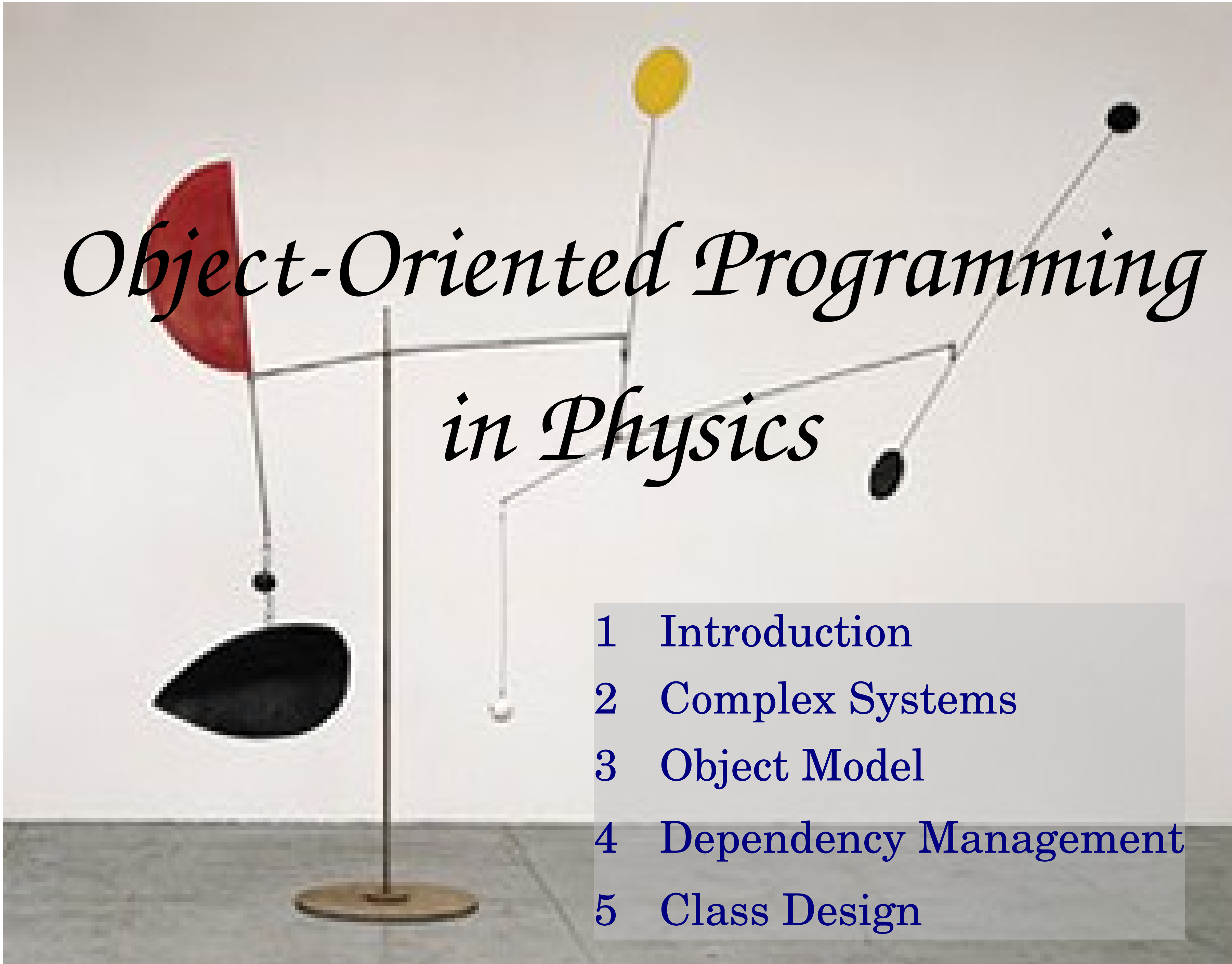


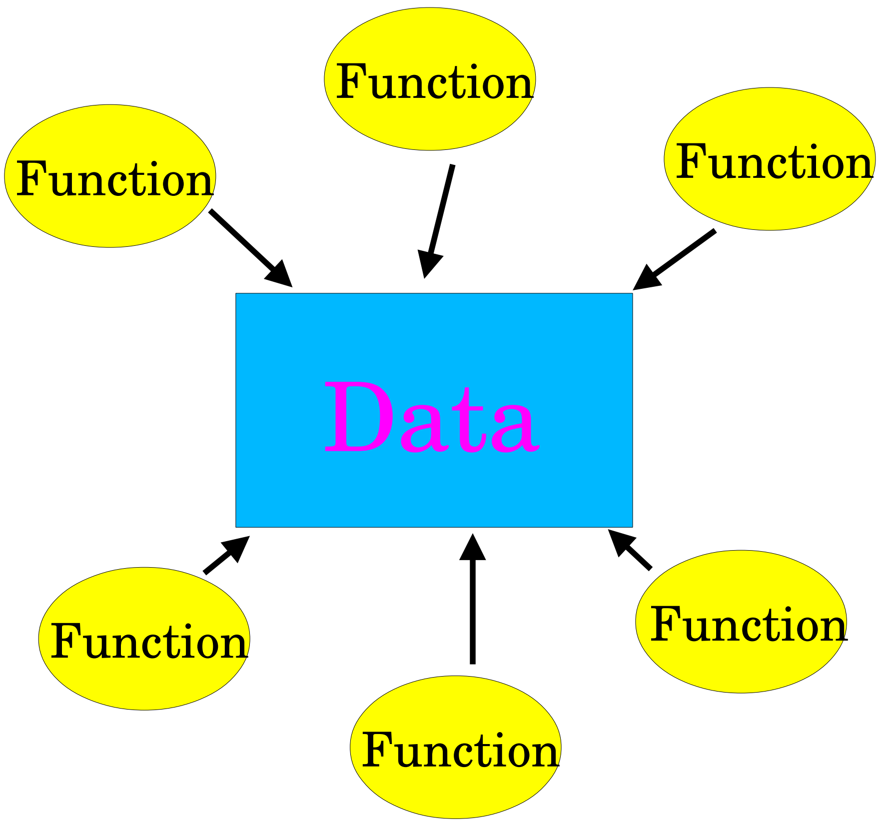
Object-Oriented Programming in Physics

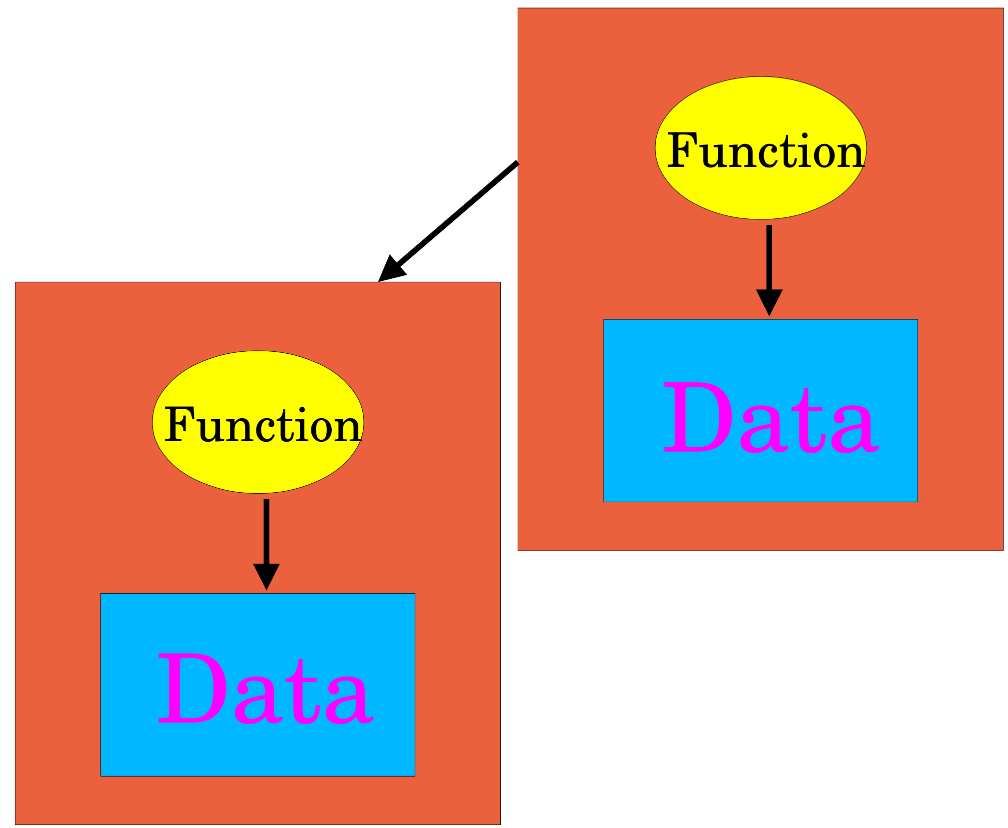
1 Introduction
2 Complex Systems
3 Object Model
4 Dependency Management
5 Class Design

1 What is OO?

- A method to design and build large programs with a long lifetime
  - e.g. O(10k) loc C++ with O(a) lifetime
  - Blueprints of systems before coding
  - Iterative development process
  - Maintainance and modifications
  - Control of dependencies
  - Separation into components

1 SA/SD and OO



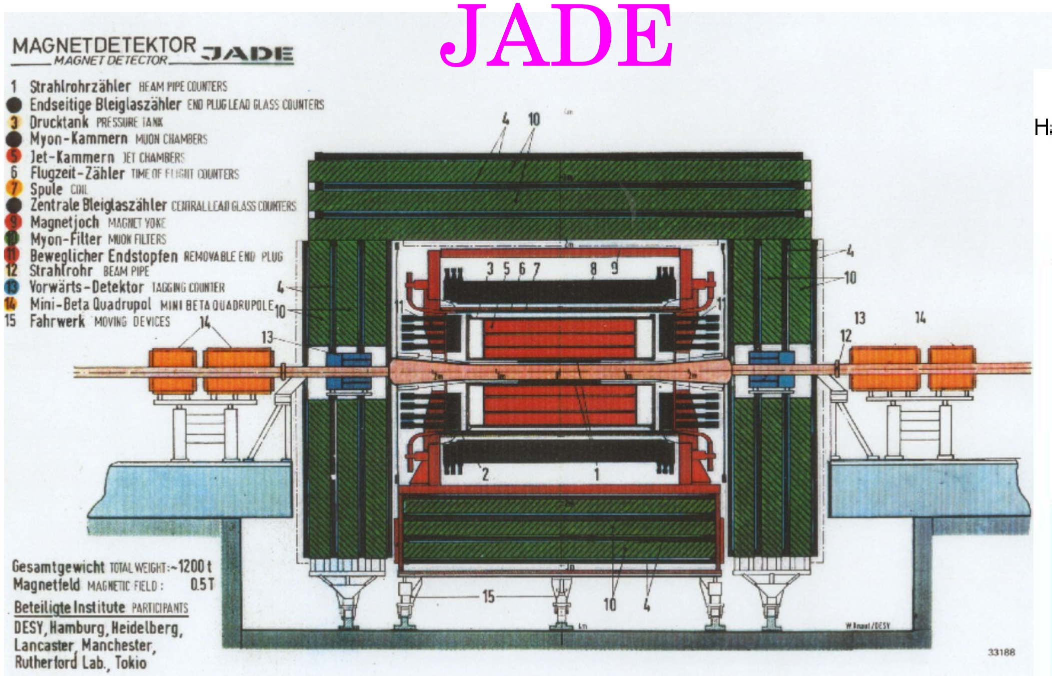


Top-down hierarchies of function calls and dependencies

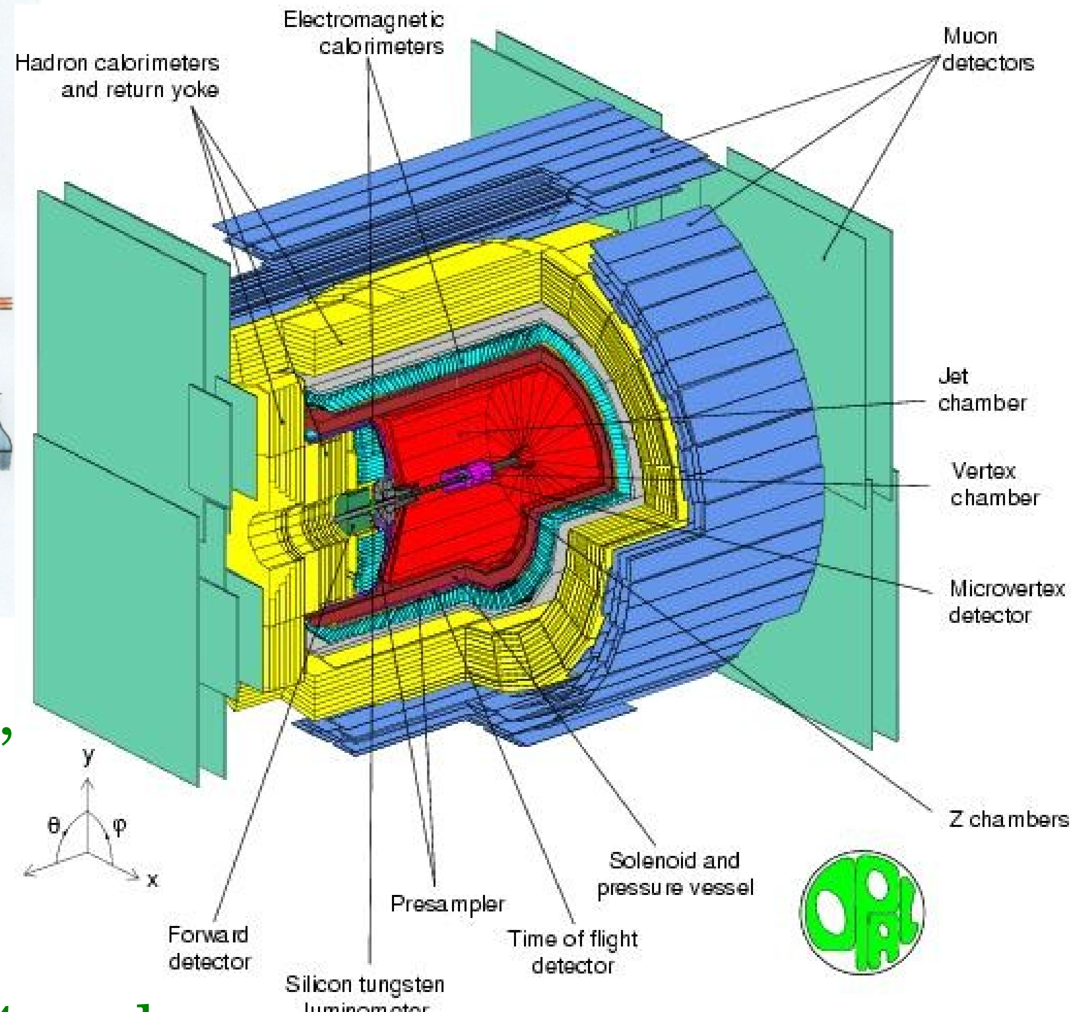
Bottom-up hierarchy of dependencies

1 Software in HEP Experiments

JADE



OPAL

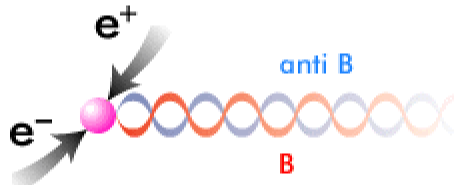


80's O(100) kloc, 2000 routines, 14 packages

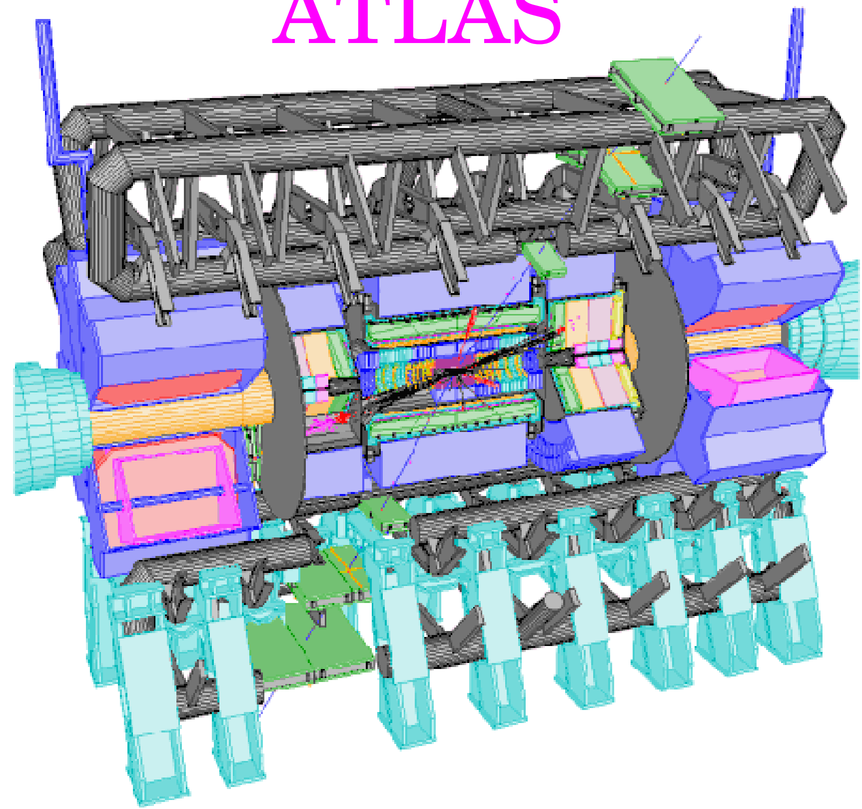
90's 500 kloc, 6900 routines, 54 packages

1 Software in HEP Experiments

BaBar



ATLAS



00's O(1) Mloc, O(10k) classes, O(1k) packages

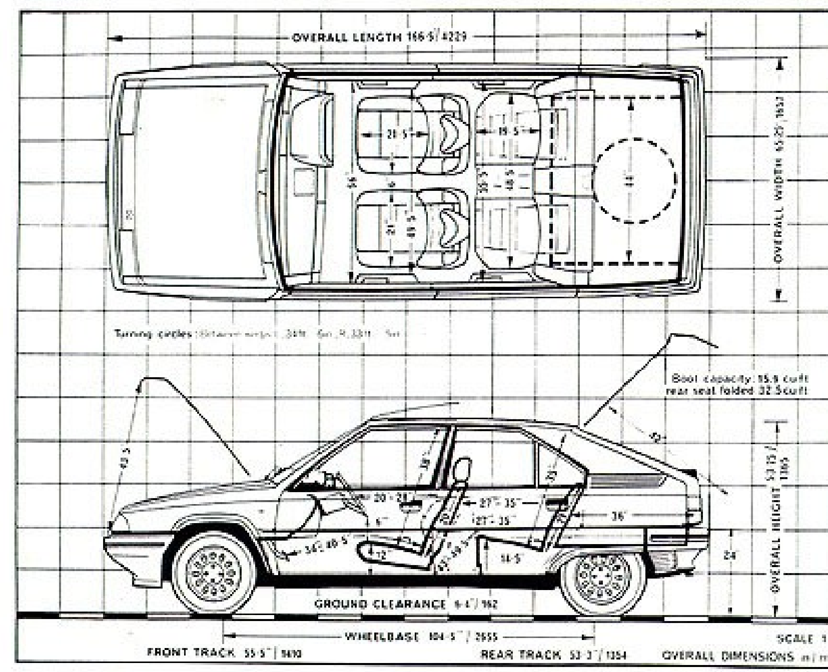
00's O(1) Mloc, O(1k) classes, O(100) packages

2 Complex Systems

- For our purpose complex systems (Booch):
  - have many states, i.e. large “phase space”,
  - are hard to comprehend in total
  - hard to predict
- Examples:
  - ant colony, an ant
  - computer
  - weather
  - a car

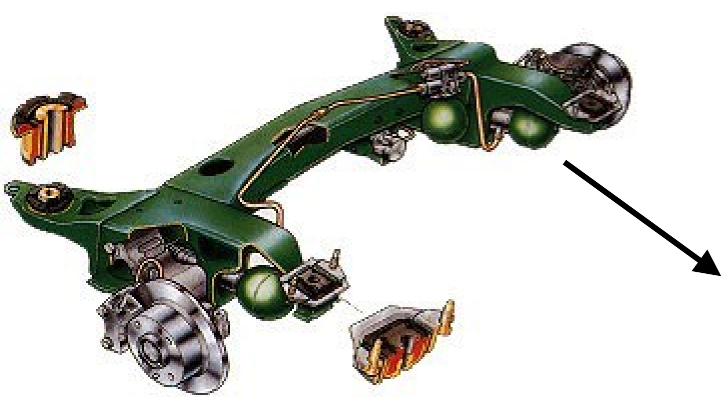
2 Complex Systems: Hierarchical

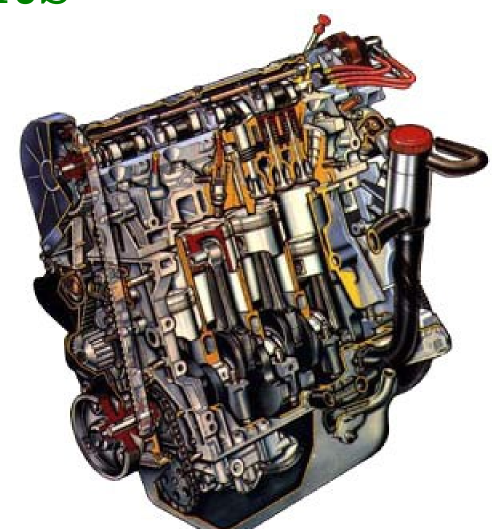
- Composed of interrelated subsystems
  - subsystems consist of subsystems too
  - until elementary component



2 Complex Systems: Components

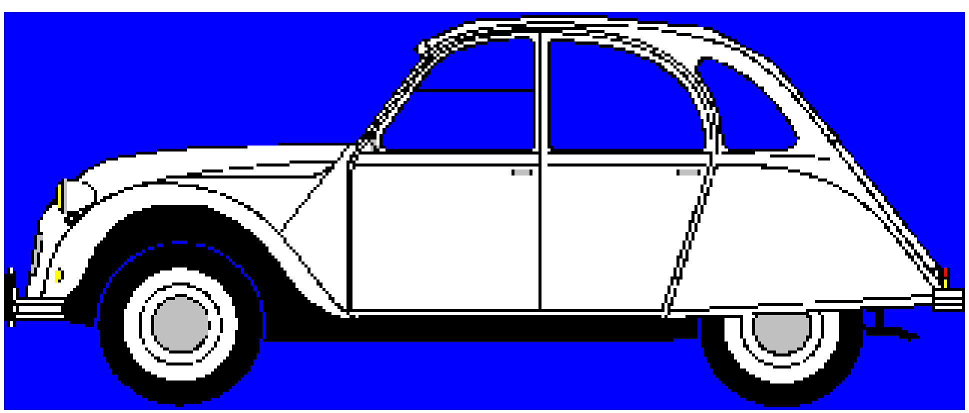
- Links (dependencies) within a component are stronger than between components
  - inner workings of components separated from interaction between components
  - service/repair/replace components





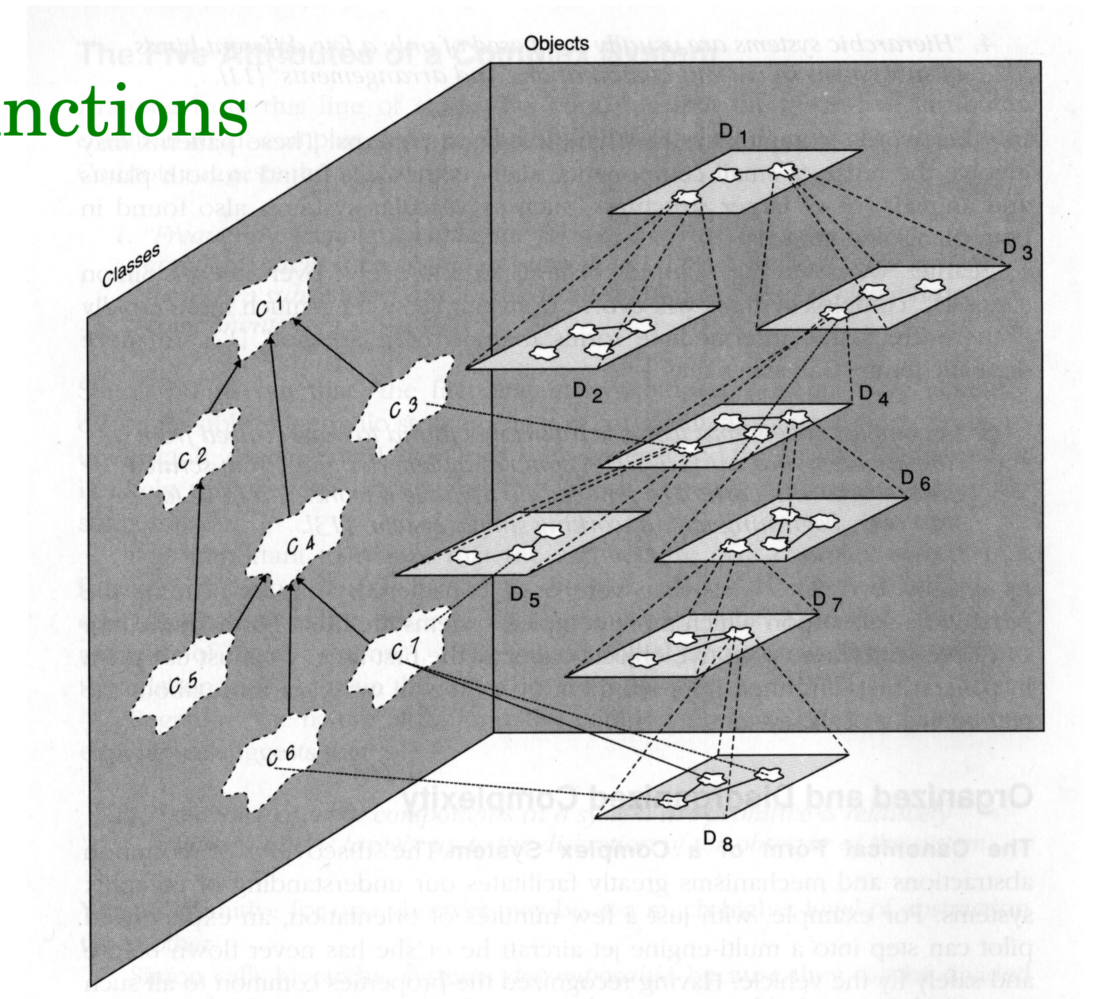
2 Complex Systems: Evolved from a simpler system

- Complex system designed from scratch rarely works
- Add new funtionality/improvements in small steps



2 Complex Systems: Two orthogonal views

- The *Object Structure*
  - “part of” hierarchy, functions
  - actual components
  - concrete
- The *Class Structure*
  - “is a” hierarchy
  - kinds of components
  - abstract



3 The Object Model

- Four essential properties
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy
- Two more useful properties
  - Type
  - Persistence

(Booch)

3 Abstraction

The characteristics of an object which make it unique and reflect an important concept

(following Booch)

Jackson Pollock, She-Wolf, 1943

3 Encapsulation

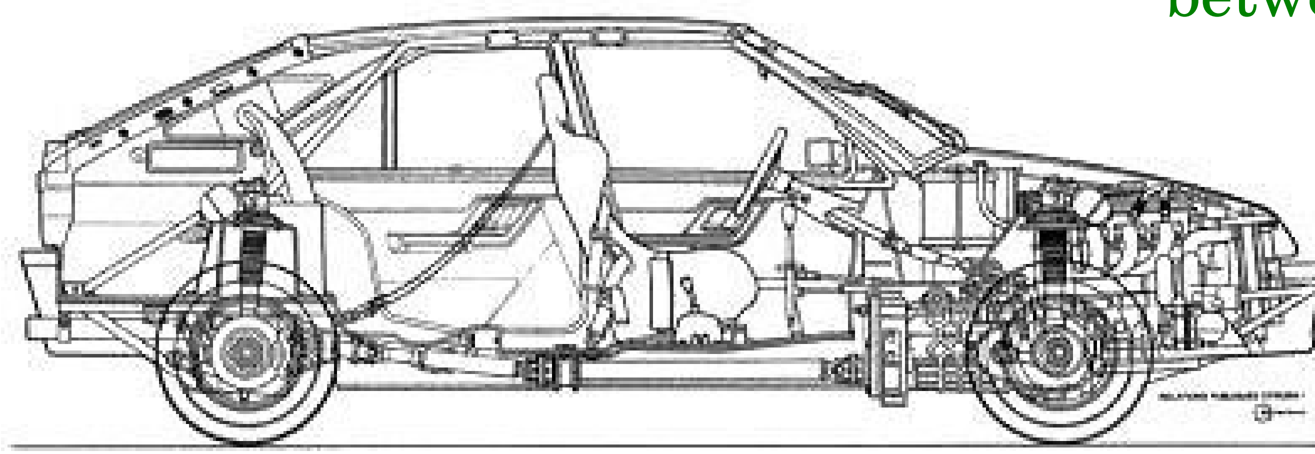
Separates interface of an abstraction from its implementation

Abstraction:	car
Interface:	steering, pedals, controls
Implementation:	you don't need to know, quite different between different makes or models

3 Modularity

Property of a system decomposed into cohesive and loosely coupled modules

Cohesive:	group logically related abstractions
Loosely coupled:	minimise dependencies between modules



3 Hierarchy

Hierarchy is a ranking or ordering of abstractions

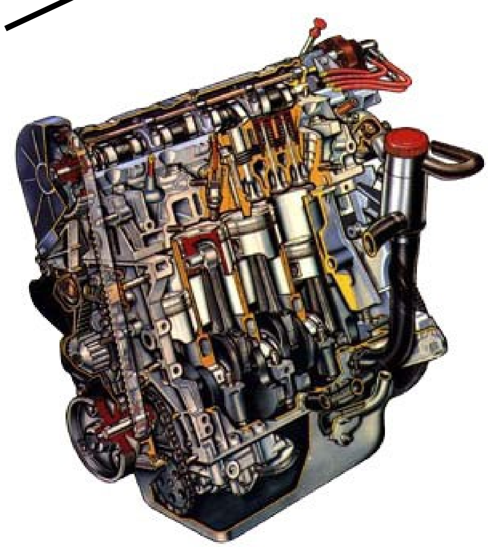
PSA A8A turbo diesel engine → Turbo diesel engine

VW 1.9 TDI → Turbo diesel engine

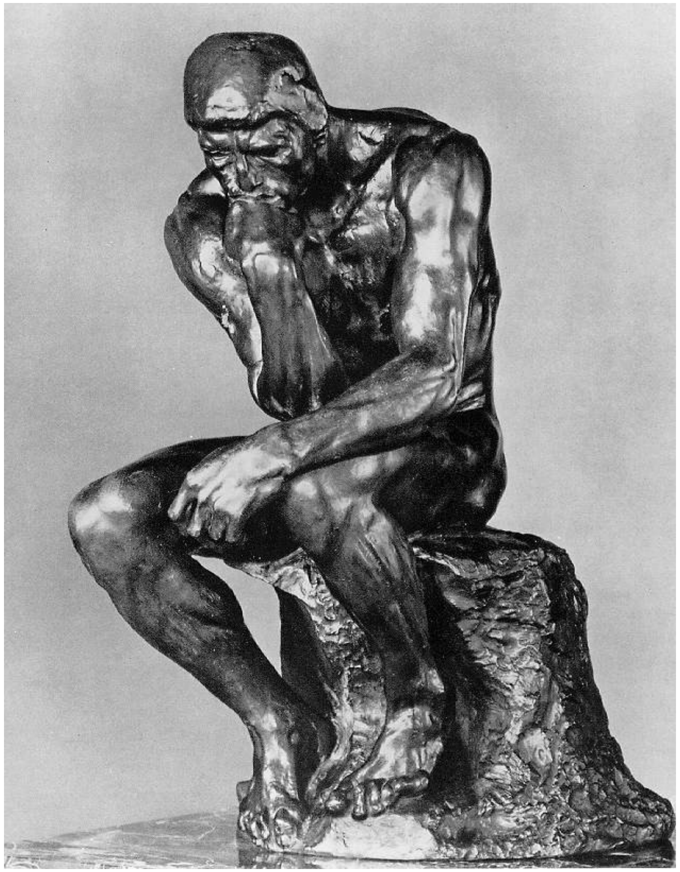
Turbo diesel engine → Diesel engine

Diesel engine → Internal combustion engine

Petrol engine → Internal combustion engine

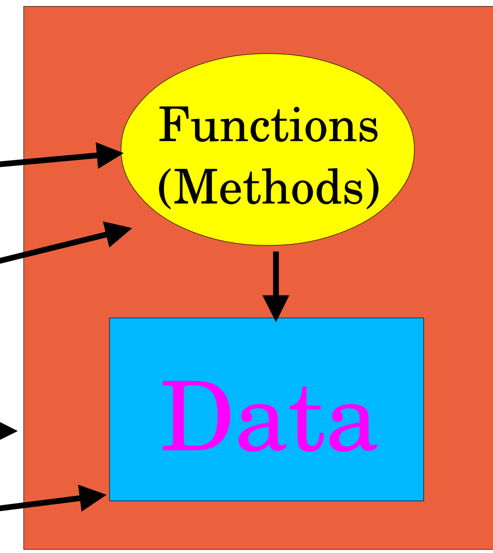


3 What is an Object?



An object has:

- interface
- behaviour
- identity
- state



Interface (*how to use it*):
  Method signatures

Behaviour (*what it does*):
  Algorithms in methods

Identity (*which one is it*):
  Address or instance ID

State (*what happened before*):
  Internal variables

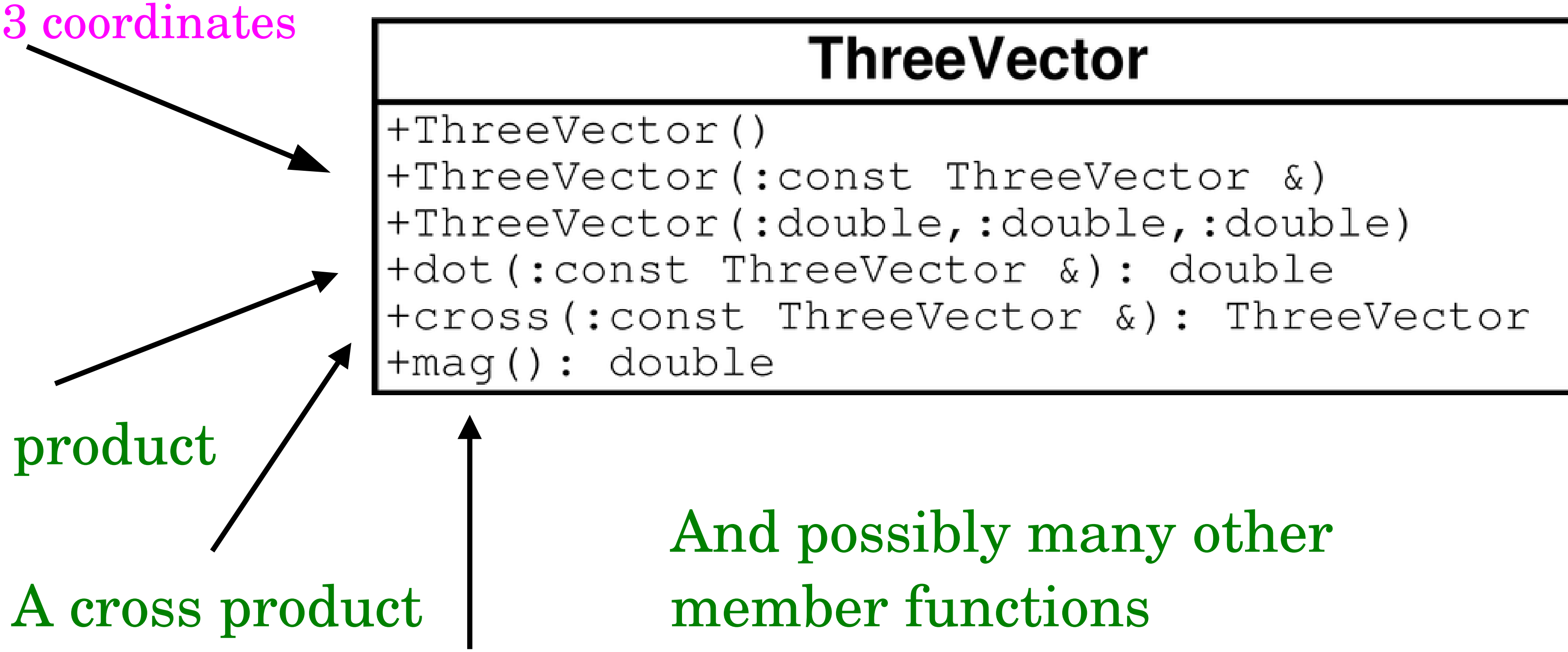
3 Object Interface

How to use it

Create an object (constructors)

- from nothing (default)
- from another object (copy)
- from 3 coordinates

The object interface is given by its *member functions* described by the objects class



A dot product

A cross product

Magnitude

And possibly many other member functions

3 Object Behaviour

What it does

```
class ThreeVector {

   public:

      ThreeVector() { x=0; y=0; z=0 };
      ...

      double dot( const ThreeVector & ) const;
      ThreeVector cross( const ThreeVector & ) const;
      double mag() const;

      ...

   private:

      double x,y,z;

}
```

Default constructor sets to 0

Dot and cross are unambigious

Magnitude, user probably expects 0 or a positive number

const means state of object does not change (vector remains the same) when this function is used

3 Object Identity

Which one is it

```
...

ThreeVector a;
ThreeVector b(1.0,2.0,3.0);

...

ThreeVector c(a);
ThreeVector d= a+b;

...

ThreeVector* e= new ThreeVector();
ThreeVector* f= &a;
ThreeVector& g= a;

...

double md= d.mag();
double mf= f->mag();
double mg= g.mag();

...
```

There can be many objects (instances) of a given class:

Symbolically:
$a \neq b \neq c \neq d \neq e$
but $f = g = a$

Pointer (*): Address of memory where object is stored; can be changed to point to another object

Reference (&): Different name for identical object

3 Object State

What happened before

Different objects of the same class have
- different identity
- different state
- possibly different behaviour
- but always the same interface

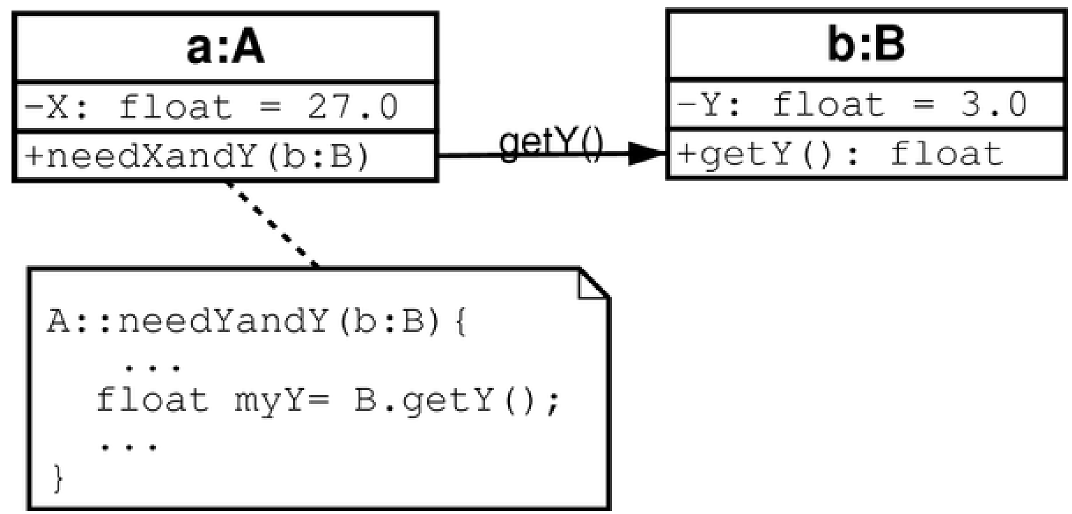
The internal state of an object is given by its data members

```
p: ThreeVector
-------------------------------------------
-x: double = 2.356
-y: double = 19.45
-z: double = -5.284
-n: int = 5
-------------------------------------------
+ThreeVector()
+ThreeVector(:const ThreeVector &)
+ThreeVector(:double,:double,:double)
+dot(:const ThreeVector &): double
+cross(:const ThreeVector &): ThreeVector
+mag(): double
```

3 Object Interactions

Objects interact through their interfaces only

Objects manipulate their own data but get access to other objects data through interfaces only



Most basic: get() / set(...) member functions, but usually better to provide "value added services", e.g.

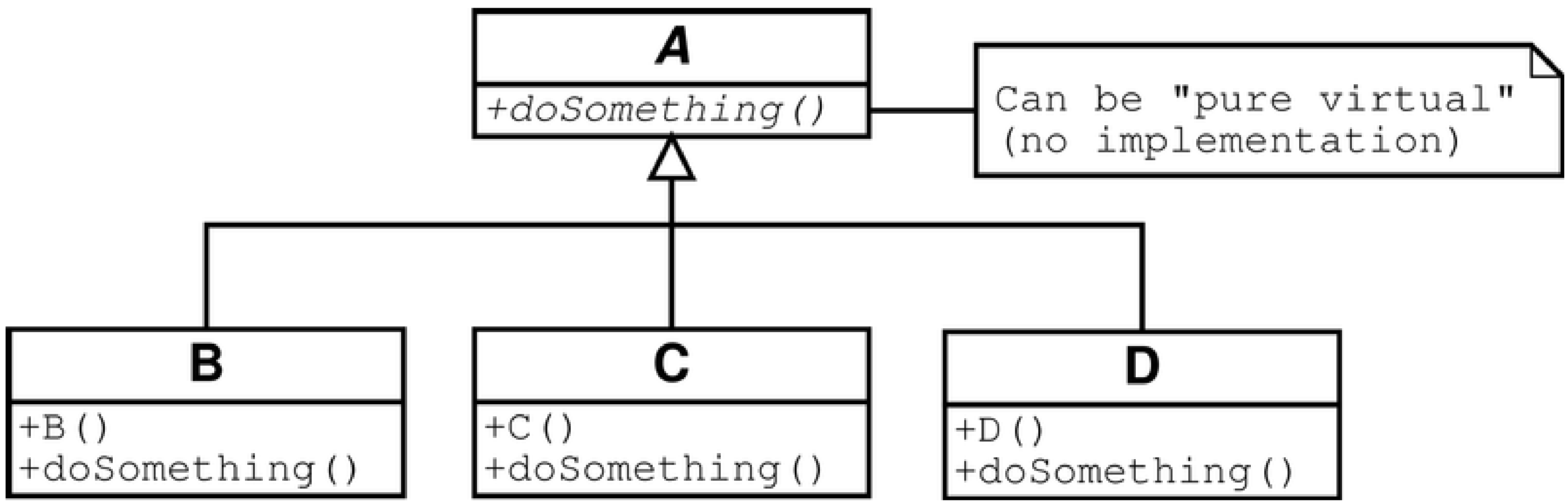
- fetch data from storage
- perform an algorithm

Also called message passing

3 Objects and Classes

- Objects are described by classes
  - blueprint for construction of objects
  - OO program code resides in classes
- Objects have type specified by their class
- Classes can inherit from each other
  - implies special relation between corresponding objects
- Object interfaces can be separated from object behaviour and state

3 Dynamic Object Polymorphism



Objects of type A are actually of type B, C or D

Objects of type A can take many forms, they are polymorph

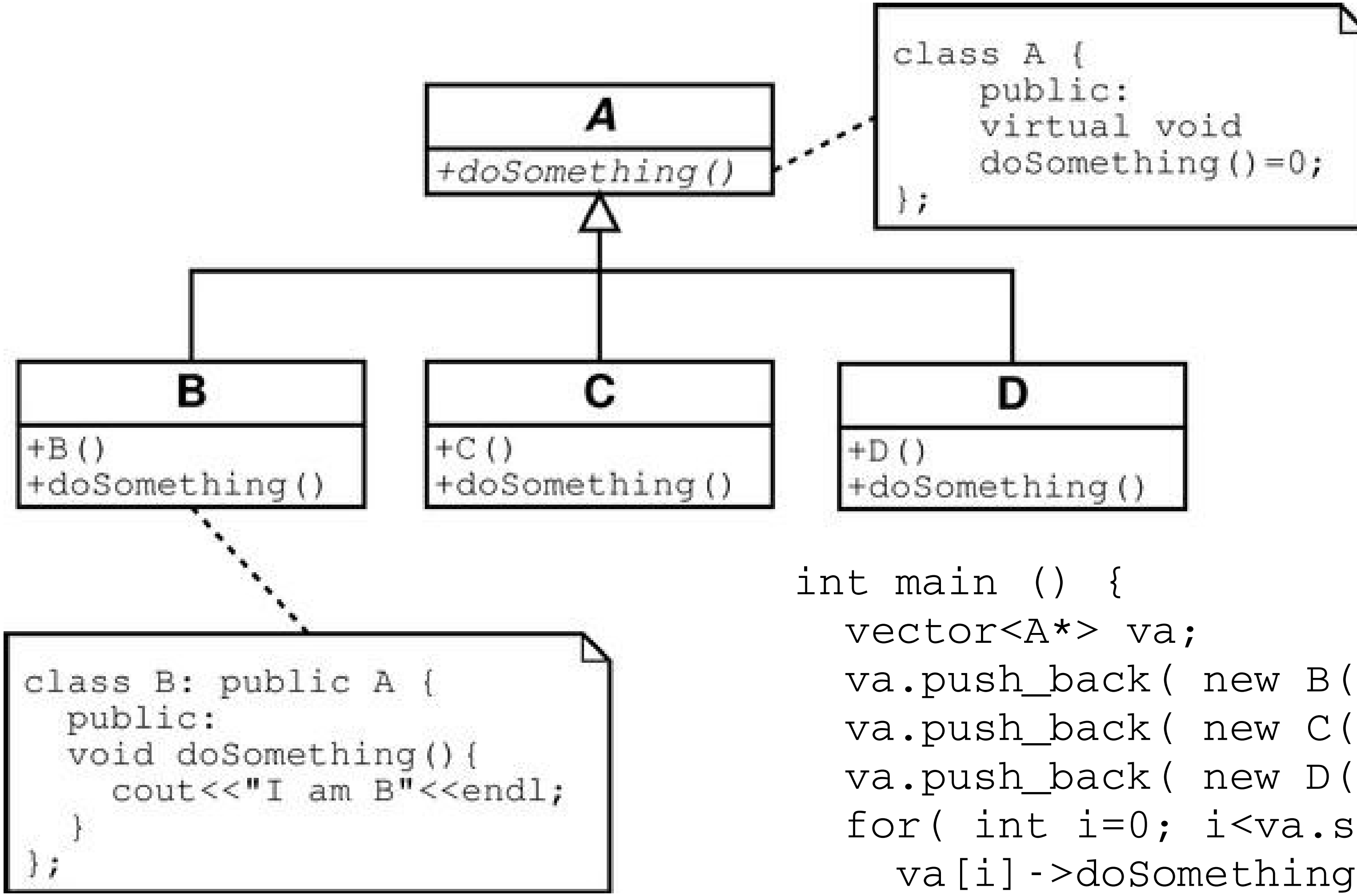
Code written in terms of A will not notice the difference

but will produce different results

Can separate generic algorithms from specialisations

Avoids explicit decisions in algorithms (if/then/else or case)

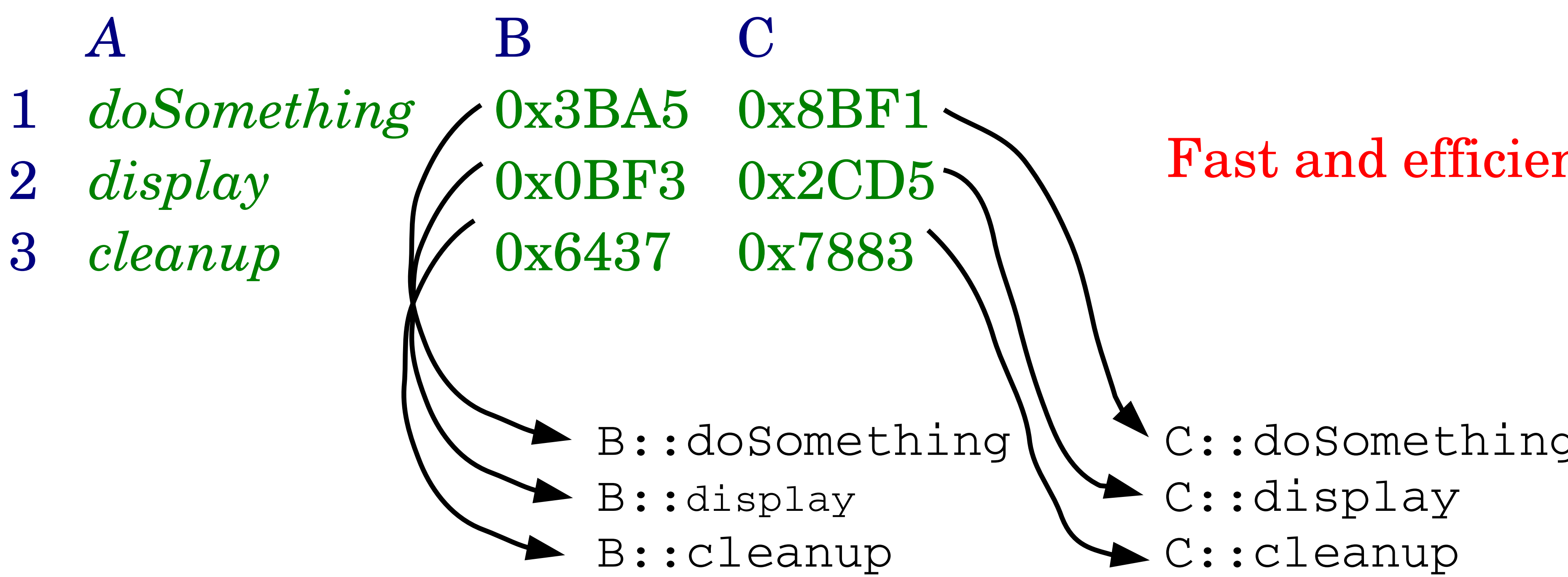
3 Dynamic Object Polymorphism




```
int main () {
  vector<A*> va;
  va.push_back( new B() );
  va.push_back( new C() );
  va.push_back( new D() );
  for( int i=0; i<va.size(); i++ ){
    va[i]->doSomething();
  }
}
```

3 Mechanics of Dynamic Polymorphism

Virtual function table with function pointers
in strongly typed languages, e.g. C++, Java



Fast and efficient!

Lookup by name in hash-tables in weak+dynamically typed languages (Perl, Python, Smalltalk)

3 Inheritance SA/SD vs OO

SA/SD (procedural):

Inherit for functionality

We need some function, it exists in class A → inherit from A in B and add some more functionality

☹

OO:

Inherit for interface

There are some common properties between several objects → define a common interface and make the objects inherit from this interface

☺

4 Dependency Management

- The parts of a project depend on each other
  - Components, programs, groups of classes, libraries
- Dependencies limit
  - flexibility
  - ease of maintainance
  - reuse of components or parts
- Dependency management tries to control dependencies

4 Problems with Software

- Rigid
- Fragile
- Not Reuseable
- High Viscosity
- Useless Complexity
- Repetition
- Opacity

These statements apply to an average physicist/programmer who develops and/or maintains some software system.

Software gurus will always find some solution in their code.

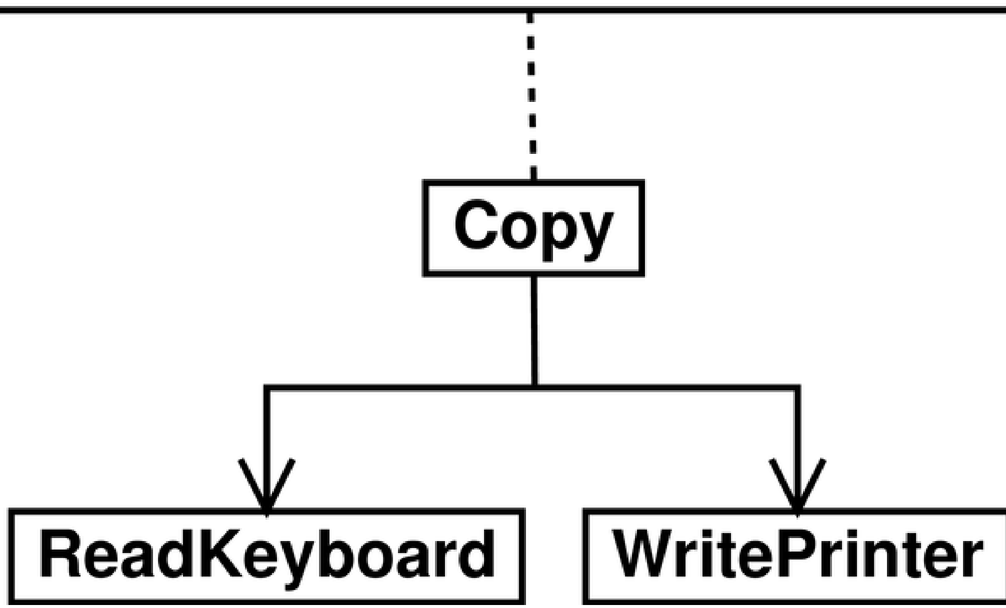
Do you want to rely on the guru? What if that person retires, finds a well-paid job or gets moved to another project?

4 Example: The Copy Routine

- Code rots
- There are many reasons for code rot
- We'll make a case study (R. Martin)
- A routine which reads the keyboard and writes to a printer

4 Copy Version 1

```
void Copy(void) {
  char ch;
  while( (ch= ReadKeyboard()) != EOF ) {
    WritePrinter( ch );
  }
}
```



A simple solution

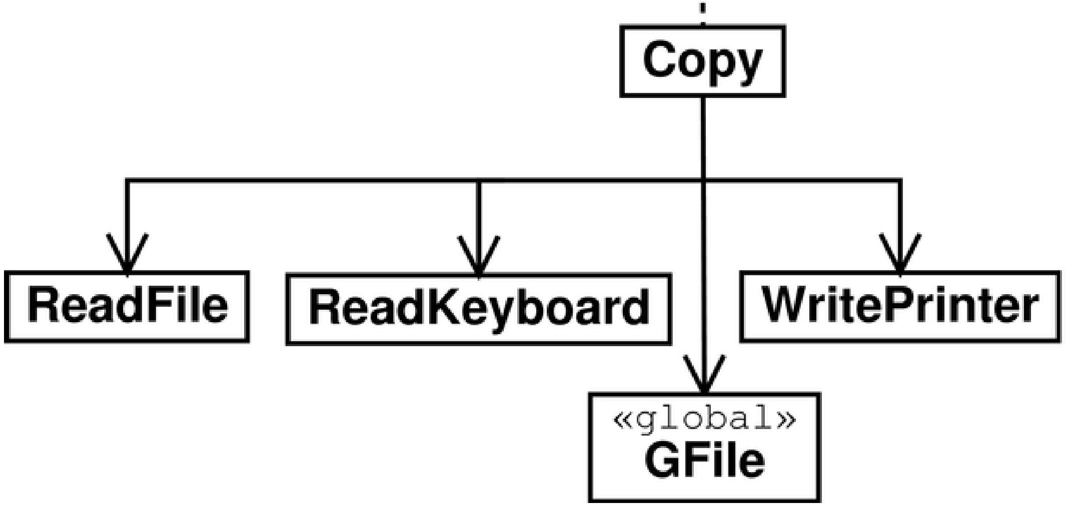
to a simple problem

ReadKeyboard and WritePrinter are probably reuseable

4 Copy Version 2

```
bool GFile;

void Copy(void) {
  char ch= 0;
  while( ch != EOF ) {
    if( GFile ) { ch= ReadFile(); }
    else { ch= ReadKeyboard(); }
    WritePrinter( ch );
  }
}
```



Many users want to read files too ...

But they don't want to change their code ... can't put a flag in the call

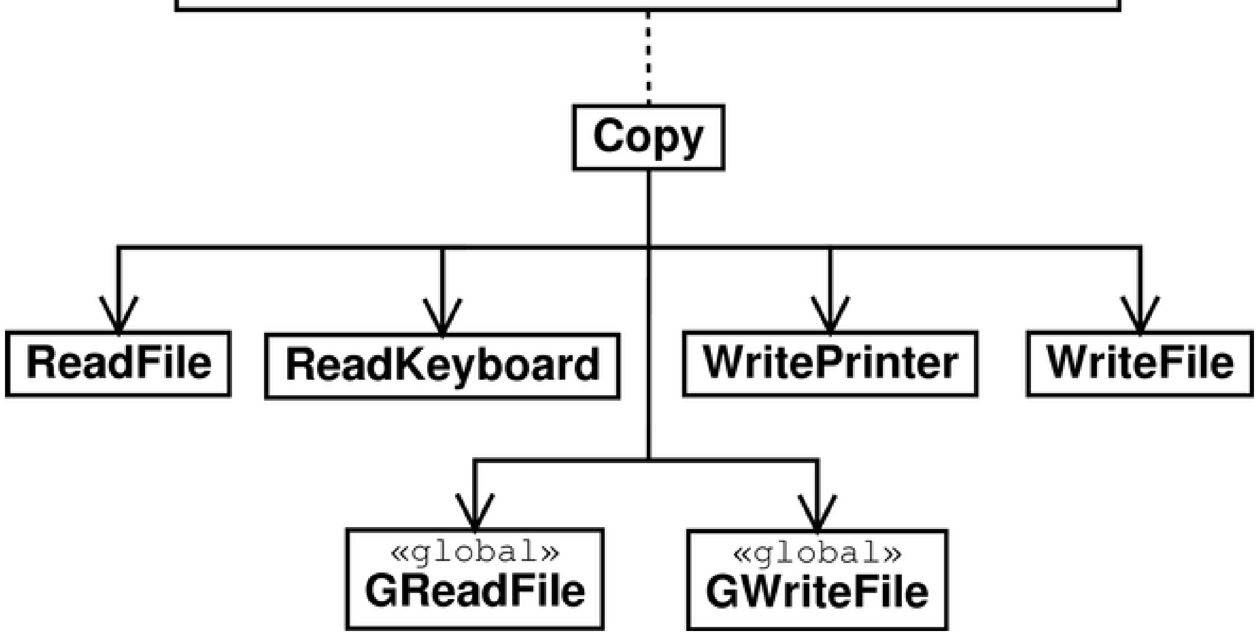
Ok, so we use a global flag

It is backwards compatible, to read files you have to set the flag first

4 Copy Version 3

```
bool GReadFile;
bool GWriteFile;

void Copy(void) {
  char ch;
  while( 1 ) {
    if( GReadFile ) { ch= ReadFile(); }
    else { ch= ReadKeyboard(); }
    if( ch == EOF ) break;
    if( GWriteFile ) { WriteFile( ch ); }
    else { WritePrinter( ch ); }
  }
}
```



Users want to write to files, of course they want it backwards compatible

We know how to do that!

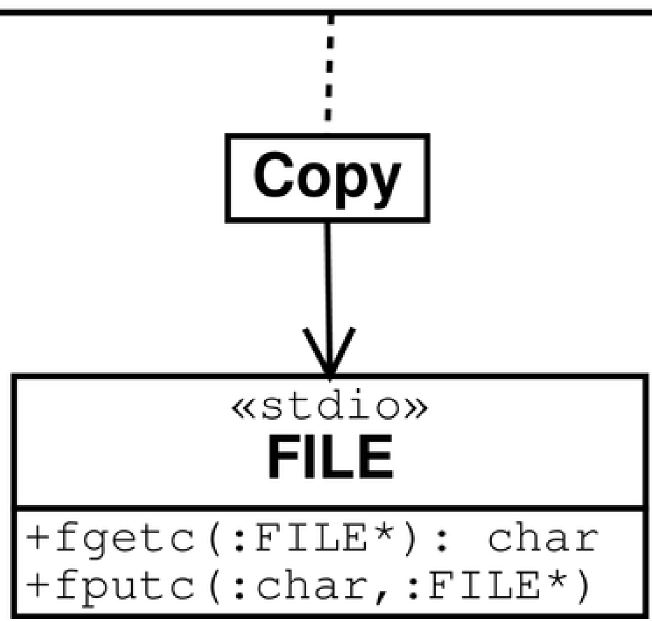
The Copy routine seems to grow in size and complexity every time a feature is added

The protocol to use it becomes more complicated

4 Copy done properly in C

```
#include <stdio.h>

void Copy( FILE* in, FILE* out ) {
  char ch;
  while( (ch= fgetc( in )) != EOF ) {
    fputc( ch, out );
  }
}
```



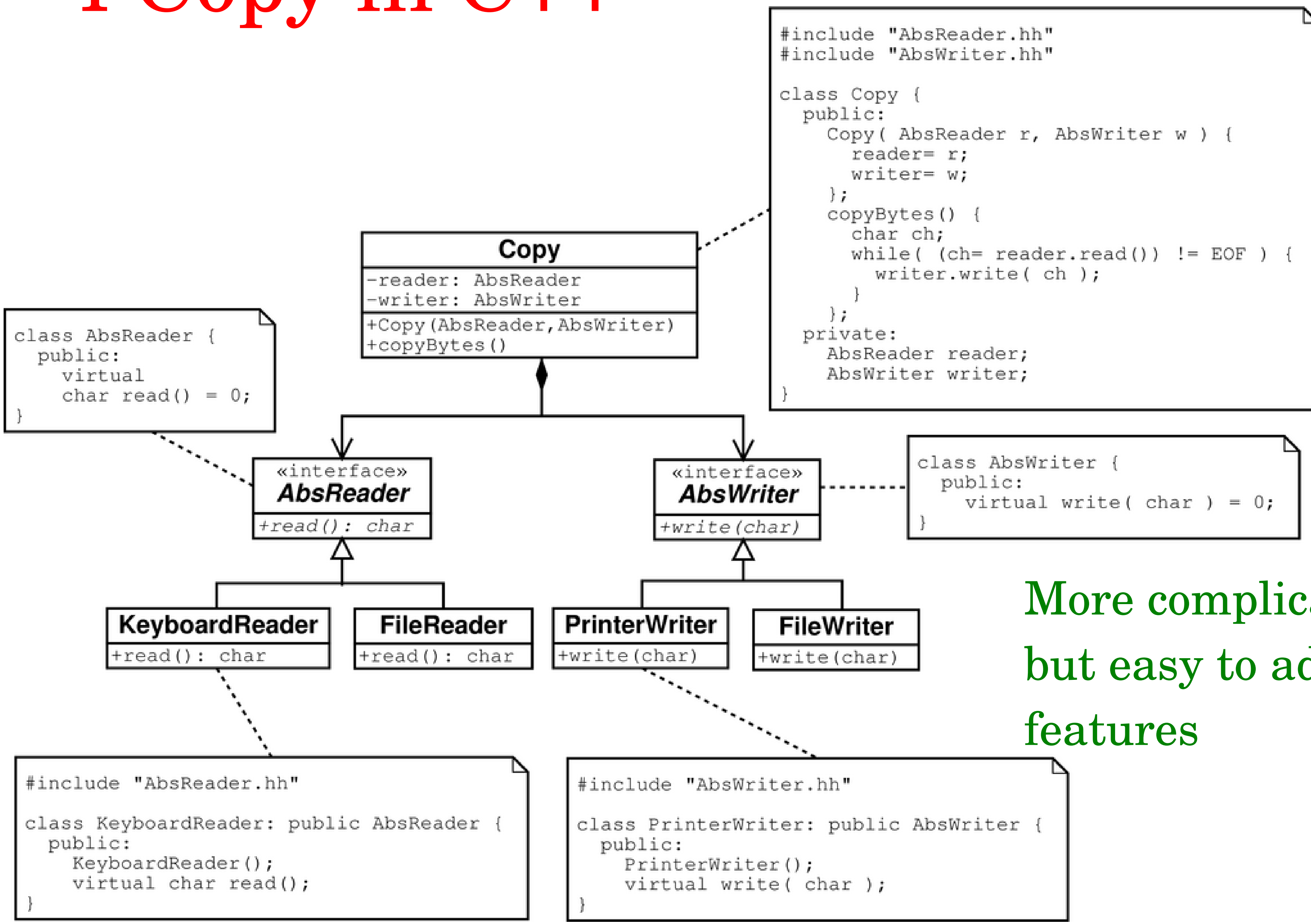
Finally a good C programmer comes to the rescue!

But this is C?!

FILE, fgetc and fputc behave like an interface class

FILE is a generic byte stream manipulated by fgetc, fputc etc.

4 Copy in C++



More complicated but easy to add new features

5 Class Design Principles

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
  - a.k.a. Design by Contract
- Dependency Inversion Principle (DIP)
- Interface Segregation Principle (ISP)

5 Single Responsibility Principle (SRP)

A class should have only one reason to change

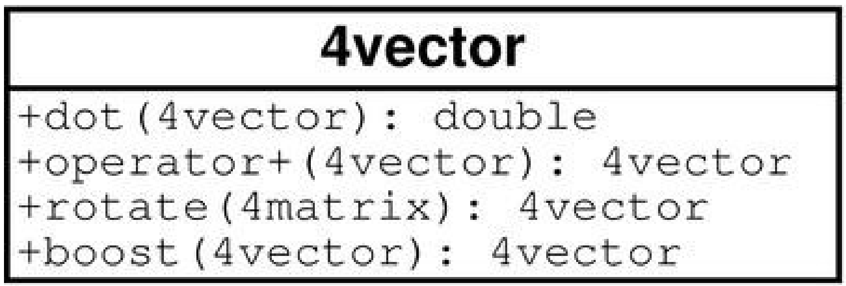
Robert Martin

Related to and derived from *cohesion*, i.e. that elements in a module should be closely related in their function

Responsibility of a class to perform a certain function is also a reason for the class to change

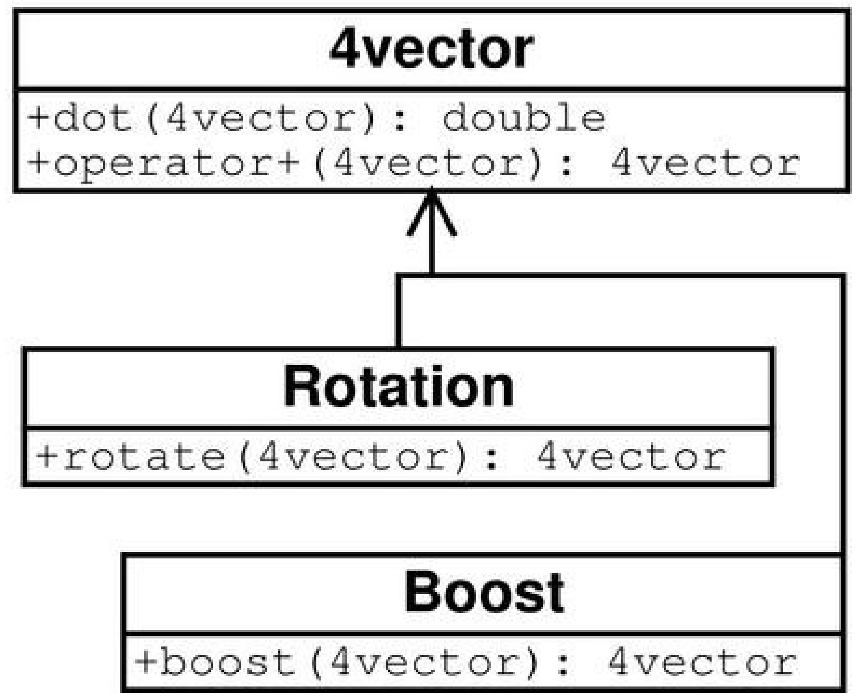
5 SRP Example

All-in-one wonder



Always changes to 4vector

Separated responsibilities



Changes to rotations or boosts don't impact on 4vector

5 Open/Closed Principle (OCP)

Modules should be open for extension,
but closed for modification

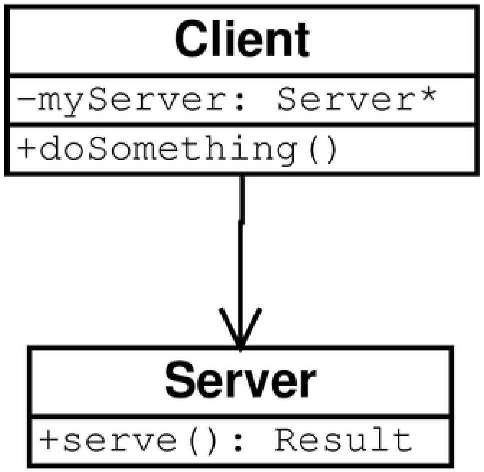
Bertrand Meyer
Object Oriented Software Construction

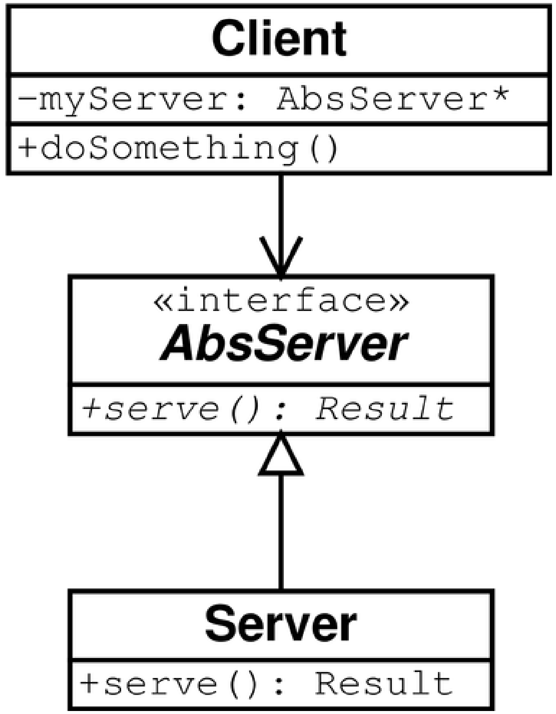
Module: Class, Package, Function

New functionality → new code, existing code remains unchanged

"Abstraction is the key" → cast algorithms in abstract interfaces
develop concrete implementations
as needed

5 Abstraction and OCP





Client is closed to changes in implementation of Server

Client is open for extension through new Server implementations

Without AbsServer the Client is open to changes in Server

5 Liskov Substitution Principle (LSP)

All derived classes must be substituteable for their base class

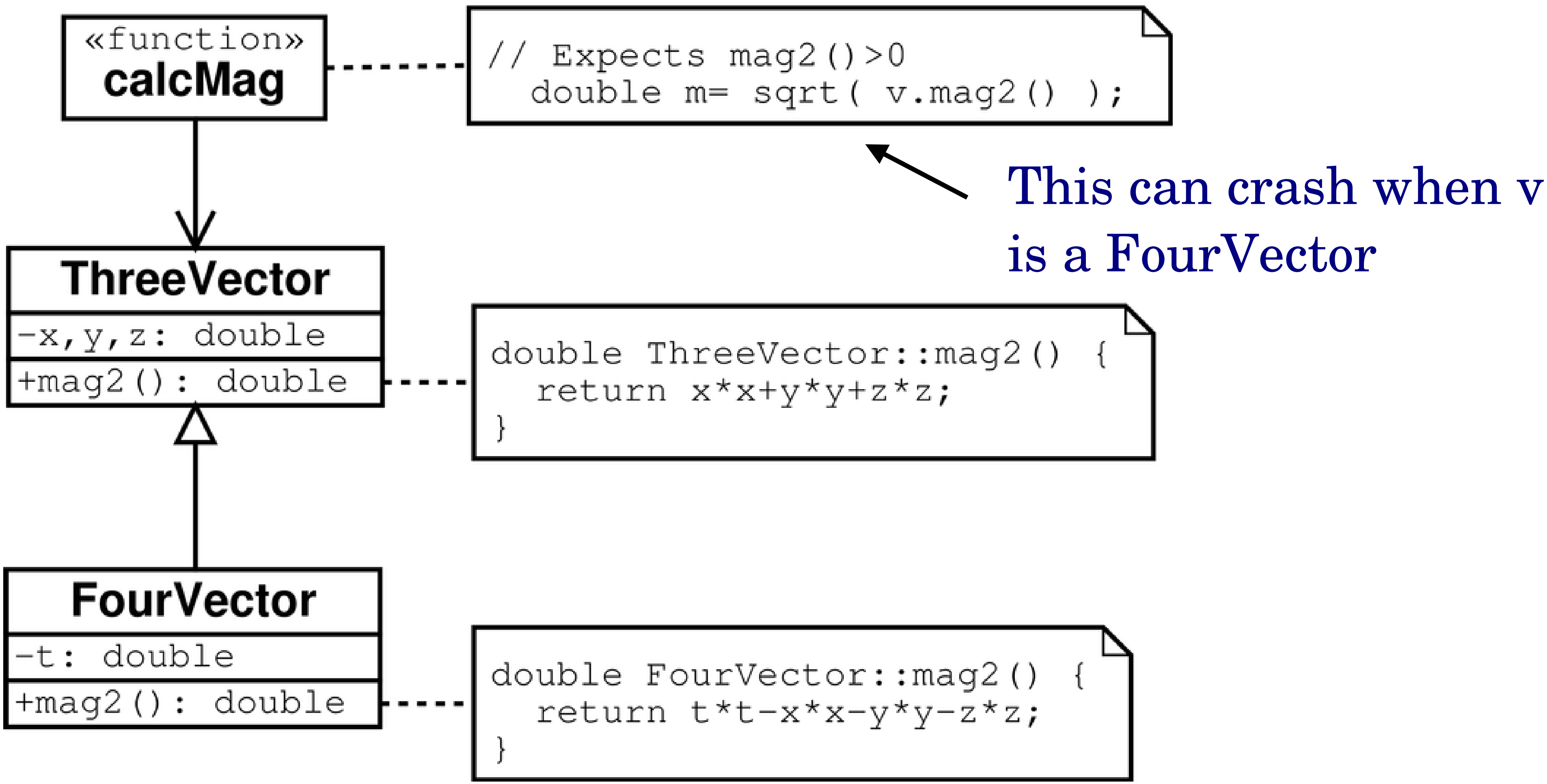
Barbara Liskov, 1988

The “Design-by-Contract” formulation:

All derived classes must honor the contracts of their base classes

Bertrand Meyer

5 LSP: FourVector Example



A 4-vector IS-A 3-vector with a time-component? Not in OO, 4-vector has different algebra → can't fulfill 3-vector contracts

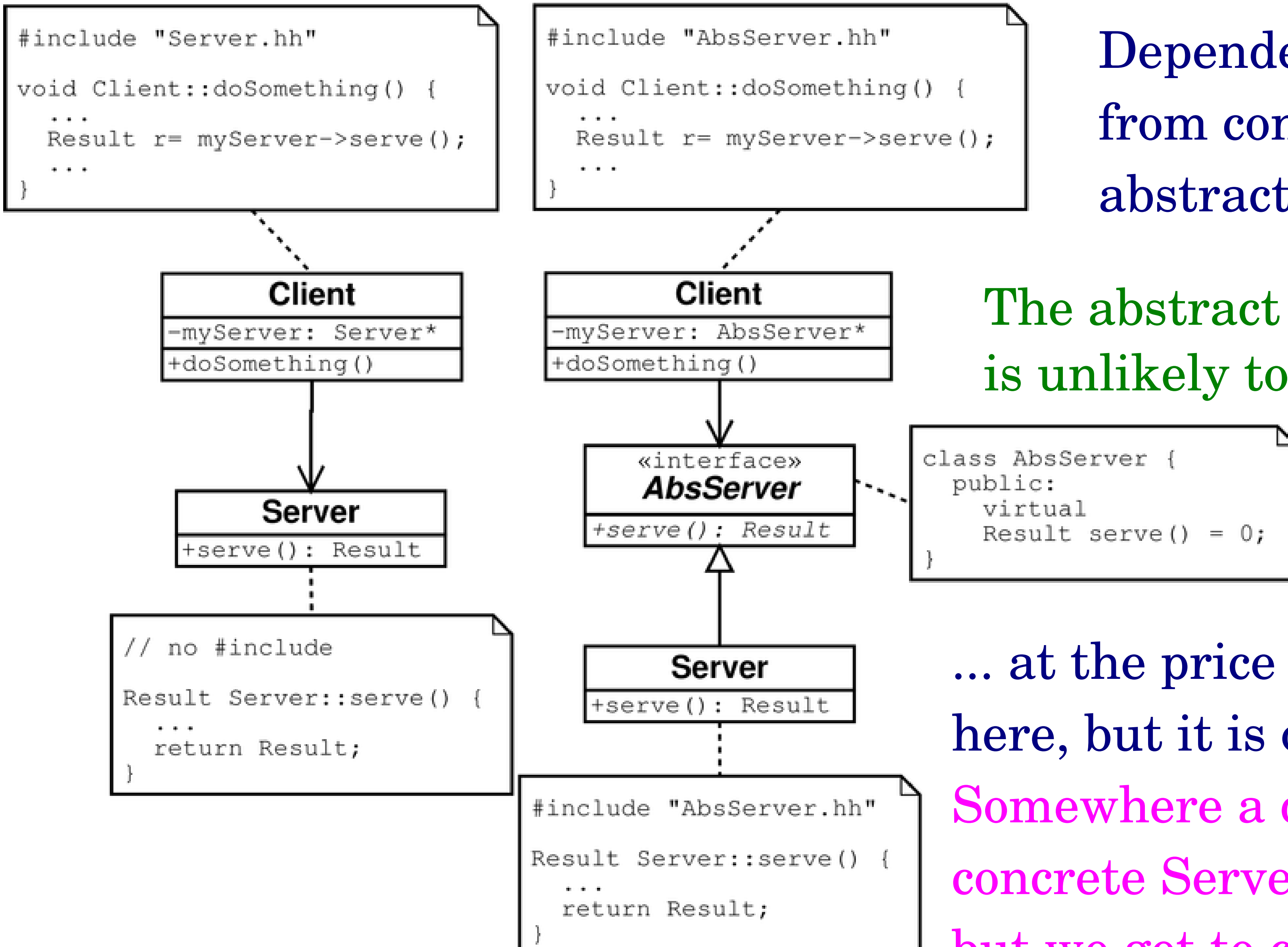
5 Dependency Inversion Principle (DIP)

Details should depend on abstractions.
Abstractions should not depend on details.

Robert Martin

Why *dependency inversion*? In OO we have ways to invert the direction of dependencies, i.e. class inheritance and object polymorphism

5 DIP Example



Dependency changed from concrete to abstract …

The abstract class is unlikely to change

… at the price of dependency here, but it is on abstraction.

Somewhere a dependency on concrete Server must exist, but we get to choose where.

5 Interface Segregation Principle (ISP)

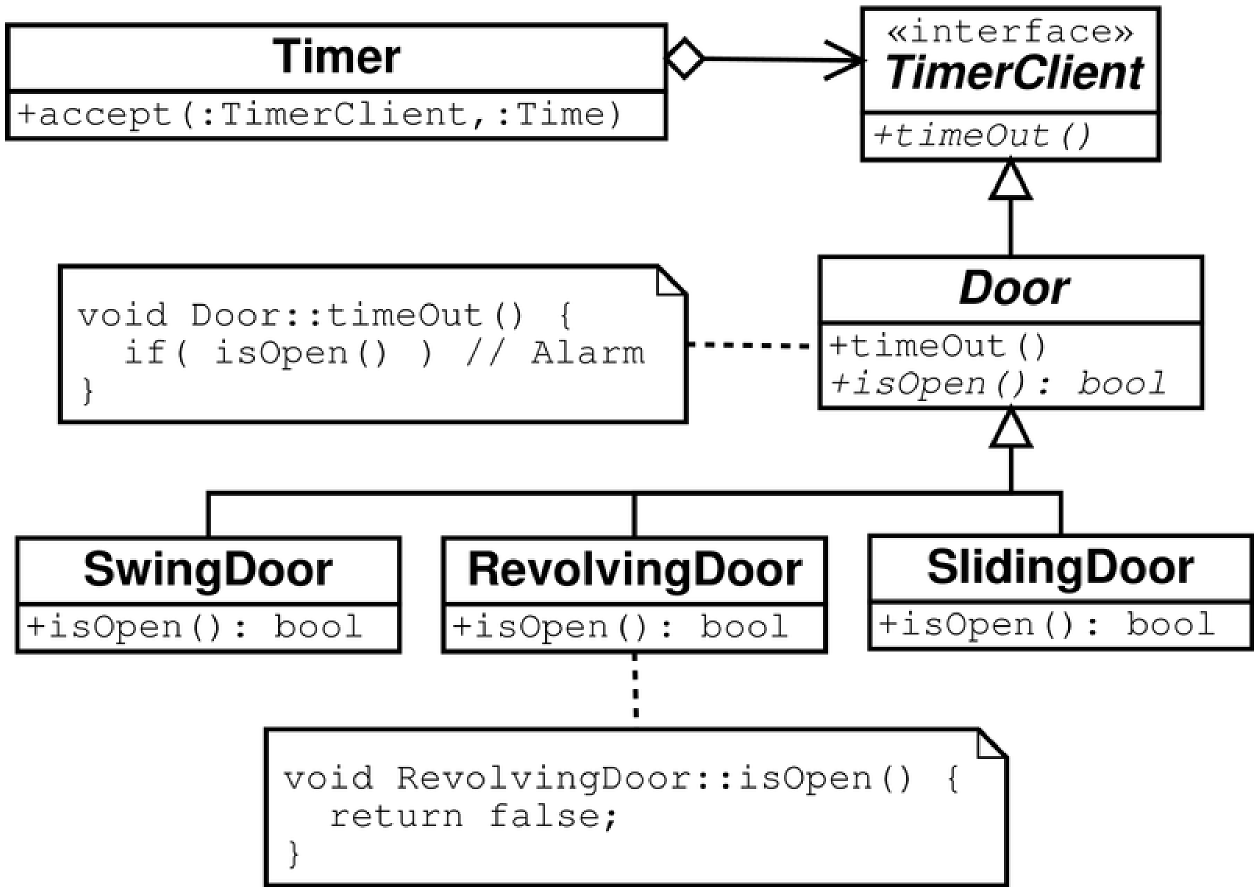
Many client specific interfaces are better than one general purpose interface

Clients should not be forced to depend upon interfaces they don't use

1) High level modules should not depend on low level modules. Both should depend upon abstractions (interfaces)
2) Abstractions should not depend upon details. Details should depend abstractions.

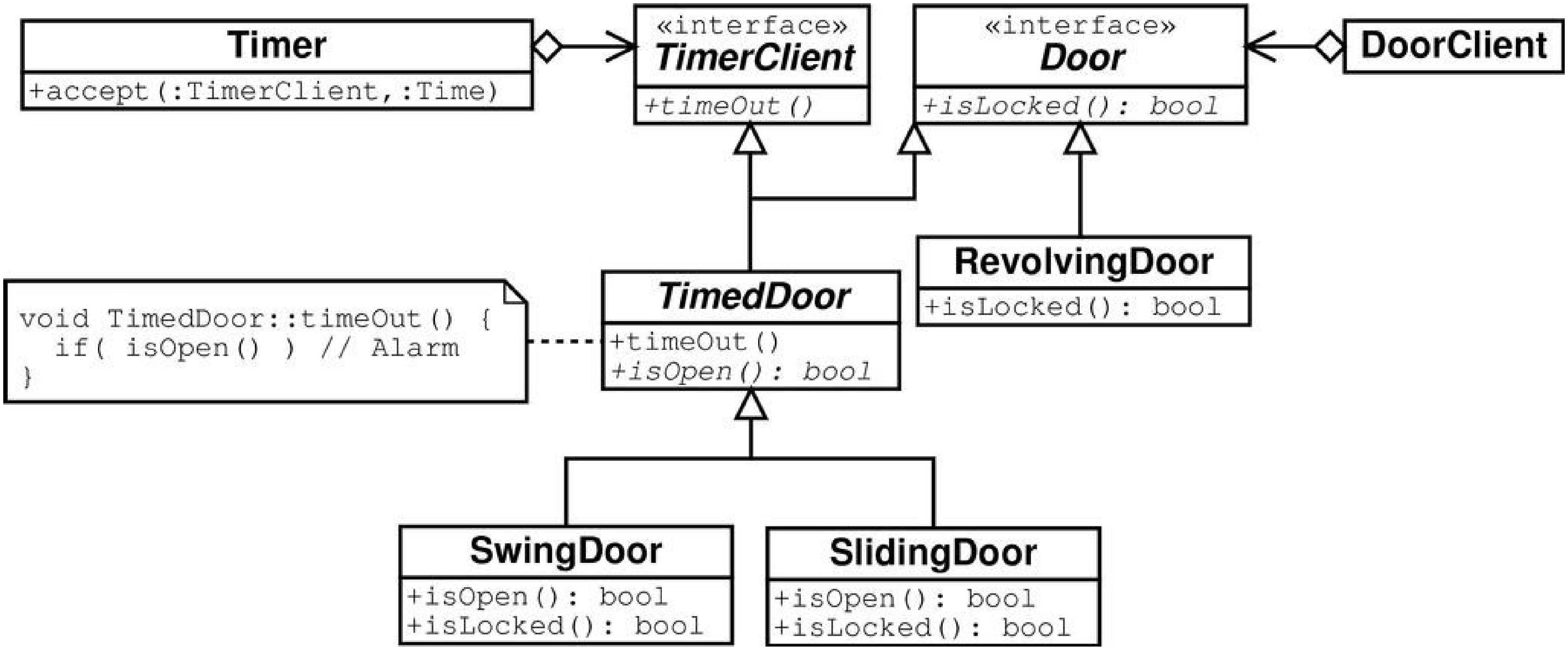
Robert Martin

5 ISP Example: Timed Door



There may be derived classes of Door which don't need the TimerClient interface. They suffer from depending on it anyway.

5 Timed Door ISP



RevolvingDoor does not depend needlessly on TimerClient

SwingDoor and SlidingDoor really are timed doors

OOAD in Physics: Summary

- Software is complex:
  - learn from other successful complex systems
- Object model:
  - Abstraction, encapsulation, modularity, hierarchy
  - Objects: building blocks for complex systems
- Class design:
  - Manage dependencies
  - SRP, OCP, LSP, DIP, ISP

1 Common Prejudices

- OO was used earlier without OO languages
  - Doubtful. A good procedural program may deal with some of the OO issues but not with all
  - OO without language support is at least awkward and dangerous if not quite irresponsible
- It is just common sense and good practices
  - It is much more than that, it provides formal methods, techniques and tools to control analysis, design, development and maintainance

1 Just another paradigm?

- Object-orientation is closer to the way problems appear in life (physical and non-physical)
- These problems generelly don't come formulated in a procedural manner
- We think in terms of "objects" or concepts and relations between those concepts
- Modelling is simplified with OO because we have objects and relations

1 Why OOAD in Physics?

- Physics is about modelling the world:
  - Objects interact according to laws of nature: particles/fields, atoms, molecules and electrons, liquids, solid states
- OOAD: create models by defining objects and rules of interaction
  - This way of thinking about software is well adapted and quite natural to physicists
- OOAD is software engineering practice
  - manage large projects professionally

3 Private Object Data

- Object state a priori unknown
- The object knows and reacts accordingly
- Decisions (program flow control) encapsulated
- User code not dependent on algorithm internals, only object behaviour
- Object state can be queried (when the object allows it)

3 Class Inheritance

- Objects are described by classes, i.e. code
- Classes can build upon other classes:
  - reuse (include) an already existing class
  - add new methods and member data
  - replace (overload) inherited methods
  - interface of new class must be compatible
  - New class has own type and type(s) of parent(s)

3 Static Polymorphism (Templates)

```
Template <class T> class U {
  public:
  void execute() {
    T t;
    t.init();
    t.run();
    t.finsish();
  }
}
```

```
Class B {
  public:
  void init();
  void run();
  void finish();
}
```

Template class U contains generic algorithm
Class B implements

```
#include “B.hh”
#include “U.hh”
int main {
  U<B> ub;
  ub.execute();
}
```

No direct dependence between U and B, but interface must match for U<B> to compile

Can't change types at run-time
Using typed collections difficult

→ don't use static polymorphism unless proven need

4 Dependency Management Summary

- Lack of sensible design leads to code rot
  - Useless complexity, repetition, opacity
- Software systems are dynamic
  - New requirements, new hardware
- A good design makes the system flexible and allows easy extensions
  - Abstractions and interfaces
- An OO design may be more complex but it builds in the ability to make changes

4 The Shape Example - Procedural

Shape.h

```
enum ShapeType { isCircle, isSquare };
typedef struct Shape {
  enum ShapeType type
} shape;
```

Circle.h

```
typedef struct Circle {
  enum ShapeType type;
  double radius;
  Point center;
} circle;
void drawCircle( circle* );
```

Square.h

```
typedef struct Square {
  enum ShapeType type;
  double side;
  Point topleft;
} square;
void drawSquare( square* );
```

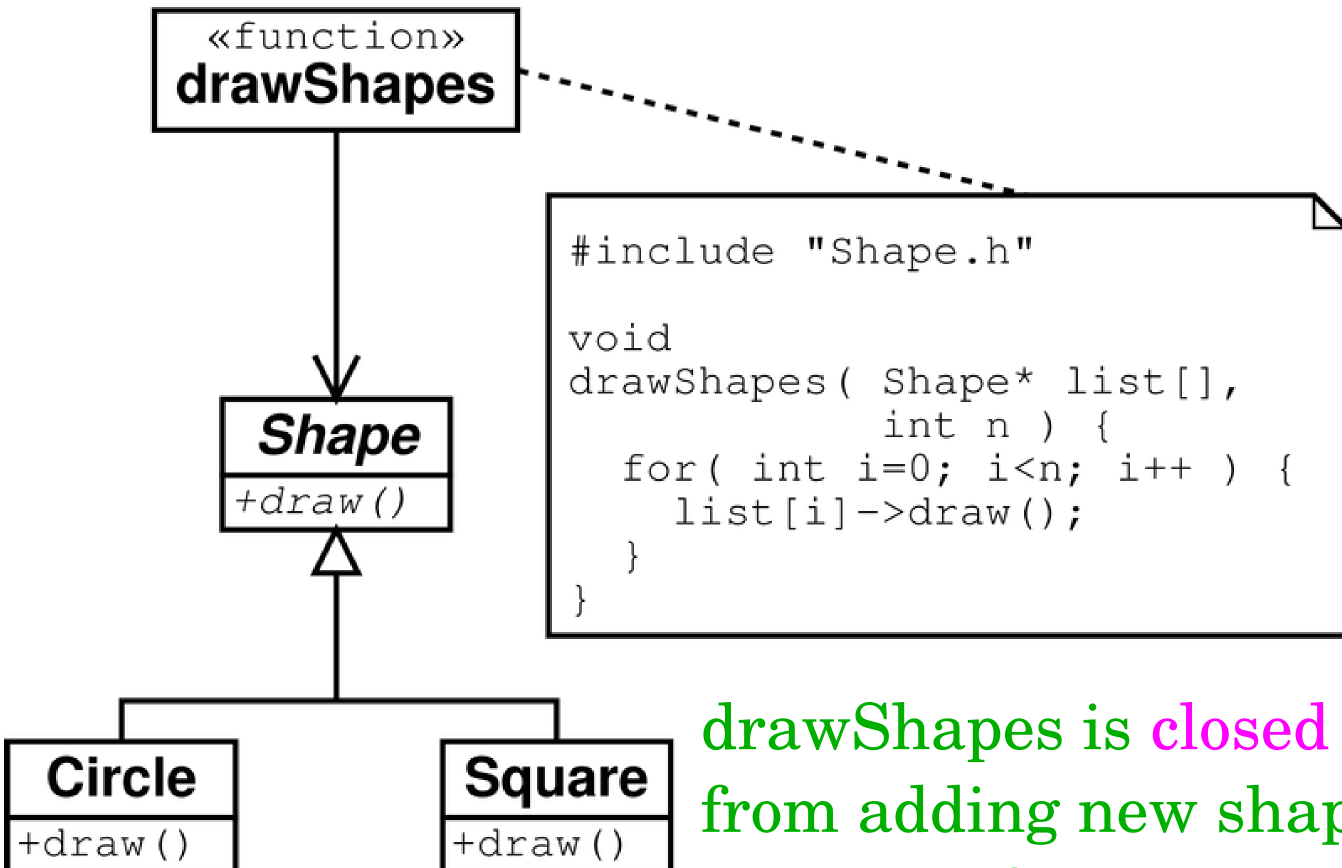
drawShapes.c

```
#include "Shape.h"
#include "Circle.h"
#include "Square.h"

void drawShapes( shape* list[], int n ) {
  int i;
  for( int i=0; i<n; i++ ) {
    shape* s= list[i];
    switch( s->type ) {
    case isSquare:
      drawSquare( (square*)s );
      break;
    case isCircle:
      drawCircle( (circle*)s );
      break;
    }
  }
}
```

RTTI a la C: Adding a new shape requires many changes

4 The Shape Example OO



drawShapes is closed against changes from adding new shapes

It is open for extension, e.g. adding new functions to manipulate shapes

Just add new shapes or functions and relink

3 Type

Typing enforces object class such that objects of different class may not be interchanged

Strong typing:	operation upon an object must be defined
Weak typing:	can perform operations on any object
Static typing:	names bound to types (classes) at compile time
Dynamic typing:	names bound to objects at run time
Static binding:	names bound to objects at compile time
Dynamic binding:	names bound to objects at run time

C++, Java:	strong+static typing + dynamic binding
Python:	strong+dynamic typing
Perl:	weak+dynamic typing
Fortran, C:	strong+static typing + static binding (except casts)

3 Classes = Types

- Class is new programmer-defined data type
- Objects have type
  - extension of bool, int, float, etc
  - e.g. type complex didn't exist in C/C++, but can construct in C++ data type complex using a class
- ThreeVector is a new data type
  - 3 floats/doubles with interface and behaviour
  - can define operators +, -, *, / etc.

3 Interface Abstraction

- Common interface of group of objects is an abstraction (abstract class, interface class)
  - find commonality between related objects
  - express commonality formally using interfaces
- Clients (other objects) depend on abstract interface, not details of individual objects
  - Polymorphic objects can be substituted
- Need abstract arguments and return values
  - or clients depend on details again

4 Contract Violation

- The contract of ThreeVector:
  - Magnitude guaranteed to be non-negative
- FourVector breaks this contract
- Derived methods should not expect more and provide no less than the base class methods
  - Preconditions are not stronger
  - Postconditions are not weaker

5 Class Design Principles

- Single Responsibility Principle (SRP)
  - Only one reason to change
- Open-Closed Principle (OCP)
  - Extend functionality with new code
- Liskov Substitution Principle (LSP)
  - Derived classes substitute their base classes
- Dependency Inversion Principle (DIP)
  - Depend on abstractions, not details
- Interface Segregation Principle (ISP)
  - Split interfaces to control dependencies