Design Patterns in HEP software

Nobu Katayama

Wilson Laboratory
Cornell University
Ithaca, New York, 14853, U.S.A.

September 1995

Computing in High Energy Physics,
Rio de Janeiro, Brazil

Where I started from

- ⋆ "Design Patterns — Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson and J. Vlissides. Addison Wesley, ISBN 0-201-63361-2,

  Foreword by G. Booch starts with:

  "All well-structured object oriented architectures are full of patterns. . . "

  Many patterns and examples will be quoted from "the book" in this talk.

- ⋆ Patterns home page:

  http://st-www.cs.uiuc.edu/users/patterns/patterns.html

- ⋆ C. Alexander, et al. "A Pattern Language," Oxford University Press, New York, 1977.

- ⋆ In this talk, I will try to convince you that:

  - A lot of patterns can be found in HEP software,

  - Using design patterns one can improve the software.

- ⋆ I will not try to show and explain all of the 23 patterns in the book to you.

What is a design pattern

⋆ C. Alexander says, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

⋆ Four essential elements of a pattern :

- Pattern name is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- Problem describes when to apply the pattern. It explains the problem and its context.
- Solution describes the elements that make up the design, their relationships, responsibilities, and collaborations.
- Consequences are the results and trade-offs of applying the pattern. Concerns for space, time, language and implementation issues. Impact on a system's flexibility, extensibility or portability should be discussed.

⋆ Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive (and therefore trivial) building block.

Example of a (micro) pattern

- ⋆ Pattern name : Initialization
- ⋆ Problem : It is important for some code to be executed only once at the beginning of the execution of the program.
- ⋆ Solution : The solution is to use a static variable that holds information on whether or not the code sequence has been executed.
- ⋆ Consequences : The solution requires the language to have a static variable that can be allocated storage at the beginning of the execution, initialized prior to the execution and remain allocated until the program termination.

By now you know what I am talking about.

Sample code

```
      SUBROUTINE MYSUB
      LOGICAL INIT
      SAVE INIT
      DATA INIT/.TRUE./
...
      IF(INIT)THEN
C
C Block of code executed once at the
C beginning of the program
C
        INIT = .FALSE.
      ENDIF
C
C Block of the code executed every time the
C subroutine is called.
C
      END
```

What a design pattern is NOT

⋆ It is not a toolkit or class library. A toolkit is a set of related and reusable classes designed to provide useful, general-purpose functionality.

⋆ It is not a framework. A framework is a set of cooperating classes that make up a reusable design for a specific class of software.

⋆ It is not a style guide. A style guide is a set of recommendations that make the software **consistent**, portable, robust and effectively achieve the goals of (object oriented) design.

Patterns are more abstract than toolkits/frameworks.

Each pattern focuses on a particular object-oriented design problem and try to solve it.

Patterns are based on practical solutions that have been implemented successfully.

Where to use a design pattern

In a development cycle :

Requirements→Analysis→Design→Implementation

⋆ Without Design Patterns

- Analysis
- do {
-   Design
- } while (Implementation is not good);

⋆ With Design Patterns

- Analysis
- Design
- Look up an appropriate pattern
- Implementation

⋆ This is of course an exaggeration and will come back to this point from another point of view of the development cycle.

How does a design pattern help

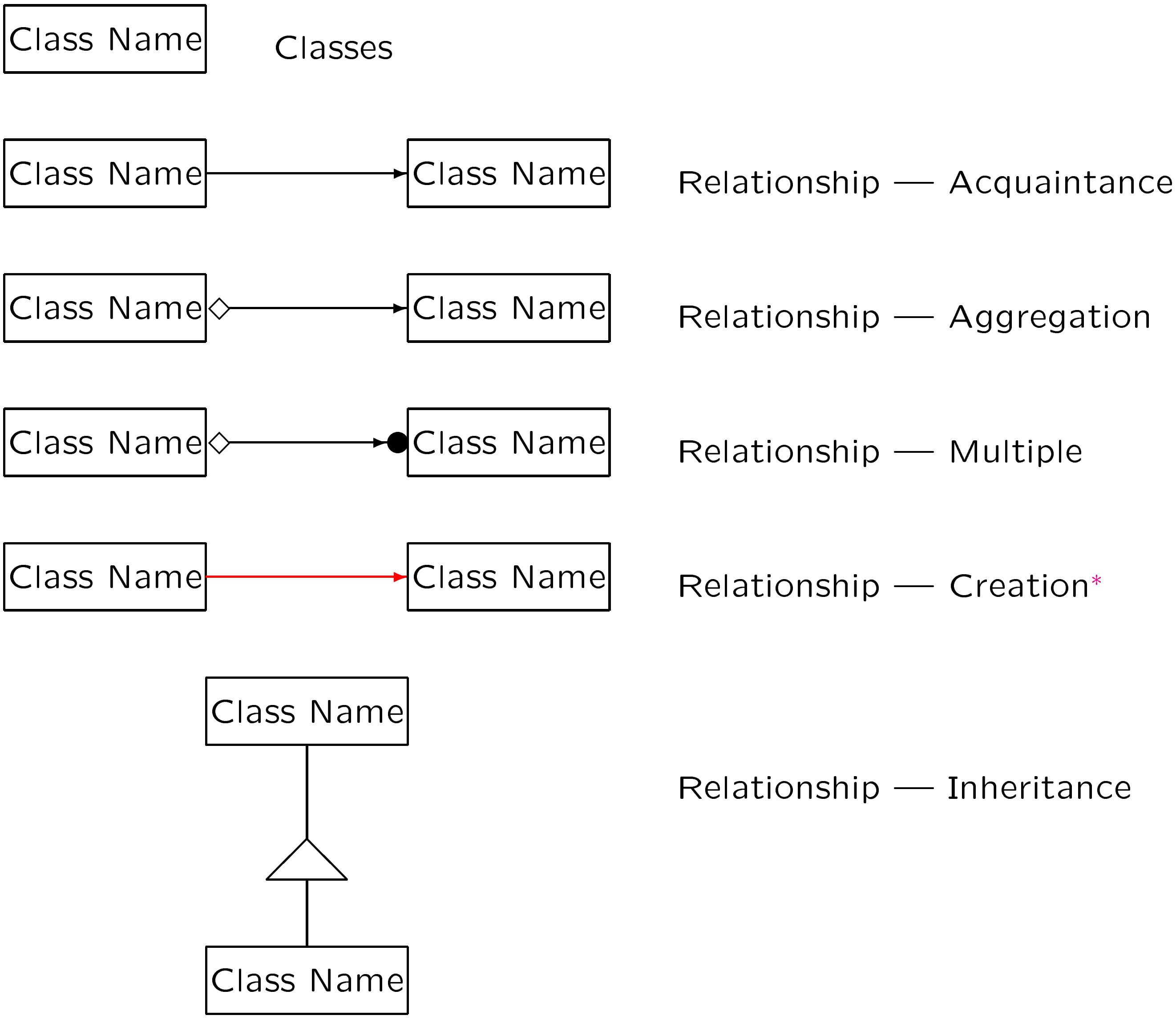
- ⋆ A design pattern rarely involves more than several classes.
- ⋆ It often gives an elegant solution to a small/local problem.
- ⋆ It helps to build an implementation of a medium (sub) project cleanly, effectively and efficiently.
- ⋆ For a bigger project, design patterns will certainly help but you will need other formal design techniques or tools.
- ⋆ It helps to make an existing class/objects more flexible, extensible, reusable.
- ⋆ It certainly suits me as the patterns help me to solve problems I deal with.

Why design patterns are object oriented

- Not necessarily,
- The design patterns in the book are for object oriented designs.
- Most patterns in the book use inheritance.
- However, it is discussed in the book again and again that composition (aggregation) is as important as inheritance as couplings are much looser and more flexible.

A few examples from the book

⋆ Notation (OMT with an addition)



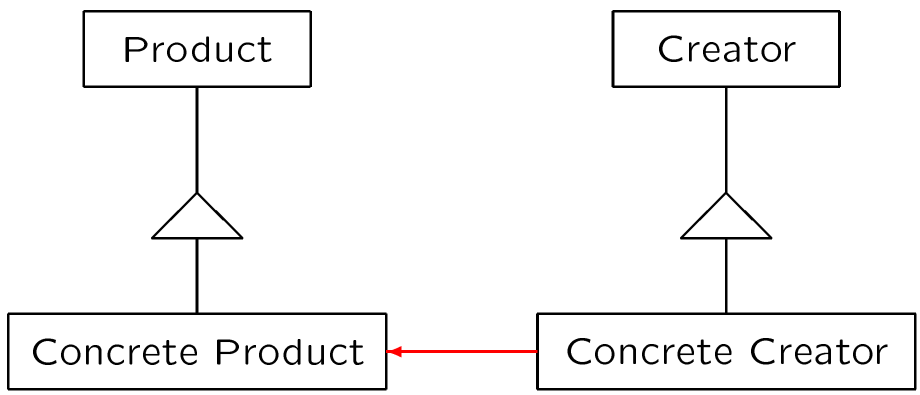
* Not in OMT.

Factory Method

⋆ Intent : Define an interface for creating an object, but let subclasses decide which class to instantiate.

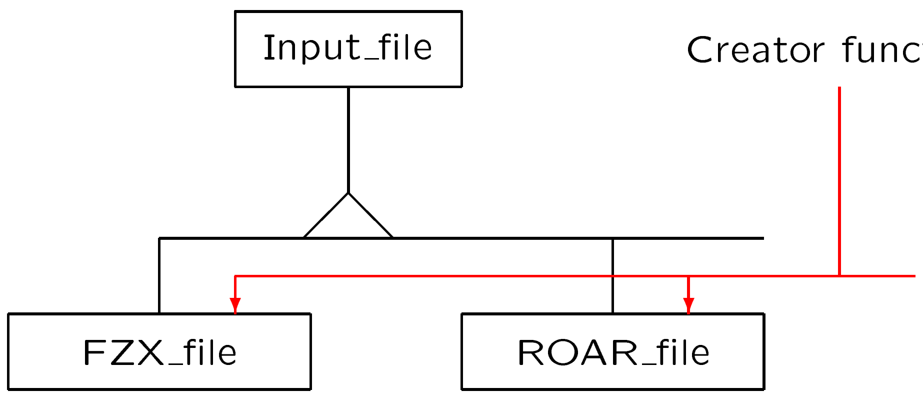
⋆ Also Known As : Virtual Constructor

⋆ Structure :



Factory Method (HEP example)

- ⋆ Read files with various formats.
- ⋆ Started with a simpler version of the Factory pattern (without knowing it's a pattern.)

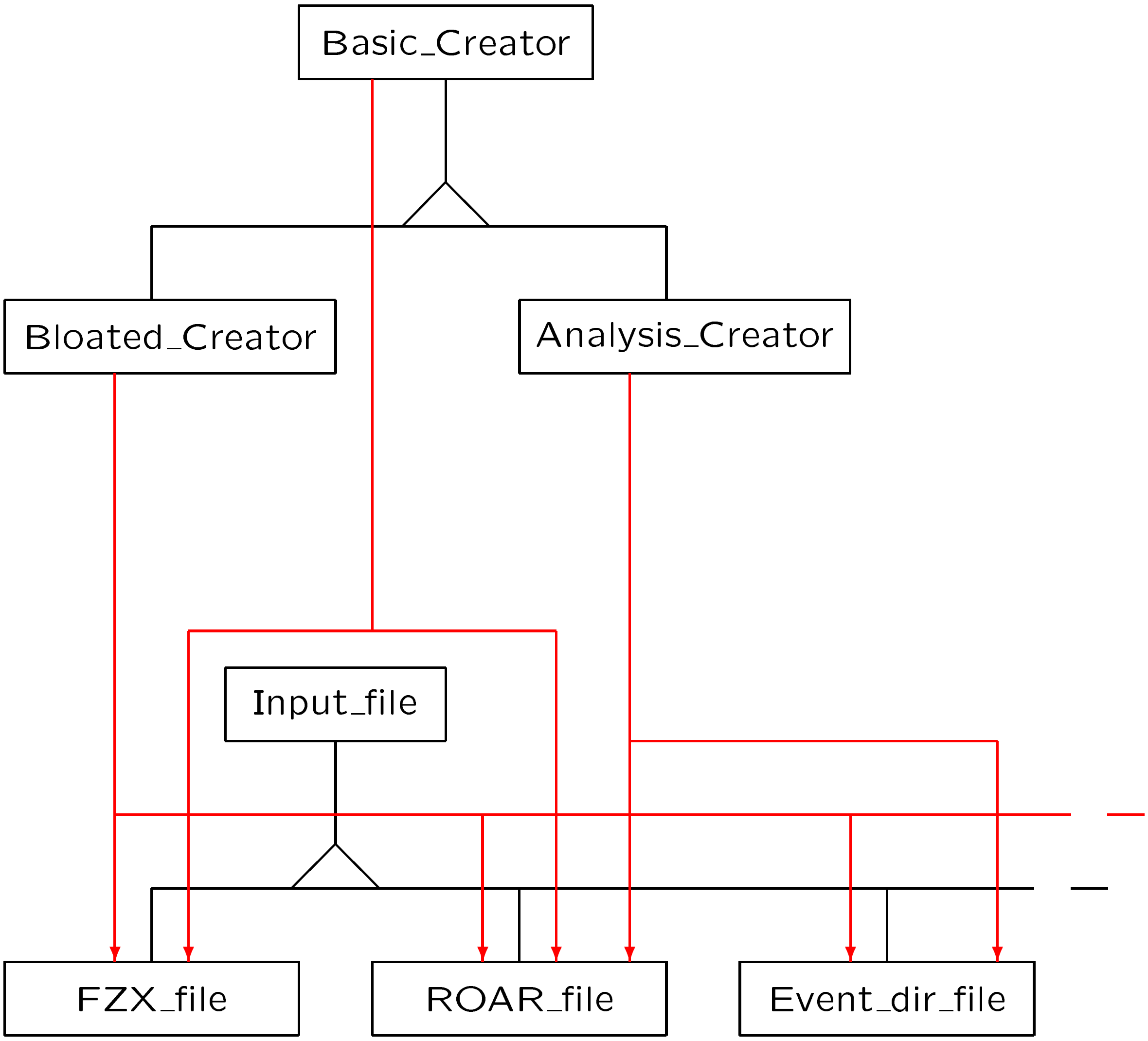


- ⋆ Design change : Add more file types.
- ⋆ Bloated executable : One does not always want to link with code for all possible formats.

→ Make the creator parameterized (as described in the Factory Method pattern).

HEP example (continued)

⋆ Use creator objects



Sample code

⋆ Basic job

```
main() {
    Basic_creator *creator = new Basic_creator;
...
    Input_file *file = creator->open_file(filename);
...
}
```

⋆ Full job

```
main() {
    Basic_creator *creator = new Bloated_creator;
...
    Input_file *file = creator->open_file(filename);
...
}
```

⋆ Tiny analysis job

```
main() {
    Basic_creator *creator = new Analysis_creator;
...
    Input_file *file = creator->open_file(filename);
...
}
```

Singleton

- Intent : Ensure a class only has one instance and provide a global point of access to it.

- A well known example

```
      SUBROUTINE ZFINIT
      LOGICAL INIT
      SAVE INIT
      DATA INIT/.TRUE./
C Body
      IF (.NOT.INIT) RETURN
      INIT = .FALSE.
C Executable code.
      END
```

- Wrapper for HBOOK. Who calls HLIMIT ?

In a large reconstruction program which is a collection of many individually written small packages, every one wants to call HLIMIT (MZEBRA,,,,).

In a C++ project, one would like to forget about HLIMIT.

Sample code

⋆ Class definition and implementation

```
class Hbook {
public:
  Hbook(int pawc_size);
  Hbook_file *open(const char *filename);
...
};
extern "C" hlimit_(const int *);

Hbook::Hbook(int pawc_size) {
  int n = pawc_size;
  hlimit_(&pawc_size);
}
```

⋆ Main program

```
extern struct {
  int pawc[1000000];
} pawc_;

main() {
  Hbook h(1000000);
  Hbook_file *f = h->open("hbook.rzn");
...
```

⋆ Using the above definition, one cannot assure that only one Hbook object exists as the constructor of the Hbook class is public.

The Singleton pattern

⋆ Make the constructor protected and use a static function to access the object.

```
class Hbook {
public:
  static Hbook *instance();
  static void set_pawc_size(int);
protected:
  Hbook(int);
private:
  static Hbook *_instance;
  static int _pawc_size;
};

static const int local_pawc_size = 1000000;
extern struct {
  int pawc[local_pawc_size];
} pawc_;
int Hbook::_pawc_size = local_pawc_size;
Hbook *Hbook::_instance = NULL;

Hbook *Hbook::instance() {
  if(_instance==NULL) {
    _instance = new Hbook(_pawc_size);
  }
  return _instance;
}
void Hbook::set_pawc_size(int new_size) {
  if(_instance!=NULL)
    throw "Hbook object already exists";
  _pawc_size = new_size;
}
```

Why use design patterns

- ⋆ A pattern is abstract. It is applicable to more than one problem. Apply to GZEBRA, MZEBRA, KUINIT, HPLINI. . . .

- ⋆ The pattern offers a solution. The solution is often an elegant one.

- ⋆ It makes a common design vocabulary. (Pattern catalog) "Let's use a Singleton here," or, "Let's make a Factory Method out of these classes."

- ⋆ It makes re-factorization easier. Three phases of the OO development : prototyping, expansionary, and consolidating. Design patterns help keep moving from one phase to another. For example in the previous cases,

  - The Factory method pattern was introduced in the expansionary phase and,

  - The Singleton pattern was used in the consolidating phase.

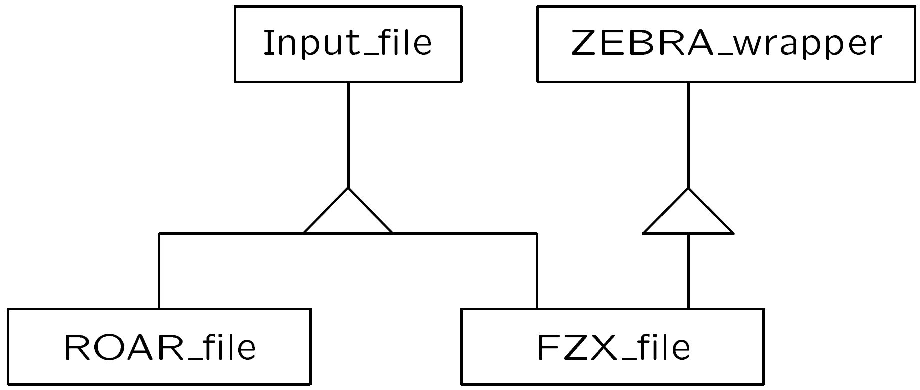
Pattern catalog

- By purpose
  - Creational patterns concern the process of object creation.
  - Structural patterns deal with the composition of classes or objects.
  - Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility.
- By scope
  - Class patterns deals with relationships between classes and their subclasses which are established through inheritance, so they are static and fixed at compile-time.
  - Object patterns deal with object relationships, which can be changed at run-time and are more dynamic.
- As the book emphasizes the importance of composition, there are many more object patterns in the book.

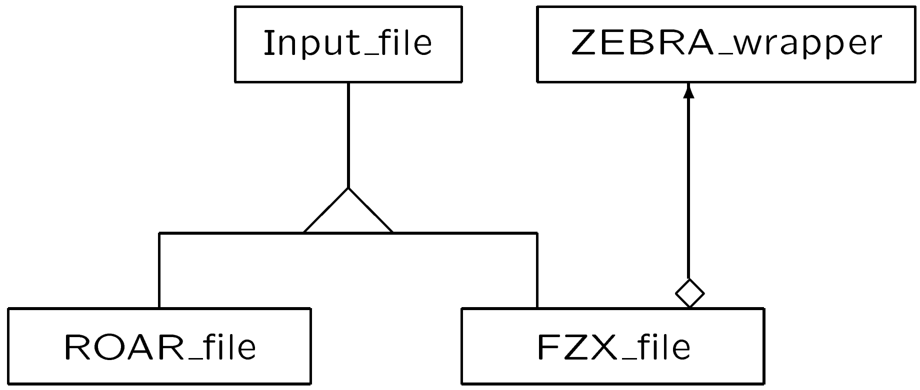
Adapter (Structural pattern)

⋆ Intent : Convert the interface of a class into another interface clients expect.

⋆ Structure : Class Adapter



⋆ Structure : Object Adapter



Adapter (continued)

- ⋆ The ZEBRA_wrapper class is a straight forward wrapper or does already exist.

- ⋆ The FZX_file class has the same interface as other file objects. We can either :

  - Inherit from ZEBRA_wrapper privately or,
  - Aggregate a ZEBRA_wrapper object.

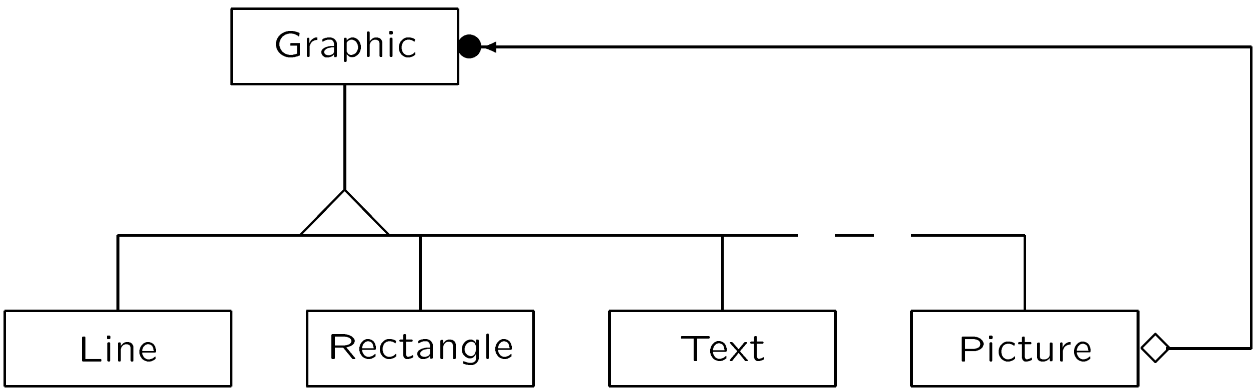
  In either case, the interface of ZEBRA_wrapper is converted to the interface of Input_file.

- ⋆ Class and object adapters have different trade-offs. The book states them well and you can decide which pattern to use.

Composite (Structural pattern)

- Composite objects into tree structures to represent part-whole hierarchies. Composite let clients treat individual objects and compositions of objects uniformly.

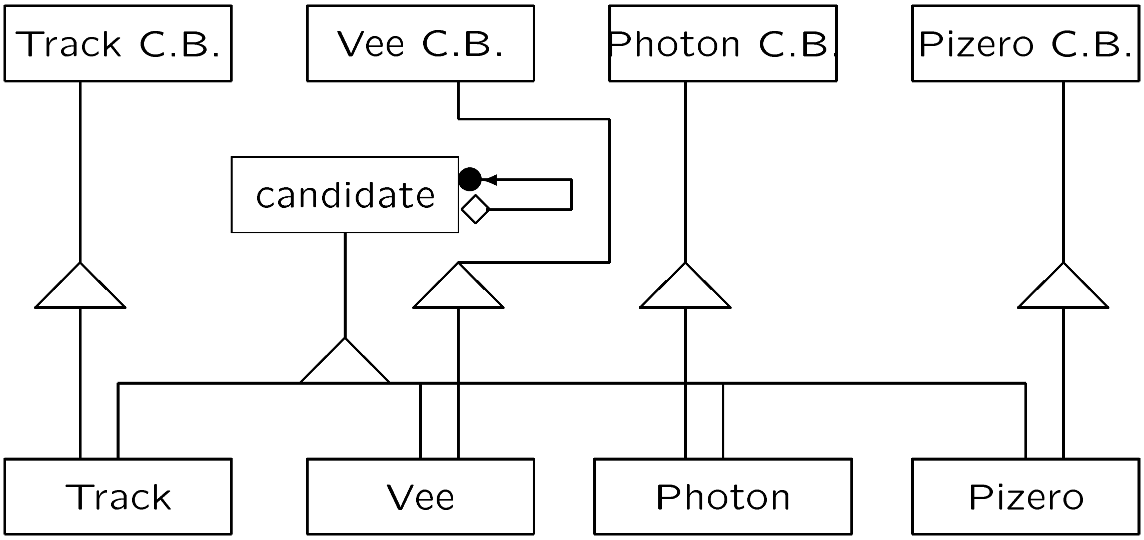
- Structure :



- You do not have to be a genius to come up with this pattern. No surprise the pattern can be seen in various HEP software. (That's why it is a pattern.)

Composite pattern in HEP

⋆ CABS candidates class : Instances of $B \to D^*\pi, D^* \to D^o\pi, D^o \to K^-\pi^+\pi^-\pi^+$ decays

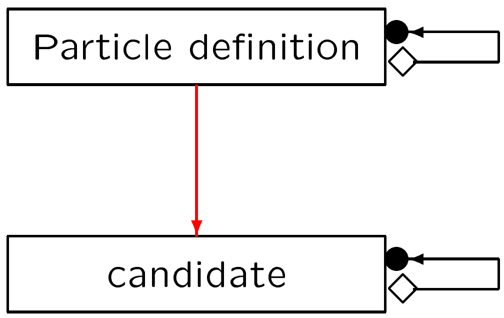


⋆ xxxx C.B. is a wrapper class to interface to variables in the FORTRAN common blocks. (The Adapter pattern.)

⋆ I chose to not use a separate "composite" class (Picture in the previous example). The trade-offs are again stated in the book.

Composite pattern in HEP (continued)

- CABS Particle definition class describes the decay definition. $B \to D^* \pi, D^* \to D^o \pi, D^o \to K^- \pi^+ \pi^- \pi^+$.



- I found the Builder pattern in my code. Each candidate is created as the particle definition structure is traversed.

Another example : CLEVER++

- ⋆ CLeo EVEnt Reconstruction program.

- ⋆ Processors (individual reconstruction packages)

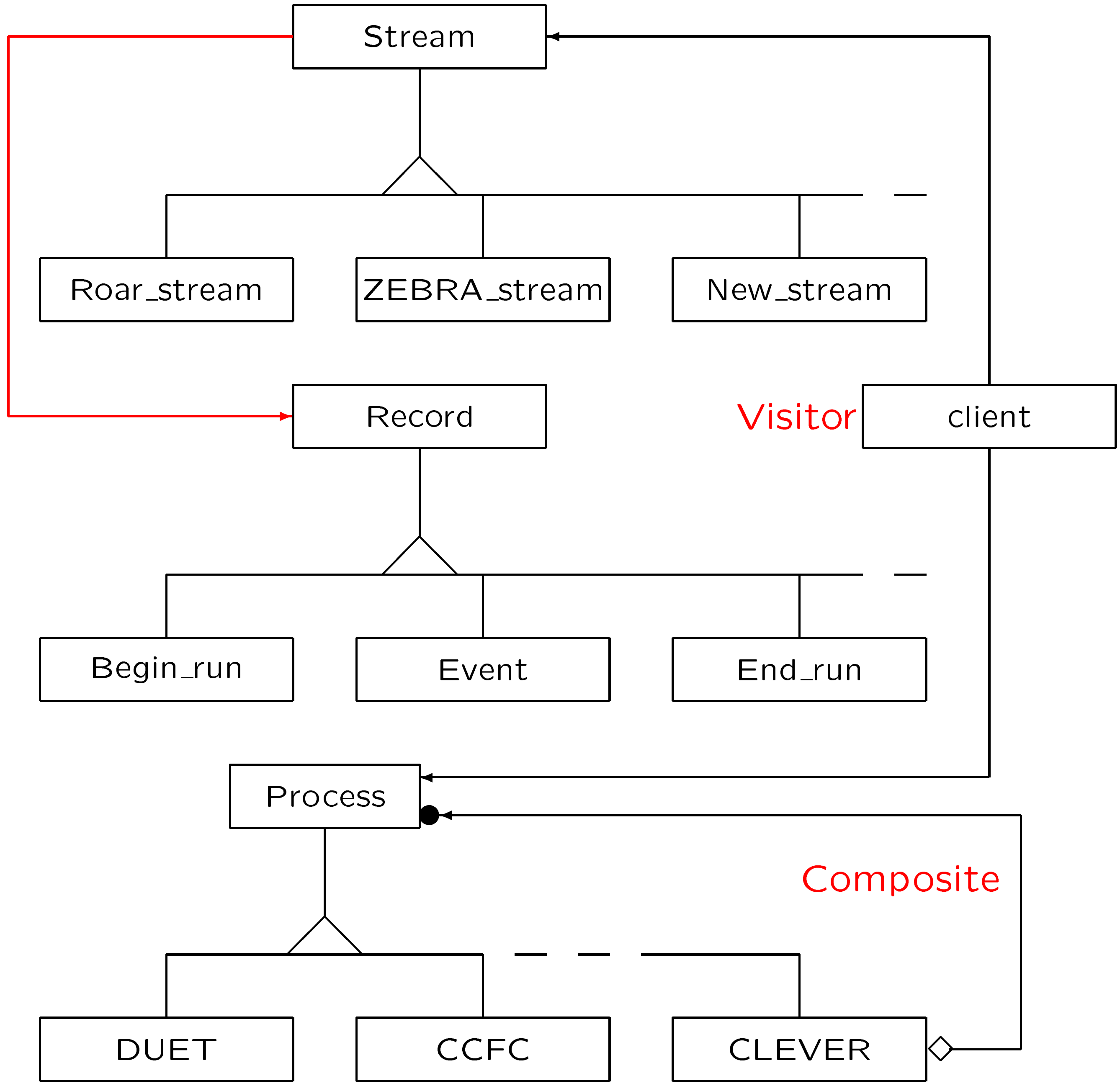
```
      PROGRAM MAIN
      CALL CLVRBJ

   10 CALL GET_NEXT_RECORD

        if (record_type.eq.event) CALL CLVREV
        else if (record_type.eq.begin_run) CALL CLVRBR
        else if (record_type.eq.end_run) CALL CLVRER
C       deal with other record types

      IF (.NOT.end_of_file) goto 10
      CALL CLVREJ
      END
```

- ⋆ CLVRXX routine (using KUIP)

```
      SUBROUTINE CLVREV
      for(int i=0;i<length;i++) {
        (*list[i])();
      }
```

where *list[i] points to subroutines xxxxEV, such as CCFCEV, DUETEV, MUTREV.

* Using C++, each processor is an object. Each type of processor is a class. (Mostly there will be a single instance of each processor.)



Sample code

⋆ Implementation of main program

```
main() {                                    Singleton

  Zfiles_init::instanciate(4);          Factory method

  Stream *in = create_stream("a.fzx", "IN");

  Clever clever;

  Ccfc ccfc;

  Ccc ccc;

  clever += &ccfc;
  clever += &ccc;
  Record *rec;
  int iev = 0;                      Visitor
  do {
    (rec=in->next())->process(clever);
    delete rec;
  } while(iev++<10);

  Zfiles_init::end();
}
```

Class definitions

Zfiles_input_fzx : Zfiles	
Zfiles_input_fzx(name:const char *, topdir:const char *)	
next	Record *
get_pending(division:int&, link:int&, wipe:int=1)	

Record	
private _input	Zfiles_input_fzx&
protected Record(:Zfiles_input_fzx&)	
virtual process	
get_pending(division:int&, link:int&, wipe:int=1)	

Begin_run : Record	
private _input	Zfiles_input_fzx&
protected Begin_run(:Zfiles_input_fzx&)	
virtual process	

Event : Record	
private _input	Zfiles_input_fzx&
protected Event(:Zfiles_input_fzx&)	
virtual process	

Class definitions (continued)

Process
Process()
~Process()
virtual begin_run(:Record&)
virtual end_run(:Record&)
virtual event(:Record&)
virtual mc_event(:Record&)

Clever : Process	
private _list	List<Process*>
Clever()	
~Clever()	
virtual begin_run(:Record&)	
virtual end_run(:Record&)	
virtual event(:Record&)	
virtual mc_event(:Record&)	
operator +=(:Process*)	

Implementation (Zfiles_input_fzx::next())

```
Record *Zfiles_input_fzx::next() {
  int nkey = 0;
  int jkey[1];
  int nhead = 100;
  int ihead[100];

  zfrhed_(topdir(), "S", &nkey, jkey, &nhead, ihead,
          strlen(topdir()), 1);

  if (ihead[1]==1) return new Event(*this);
  else if (ihead[1]==4) return new MC_event(*this);
  else if (ihead[1]==5) return new MC_constants(*this);
  else if (ihead[1]==100) return new Begin_run(*this);
  else if (ihead[1]==140) return new End_run(*this);
  else if (ihead[1]==180) return new MC_begin_run(*this);
  else return new Other(*this);
}
```

Implementation (Record)

```
Record::Record(Zfiles_input_fzx&in) : _input(in) {}

void Record::get_pending(int&division, int&link, int f) {
  _input.get_pending(division, link, f);
}

Begin_run::Begin_run(Zfiles_input_fzx&in) : Record(in) {}

void Begin_run::process(Process &p) {
  p.begin_run(*this);
}

Event::Event(Zfiles_input_fzx&in) : Record(in) {}

void Event::process(Process &p) {
  p.event(*this);
}

MC_event::MC_event(Zfiles_input_fzx&in) : Event(in) {}

void MC_event::process(Process &p) {
  p.mc_event(*this);
}
```

Implementation (Clever)

```
Clever::Clever() {
}

void Clever::operator+=(Process *p) {
  _list += p;
}
```

Observer

```
void Clever::begin_run(Record&r) {
  // Always process the next record for now
  r.get_pending(icldiv, lclev);
  Zfiles_init::begin_run(cliq[clkq+lclev+4]);
  for(List_iter<Process*> i(_list);!i.end();i++) {
    i()->begin_run(r);
  }
}
```

Observer

```
void Clever::event(Record&r) {
  // Always process the next record for now
  r.get_pending(icldiv, lclev);
  Zfiles_init::begin_event(cliq[clkq+lclev+5]);
  for(List_iter<Process*> i(_list);!i.end();i++) {
    i()->event(r);
  }
}
```

Implementation (Ccfc, a wrapper for a FORTRAN package)

```
Ccfc::Ccfc() {
  ccfcbj_();
}

void Ccfc::begin_run(Record&r) {
  ccfcbr_();
}

void Ccfc::end_run(Record&r) {
  ccfcer_();
}

void Ccfc::event(Record&r) {
  ccfcev_();
  int flag = 1;
  ccprcl_(&flag);
  flag = 0;
  gpkine_(&flag);
}

Ccfc::~Ccfc() {
  ccfcej_();
}
```

Patterns found in CLEVER++

- CLEVER++ was a prototype written before reading the book carefully. I found that Composite, Visitor and Factory method patterns are used. (I was very disappointed to find :

  ```
  do {
    input_stream->next()->process(clever);
  } while(1);
  ```

  is the Visitor pattern in the book. In the book it was also mentioned that this technique is called "double-dispatch" and is used often. . .

- After reading the book, I now know I can use State pattern to deal with out-of-order / pathological input record sequence.

- Abstract Factory pattern can be use to create Stream and Processors in a more flexible way.

Other patterns found in HEP software

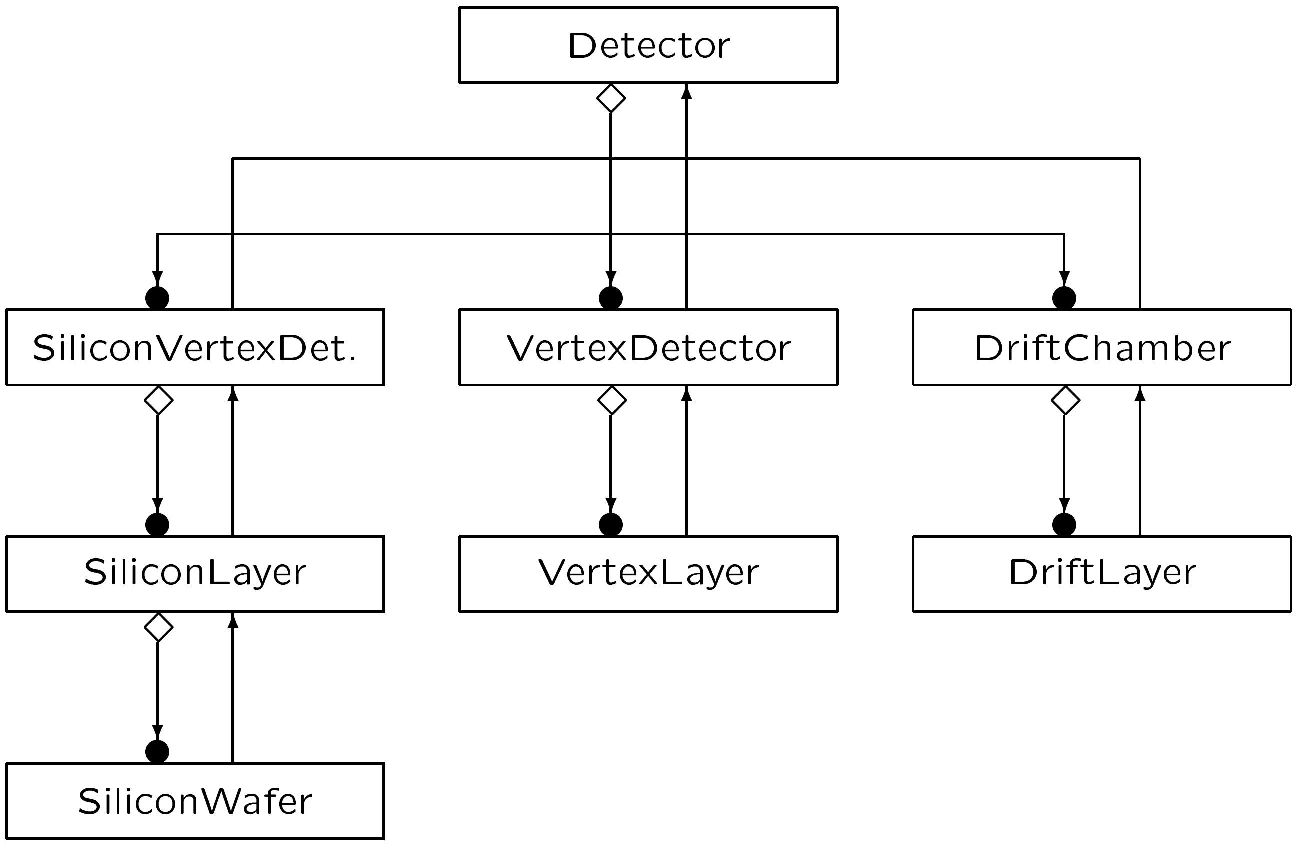
- Geometry tree search algorithm is important in HEP simulation and reconstruction code. The Chain of responsibility pattern is often used to implement some of the operations of the geometry package.

- The example in the book describes how to implement a context sensitive help facility using the chain of responsibility pattern.

- Intent : Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

- Our tracking code (See CHEP 94 proceedings) uses this pattern for several operations of the geometry objects.

Detector structure



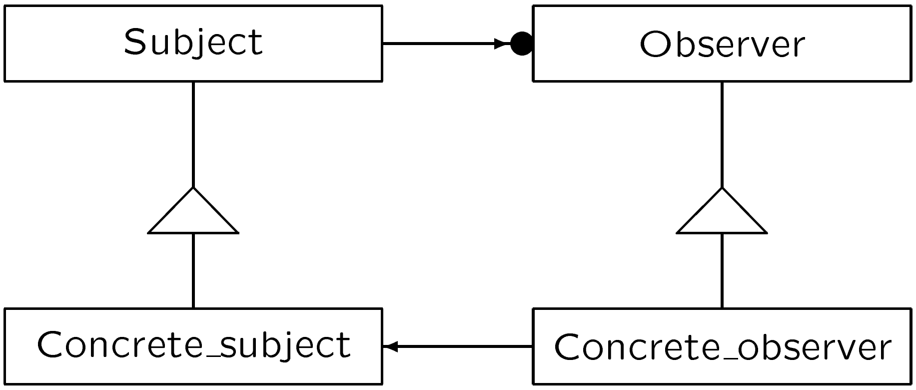
⋆ To efficiently assign hits, the list of hits is handled by each individual object. When a hit unknown to the object is found, the object passes the list to another object in line which is responsible for handling the rest of hits.

Other patterns in HEP software

- ⋆ ZFILES (a constants database on top of ZEBRA, written in FORTRAN many years ago) uses Observer pattern to update constants when a new begin-run record is read from the input stream.
- ⋆ For each set of constants, when it is copied as a BANK in a ZEBRA common block from an RZ file, a reference link will be added to the list of active constants.
- ⋆ When the begin-run record is encountered, the validity of all active constants are checked through the list of reference links. If a set of constants becomes invalid for the new run, a new set is read and replaces the old, invalid constants.
- ⋆ Intent of the Observer pattern : Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- ⋆ Also known as : Dependents, Publish–Subscribe.

Other patterns in HEP software

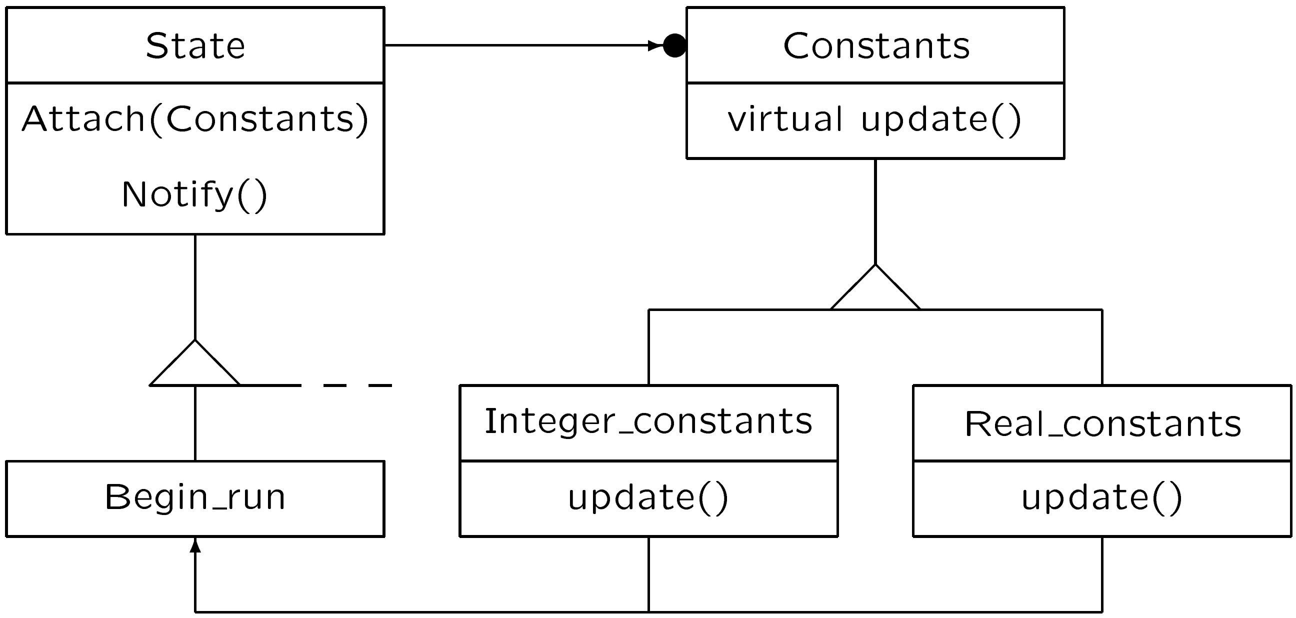
- Structure :



- After reading the book a prototype ZFILES wrapper was improved by clearly implementing the Observer pattern.

- A clear understanding of the mechanism of the pattern described in the book (as opposed to vague thoughts in my mind) was especially important when implementing the constants objects as remote objects using CORBA IDL.

Updating the constants



⋆ When a new begin run is encountered,

```
void State::Notify() {
  for all c in constants_list {
    c->update();
  }
}
```

will be executed. The update function is responsible to update the constants.

ODL front-end

⋆ What is ODL, IDL, CORBA, ORB, OMG, ODMG, SOM, OpenDoc. . . ? (Jamie's talk on RD45, HEP persistent objects)

⋆ ODMG standard for ODBMS (Object database management system)

- Object Model
- Object Definition Language
- Object Query Language
- C++ language binding
- . . .

⋆ Defines source code level portability

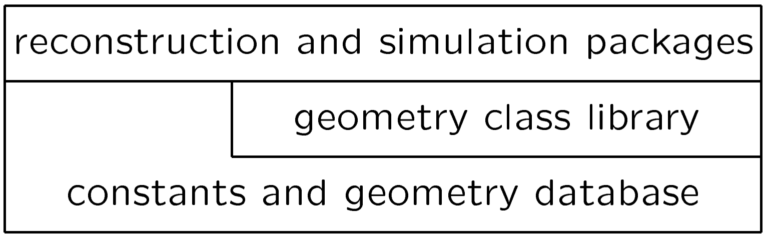
⋆ Why is it important for HEP ?

$\Rightarrow$ Description of the geometry of the detector

⋆ The geometry package for the detector simulation and reconstruction will most likely be a class library for each detector (CLEO, BaBar, Belle, ATLAS, CMS. . .).

ODL front-end (continued)

⋆ As it will be very complex I doubt if one can make a database (of attributes) out of it. I envision that it will be a hierarchical collection of objects which interacts with simulation and reconstruction packages of sub-detectors.



⋆ However, the interface should be as standardized as possible so that it can easily talk to other packages such as :

- GEANT 3 / 4
- CAD / STEP
- Interactive Graphical display package
- ODBMS
- CORBA / ORB

⇒ It will be a tremendous advantage to be able to describe the geometry objects and their interface using a standard definition language. **ODL** is clearly one of the best choices.

ODL front-end (continued)

- ⋆ A front-end of ODL compiler was written taking the syntax from the proposed ODMG–93 ODL Grammar BNL. (ISBN 1–55860–302–6). In the compiler a Interpreter pattern was found. The pattern describes how to define a representation of the grammar of a language.
- ⋆ It was learned from the book that using the Visitor pattern (again) enables to decouple the operation (compilation of the ODL) from the object structure (the parsed source code).
- ⋆ Again, the book states clearly the trade-offs which we took very seriously before adopting the Visitor pattern.
- ⋆ The compiler front-end will allow us to learn about the language, add our own implementation in C++ to write the geometry class library using the standard interface.

Conclusions

- ⋆ Indeed one can find a lot of patterns in one's code. HEP software is no exception, especially because it is written by people as smart, creative and experienced as you are.

- ⋆ I did not show examples of more algorithmic code but you can find patterns there as well though they may be subtle and often hidden underneath lines of formulae.

- ⋆ One can improve one's code by applying appropriate patterns.

- ⋆ One can quickly move in the analysis → design → implementation loop without unnecessary iterations.

- ⋆ It makes re-factorization easier. In the three phases of the OO development; prototyping, expansionary, and consolidating, design patterns help when moving from one phase to another. Using patterns, one can extend and/or consolidate one's design and implementation efficiently.