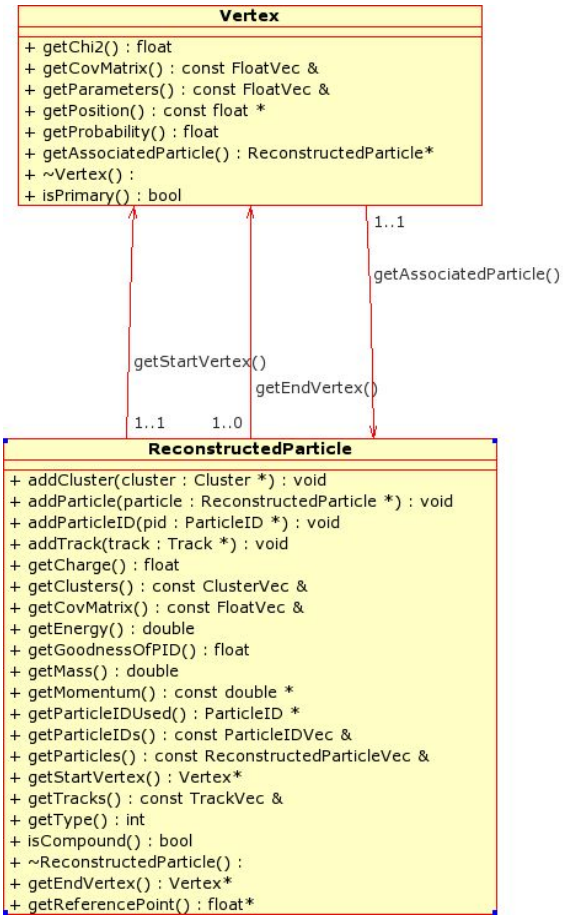
Vertices in LCIO

F.Gaede, DESY for the LCIO group, 05.07.2006

A recent request by the LCFI group to include a dedicated vertex class in LCIO has led to some discussion[1]. Originally the idea was to store vertices in LCIO in collections of *ReconstructedParticle* objects, interpreting a vertex as part of a particle. Even though this is a valid and consistent approach it is somewhat abstract and a lot of people regard a vertex as an entity in its own right that should be used to create a reconstructed particle.
The aim of this document is to make a proposal that unifies the different views without compromising the consistency and design philosophy of the LCIO event data model.

In general a vertex is a reconstructed point inside the detector where a particle decays into N decay products. One simple example for a vertex is the decay $K_s^0 \rightarrow \pi^+ \pi^-$. There are actually two vertices associated with the K_s^0 , one is the IP where the particle is created and the second is the endpoint, i.e. the point where the two pions are created. Conversely every vertex is typically associated with two sets of particles, the parent(s) and the daughters of the interaction of which the vertex is the origin and endpoint respectively.
The classes shown below reflect this association between vertices and particles:



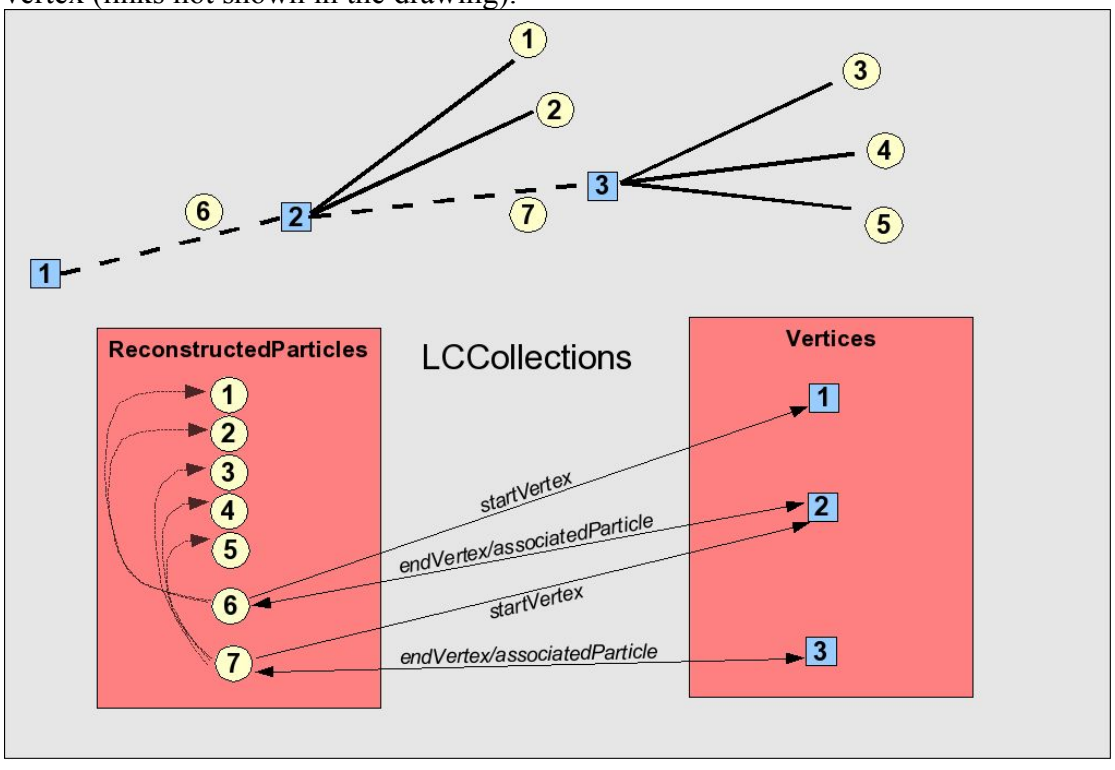
1 See posting „Vertex class“ in LCIO forum at http://forum.linearcollider.org/index.php?t=threadt&frm_id=5

The *Vertex* class holds all information that defines the position, its error, the quality of the fit as well as an arbitrary set of parameters related to the fit. Also there is exactly one *ReconstructedParticle* associated to the vertex that holds the kinematic quantities such as the mass and momentum of the decayed particle. This particle then points to the vertex as its endpoint. In the example $K_s^0 \to \pi^+ \pi^-$ there will be one vertex object for the position of the decay that is pointed at by the pions as *startVertex* and there will be one *ReconstructedParticle* for the K_s^0 that is associated to the vertex and has the vertex as its *endVertex*.

This class design introduces a dedicated vertex class and adds to the existing *ReconstructedParticle* class two pointers to vertices – start and end vertex.
The *referencePoint* is kept as this defines the point where the kinematics of the particle is defined as long as no vertexing has been applied.

Example: Secondary Vertices in Heavy Flavor Jet

A more complex example for how to use the *Vertex* and *ReconstructedParticle* classes is the vertex finding in heavy flavor jets. The drawing below shows a heavy flavor jet with five particles labeled 1-5. After reconstruction one has a collection "ReconstructedParticles" that holds these five particles and a collection "Vertices" that hold one vertex object for the IP (there always has to be at least this one vertex in the event). All particles will have this primary vertex as *startVertex*.
Assuming that the vertex finding algorithm has found two additional vertices these will be added to the "Vertices" collection (2,3) and at the same time two associated particles (6,7) will be created and added to the "ReconstructedParticles" collection. These particles hold the list of particles that belong to the vertex (their decay products) and the decay particles' *startVertex* will be updated to be the newly found vertex (links not shown in the drawing).



Discussion of proposed classes

1. The event data model in LCIO incorporated a well defined hierarchy with the *ReconstructedParticle* as the top level object from which pointers relate back to lower level objects like *Tracks*, *Clusters* and from there to hits and Monte Carlo truth objects. This will be changed with the introduction of the *Vertex* class that points to *ReconstructedParticle* and is pointed at by *ReconstructedParticle* at the same time. However this reflects the iterative nature of vertex reconstruction where particles are needed as input to the vertexing and then the particle information (origin/kinematics) are updated making use of the vertex constraint.

2. The introduction of a vertex for every reconstructed particle together with the *isPrimary*() method allows to conveniently loop over the reconstructed particles without double counting. One simply requires the particles to originate from the primary vertex (IP) in the event:

```
ReconstructedParticle* recP = ... ;

if(  recP->getStartVertex()->isPrimary() ) {

        /* do sth. */
}
```

   Note that it is required that exactly one primary vertex exists in the event. This will have no associated particle.

3. The proposed scheme of having two classes for vertices and reconstructed particles requires that the event is updated in a consistent way when a vertex finding procedure is applied. This requires that a mechanism exists in the reconstruction framework to update/modify the LCIO event that has been read from file. This is currently not the case for the Marlin framework. Until such an update mechanism is introduced one has to create the "ReconstructedParticles" collection in the same job before running the vertexing processor.

4. It has been suggested by the LCFI group that vertices should have a method that allows to create a linked list of vertices, e.g. to chain the vertices found in one jet. In the proposed vertex class design these links are implicitly included through the associated particle and their links to the constituent particles (the decay products). However we could introduce an additional link in the *Vertex* class that allows to set and follow this link directly, such as *nextVertex*() and *previousVertex*(). Another mechanism to relate vertices to each other of course exists through the *LCRelation* objects.

5. The *getParameters*() method in the *Vertex* class allows to store arbitrary parameters with the vertex. These might be different for every algorithm, e.g. The ZVTop code might provide parameters that can be used for jet flavor tagging. The meaning of the parameters will be encoded in a collection parameter. As planned for a while now LCIO should provide some code that allows to conveniently access such decoded additional user information.

6. The *getReferencePoint*() method of *ReconstructedParticle* is kept to hold the point where the kinematics of the particle is defined. The question is, whether one needs

errors of this point or even a full covariance matrix (including the 4-momentum) ?

7. What should be the behavior of LCIO when reading old files that have no vertices ? Probably it would be good to simply add null pointers to the vertices so that code relying on vertex information will fail – possibly one could also throw an Exception if the Vertex information is accessed in an old LCIO event.

8. It might be a good idea and convenient for the user to have an *LCEvent::getPrimaryVertex*() method.

9. The proposed *endVertex*() method in *ReconstructedParticle* has been introduced for user convenience as it is typically the end (decay) of the particle that is associated with the reconstructed vertex – as described in the simple K_s^0 decay above. However the information is of course redundant with the *startVertex*() of all decay particles. So we might want to choose to implement this as a convenience method that actually returns the *startVertex*() of the (first) decay particle. This would then also enforce the consistent update of the *startVertex* of the particles that have been used in the vertex fitting.