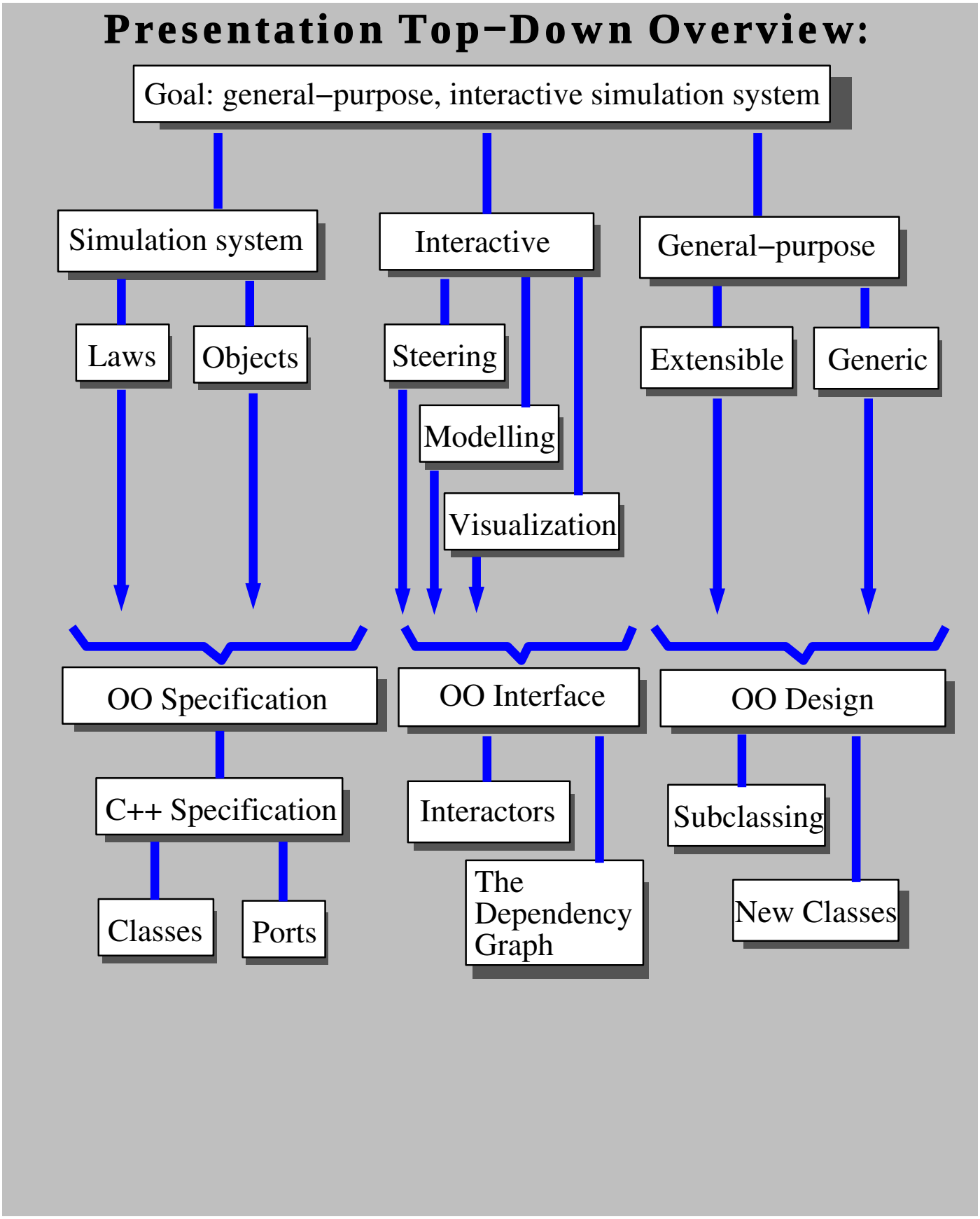
A General–Purpose Interactive Simulation System

The Design Path From Specifications
To An Object–Oriented Implementation

Alexandru Telea

Fac. Wiskunde en Informatica

Presentation Top-Down Overview:



The Simulation System Concept:

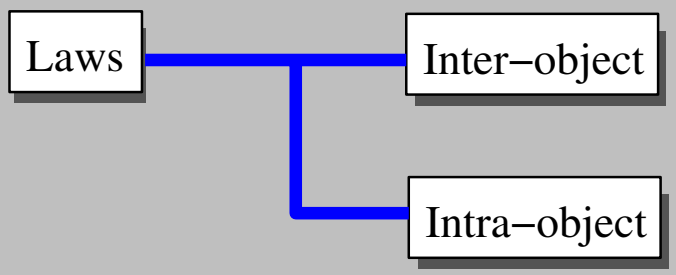
(as compared to the imperative programming concept)

A Simulation = Objects + Laws

Objects = State variables

An object groups related state variables and treats them as an entity.

Example: a point, a vector, a field, a time–dependent PDE

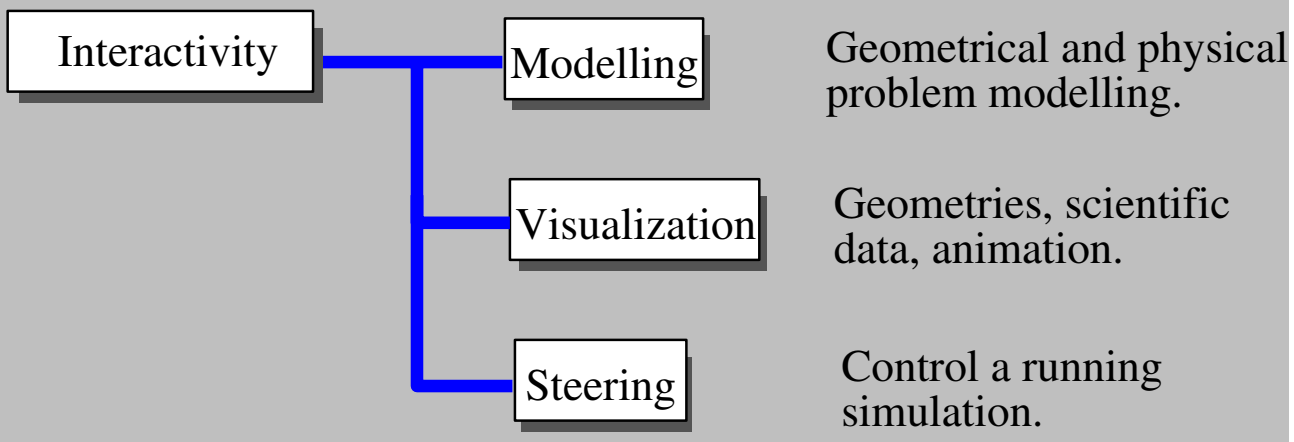


Express constraints between objects.

Express constraints between state variables of the same object.

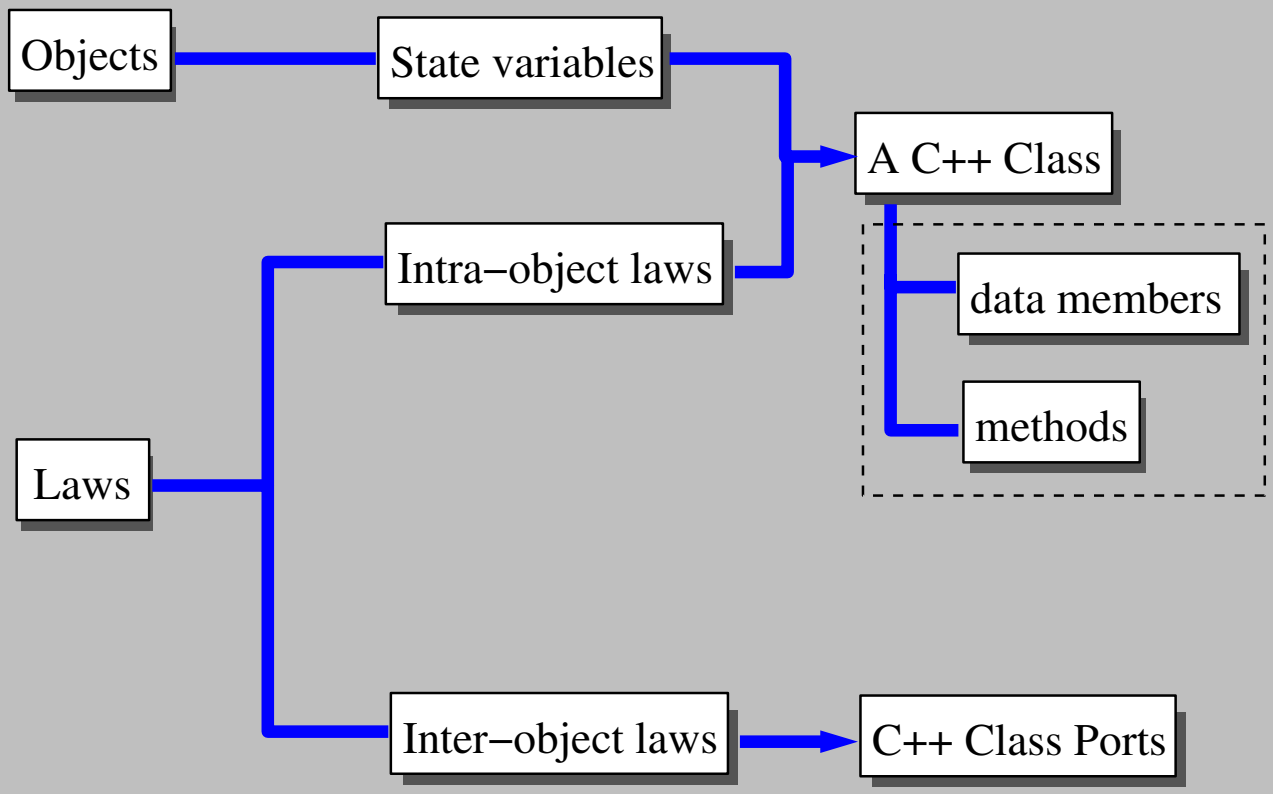
The Interactivity Concept:

(as compared to the offline simulation concept)



From Objects and Laws to Classes:

From the abstract object and law concepts we derive the concrete OO (C++) implementation concepts:

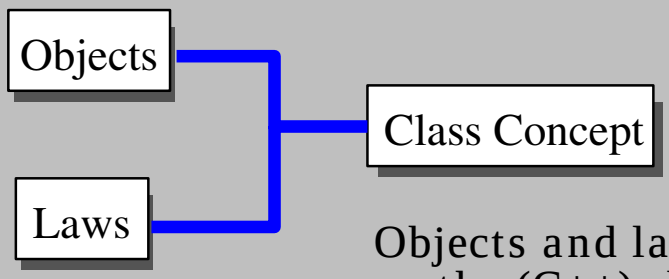


Overall: We implement objects and intra–object laws by C++ classes (data members and methods).

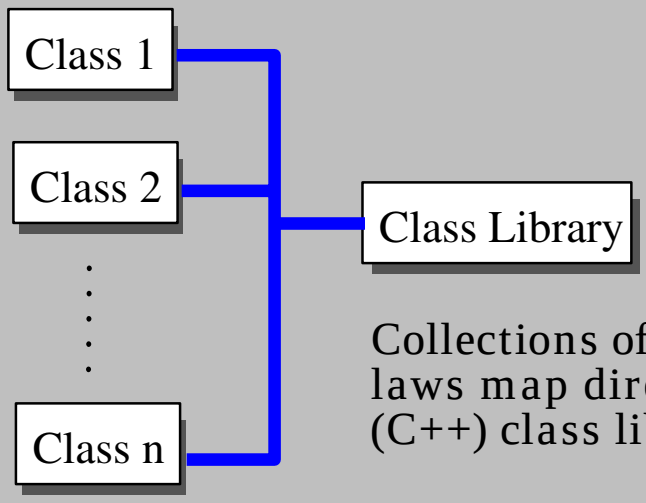
We implement inter–object laws by C++ class ports.

OO Modelling: Objects and Laws

Reasons for using OO modelling and C++:



Objects and laws map directly on the (C++) class concept

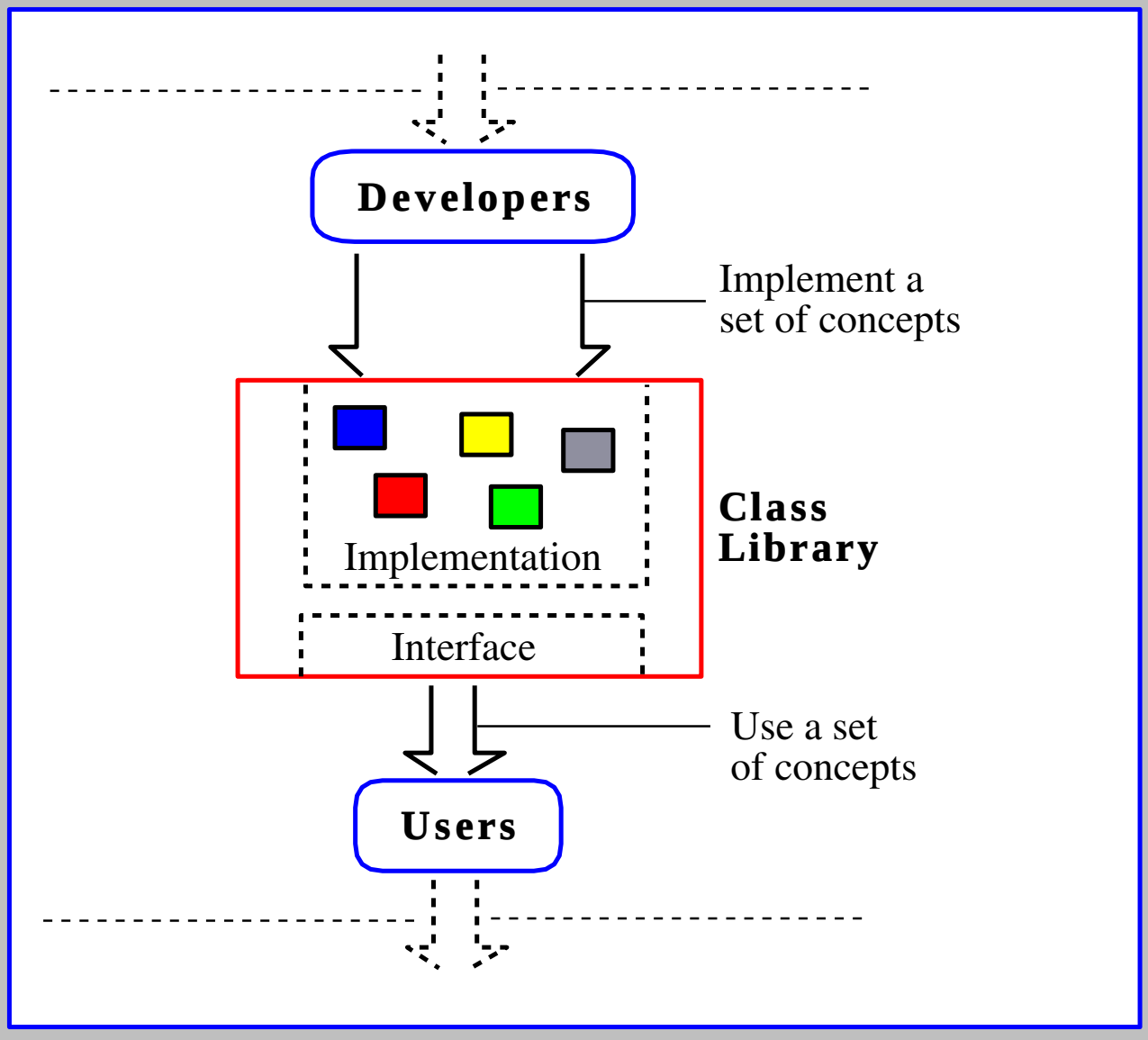


Collections of objects and laws map directly on the (C++) class library concept

The Class Library:

Is the central concept of reusing OO design.

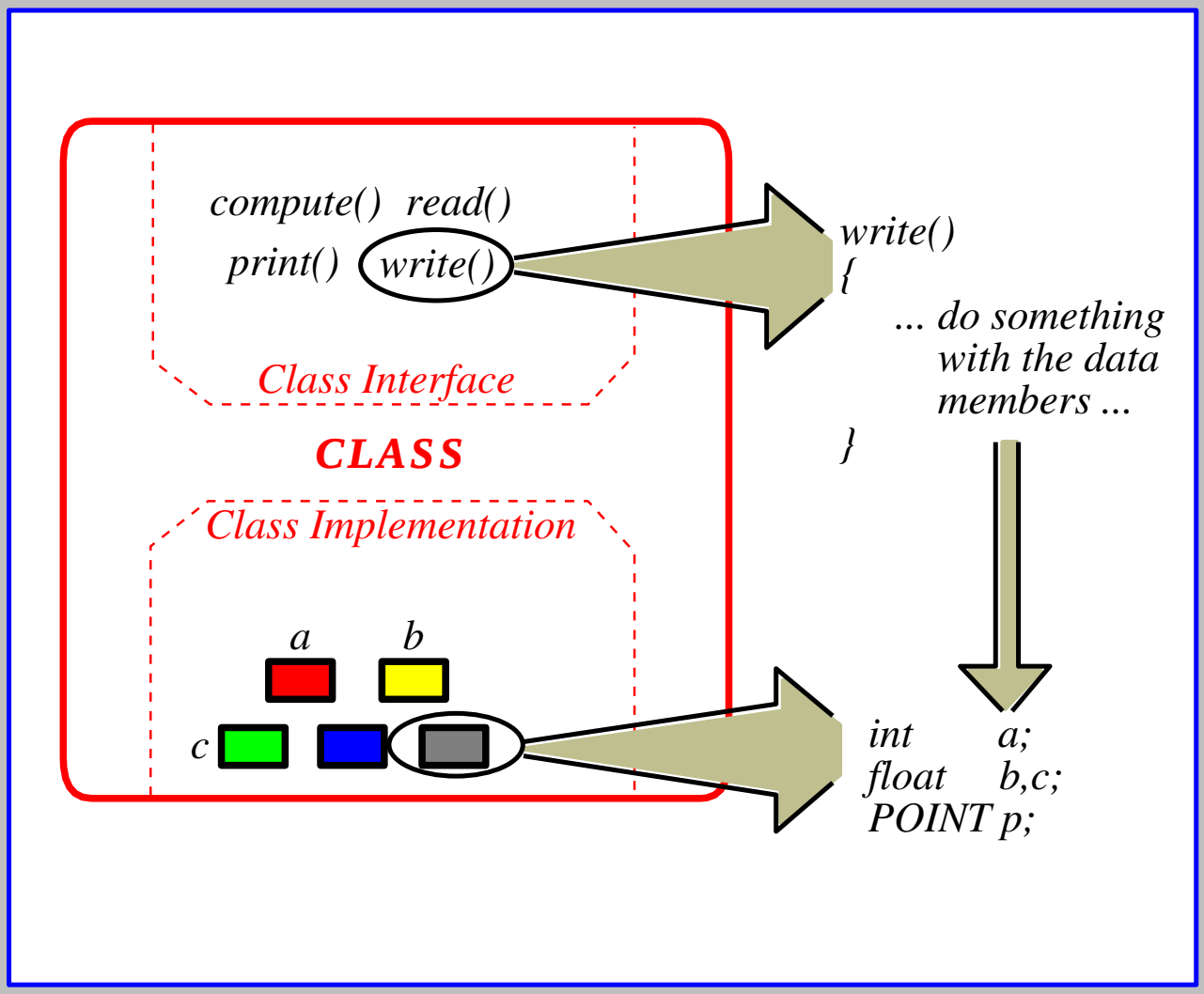
A class library is a set of cooperating *classes*.



The Class:

Is the central concept of a class library.

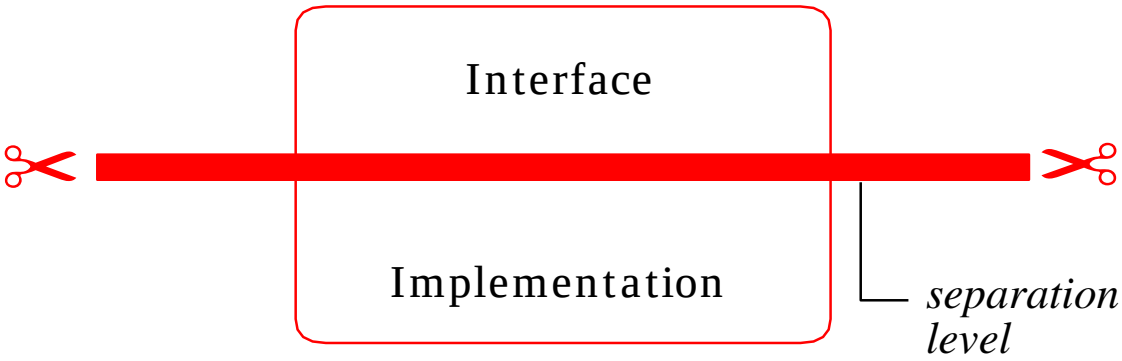
A class groups together *data* and *functions*.



Designing A Class:

Main rule of OO design:

separate design of interface from design of implementation

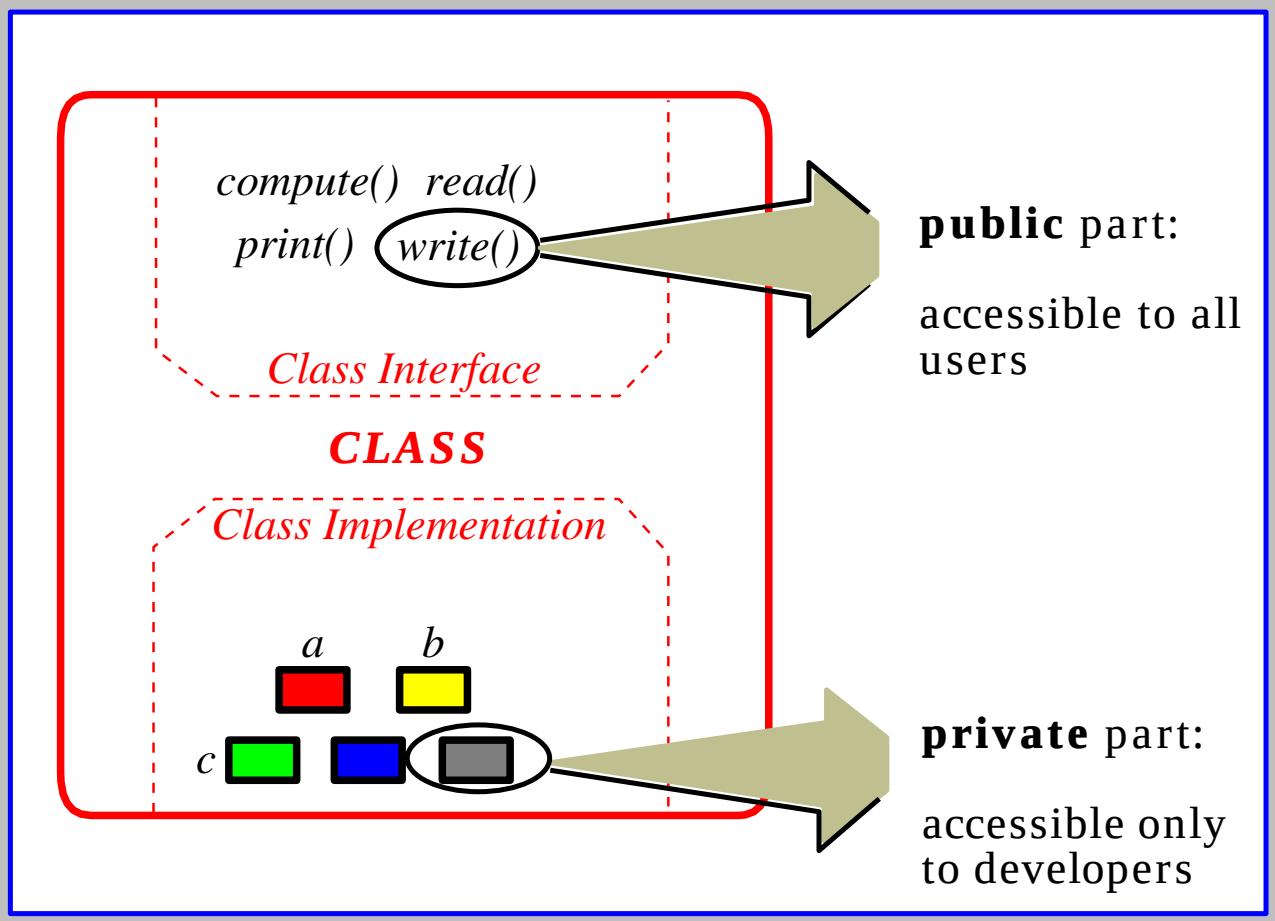


Advantages:

- implementation changes don't affect users
- minimize code rewriting and recompilation
- users program in terms of interfaces and ***NOT*** implementations

Class Concepts: Encapsulation

Basic tool for hiding implementation details:




```
class A
{
public:
        compute();
private:
        int a,b;
}
```

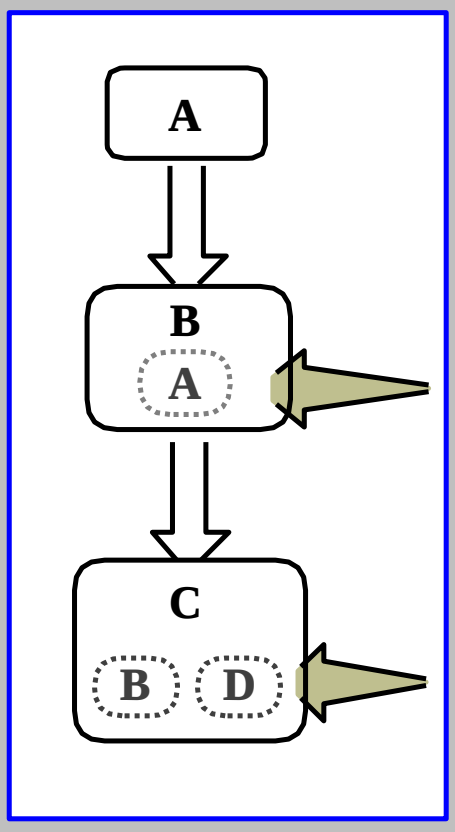
user accessible

hidden to user

Class Concepts: Inheritance

Powerful tool for code reuse and class specialization:

- implement a class in terms of other classes
  - → ***code reuse***
- add new features to an existing class
  - → ***class specialization***



B inherits from A

```
class B : A
{ ...
}
```

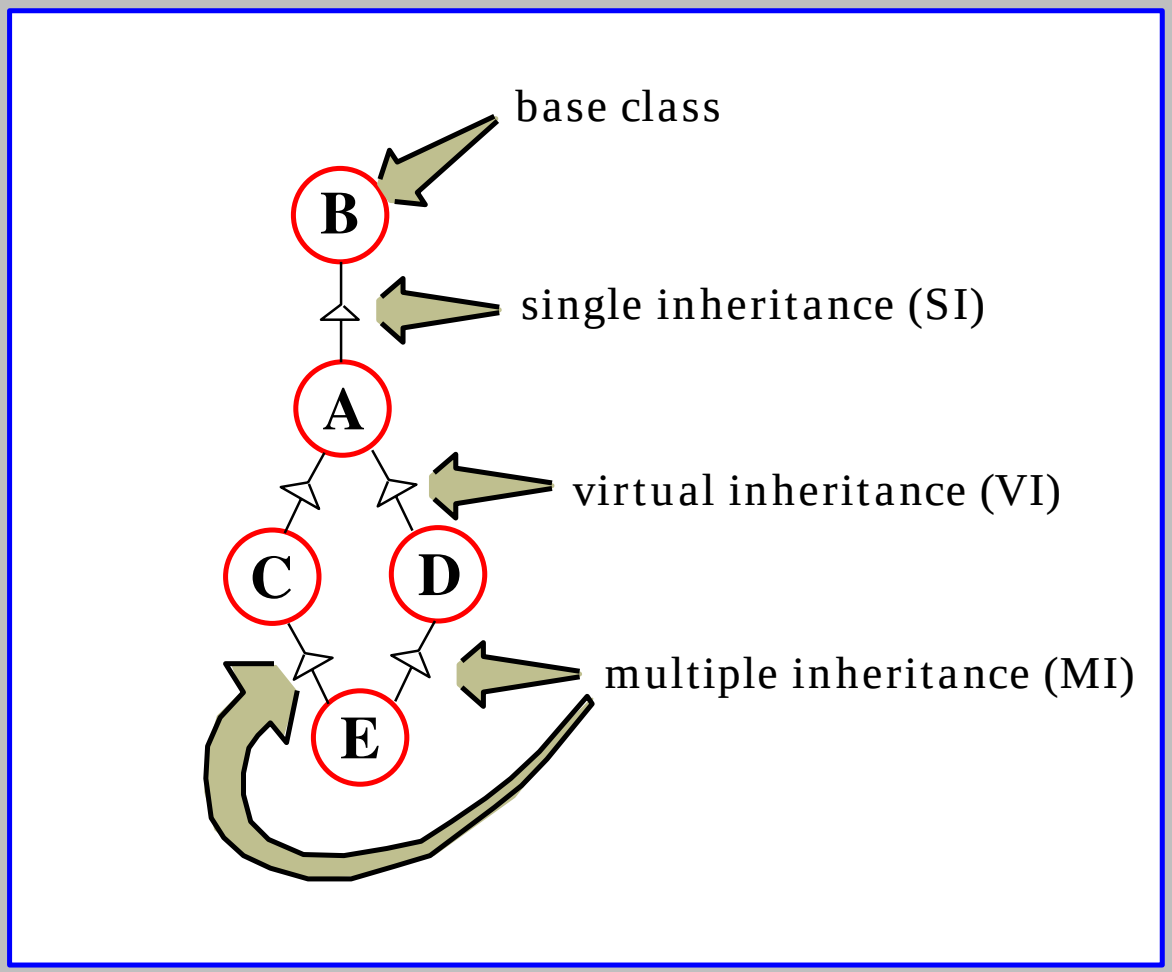
C inherits from B,D

```
class C : B,D
{ ...
}
```

Class Concepts: Inheritance (cont.)

Inheritance creates *class hierarchies*

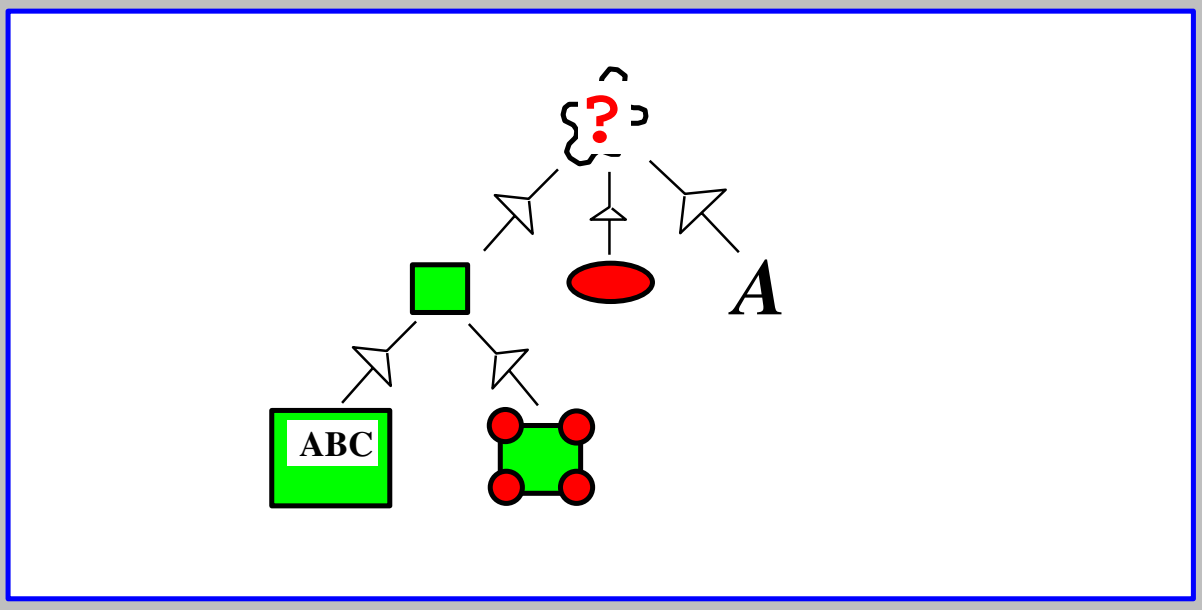
(directed acyclic graphs of classes):

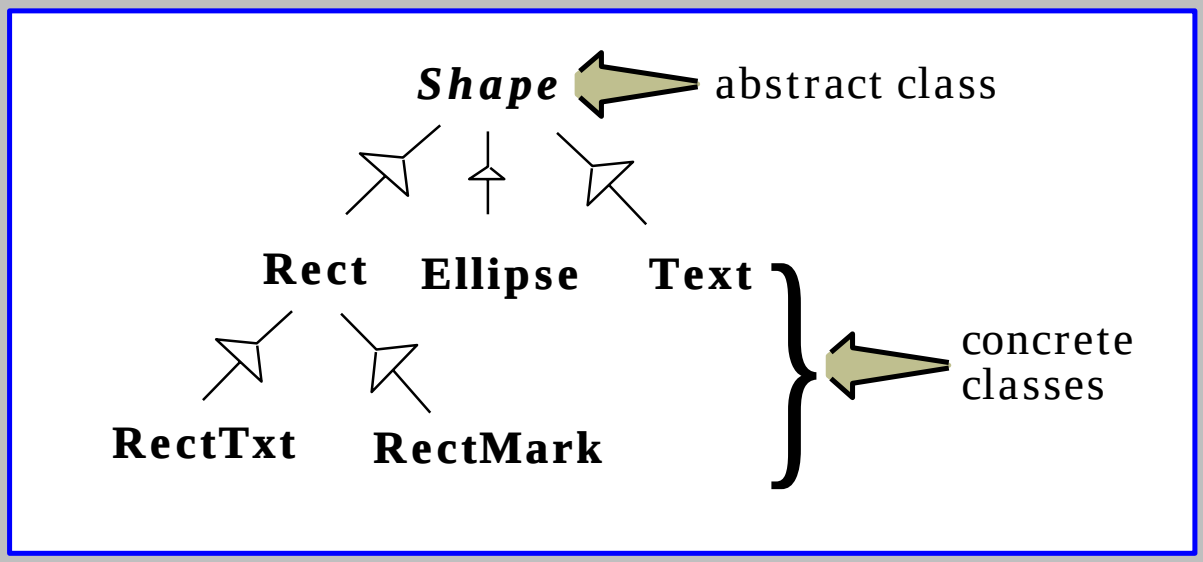


Class Concepts: Polymorphism

Is the key concept to extensible software:

Example: a class hierarchy of graphic shapes

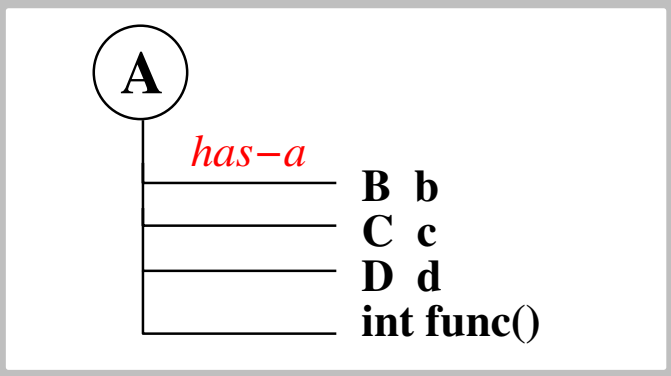




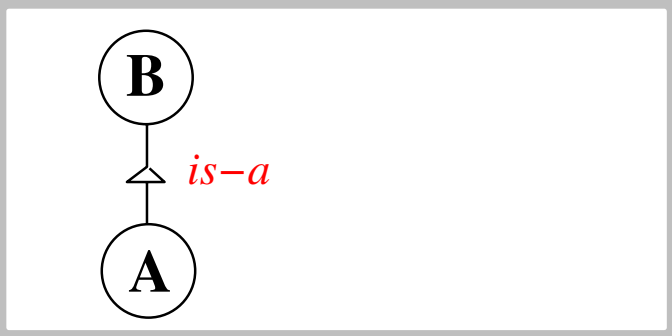
Class and Object Relationships:

Classes and objects can participate in relationships:

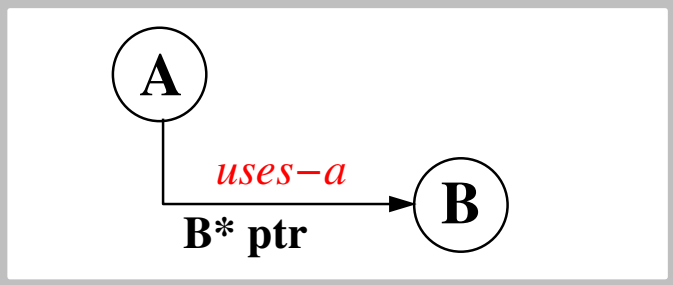
has-a: a class A has-a B if B is a member of A.



is-a: a class A is-a B if A is derived from B.

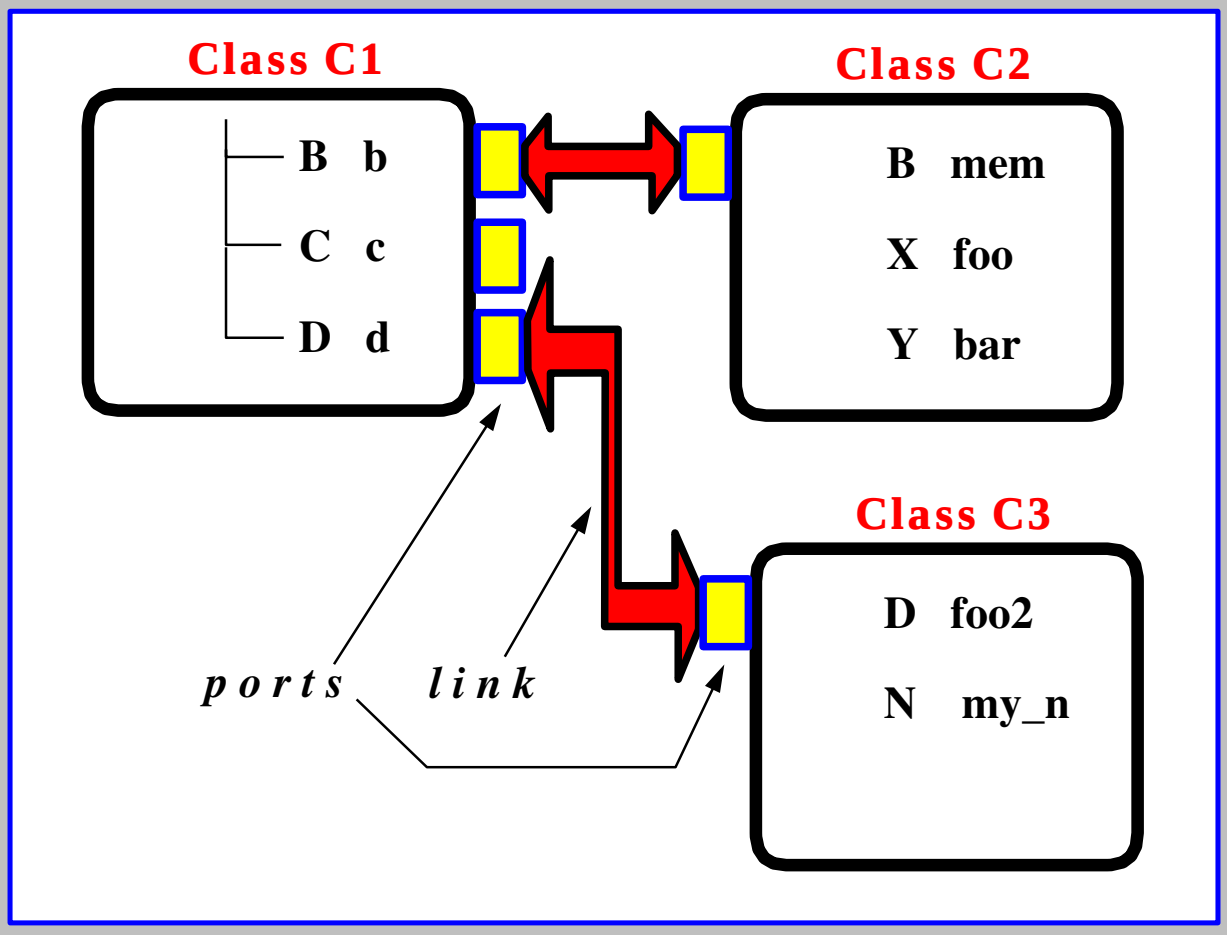


uses-a: a class A uses-a B if it has a B* member (a pointer-to-B member)



Class Ports:

Classes are provided with ports to establish inter−class relationships:



Example of inter−class constraints:

C1 :: d = C3 :: foo2

C1 :: b = C2 :: mem

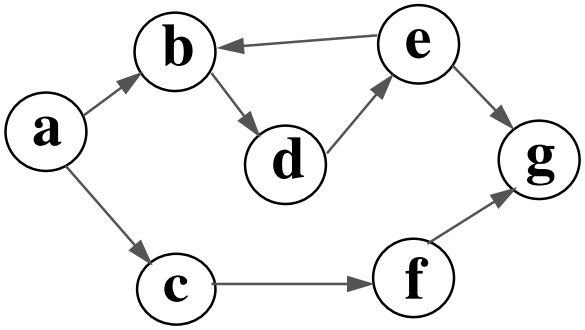
The Dependency Graph:

Inter−class constraints establish a dependency graph at simulation level.

Example: having the following constraints between objects `a,b,c,d,e,f`:

```
b = f1(a)
d = f2(b)
e = f3(d)
c = f4(a)
f = f5(c)
g = f6(c,f)
```

we obtain the equivalent dependency graph:



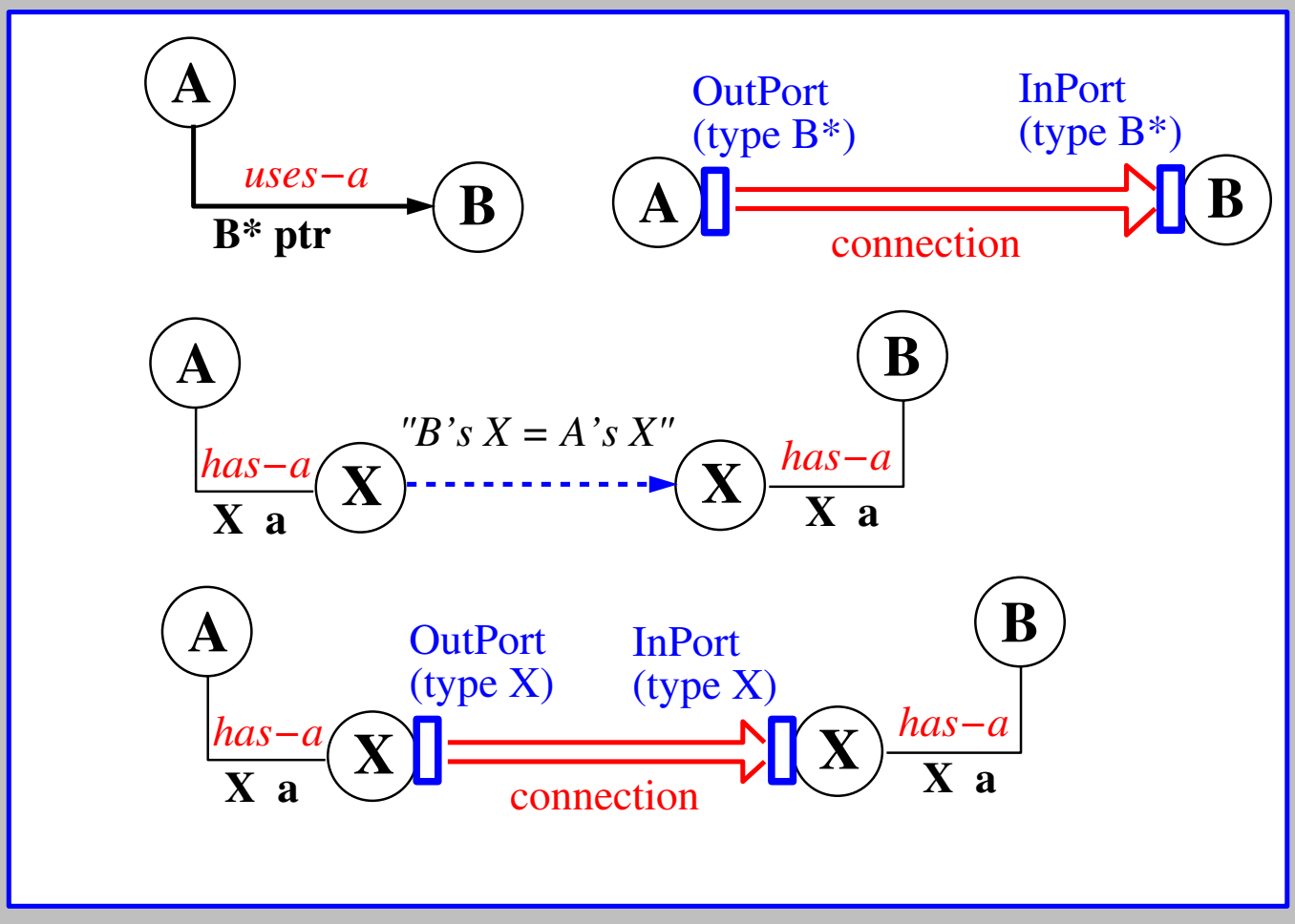
The Dependency Mechanism:

We create a constraint specification and management system over the C++ simulation classes.

Constraint specification is done by *ports*.

Ports:

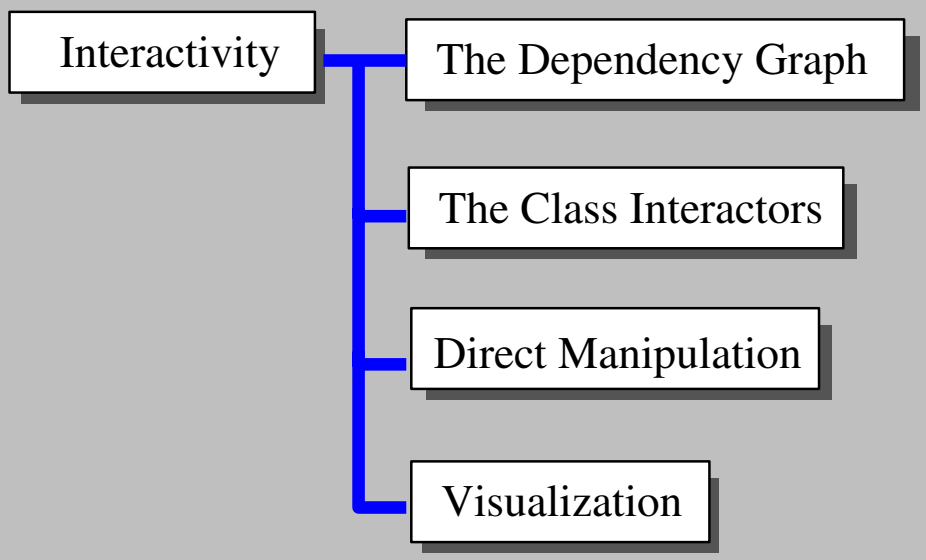
- are typed entities representing state parameters.
- are attached to classes.
- use class's parameter read/write methods.
- constraints are specified connecting ports of compatible types:



Interactivity:

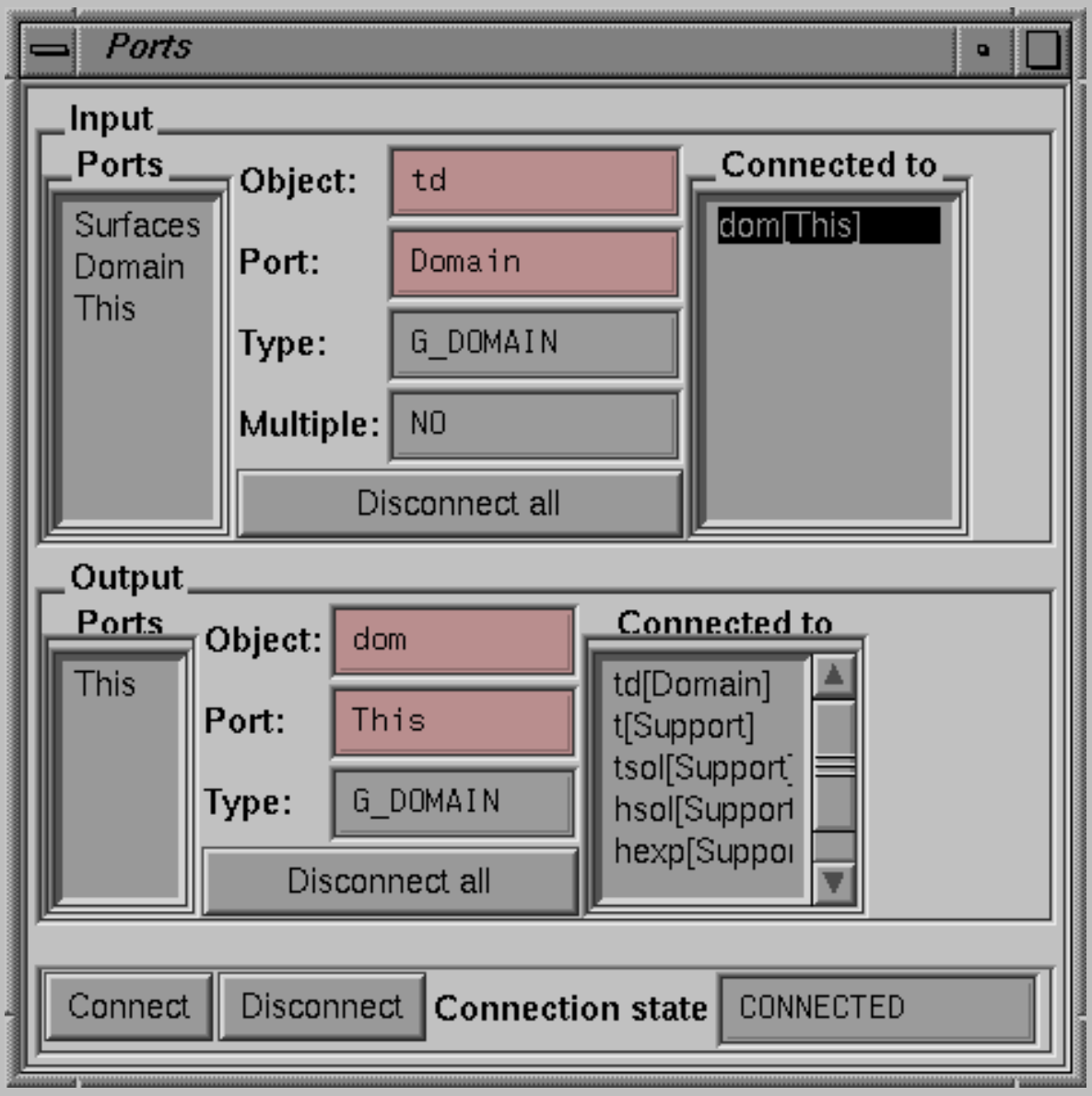
Interactivity has the following components:

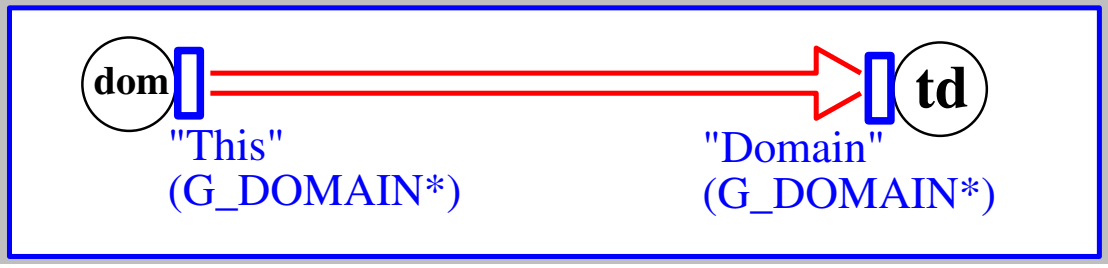
- building the dependency graph
- object manipulation via class interactors
- direct manipulation via cameras (OpenInventor)
- visualization via cameras (OpenGL, OpenInventor)



Building the Dependency Graph:

The user can explicitly establish data dependencies by connecting/disconnecting ports:



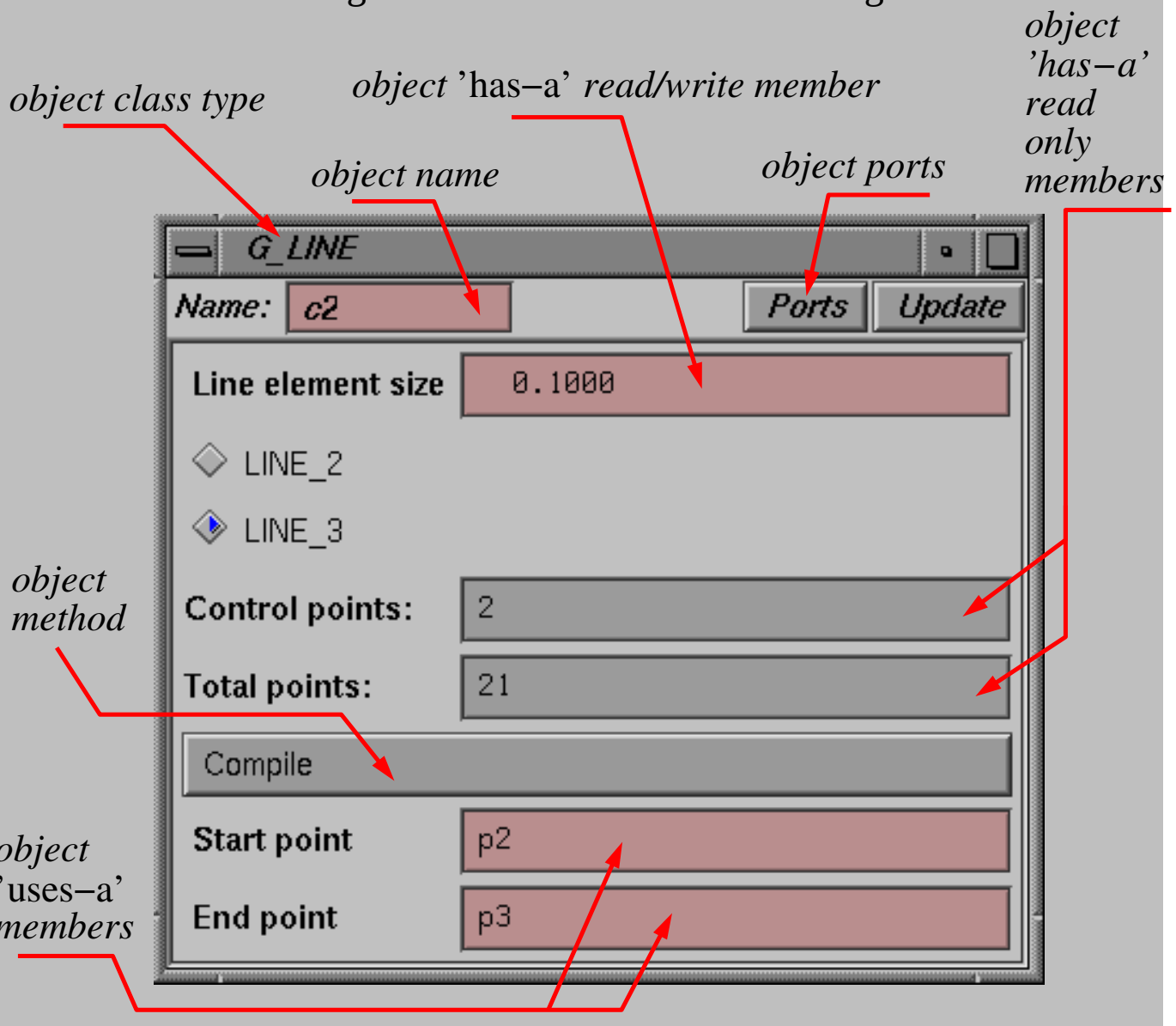


Class Interactors:

In order to interact with a class object, the system provides *interactors*.

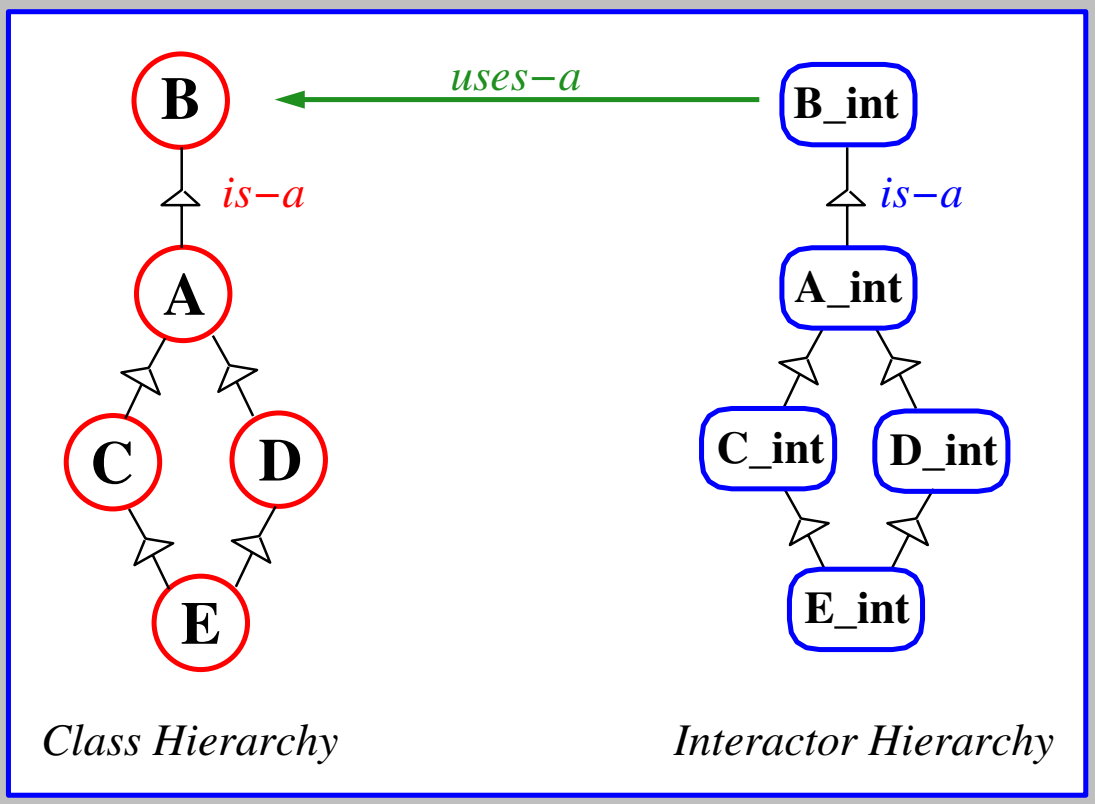
Interactors:

- are GUI representations of classes.
- allow reading/writing class members and calling class methods via GUI widgets.



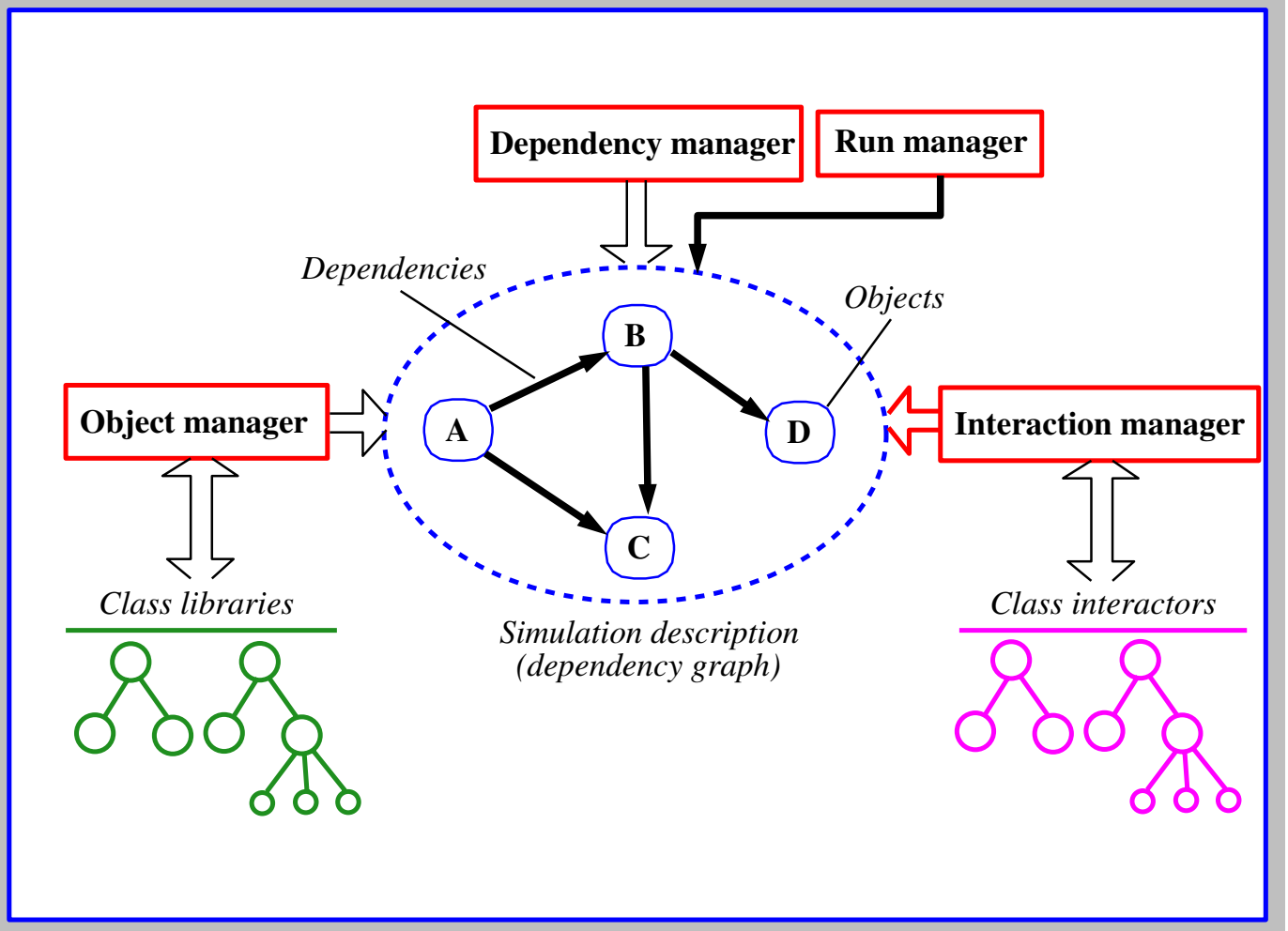
Class Interactors(cont.):

- The '*uses−a*' relations established by interactors are automatically translated into explicit (by reference) dependencies.
- A run−time type information (RTTI) component is used to check if dependencies are established between objects of the correct type.
- Class hierarchies are paralleled by interactor hierarchies:



- Class hierarchies are designed completely independent on interactor hierarchies (one−way loose coupling)

Simulation System Overview:

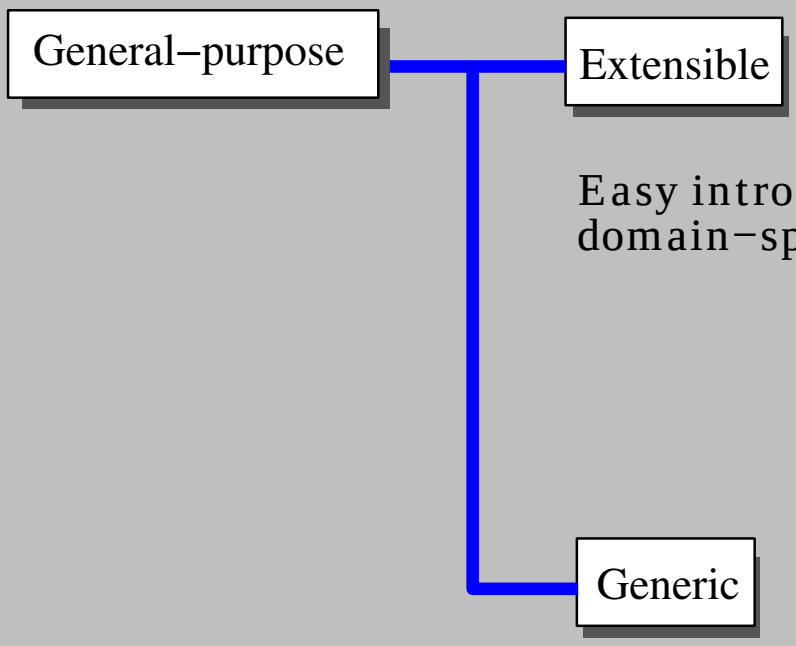


Legend:

- System functional components (managers)
- Simulation specification (data)
- Simulation class libraries (problem–specific classes)
- Class interactor libraries (for the simulation classes)

The General–Purpose Concept:

A general–purpose simulation system should easily accomodate applications coming from various scientific domains.



Visualization:

Here is an example of visualization using an OpenInventor−based camera: