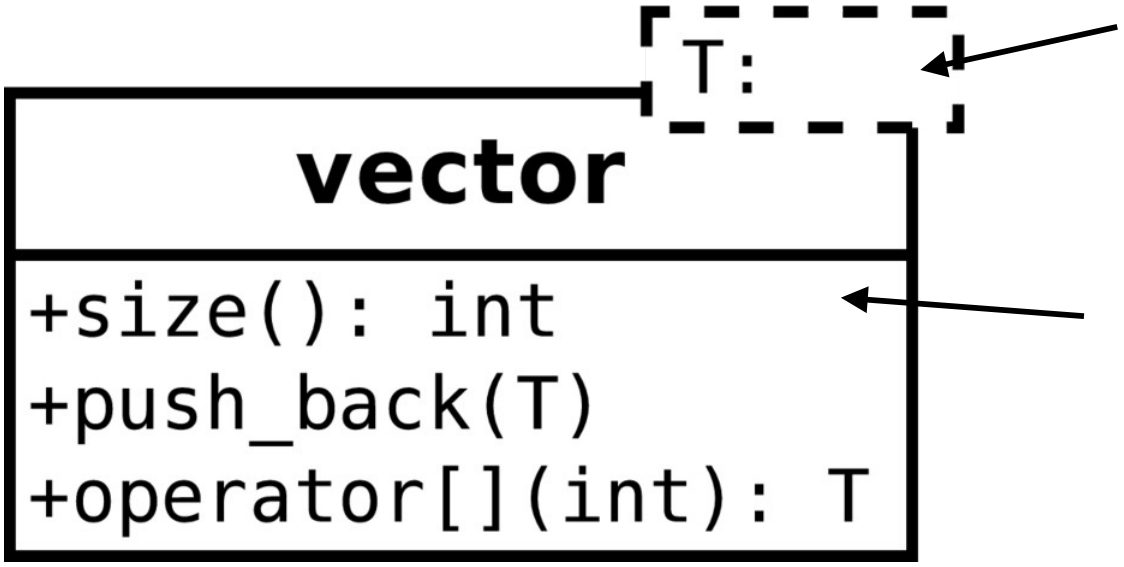
visibility name : type

2.2 Visibility

+	–	#
public	private	protected
Anyone can access	No-one can access	Subclasses can access
Interface operations	Data members	Operations where sub-classes collaborate
Not data members	Helper functions	Not data members (creates dependency of subclass on im-plementation of parent)
	“Friends” are allowed in though	

2.2 Template Classes

Generic classes depending on parametrised types



Type parameter(s)

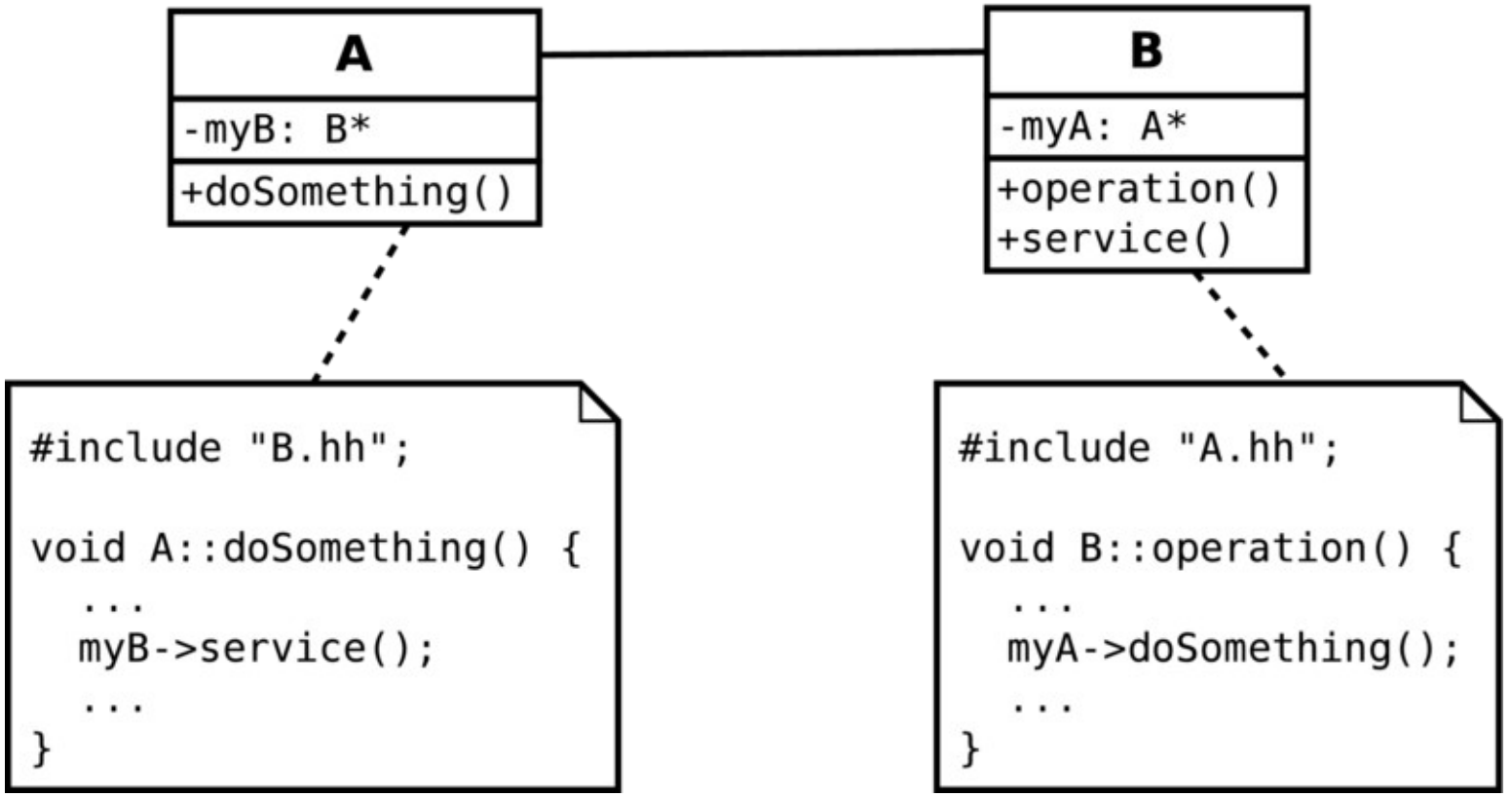
Operations compartment as usual, but may have type parameter instead of concrete type

2.3 Relations

- Association
- Aggregation
- Composition
- Parametric and Friendship
- Inheritance

2.3 Binary Association

Binary association: both classes know each other



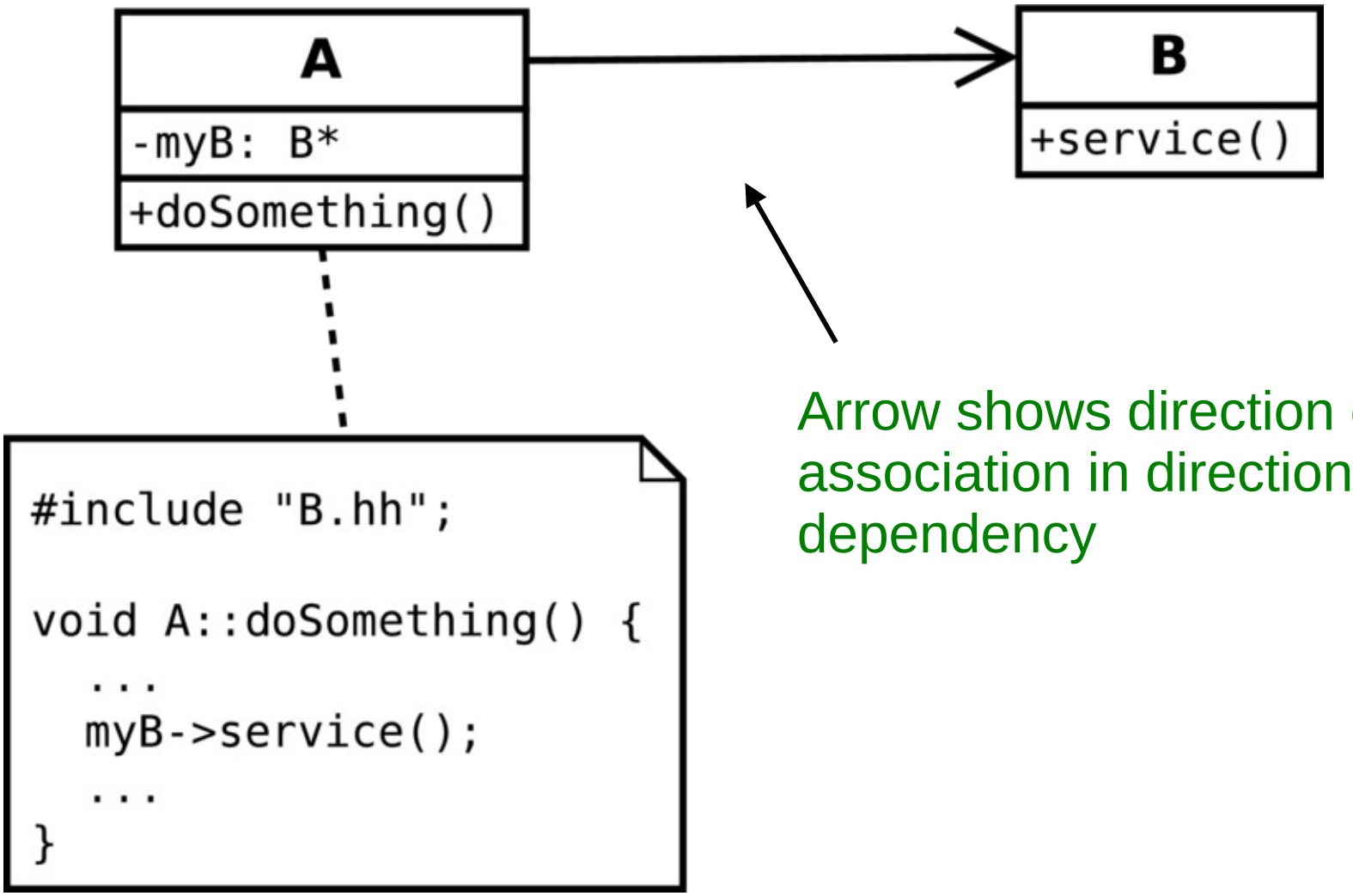
Usually “knows about” means a pointer or reference

Other methods possible: method argument, tables, database, ...

Implies dependency cycle

2.3 Unary Association

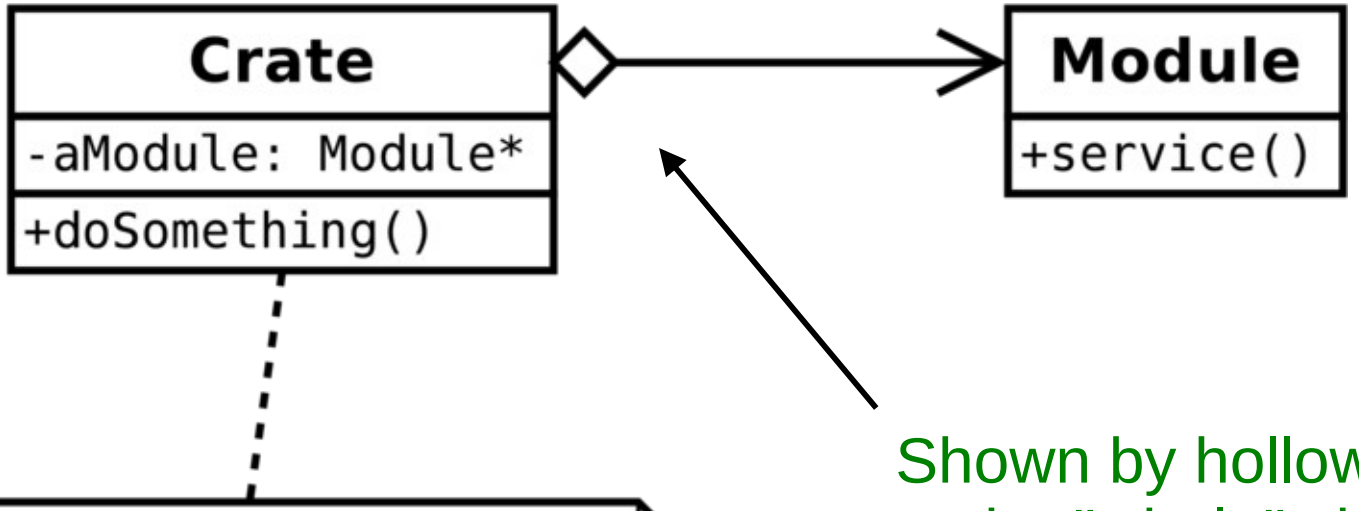
A knows about B, but B knows nothing about A



Arrow shows direction of association in direction of dependency

2.3 Aggregation

Aggregation = Association with “whole-part” relationship



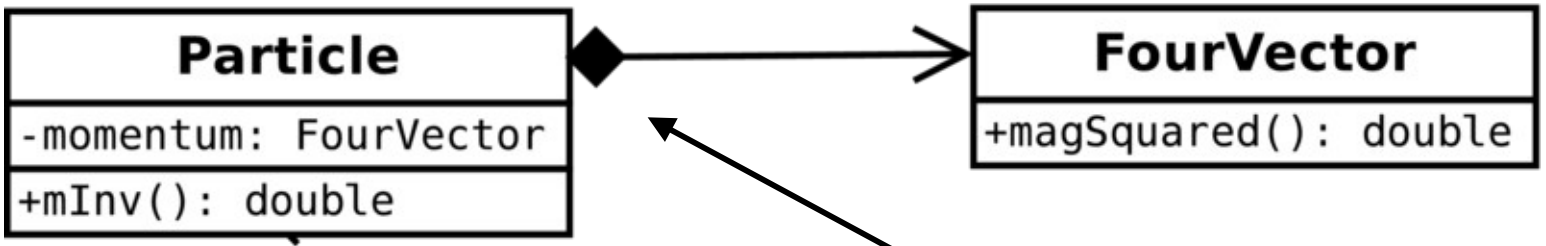

```
#include "Module.hh";

void Crate::doSomething() {
  ...
  aModule->service();
  ...
}
```

Shown by hollow diamond at the "whole" side

No lifetime control implied

2.3 Composition

Composition = Aggregation with lifetime control




```
double Particle::mInv() {
  double minv2= momentum.magSquared();
  return minv2<0?-sqrt(-minv2):sqrt(minv2);
}
```

Shown by filled diamond at the “owner” side

Lifetime control implied

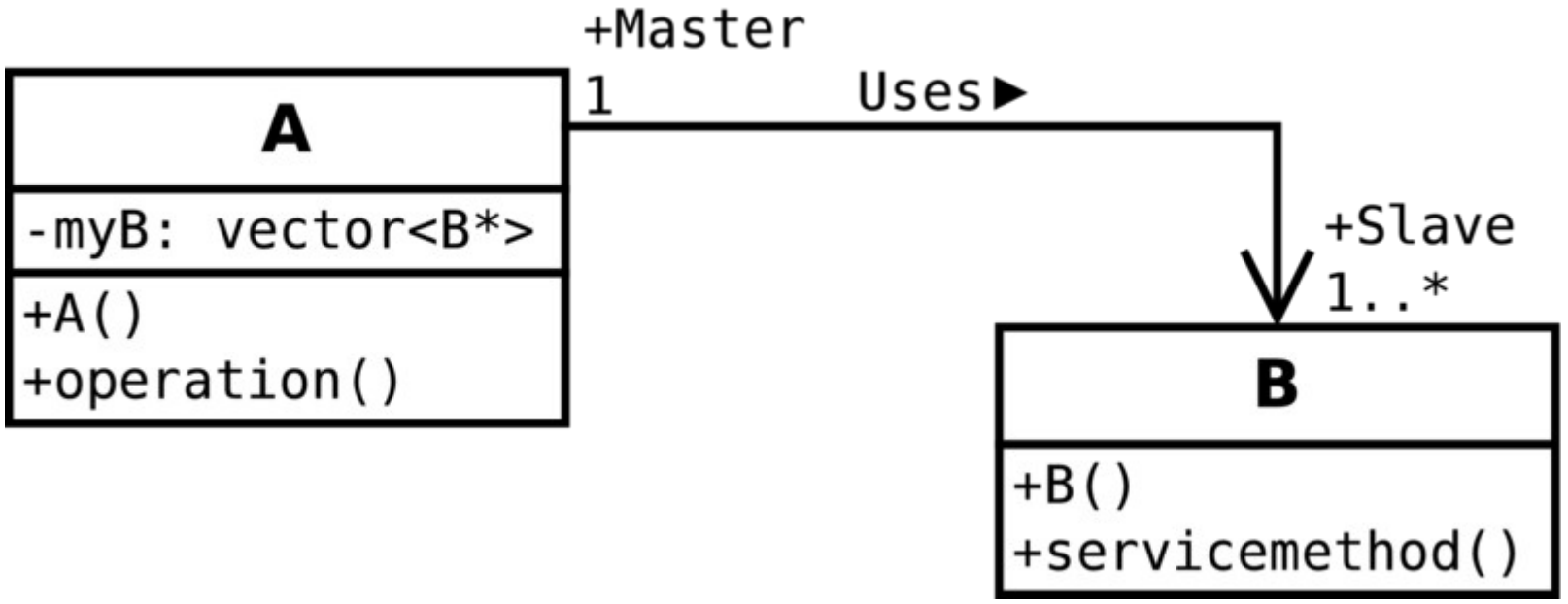
Lifetime control can be transferred

Lifetime control: construction and destruction controlled by “owner”

→ call constructors and destructors (or have somebody else do it)

2.3 Association Details

Name gives details of association
Name can be viewed as verb of a sentence



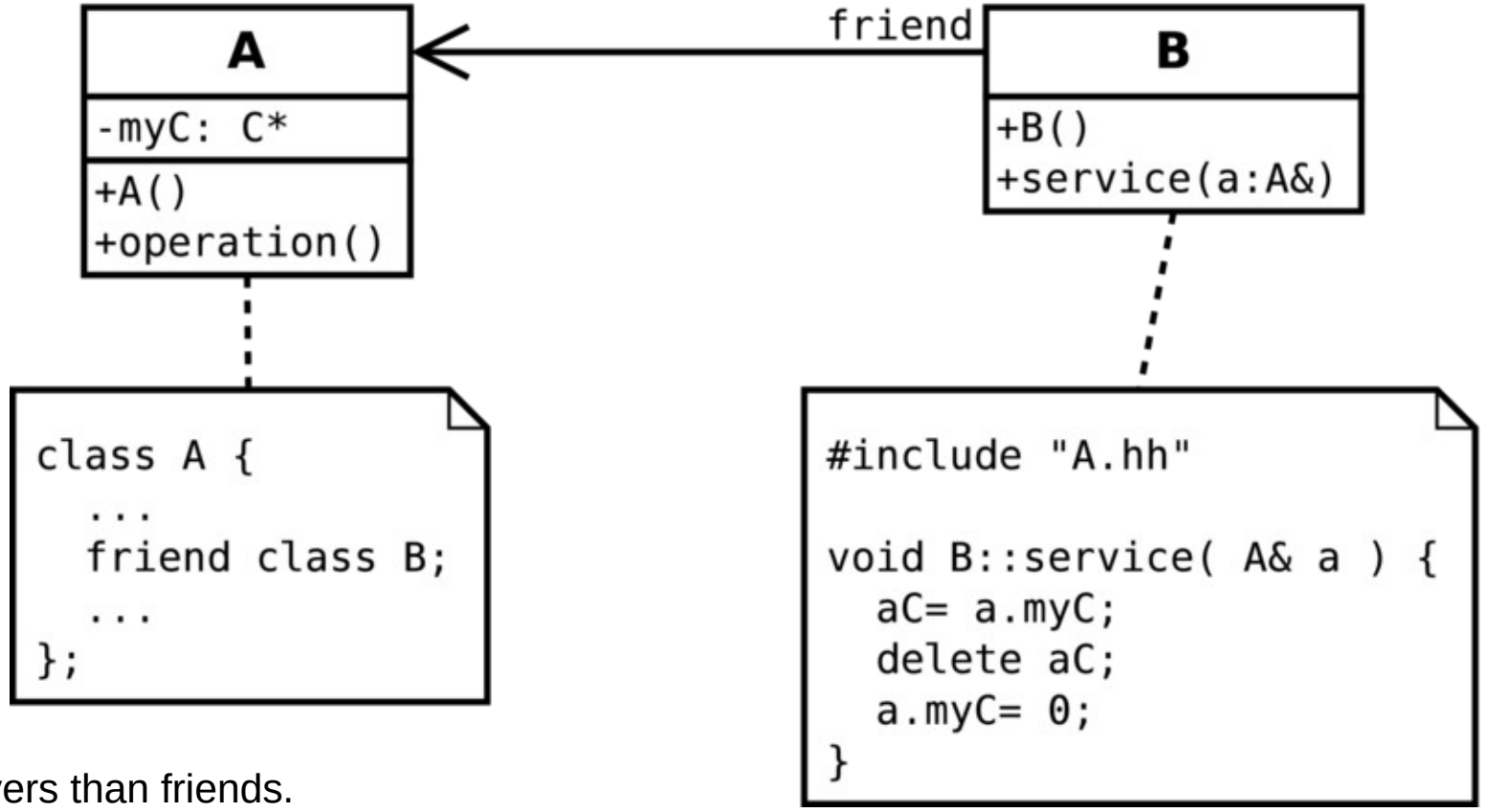
Notes at association ends explain “roles” of classes (objects)

Multiplicities show number of objects which participate in the association

2.3 Friendship

Friends are granted access to private data members and member functions

Friendship is given to other classes, never taken

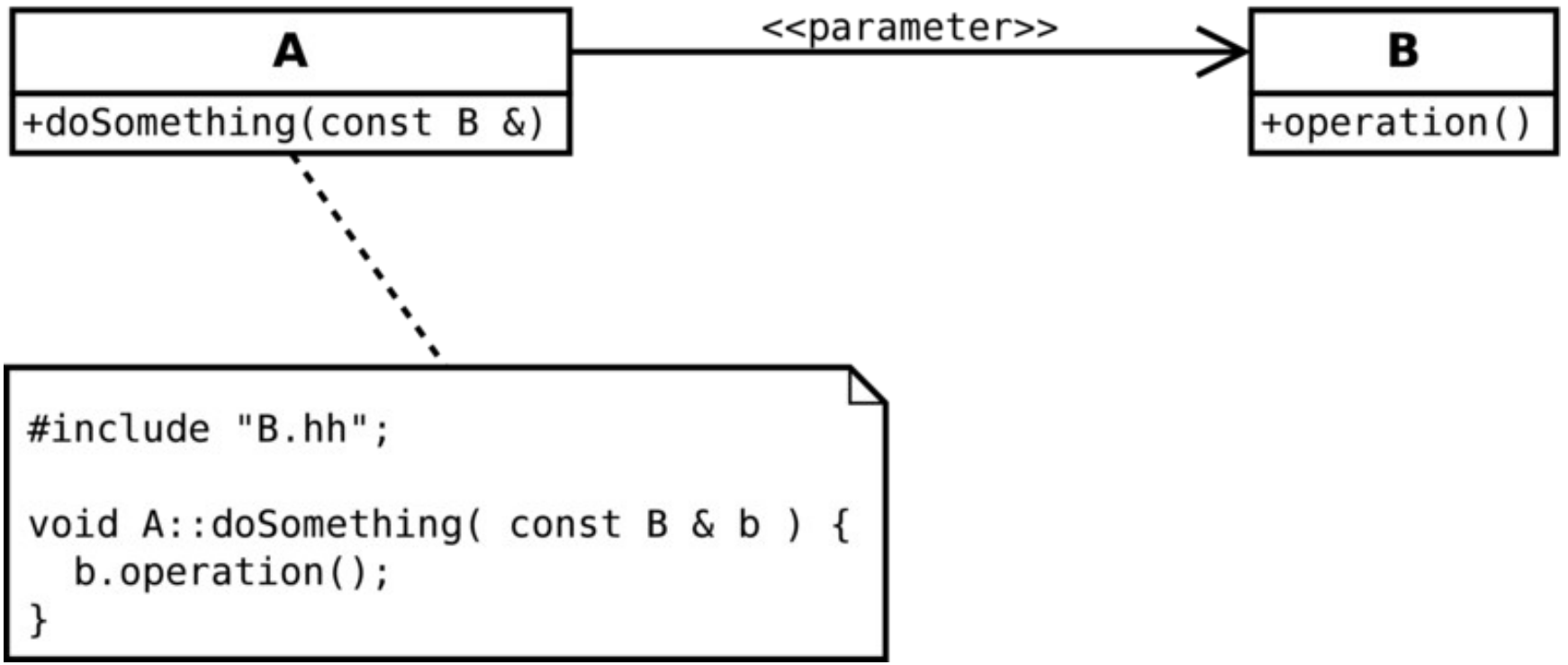


Bob Martin:
More like lovers than friends.
You can have many friends,
you should not have many lovers

Friendship breaks data hiding, use carefully

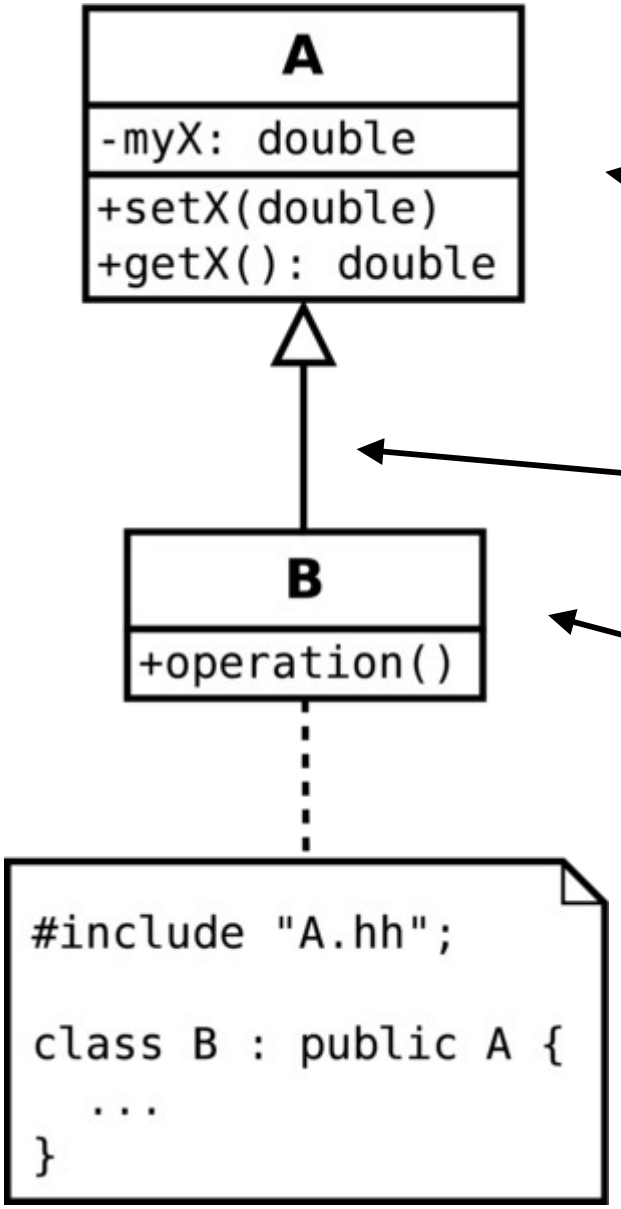
2.3 Parametric Association

Association mediated by a parameter (function call argument)



A depends upon B, because it uses B
No data member of type B in A

2.3 Inheritance



Base class or super class

Arrow shows direction of dependency

Derived class or subclass

→ B inherits A's interface, behavior and data members

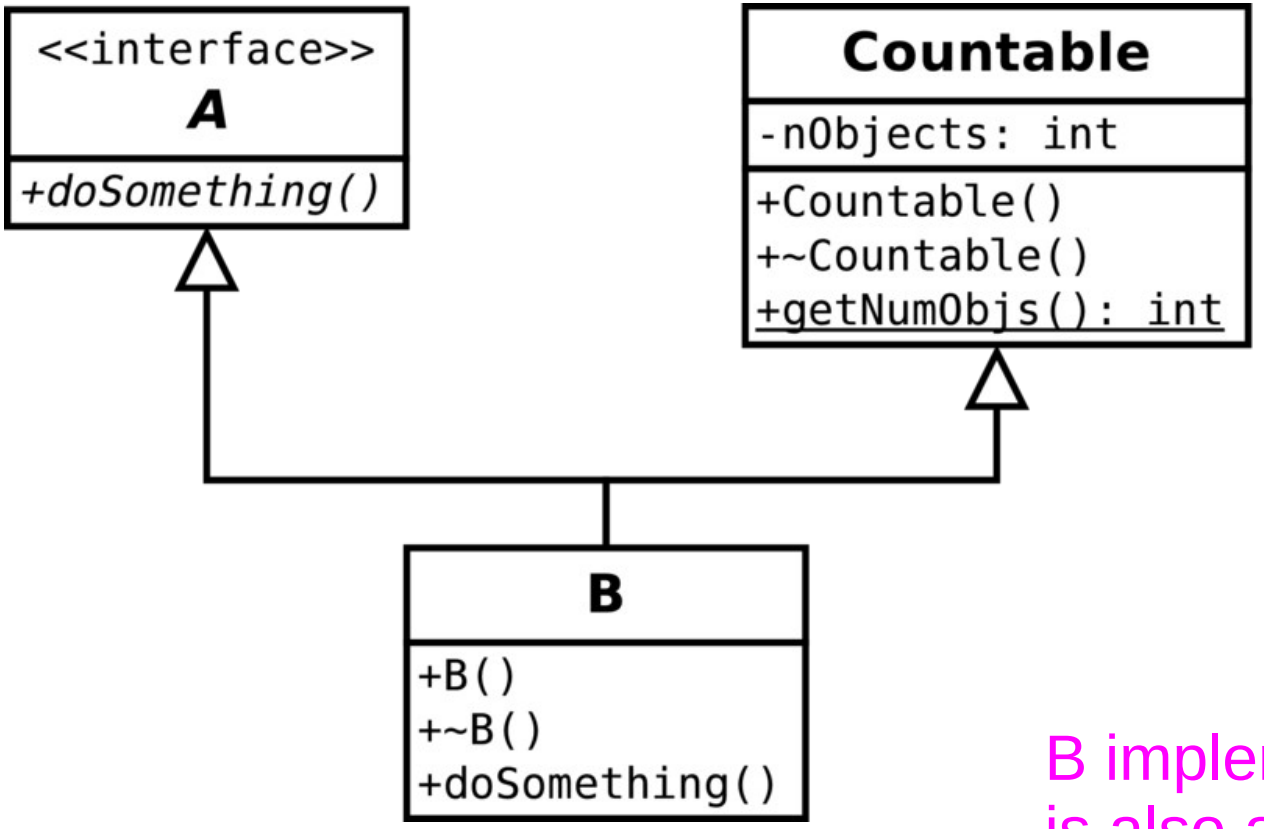
→ B can extend A, i.e. add new data members or member functions

→ B depends on A, A knows nothing about B

2.3 Associations Summary

- Can express different kinds of associations between classes/objects with UML
  - Association, aggregation, composition, inheritance
  - Friendship, parametric association
- Can go from simple sketches to more detailed design by adding *adornments*
  - Name, roles, multiplicities
  - lifetime control

2.3 Multiple Inheritance



The derived class inherits interface, behavior and data members of all its base classes

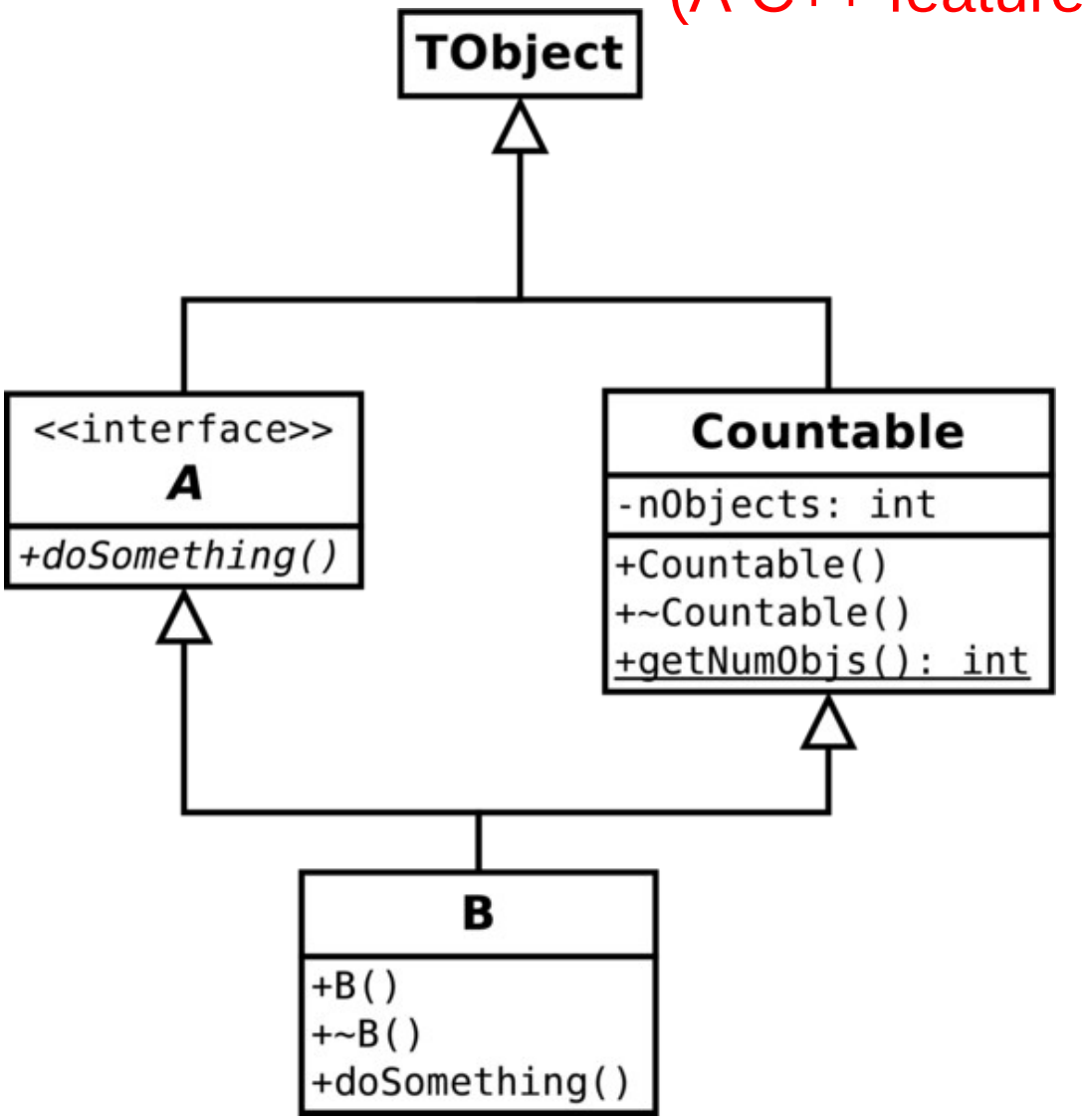
Extension and overriding works as before

B implements the interface A and is also a “countable” class

Countable also called a ”Mixin class”

2.3 Deadly Diamond of Death

(A C++ feature)



Now the @*#! hits the %&$?

Data members of TObject are inherited twice in B, which ones are valid?

Fortunately, there is a solution to this problem:

→ virtual inheritance in C++: only one copy of a multiply inherited structure will be created

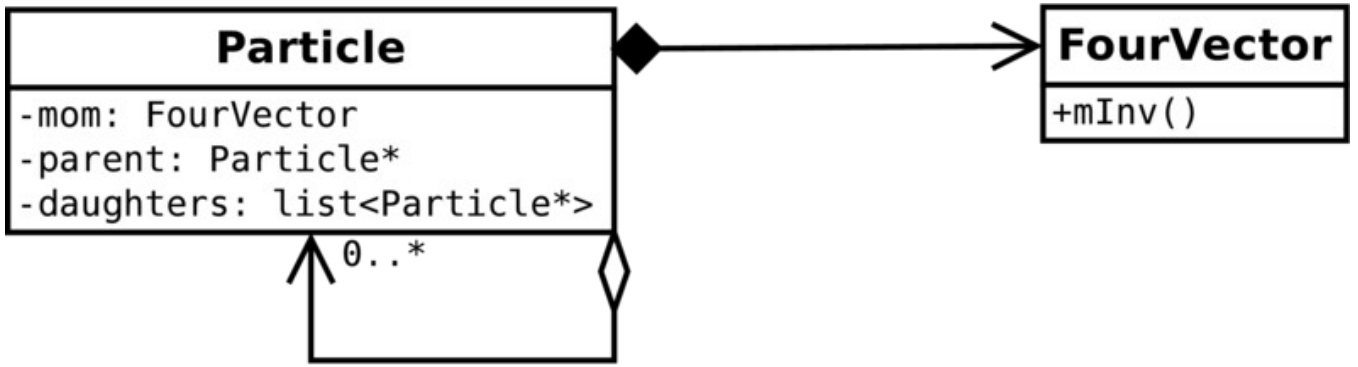
2.4 Static and Dynamic Design

- Static design describes code structure and object relations
  - Class relations
  - Objects at a given time
- Dynamic design shows communication between objects
  - Similarity to class relations
  - can follow sequences of events

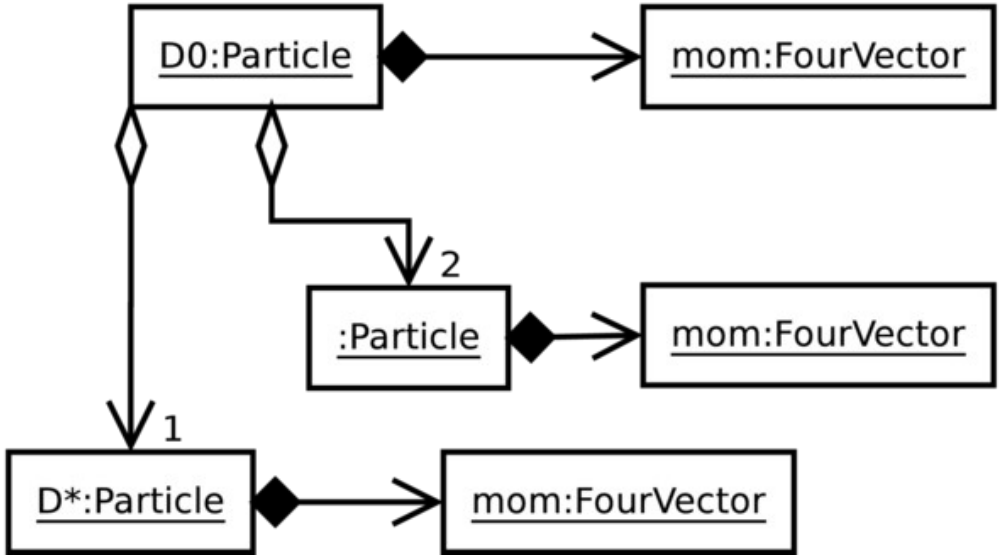
2.4 Class Diagram

- Show static relations between classes
  - we have seen them already
  - interfaces, data members
  - associations
- Subdivide into diagrams for specific purpose
  - showing all classes usually too much
  - ok to show only relevant class members
  - set of all diagrams should describe system

2.4 Object Diagram



Class diagram never changes

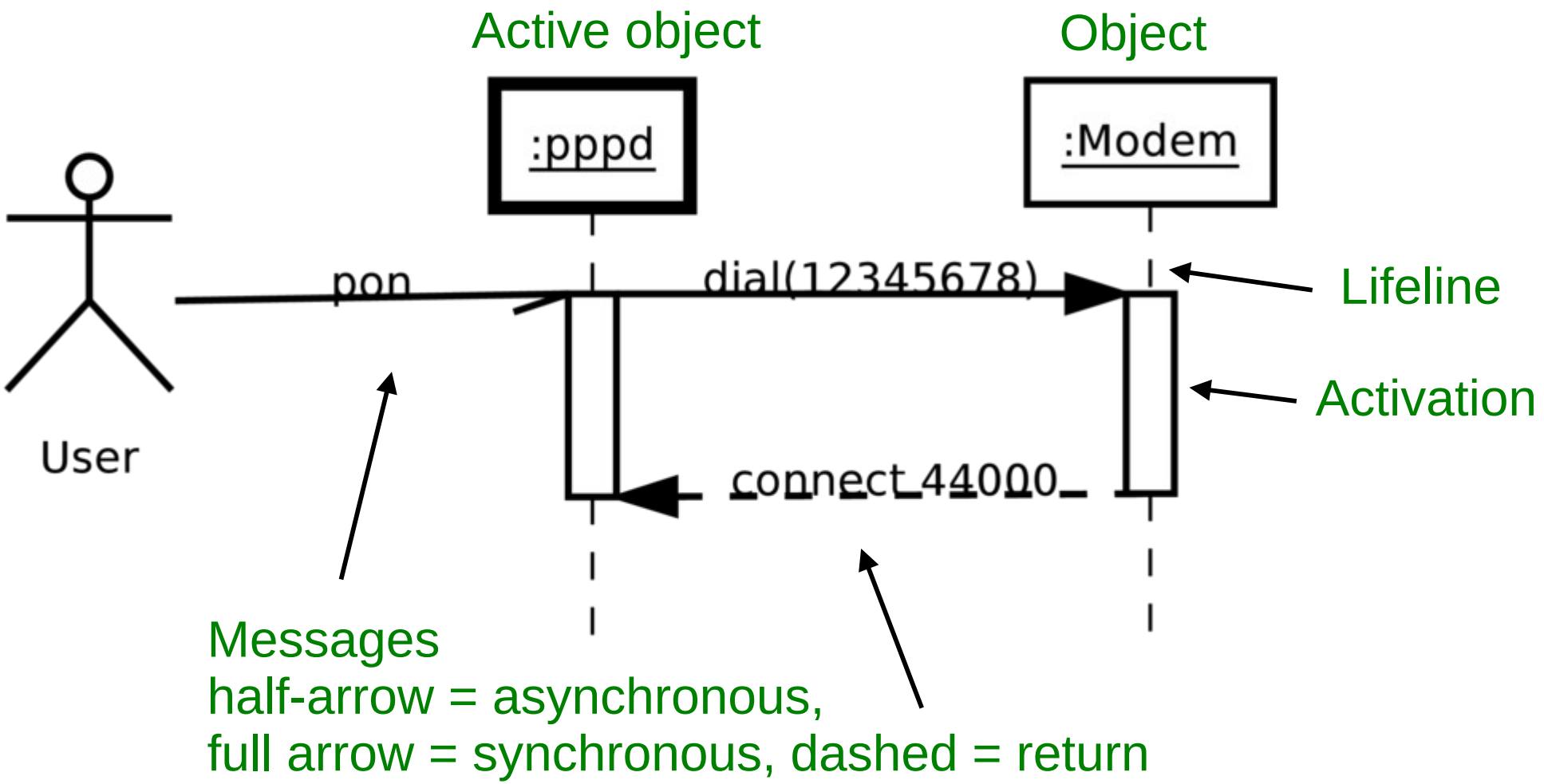


Object diagram shows relations at instant in time (snapshot)

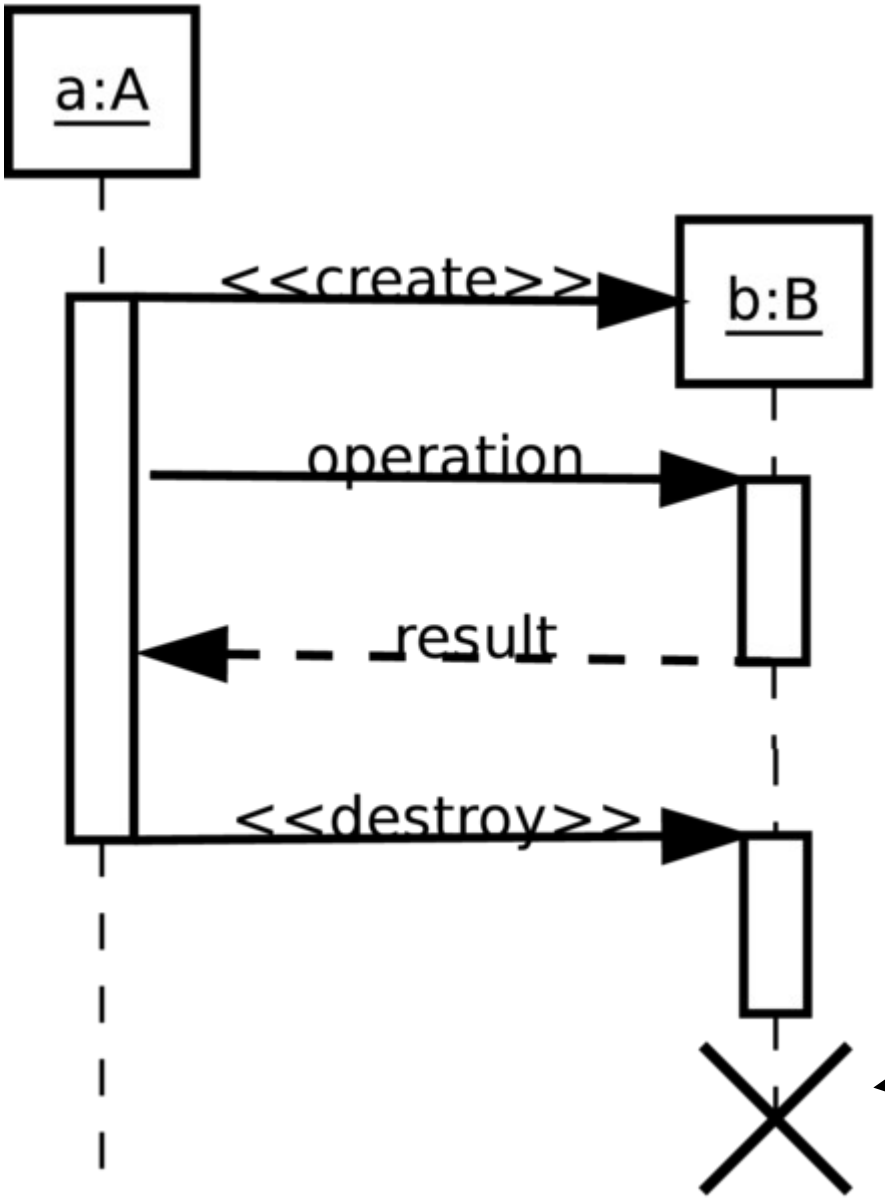
Object relations are drawn using the class association lines

2.4 Sequence Diagram

Show sequence of events for a particular use case



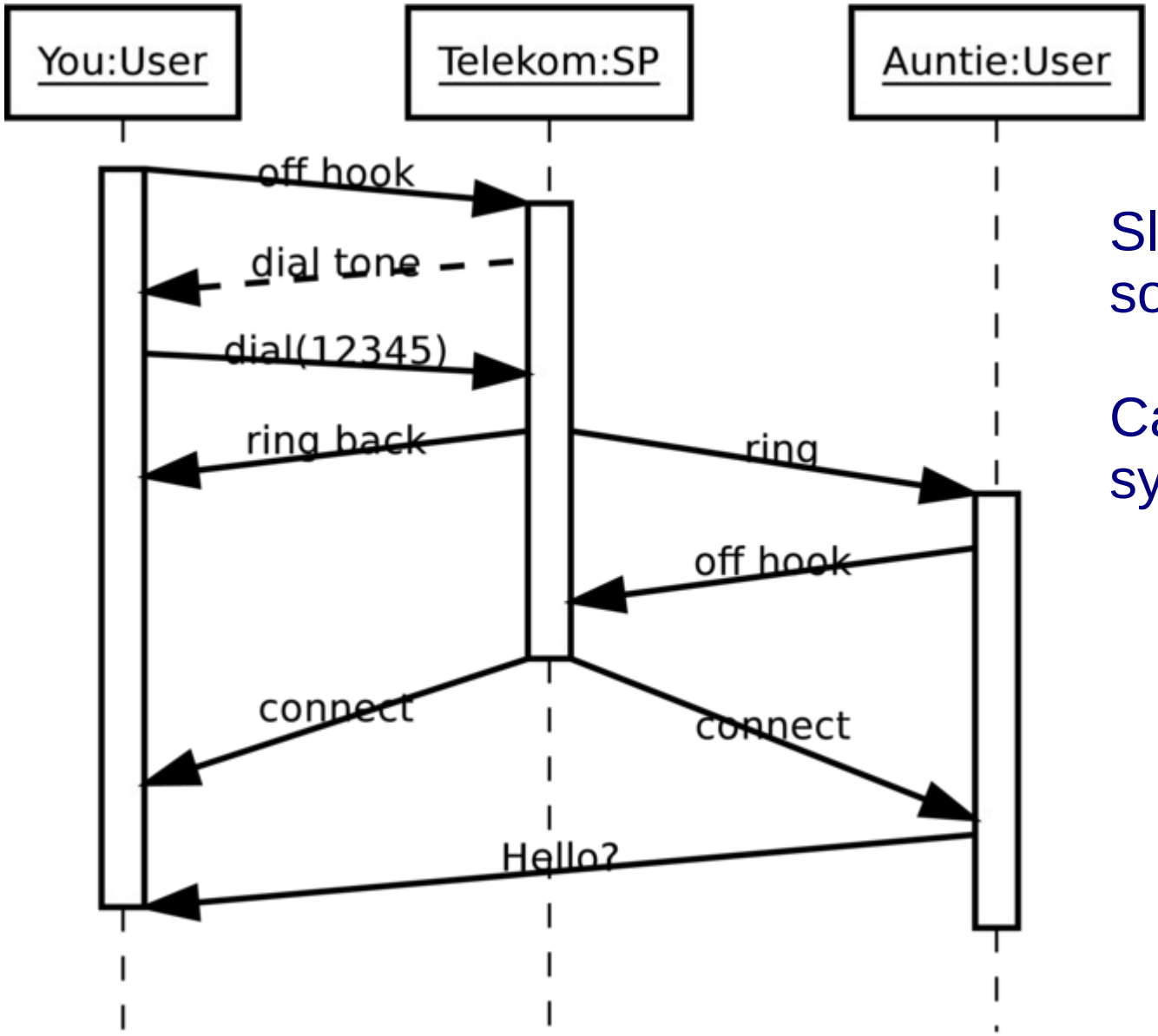
2.4 Sequence Diagram



Can show creation and destruction of objects

Destruction mark

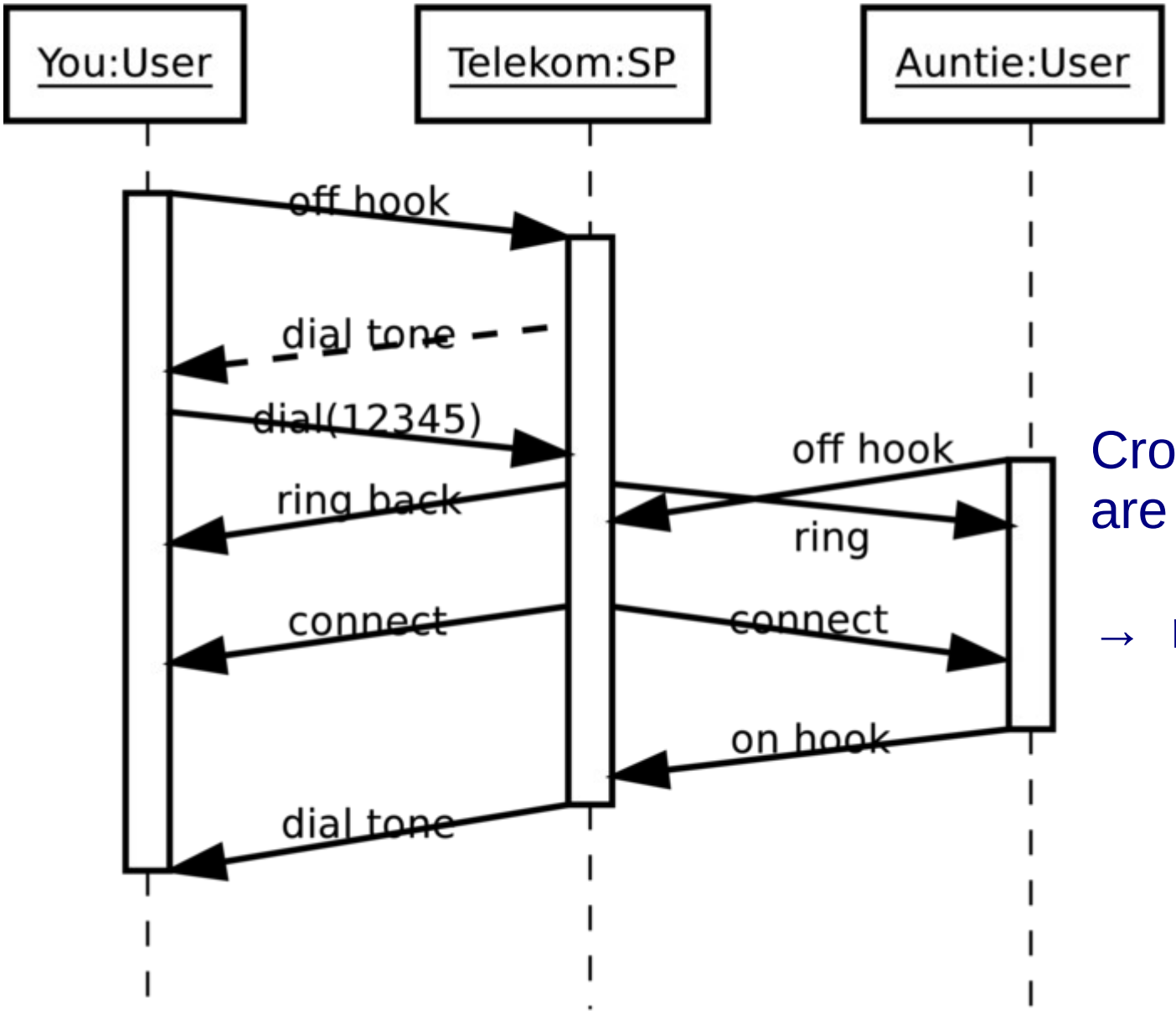
2.4 Sequence Diagram



Slanted messages take some time

Can model real-time systems

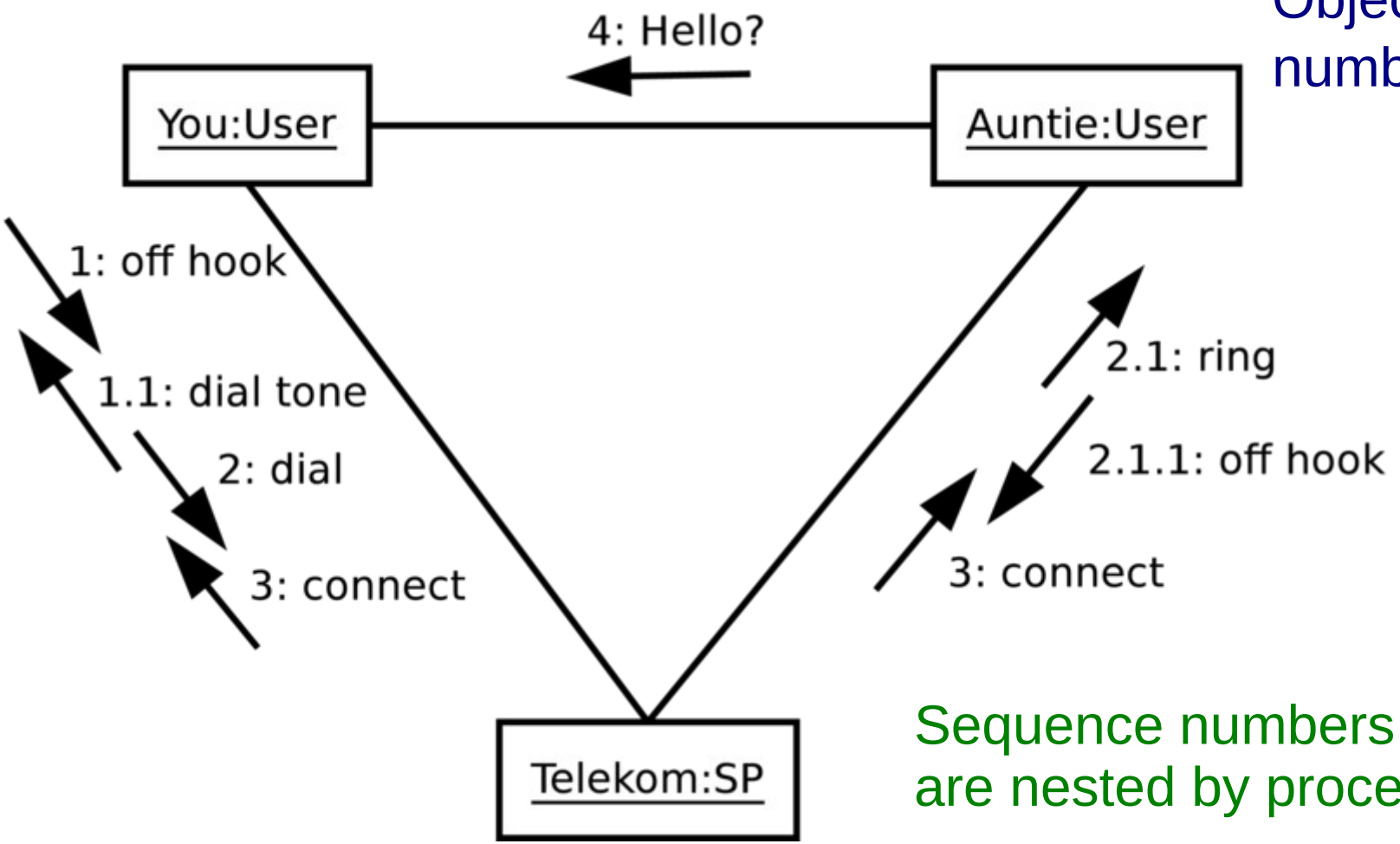
2.4 Sequence Diagram



Crossing message lines are a bad sign

→ race conditions

2.4 Collaboration Diagram



Object diagram with numbered messages

Sequence numbers of messages are nested by procedure call

2.4 Static and Dynamic Design Summary

- Class diagrams → object diagrams
  - classes → objects; associations → links
- Dynamic models show how system works
  - Sequence and collaboration diagram
- There are tools for this process
  - UML syntax and consistency checks
- Sketches by hand or with simple tools
  - aid in design discussions

Some Comments

- Design-heavy development processes
  - several 10% of person-power/time spent on design with formal UML from requirements
  - start coding when the design is consistent
  - large software houses may work this way
- Lighter processes
  - a few % of person-power/time spent with UML
  - UML as a discussion and documentation aid
  - probably more adequate in HEP