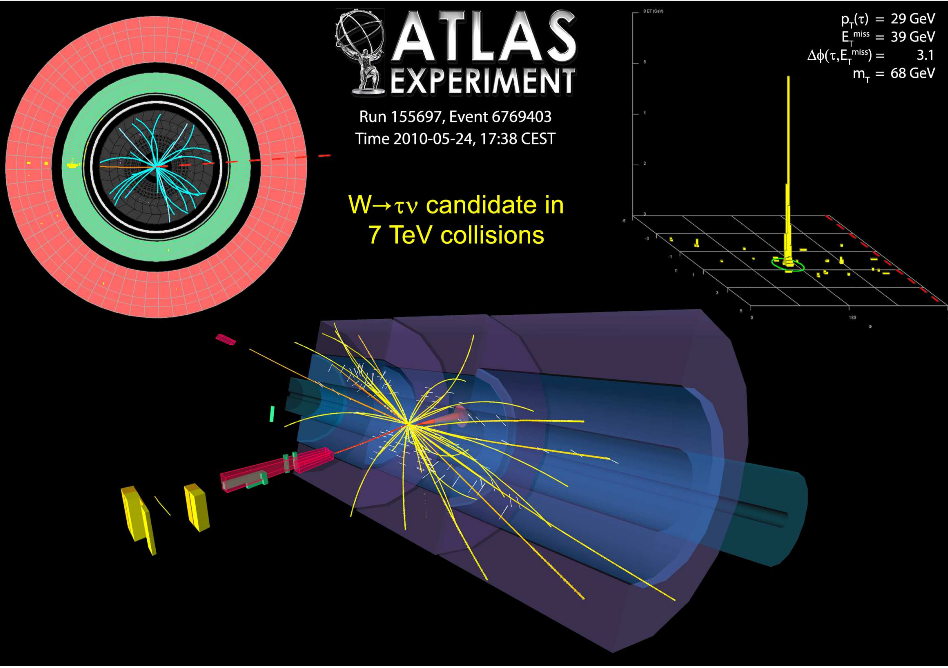
**‘Fortran? C++? Egal!
Ein guter Programmierer kann Spaghetti-Code
in jeder Sprache schreiben!’**

Wohl nicht ganz ernstgemeinte Bemerkung eines unbekannten Software-Gurus
(Gehört irgendwann Ende der 90er Jahre)

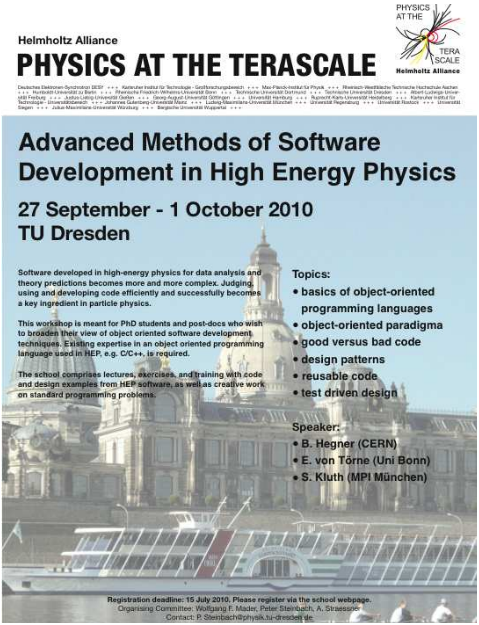


ATLAS EXPERIMENT

Run 155697, Event 6769403
Time 2010-05-24, 17:38 CEST

W→τν candidate in 7 TeV collisions

$p_T(\tau) = 29$ GeV
$E_T^{miss} = 39$ GeV
$\Delta\phi(\tau, E_T^{miss}) = 3.1$
$m_T = 68$ GeV



Helmholtz Alliance

PHYSICS AT THE TERASCALE

PHYSICS
AT THE
TERA
SCALE
Helmholtz Alliance

Deutsches Elektronen-Synchrotron DESY +++ Karlsruher Institut für Technologie - Großforschungsbereich +++ Max-Planck-Institut für Physik +++ Rheinisch-Westfälische Technische Hochschule Aachen +++ Humboldt-Universität zu Berlin +++ Rheinische Friedrich-Wilhelms-Universität Bonn +++ Technische Universität Dortmund +++ Technische Universität Dresden +++ Albert-Ludwigs-Universität Freiburg +++ Justus-Liebig-Universität Gießen +++ Georg-August-Universität Göttingen +++ Universität Hamburg +++ Ruprecht-Karls-Universität Heidelberg +++ Karlsruher Institut für Technologie - Universitätsbereich +++ Johannes Gutenberg-Universität Mainz +++ Ludwig-Maximilians-Universität München +++ Universität Regensburg +++ Universität Rostock +++ Universität Siegen +++ Julius-Maximilians-Universität Würzburg +++ Bergische Universität Wuppertal +++

Advanced Methods of Software Development in High Energy Physics

27 September - 1 October 2010
TU Dresden

Software developed in high-energy physics for data analysis and theory predictions becomes more and more complex. Judging, using and developing code efficiently and successfully becomes a key ingredient in particle physics.

This workshop is meant for PhD students and post-docs who wish to broaden their view of object oriented software development techniques. Existing expertise in an object oriented programming language used in HEP, e.g. C/C++, is required.

The school comprises lectures, exercises, and training with code and design examples from HEP software, as well as creative work on standard programming problems.

Topics:

- basics of object-oriented programming languages
- object-oriented paradigma
- good versus bad code
- design patterns
- reusable code
- test driven design

Speaker:

- B. Hegner (CERN)
- E. von Törne (Uni Bonn)
- S. Kluth (MPI München)

Registration deadline: 15 July 2010. Please register via the school webpage.
Organising Committee: Wolfgang F. Mader, Peter Steinbach, A. Straessner
Contact: P. Steinbach@physik.tu-dresden.de

https://indico.desy.de/conferenceDisplay.py?confId=3155

Outline of this Course

1) UML: The Feynman Diagrams of Software Design
2) Class Design Principles: Efficient Methods of Developing Re-Usable Code in High Energy Physics
3) Design Patterns (Selected Examples and Use-Cases in High Energy Physics)
4) Hands-on: Exercise on Software Design

> **I liked it a lot! First I was thinking that software development with pen and paper is absolutely boring. But in the end it turned out to be very inspiring.**

A Workshop Participant
'Advanced Methods of Software Development in High-Energy Physics'
Dresden – 9/2010

Outline of this Course

1) UML: The Feynman Diagrams of Software Design
2) Class Design Principles: Efficient Methods of Developing Re-Usable Code in High Energy Physics
3) Design Patterns (Selected Examples and Use-Cases in High Energy Physics)
4) Hands-on: Exercise on Software Design

> **I liked it a lot! First I was thinking that software development with pen and paper is absolutely boring. But in the end it turned out to be very inspiring.**

A Workshop Participant
'Advanced Methods of Software Development in High-Energy Physics'
Dresden – 9/2010

UML – The Feynman-Diagrams of Software Design

Dr. Wolfgang F. Mader[1], Peter Steinbach[1]

[1] Institut für Kern- und Teilchenphysik, TU Dresden

Blockkurs Graduiertenkolleg 'Masse, Spektren, Symmetrie', Rathen 2010
10.March 2011

Outline of this Lecture

1) What is UML?
2) Classes in UML
3) Relations between Classes
4) Example of Class Design

UML is a Language to Talk about the Design of your Software Package (like FEYNMAN Diagrams in High-Energy Physics)

We Hope to Convince you that Thinking about the Design of your Software before Starting to Write Code Makes Perfect Sense...

Outline of this Lecture

1) What is UML?
2) Classes in UML
3) Relations between Classes
4) Example of Class Design

UML is a Language to Talk about the Design of your Software Package (like FEYNMAN Diagrams in High-Energy Physics)

We Hope to Convince you that Thinking about the Design of your Software before Starting to Write Code Makes Perfect Sense...

Outline of this Lecture

1) What is UML?
2) Classes in UML
3) Relations between Classes
4) Example of Class Design

UML is a Language to Talk about the Design of your Software Package (like Feynman Diagrams in High-Energy Physics)

We Hope to Convince you that Thinking about the Design of your Software before Starting to Write Code Makes Perfect Sense...

What is UML? A Definition.

‘The Unified Modeling 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 re-usable software components’

Grady Booch, Ivar Jacobsen, Jim Rumbaugh
Rational Software Coorporation
The Unified Modelling Language User Guide, Addison-Wesley, 2003

A Bit of History

1980

- First Object Oriented (OO) Modeling Languages
- Other Techniques, e.g. SA/SD

1990

- ‘OO Method Wars’
- Many Modeling Languages

End of 1990s

- UML as Combination of Best Practices

Strukturierte Analyse (SA)

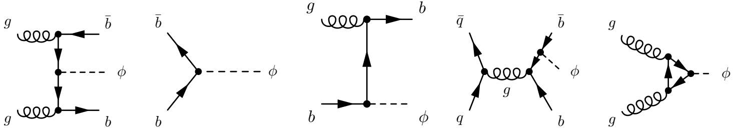
Das Ergebnis ... ist ein hierarchisch gegliedertes Anforderungsdokument für Umfang und Inhalt der betrieblichen Anwendung, die in dem geplanten Softwaresystem realisiert werden soll. Die Strukturierte Analyse ist eine graphische Analysemethode, die mit Hilfe eines Top-Down-Vorgehens ein komplexes System in immer einfachere Funktionen bzw. Prozesse aufteilt und gleichzeitig eine Datenflussmodellierung durchführt. In ihrer Grundform ist die SA eine statische Analyse ...

Strukturiertes Design (SD)

...ist ein Entwurfsmuster in der Softwaretechnik ... welches modulares Design unterstützt, um neben der reinen Funktionshierarchie auch die Wechselwirkungen von übergeordneten Modulen zu beschreiben. SD wird mit der Strukturierten Analyse (SA) in der Softwaretechnik verwendet. Das Strukturierte Design schlägt eine Brücke zwischen der technologieneutralen Analyse und der eigentlichen Implementierung. Im Strukturierten Design werden technische Randbedingungen eingebracht und die Grobstruktur des Systems aus technischer Sicht festgelegt. Es stellt damit die inhaltliche Planung der Implementierung dar.

Why Using UML...?

- **Physicists Know Formal Graphical Modeling**
  - Mathematics to Describe Nature
  - Feynman Diagrams to Calculate e.g. Cross Sections of Physics Processes



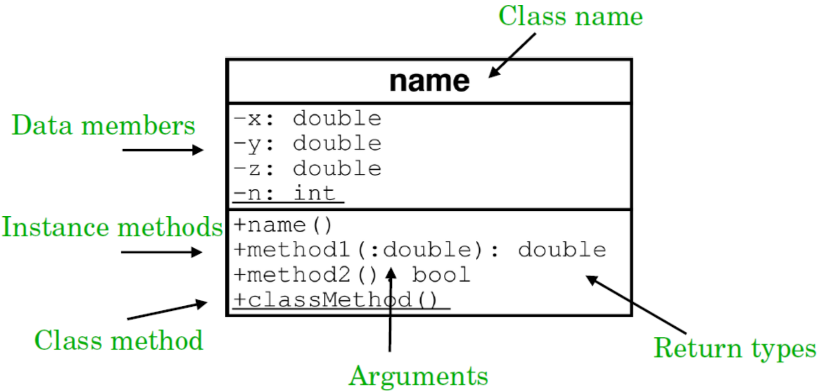
- **A Common Language is Needed to Talk about Software Design**
  - Discuss Software Design on Blackboard
  - Documentation of Software Packages
  - UML is Important Part of that Language
  - UML Provides the 'Words and Grammar'

Classes in UML

- **Classes Describe Objects**
  - Interface to the Class (Member Function Signature)
  - Behavior (Member Function Implementations)
  - State of Book-Keeping (Values of Data Members)
  - Creation and Destruction of Classes

- **Collaboration between Classes**
  - Class Relations (Object Relations)
  - Dependencies between Classes

Classes in UML



Class name

Data members

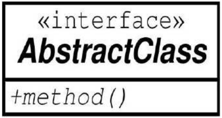
Instance methods

Class method

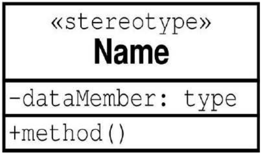
Arguments

Return types

Classes in UML

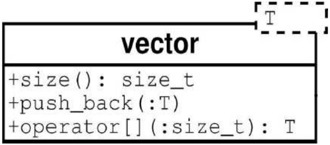


«interface»
AbstractClass
+method()



«stereotype»
Name
-dataMember: type
+method()

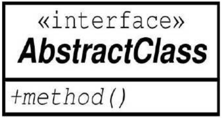
- Top Compartment Contains Name of Class
- Abstract Classes have Name in Italics
- Abstract Methods have Name in Italics
- **Or:** 'Stereotypes' to Identify Groups of Classes (e.g. Interfaces)

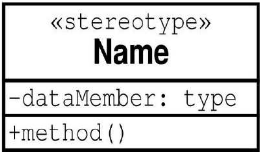


T
vector
+size(): size_t
+push_back(:T)
+operator[](:size_t): T

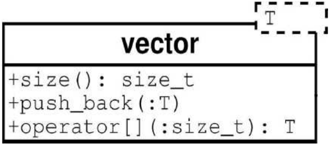
- Parameter Type (T) in Top-Compartment
- Operations Compartment as Usual, but May Have Type Parameter instead of Concrete Type

Classes in UML





- Top Compartment Contains Name of Class
- Abstract Classes have Name in Italics
- Abstract Methods have Name in Italics
- **Or:** 'Stereotypes' to Identify Groups of Classes (e.g. Interfaces)



- Parameter Type (T) in Top-Compartment
- Operations Compartment as Usual, but May Have Type Parameter instead of Concrete Type

Visibility of Class Methods and Members

- +: **Public**
  - Accessible by other Classes
  - Interface Operations
  - Not Data Members
- -: **Private**
  - Only Accessible by Class Itself
  - Data Members
  - Helper Functions
  - 'Friends' are Allowed to Access
- #: **Protected**
  - Subclasses Can Access Method/Data Member
  - Operations where Sub-Classes Collaborate
  - Not Data Members
    (Dependency of Subclasses on Implementation of Parent Class)

Visibility of Class Methods and Members

- +: **Public**
  - Accessible by other Classes
  - Interface Operations
  - Not Data Members

- -: **Private**
  - Only Accessible by Class Itself
  - Data Members
  - Helper Functions
  - ‘Friends’ are Allowed to Access

- #: **Protected**
  - Subclasses Can Access Method/Data Member
  - Operations where Sub-Classes Collaborate
  - Not Data Members
    (Dependency of Subclasses on Implementation of Parent Class)

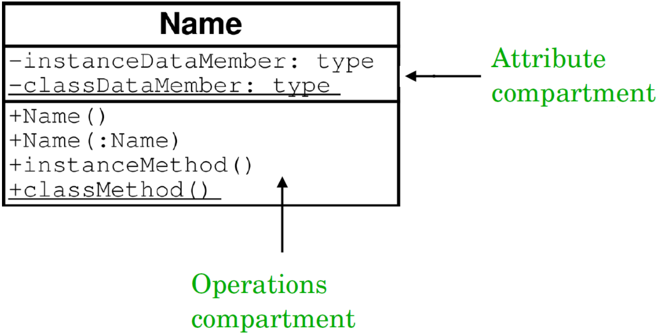
Visibility of Class Methods and Members

- **+: Public**
  - Accessible by other Classes
  - Interface Operations
  - Not Data Members

- **-: Private**
  - Only Accessible by Class Itself
  - Data Members
  - Helper Functions
  - 'Friends' are Allowed to Access

- **#: Protected**
  - Subclasses Can Access Method/Data Member
  - Operations where Sub-Classes Collaborate
  - Not Data Members
    (Dependency of Subclasses on Implementation of Parent Class)

Class Attributes and Operations



Name

-instanceDataMember: type
-classDataMember: type

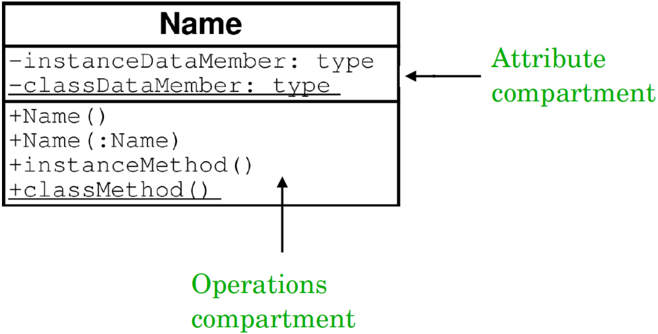
+Name()
+Name(:Name)
+instanceMethod()
+classMethod()

Attribute compartment

Operations compartment

- **Class Attributes**
  - Attributes are Instance and Class Data Members
  - Underlined Class Data Members are Shared between all Instances of Given Class
  - Data Type is Shown after ':'
- **Class Operations**
  - Operations are Class Methods with Arguments and Return-Types
  - Public (+) Operations Define Class Interface
  - Underlined Methods Have only Access to Class Data Members (No Need for Class Instance)

Class Attributes and Operations



Name

-instanceDataMember: type
-classDataMember: type

+Name()
+Name(:Name)
+instanceMethod()
+classMethod()

Attribute compartment

Operations compartment

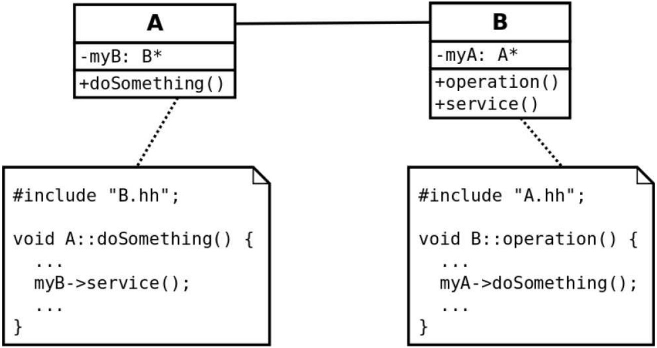
- **Class Attributes**
  - Attributes are Instance and Class Data Members
  - Underlined Class Data Members are Shared between all Instances of Given Class
  - Data Type is Shown after ':'

- **Class Operations**
  - Operations are Class Methods with Arguments and Return-Types
  - Public (+) Operations Define Class Interface
  - Underlined Methods Have only Access to Class Data Members (No Need for Class Instance)

Relations between Classes in UML

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

Binary Association between Classes

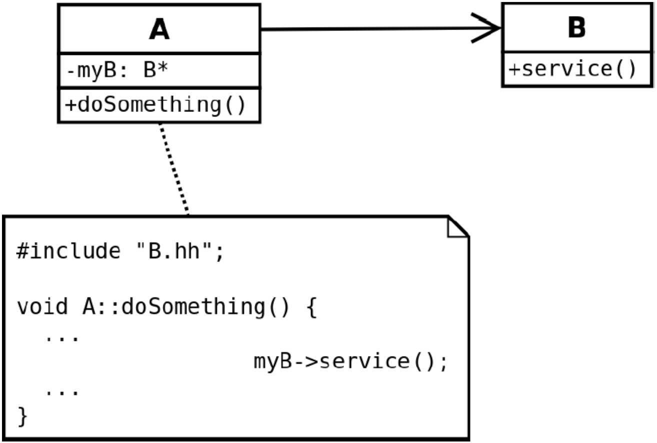


```
#include "B.hh";

void A::doSomething() {
  ...
  myB->service();
  ...
}
```

```
#include "A.hh";

void B::operation() {
  ...
  myA->doSomething();
  ...
}
```

- **A** depends on Implementation of **B**
- If **A** is Changed (Data Members or Access Method) **B** Needs to Adapt
- Implies Dependency Cyle

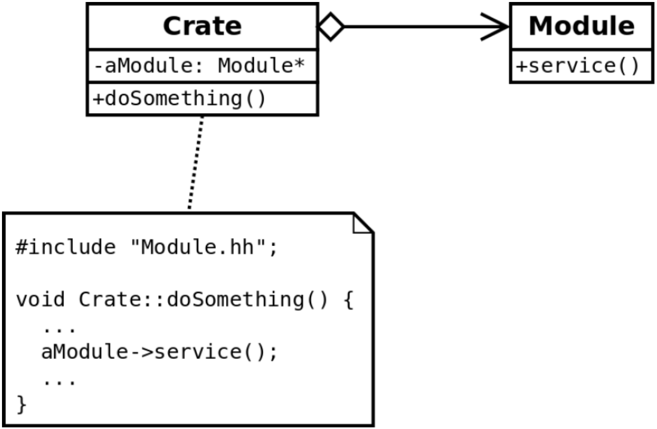
Unary Association between Classes



```
#include "B.hh";

void A::doSomething() {
   ...
                   myB->service();
   ...
}
```

- **A** Knows about **B**, but ...
- ... **B** Knows Nothing about **A**
- Arrow Shows Direction of Association in Direction of Dependency

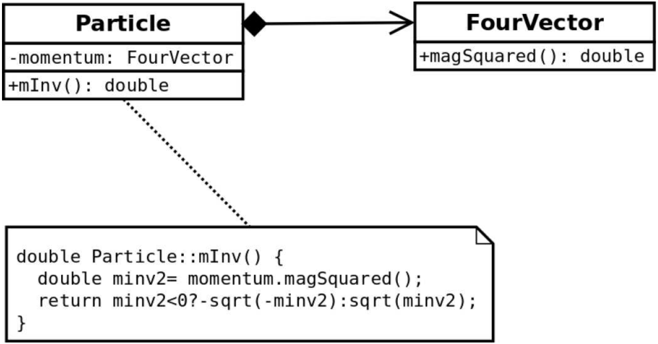
Aggregation



```
#include "Module.hh";

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

- Aggregation: Association with ‘whole-part’ Relationship
- Symbolized by hollow Diamond
- ‘Create’ Does not Control the Lifetime of ‘Module’

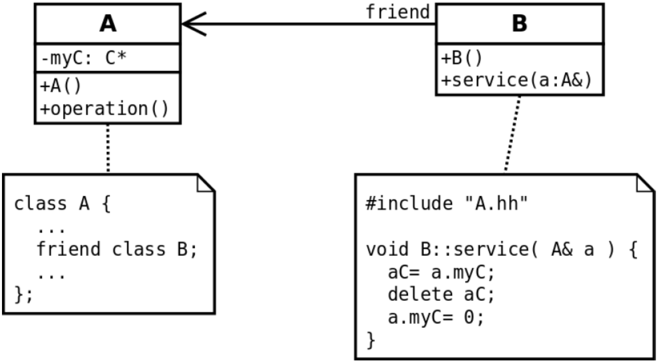
Composition



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

- Composition: Aggregation with Lifetime Control
- Symbolized by Filled Diamond
- ‘Particle’ Responsible for **Creation** and **Destruction** of ‘FourVector’ (Might be Delegated)

Friendship between Classes

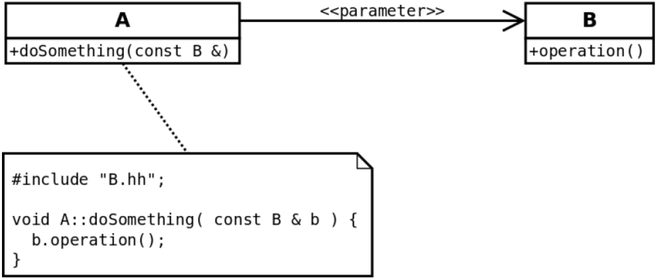


```
class A {
  ...
  friend class B;
  ...
};
```

```
#include "A.hh"

void B::service( A& a ) {
  aC= a.myC;
  delete aC;
  a.myC= 0;
}
```

- 'Friends' Have Access to Private Data Members and Functions
- Friendship Breaks Data Hiding Policy (Use with Care)

Parametric Association



A

+doSomething(const B &)

<<parameter>>

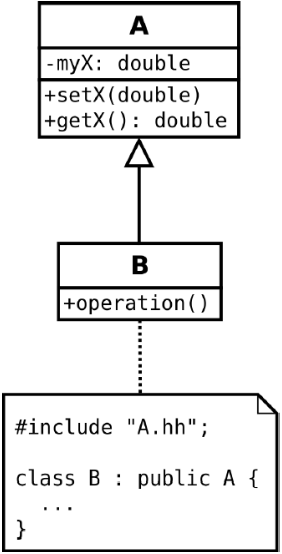
B

+operation()

```
#include "B.hh";

void A::doSomething( const B & b ) {
  b.operation();
}
```

- **A** Depends on **B** (it Uses **B**)
- No Data Member of Type **B** in **A**

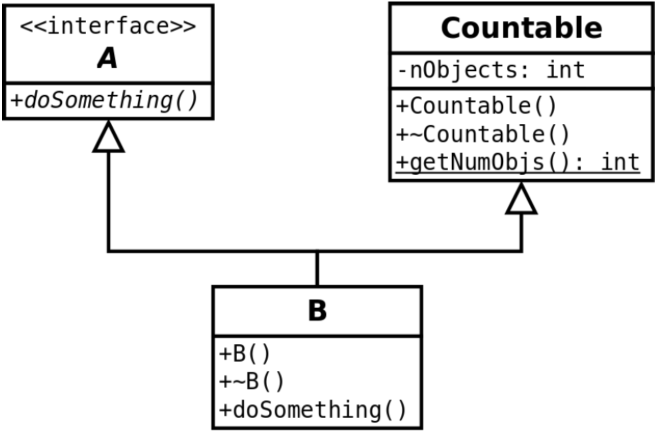
Inheritance



```
#include "A.hh";

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

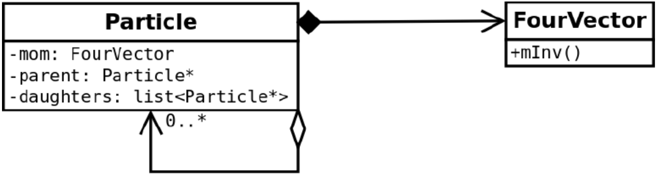
- **A** is Called ‘Base Class’ or ‘Super Class’
- Arrow Shows Direction of Dependency
- **B** Inhertis **A**’s Methods and Data Members
- **B** Can Extend **A**
- **B** Depends on **A**, but…
- … **A** Know Nothing about **B**

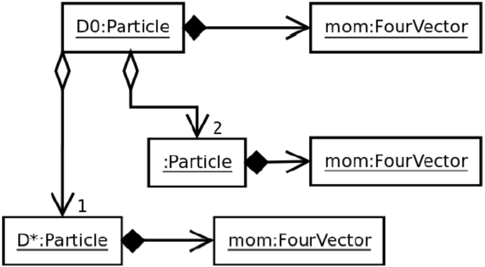
Multiple Inheritance



- Derived Class Inherits Interfaces, Data Members and Behavior of all its Base Classes
- Extension and Overriding Works as Well
- **B** Implements the Interfaces of **A** and is also a Countable Class

Class Diagrams vs. Object Diagrams





- Class Diagrams (Top) Never Change
- Used to Show Specific Relations between (a Part of the) Classes of a Software Package at Given Instant in Time
- Object Relations are Drawn Using Class Association Lines

Some Comments Close to the End...

- **Design-Heavy Development Process**
  - Substantial Amount of Person-Power Spent on Design of Software Package Using UML
  - Start Coding ONLY when Design is Consistent
  - Recommended Way for Really Large Software Packages
- **Light-Weight Development Process**
  - Limited (but not Negligible) Amount of Person-Power Spent on Design
  - UML Used as a Tool to Discuss Program Structure AND to Document the Implementation
  - Probably More Adequate in Day-to-Day Work of High-Energy Physicists

… and now a Real-Life Example: The Copy Routine

Code Rots!!!!

- The Are Many Reasons for Code to Rot…
- Case-Study Based on an Example by Bob Martin
- A Routine which Reads the Keyboard and Writes to a Printer

… and now a Real-Life Example: The Copy Routine

Code Rots!!!!

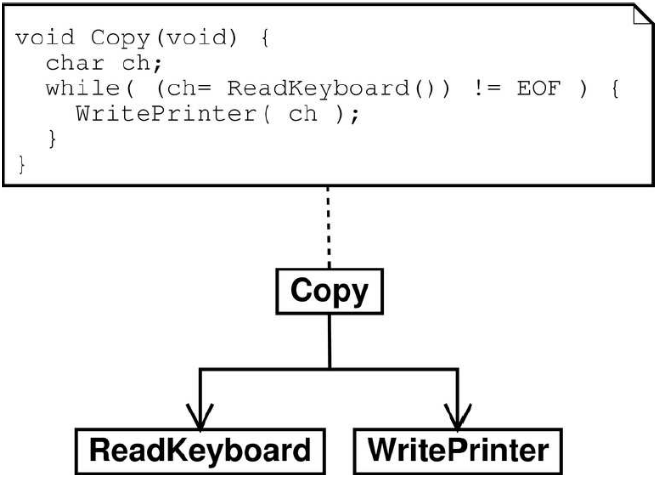
- The Are Many Reasons for Code to Rot…
- Case-Study Based on an Example by Bob Martin
- A Routine which Reads the Keyboard and Writes to a Printer

… and now a Real-Life Example: The Copy Routine

Code Rots!!!!

- The Are Many Reasons for Code to Rot…
- Case-Study Based on an Example by Bob Martin
- A Routine which Reads the Keyboard and Writes to a Printer

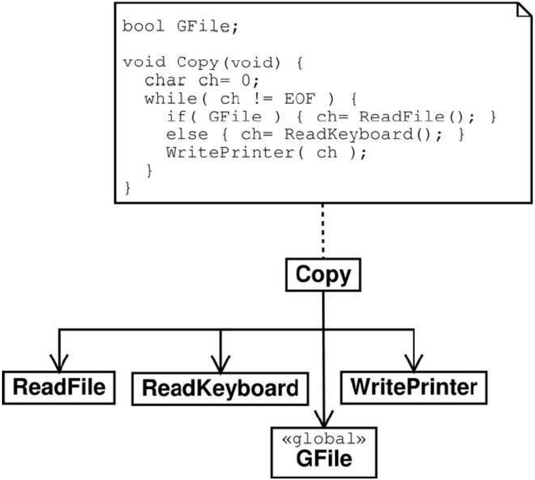
The Copy Routine (First Version)



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

- Simple Solution to Simple Problem
- ReadKeyboard and WritePrinter Probably Easily Re-Usable

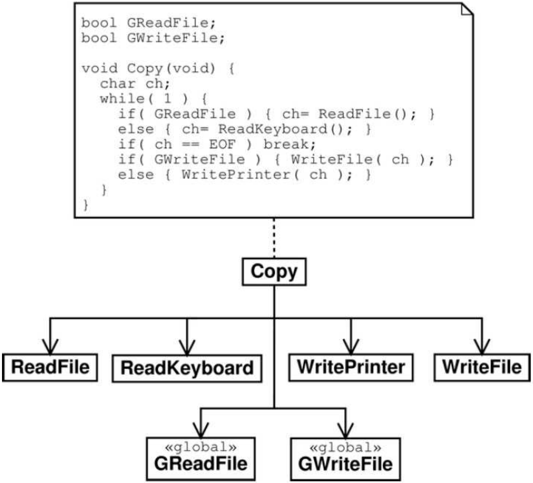
The Copy Routine (Slightly Revised Version)



```
bool GFile;

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

- Well...
- ...Maybe Users Want to Read Files as well w/ Changing their Code
- Used Global Variable (;-)) but Backwards-Compatible

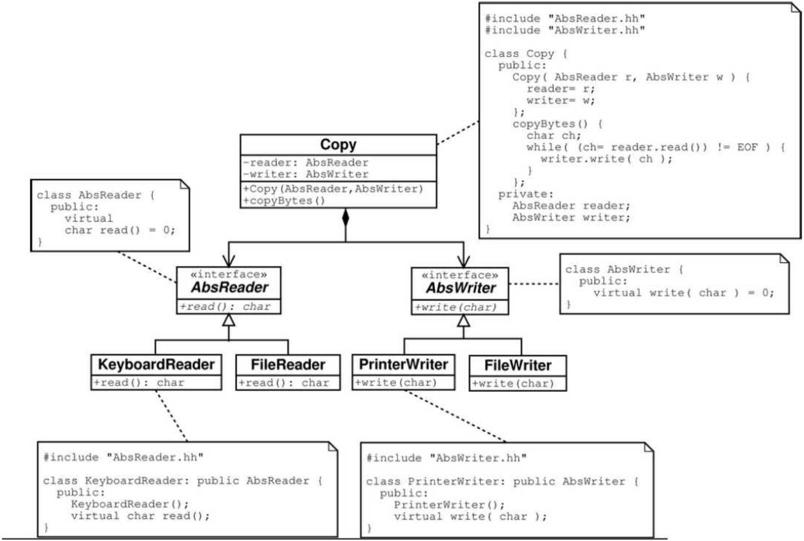
The Copy Routine (...Revised again!?!?!!???)



```
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 ); }
  }
}
```

- Backwards-Compatible, but...
- ...another Global Variable ;-((Things Get Increasingly Complicated...)

The Copy Routine (Doing it Properly!)



```
#include "AbsReader.hh"
#include "AbsWriter.hh"

class Copy {
  public:
    Copy( AbsReader r, AbsWriter w ) {
      reader= r;
      writer= w;
    };
    copyBytes() {
      char ch;
      while( (ch= reader.read()) != EOF ) {
        writer.write( ch );
      }
    };
  private:
    AbsReader reader;
    AbsWriter writer;
}
```

```
class AbsReader {
  public:
    virtual
    char read() = 0;
}
```

```
class AbsWriter {
  public:
    virtual write( char ) = 0;
}
```

```
#include "AbsReader.hh"

class KeyboardReader: public AbsReader {
  public:
    KeyboardReader();
    virtual char read();
}
```

```
#include "AbsWriter.hh"

class PrinterWriter: public AbsWriter {
  public:
    PrinterWriter();
    virtual write( char );
}
```

- Dependency between Readers/Writers Broken
- Easy to Add New Features without Need to Change ‘Copy’ Itself!!

Thank you for your Attention!!!!!!!!

References

1) **Design Patterns. Elements of Reusable Object-Oriented Software**
E. Gamma, R. Helm, R.Johnson, and J. Vlissden
Addison-Wesley Longman, Amsterdam

2) **Agile Software Development. Principles, Patterns, and Practices**
R. Marting
Prentice Hall International

3) **Communicating Software Patterns**
Dr.Stefan Kluth
Presentation at Workshop on Advanced Software Design, Dresden, 9/2011

4) Software Packages for UML

4a) **Dia**
http://projects.gnome.org/dia/

4b) **BOUML**
http://bouml.free.fr/

4c) **Umbrello UML Modeller**
http://uml.sourceforge.net/