Applying an Object Oriented Approach to Off-line Reconstruction at the LHC

A. (RD) Schaffer
LAL, Orsay, France

Abstract

Using an example taken from work on a prototype for electron track reconstruction in the Atlas detector, this paper will try to guide the reader through an object oriented (OO) design. It will attempt to illustrate how to approach a problem in an OO manner using a few different modelling techniques. In particular, we will introduce the use of *patterns* to describe object structure and behaviour.

1 Introduction

When one is used to programming in a procedural language, such as C or Fortran, and he/she first attempts to do an object-oriented development, it is often confusing as to where to begin. I do believe that it is important to learn to approach software development in a systematic way, for instance by using a global process development model as is described by the ESA standard [1], matched with a software methodology such as Booch [2] or OMT [3] for object-oriented developments. However, when first starting off, it can be difficult to see how '*theory*' can be applied.

In the following, I will try to explain in a simple manner how one can attempt to solve a problem using an object-oriented approach. We will follow an example which is derived from work on a prototype* which attempts to reconstruct electrons in the Atlas detector.[1] The steps we will follow can be summarized as:

- Understand the reconstruction algorithm
- Identify some initial categories of classes
- Describe some of the basic scenarios, i.e. jobs to be done
- Walk through the different scenarios to identify the classes needed and to decide '*who does what*'

One of the central activities of an object-oriented development is to identify the *key* abstractions of the problem. This is true whether one is trying to identify the different objects which are communicating with each other, or even to abstract different '*patterns*' of object structure or behaviour. I highly recommend the interested reader to have a look an excellent discussion on design patterns [5] which has recently appeared. This book captures a large amount of experience in object-oriented design and can help boot-strap oneself by learning from others' experience. In the following, I will use their patterns to help establish a standard design vocabulary, and will also adopt their diagram notation.

* This work is currently being carried out within the Moose Collaboration [4], a research project investigating the feasibility of adapting an Object Oriented approach to off-line software development for the LHC.

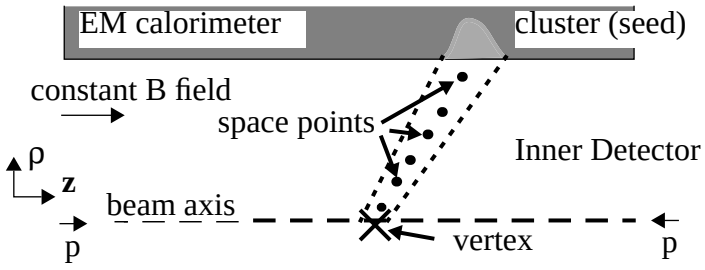
[1] Atlas will be one of two experiments at the LHC at CERN exploring 17 TeV centre-of-mass energy pp interactions. It is expected to turn on in 2004-2005.

2 Overview of the problem: electron reconstruction

Electrons can be recognized by a characteristic signature in the detector:

- An electromagnetic shower in the calorimeter, and
- a track found in the Inner Detector which points to the shower cluster, and has a measured momentum which agrees with the energy seen in the calorimeter (E/P).

This can be seen schematically in Figures 1 and 2:



The electron trajectory is a straight line in this projection.

Figure 1: Transverse view (ρ - z) of an electron signature

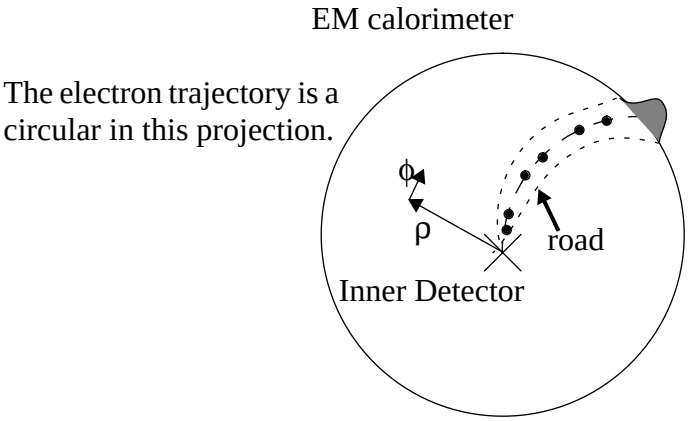


Figure 2: End (ρ – ϕ) view of electron signature

An electron track is made of 6 **space points** which fit to a helical trajectory with deviations due to multiple scattering and Bremsstrahlung. Each space point is made from a hit in 2 adjacent layers of precision **Si-strip detectors**. The Inner Detector is composed of six cylindrical **superlayers**, each with a pair of Si-strip detectors:

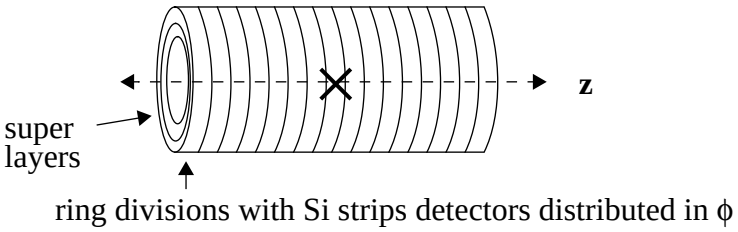


Figure 3: View of divisions of Si-strip detectors

A key point driving the tracking algorithm is the high luminosity at LHC:

- At each beam crossing there are on average 20 interactions.

This implies that an interesting physics event has 100's of 'background' tracks. Thus the strategy to approach the problem is *not* to try and reconstruct *all* tracks, but to '**seed**' the search with information from another detector. In the case of electrons tracks, we seed the reconstruction Inner Detector in with an electromagnetic cluster found in the calorimeter. This search path is called a **road**.

This leads to the following *simplified* procedure for track fitting:

1. Construct a wide road from a calorimeter seed
2. Collect Si_strip_detectors which lie inside the road
3. Construct space points in outer superlayers (lower density of hits)
4. For all pairs of space points
   - fit with vertex and seed
   - construct a narrow road and
   - collect all hits in road
   - fit set of hits
   - select candidates satisfying certain criteria, e.g. lowest χ^2, etc.

Thus, one is looking for the six space points in Figure 2 which in a real event will be superimposed with thousands of other hits from other tracks, noise, etc.

3 Identifying the classes and scenarios

From the description of the problem, it is fairly easy to come up with a list of candidates for objects:

- Detector objects: Inner detector, Si strip detectors, superlayers, Si rings, electromagnetic calorimeter
- Reconstruction objects: EM energy clusters, inner detector hits, space points, seeds, roads, tracks, electrons

What are some of the possible scenarios?:

1. Clearly, for each event we will need to '*input*' the **raw data** produced by the different detectors, e.g. inner detector hits and calorimeter cells with energy.
2. There is initial processing of detector data, e.g. electronic calibration and initial clustering of data.

   As we begin the track reconstruction, we need to:

3. Generate roads from calorimeter clusters
4. Collect detectors and hits in road
5. Build space points from hits
6. Fit and iterate 4-6.

3.1 Dividing up the problem

The next step is to divide up the problem domain by identifying some of the basic categories, or groupings, of objects and defining their primary responsibilities. For the present problem, we can identify 3 categories:

- The ‘raw data’ and the subsequent reconstruction quantities which are to be saved should be grouped together as an **event**.
- We need to have a ‘model’ of the **detector** which is responsible for:
  - ‘knowing’ its geometry
  - accessing the raw data of each new event
  - perform the ‘initial’ steps in reconstructing its own data
- We will probably need to ‘invent’ some ‘reconstructor’ objects to help in the track finding

We conclude that we need **event**, **detector**, and **tracking** categories.

We will mostly concentrate on the **detector** model, where our underlying ‘design’ philosophy is to allow the detector to do ‘*as much possible*’ before inventing new objects to do the work.

4 Definition of notation

We will use three types of diagrams[2] to illustrate the design which we develop: Class diagrams, Object diagrams, and Interaction diagrams. Figure 4 shows the various relationship between classes:

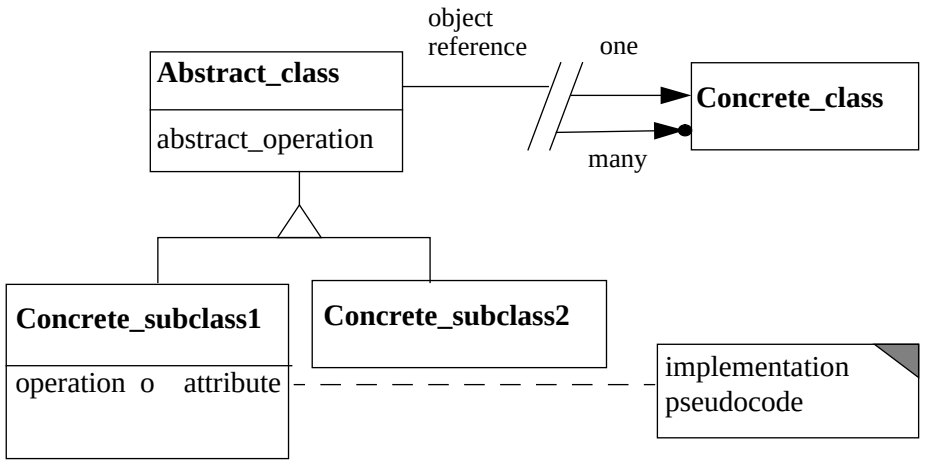


Figure 4: Generic Class diagram

Here subclasses which inherit from an abstract class is represented by a triangle. And a class can ‘refer’ to another class which is given by a straight line, terminated in either an simple arrow or arrow plus dark circle. This termination identifies the *number* of ‘destination’ objects referred to by each ‘source’ object.[3] One can also indicate the functionality of an operation with pseudocode.

Class diagrams represent a static vision of classes. However, when a program runs it is the instances of classes, often just called objects, which are created and interact. It is often useful to

[2] This notation is taken from reference [5].

[3] For example, a one to one relationship means an Abstract_class object will hold a reference to one Concrete_class object, and a one to many relationship means an Abstract_class object will hold a reference to a List or Set of several Concrete_class objects.

give an example of the object structure at a certain moment during a program run. This is expressed in an object diagram:

Figure 5: Generic Object diagram

Here the rounded box refers to an object, and the reference from one object to another is given by an arrow. Note that the explicit name of the object reference is not always shown.

The last diagram, an interaction diagram, describes how a group of objects carry out a scenario. This diagram will be presented and explained later in an example.

5 The detector model

The detector model is a natural place to introduce a parent/child hierarchy or tree structure.

For example, we would like to have some structure of objects such as:

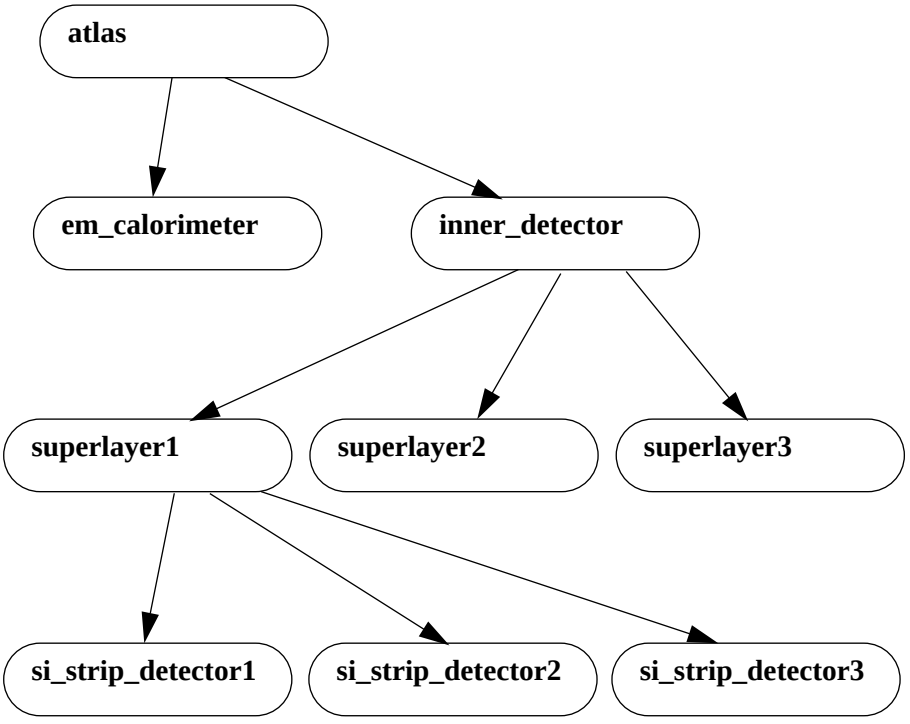


Figure 6: Detector object diagram

In words, one would describe this as an atlas object contains an em calorimeter object and an inner detector object. Similarly, the inner detector contains three superlayer objects, etc. Each of these objects would probably be instances of a Detector class.

5.1 The Composite pattern

This detector hierarchy can be well described by the Composite pattern:

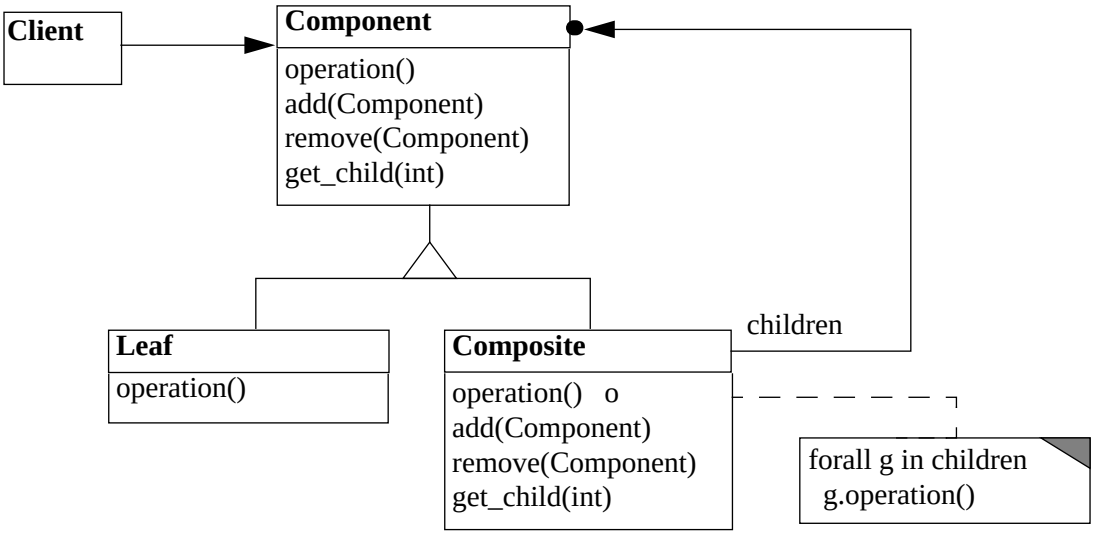


Figure 7: Composite pattern

This pattern combines into the single interface of Component all of the operations that a client needs to access for both containers and primitives. One can 'read' this diagram as follows: a client refers to a single instance of Component. This component can either be a Composite or a Leaf object. A Composite object refers to other Component objects through the *children* reference. And again, these components are either Composites or Leaves.[4] Thus when a client make requests of a component, if it is a Leaf, requests are handled directly. If it is a Composite, requests are forwarded to its children, with Composite possibly performing additional operations.

So how would this look like for our detector hierarchy?:

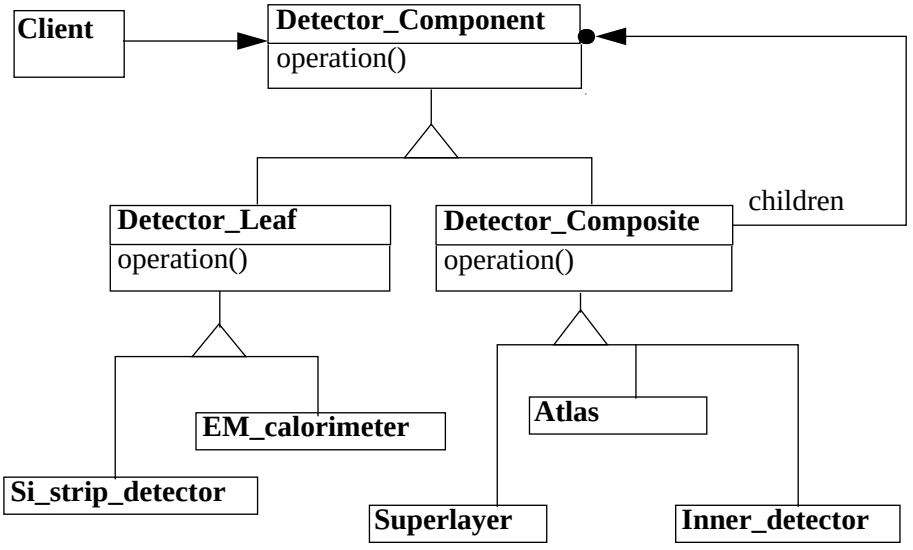


Figure 8: Detector hierarchy

[4] This structure describes a tree which is composed of nodes where each node is either terminal (leaf) or not (composite).

Referring back to Figure 6, one sees that the Leaf objects are the EM_calorimeter and Si_strip_detectors, and that the rest are Composites.
Note that when assigning operations to the different classes:

- default operations are put into Detector_Component, Detector_Composite and Detector_Leaf.
- and these are over-ridden by the subclasses where specialization is needed.

6 Assigning responsibilities to the detector hierarchy

Walking through scenarios is a useful way of identifying the various operations that are needed and who is responsible for performing them. This further clarifies the roles of the different objects and is useful to uncover new objects/classes that may be needed.

6.1 Creating the detector hierarchy

The first job to be done is to create and initialize our detector hierarchy. We define an *initialize* operation which will be abstract[5] in Detector_Component and implemented in Detector_Leaf and Detector_Composite. The initialize operation simply reads a *geometry description file* of the form:

```
atlas              {  class { Atlas }
                      children { inner_detector, em_calorimeter }        }
em_calorimeter {  class { EM_calorimeter }                               }
inner_detector  {  class { Inner_detector }
                      children { superlayer1, superlayer2, superlayer3 }  }
etc.
```

and, depending upon what is in the file, it will initialize the geometry of each object, create the children objects and continue the initialization recursively. This can be described as:

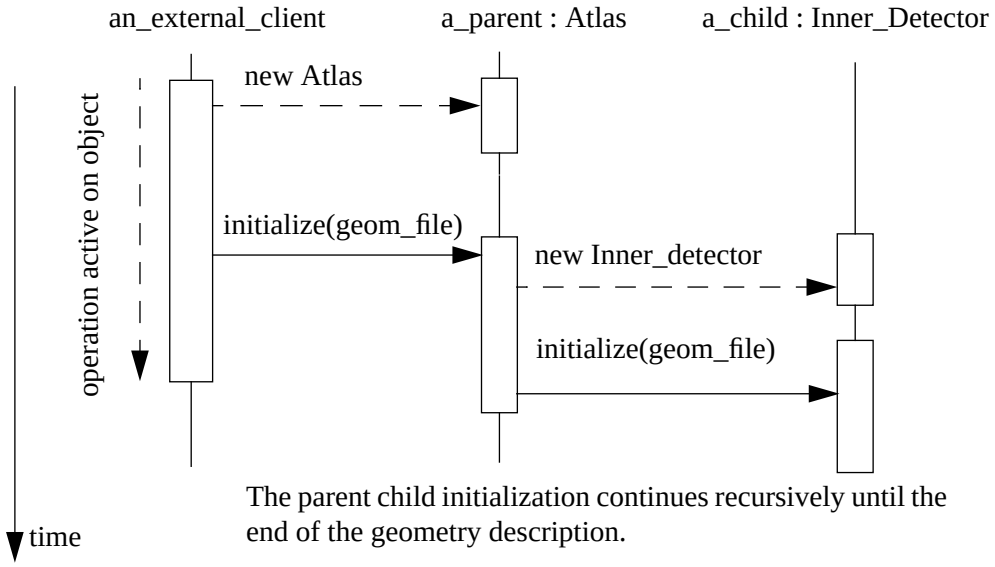


Figure 9: The *Initialize* interaction diagram

[5] An abstract operation is one whose calling signature is defined in the superclass and whose body is implemented in subclasses.

This diagram presents a sequence of messages between objects where each object is represented by a vertical line and the duration of an operation is shown as a box. The arrows between objects are the ‘messages’ or operation invocations, where creation is indicated by a dashed arrow.

Figure 9 presents only the first few top level interactions showing how an Atlas object is created, initialized and continues to initialize the rest of the hierarchy, driven by the description in the geom_file. Note that an external client is required to initiate the creation of the detector hierarchy. We leave this client unidentified for the moment.

6.2 Accessing detector data

With the detector hierarchy in place, we can look back at our first scenario:

1. ‘input’ the raw data produced by the different detectors

The first question we must ask ourselves is ‘who has data?’. In our simple model only the Detector_Leaf subclasses have data, that is the EM_calorimeter has various cells with energy and the Si_strip_detectors have position measurements or hits:

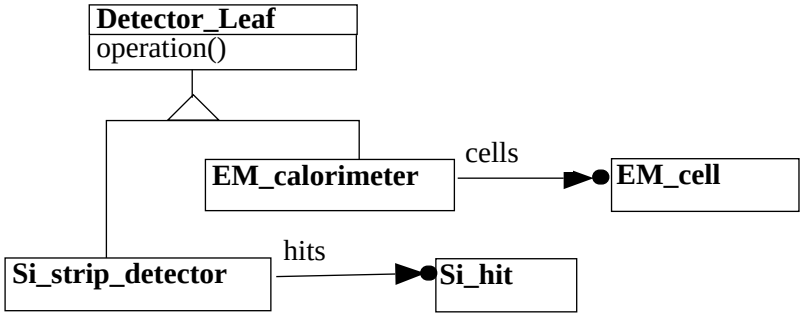


Figure 10: Class diagram of detectors with data

We need an ‘event structure’ to hold the raw data and eventually the results of the reconstruction. This we represent symbolically as an event object with ‘*raw data attached to it*’:

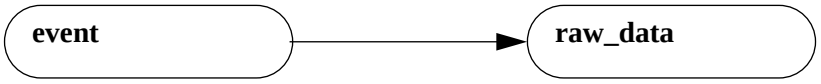


Figure 11: Simplified view of event raw data

So how does the detector hierarchy access its raw data? This can be done in a similar way to the *initialize* operation with a *trigger*(event) operation which pass an event through out the detector hierarchy:

- default operations in Detector_Leaf and Detector_Composite either do nothing or simply pass the event to the children.
- these operations are over-ridden in EM_calorimeter and Si_strip_detector. These specialized trigger operations must know how to ‘navigate’ from **event** to **raw_data** and extract their corresponding data.

An object interaction diagram for *trigger*(event) would be similar to Figure 9, where again an external client is responsible for looping over events, sending each one to the hierarchy.

7 Beginning the reconstruction

So who starts things off? Let's give **atlas** a *reconstruct* operation who first job, scenario 2, is to request all detectors in the hierarchy to '*preprocess*' their data (i.e. electronic calibration, initial cluster building, etc.):

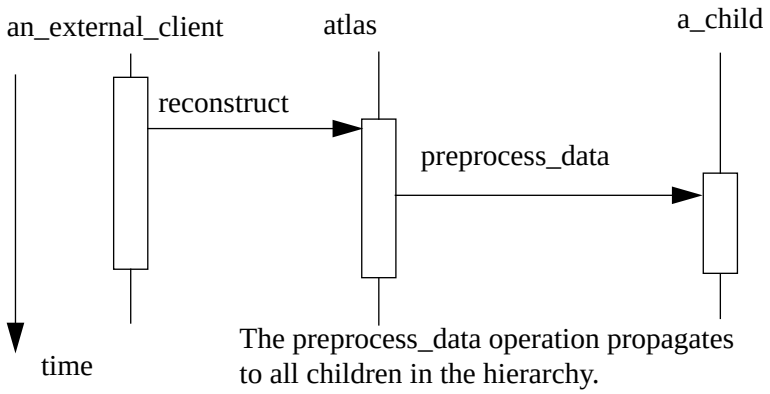


Figure 12: The *reconstruct* interaction diagram

Now recall the next two steps in the track reconstruction:

3. Generate roads from calorimeter clusters
4. Collect detectors and hits which are inside a road

We clearly need a Road class, and we could leave the responsibility for the rest of the track finding, from step 4 onwards, up to the inner_detector:

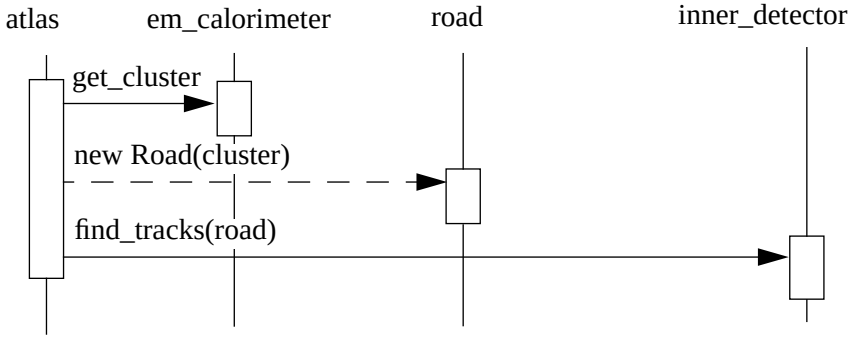


Figure 13: Continuation of Atlas *reconstruct* operation

7.1 Collecting detectors and hits

We want inner_detector to use a road to collect the subset of tracking detectors and their corresponding hits which fall inside the road. There are the following constraints on this subset:

- Want to be able to select further subsets while looking for track candidates, and
- Would like to preserve the 'structuring' of the hits which is provided by the detector hierarchy.

The solution we have chosen is to construct a **Tree** of **Node** objects, where each node refers (points) to a selected detector component or hit:

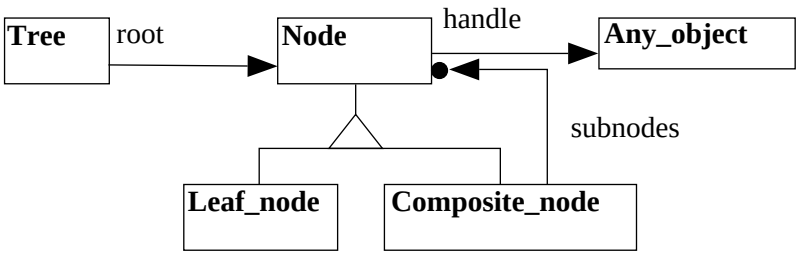


Figure 14: Class diagram of a Tree

Again we encounter our Composite pattern, where we have a hierarchy of nodes with a Tree object pointing to the root node. Here Any_object can be either a Detector_Component or a Si_hit.

In order to construct the set of nodes and the tree, inner_detector will:

1. create a tree and a root node
2. attach itself as an Any_object, and
3. pass the road and root to it's children through a build_tree method

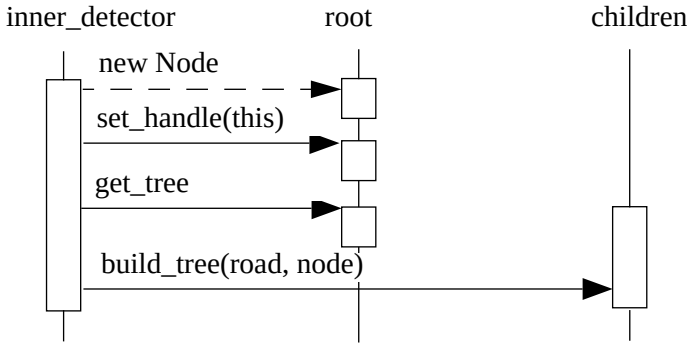


Figure 15: Build tree interaction diagram

After sending the build_tree method through the detector hierarchy, we end up with the following set of objects:

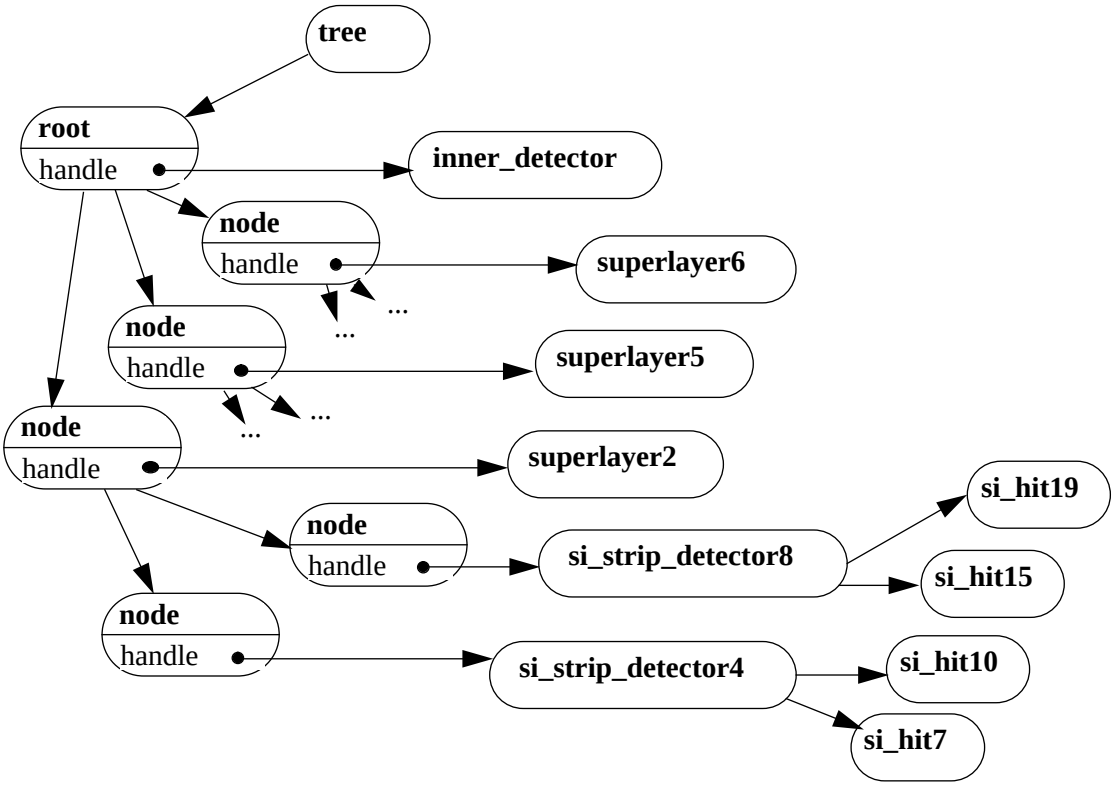


Figure 16: Object diagram of tree with detector components and hits attached to nodes

Figure 16 gives a typical snapshot of the collected detector components inside the road. The arrows pointing to ... in the figure indicate that there are nodes, si_strip_detectors and si_hits which have not been drawn.

8 The Visitor pattern

Now before we begin tracking with our tree, let's review the interface of Detector_Component:

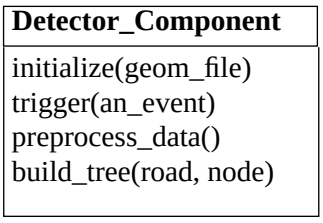


Figure 17: Class interface of Detector_Component

At this point, we can worry a bit about evolution of our model:

What is most likely to change in the future:

a) the **Detector_Component** hierarchy, or

b) the *operations* to be performed on the hierarchy?

Since the detector hierarchy closely resembles the hardware structure of the detector, which evolves fairly slowly, I would expect the operations to be more variable.

In this case, we can employ the VISITOR pattern:

We replace the four operations of Detector_Component by a single accept operation which takes as an argument a Visitor object:

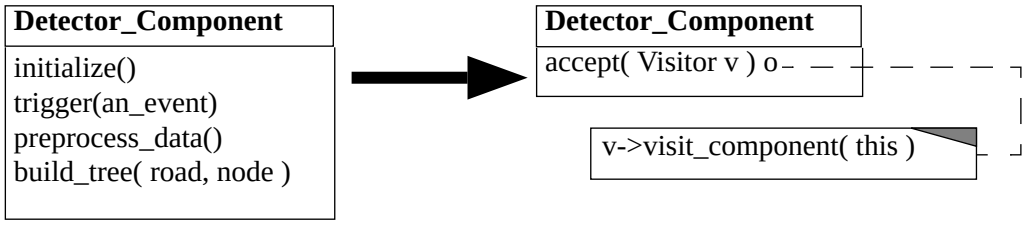


Figure 18: Simplification of the Detector_Component interface

In Figure 18, the pseudocode indicates that the **Visitor**, v, is simply requested to *visit_component* and a reference to the current object, *this*, is passed as argument. This is what is called a *call-back* mechanism.[6] Note that *visit_component* is a **specific** operation for the Detector_Component class.

The functionality of the removed operations is captured in different Visitor classes:

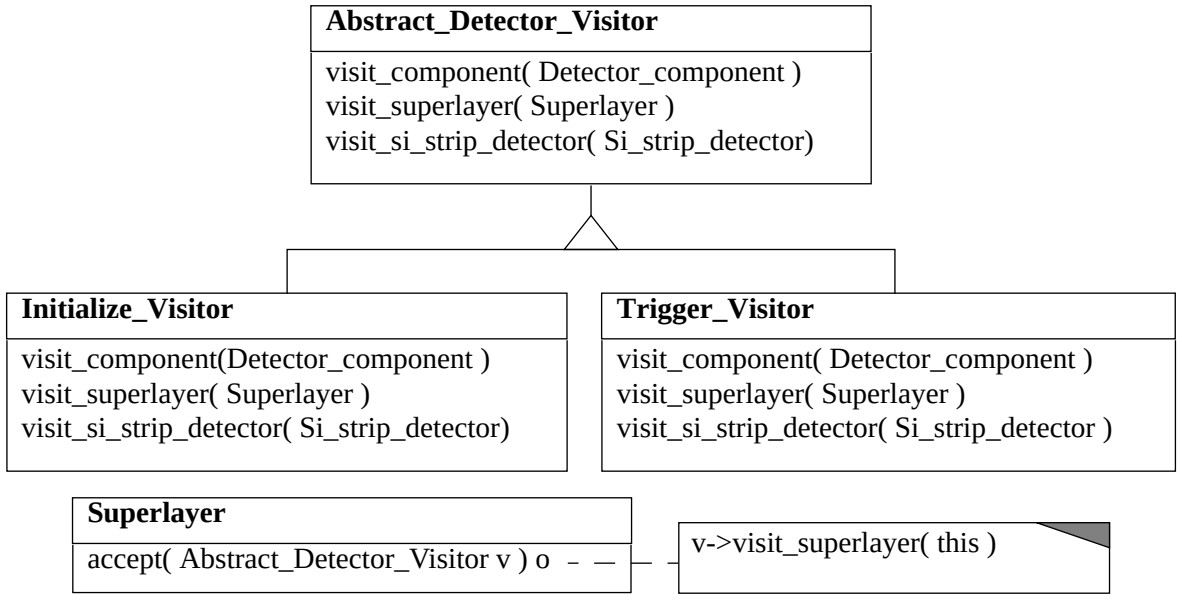


Figure 19: Class diagram of detector visitors and superlayer

With visitors the functionality of an operation, for example initialization, is grouped together into a single class. And each visitor must provide a specific operation for each of the classes which will *call-back* to the visitor. Of course default operations can be provided, e.g. in Abstract_Detector_Visitor. This pattern allows the adding or the changing of operations in an easy way, since the Detector_Component classes can accept any visitor which inherits from Abstract_Detector_Visitor. But as a consequence, when adding a new type of Detector_Component, each of the visitors must add a new method.

[6] For the more advanced readers, this mechanism is called *double-dispatch* where the *operation* which is executed depends **not only** on the type of the object receiving the request, but also on the type of the object sending it.

9 Track finding, cont.

We will now introduce a few more classes to schematically outline how the rest of the track finding will proceed. This will lead us to introduce our final pattern: Strategy.

After the detectors and hits have been collected in a tree, we allow Inner_detector to delegate to a Track_finder object to extract hits from the tree and generate the various Track_candidates:

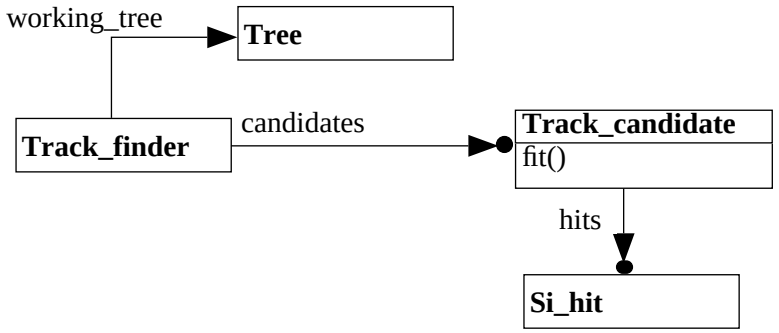


Figure 20: Track finder extracts hits from a tree to create track candidates

Clearly, Inner_detector will have to pass to the Track_finder the Tree and the Road. We can then implement the Visitor pattern for our tree hierarchy so that a set of Tracking_tree_visitors can be used to collect the hits, etc. The overall pattern recognition algorithm is implemented in the methods of our Track_finder.

9.1 Introducing the Strategy pattern

Suppose that the Track_finder wants to request its candidates to fit their hits with different fit methods. How does one avoid putting the multiple fit methods into the Track_candidate's interface? One solution is the strategy pattern:

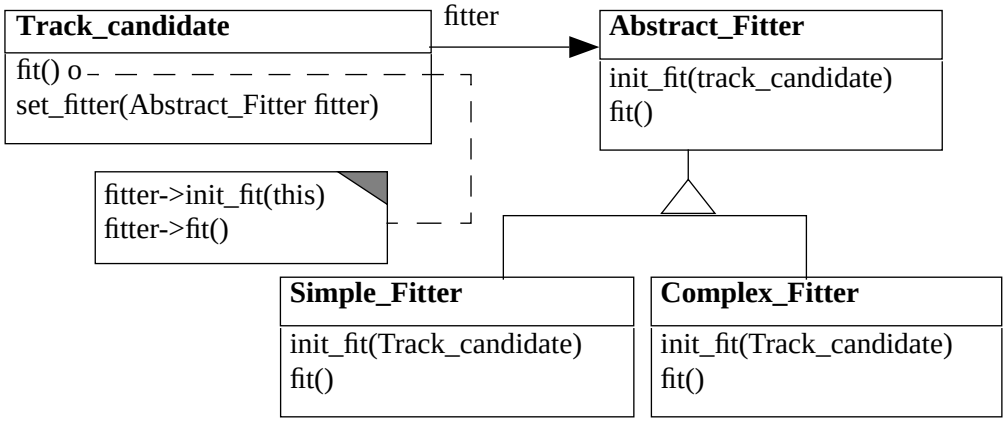


Figure 21: Track_candidate using a Fitter strategy

The Strategy pattern allows one to vary transparently the way something is done. In the present situation, the Track_finder may want to begin with a simple fit algorithm and to progress to a more sophisticated one. Thus Track_finder creates the fitter of interest and passes it to the Track_candidate through the *set_fitter*(Abstract_Fitter fitter) operation. When Track_finder later requests Track_candidate to fit(), the Track_candidate will use its reference to fitter. As can be seen in the pseudocode of the figure, the fitter is passed a reference to the current

Track_candidate, *fitter->init_fit*(this), which allows the fitter to access. i.e. call-back, the Track_candidate to get the needed information for
fitting.

The strategy pattern shows how one can cleanly encapsulate a variation: a Track_candidate only knows that it has some type of Abstract_Fitter, and the true fitter is determined by the client who is controlling the fitting action of the Track_candidate.

10 Summary and conclusions

Learning to design software from an object-oriented point of view can be lots of fun, although it may not be so easy to learn at first (at least for the 'older' generation).

A simple procedure which can help is:

1. State the problem
2. Generate a number of scenarios of things to be done
3. Walk through the scenarios to identify the participating classes and their responsibilities.
4. Use standard patterns, wherever possible, for object structures and behaviour.

It is extremely important to discuss with others while designing and to have others review your work.

References

[1] C. Mazza, et. al., Software Engineering Standards, Prentice Hall (1994).

[2] G. Booch, *Object Oriented Analysis and Design with Applications,* Benjamin/Cumming (1994).

[3] J. Rumbaugh, et al., *Object-Oriented Modelling and Design, Prentice Hall* (1991).

[4] The complete information about the Moose project can be found on the web: **http://www.cern.ch/OORD/Home_oord.html**

[5] E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley (1995).