Opal Element Layout

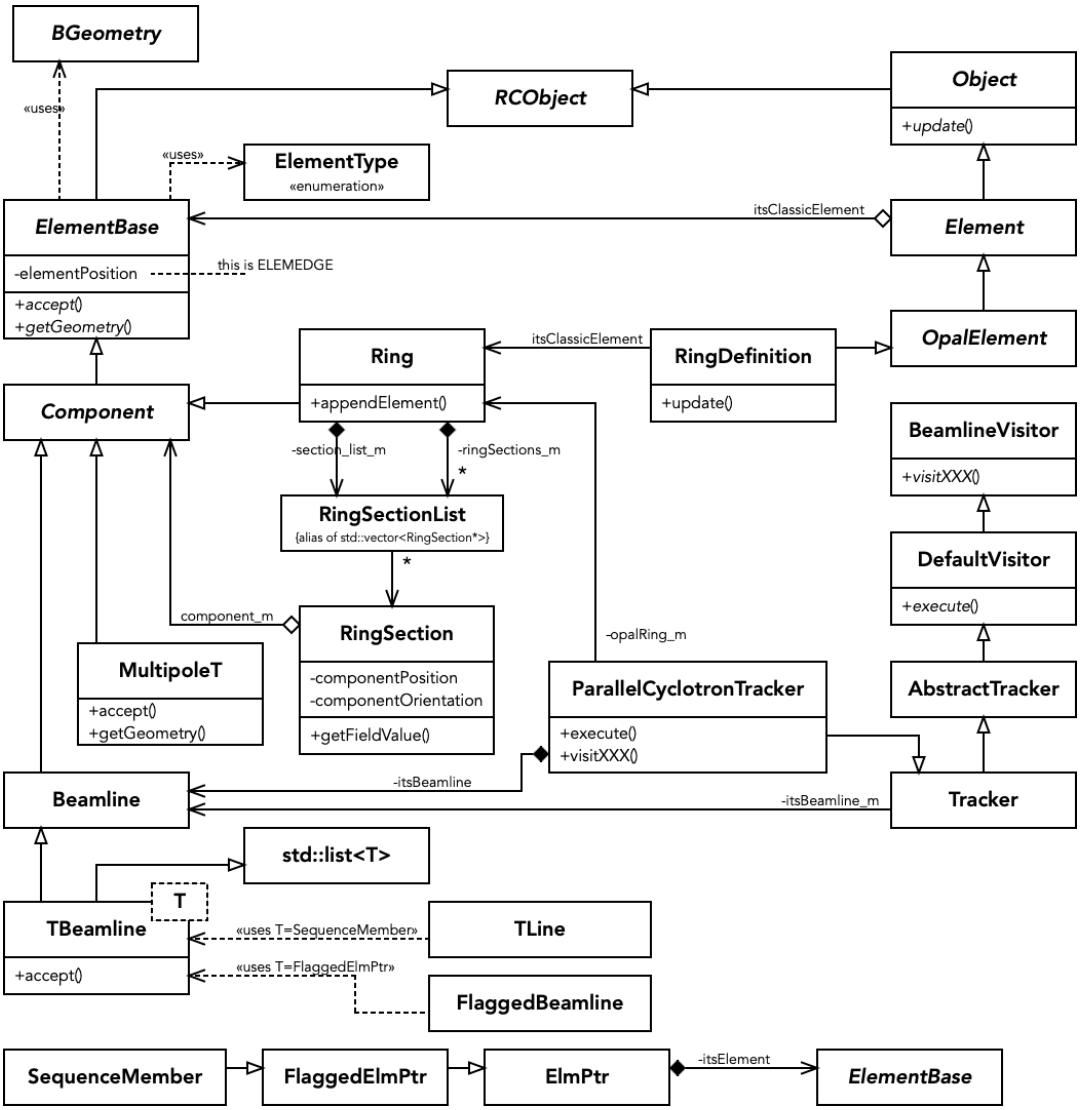
Old Opal supported three distinct element layout schemes, two of which were valid for Opal-t and the third was for Opal-cyclotron. As part of the Opal-X rewrite, all schemes should be valid for all elements at all times.

Old opal

This note firstly explores what currently exists in old Opal and how it is implemented, before proceeding to proposing a set of changes to be made for Opal-X.

What exists in Opal-cyclotron

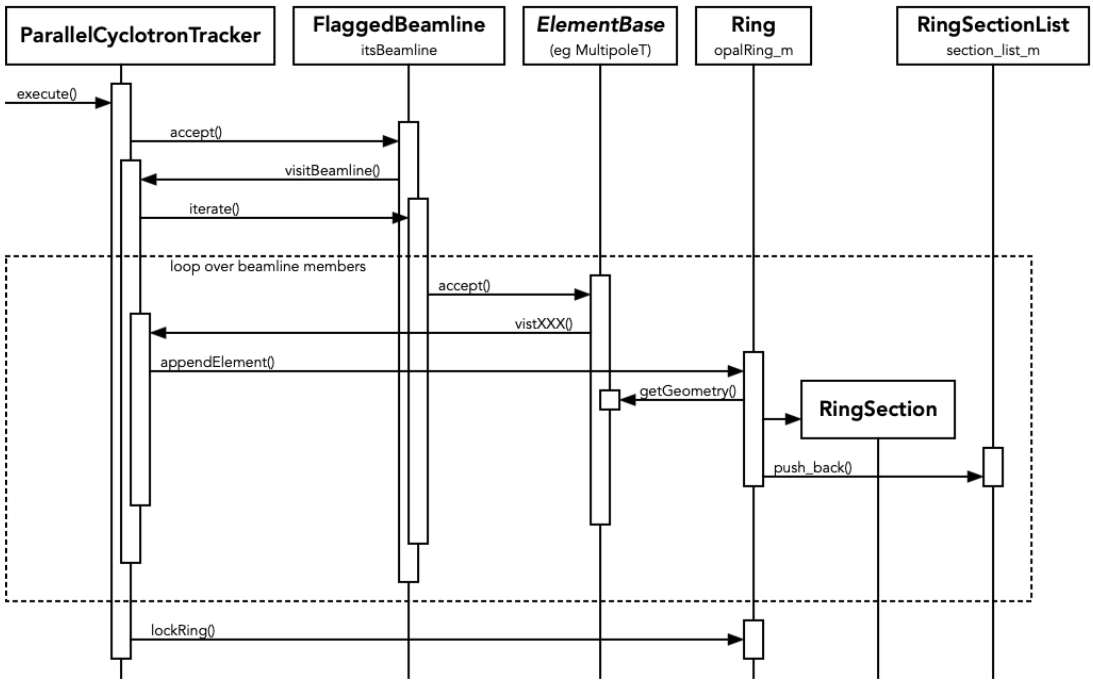
The implementation of the layout scheme provided by Opal-cyclotron is the simplest to understand so we shall start with that. A class diagram is below.



The scheme is a 'Lego block' style layout system, which can be described as relative longitudinal. Starting from a position and orientation defined by the RINGDEFINITION element, elements are laid out beginning to end along the longitudinal (or S) coordinate of the design

orbit. There are four different offset elements that can be used to provide gaps (i.e. Drifts) or other requirements for jumps in the layout.

The Ring class (the component that corresponds to the RINGDEFINITION element in Opal-cyclotron) calls the element's getGeometry() in its appendElement function. This uses the geometry's getTotalTransform() function to get a coordinate transform object that is then used to define the end position of the element, which becomes the next start position. For each element that is added, a RingSection object is created and given a copy of the added element along with start and end positions and normals. Here's the call tree sketch for the layout related part of the ParallelCyclotron::execute() function:

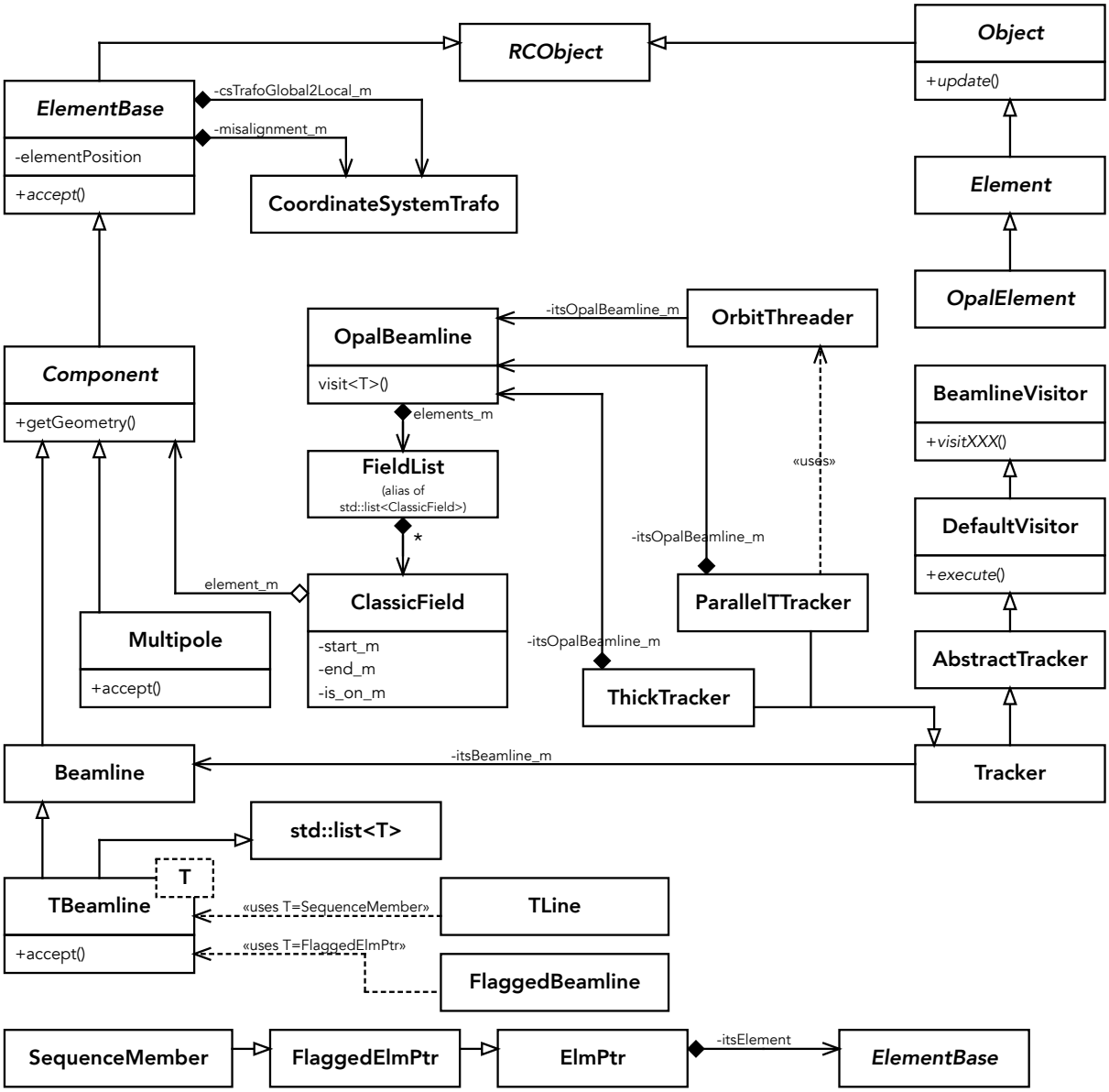


A few observations.

- The difference between Element and ElementBase is not really reflected in the names. The Element class is a base class for the user interface classes. Is it actually needed or can OpalElement perform this function alone?
- TBeamline is a template and has std::list<T> as a base class. It is not recommended to use std::list (or the other container templates) as base classes as their destructors are not virtual. Is TBeamline really a kind of std::list or is TBeamline has a std::list a better way of thinking of it?
- The whole Beamline, TBeamline<T>, TLine, and FlaggedBeamline hierarchy is quite complex. Is it necessary?
- This layout mechanism ignores the LINE statement's ORIGIN and ORIENTATION attributes and uses RINGDEFINITION instead.
- Any global layout information the elements may have been given (X, Y, Z, THETA, PHI, PSI attributes) is ignored. Ignoring PSI means that this scheme as implemented cannot bend the beam out of the horizontal plane.
- The ELEMEDGE attribute of elements is not used.

What exists in Opal-t

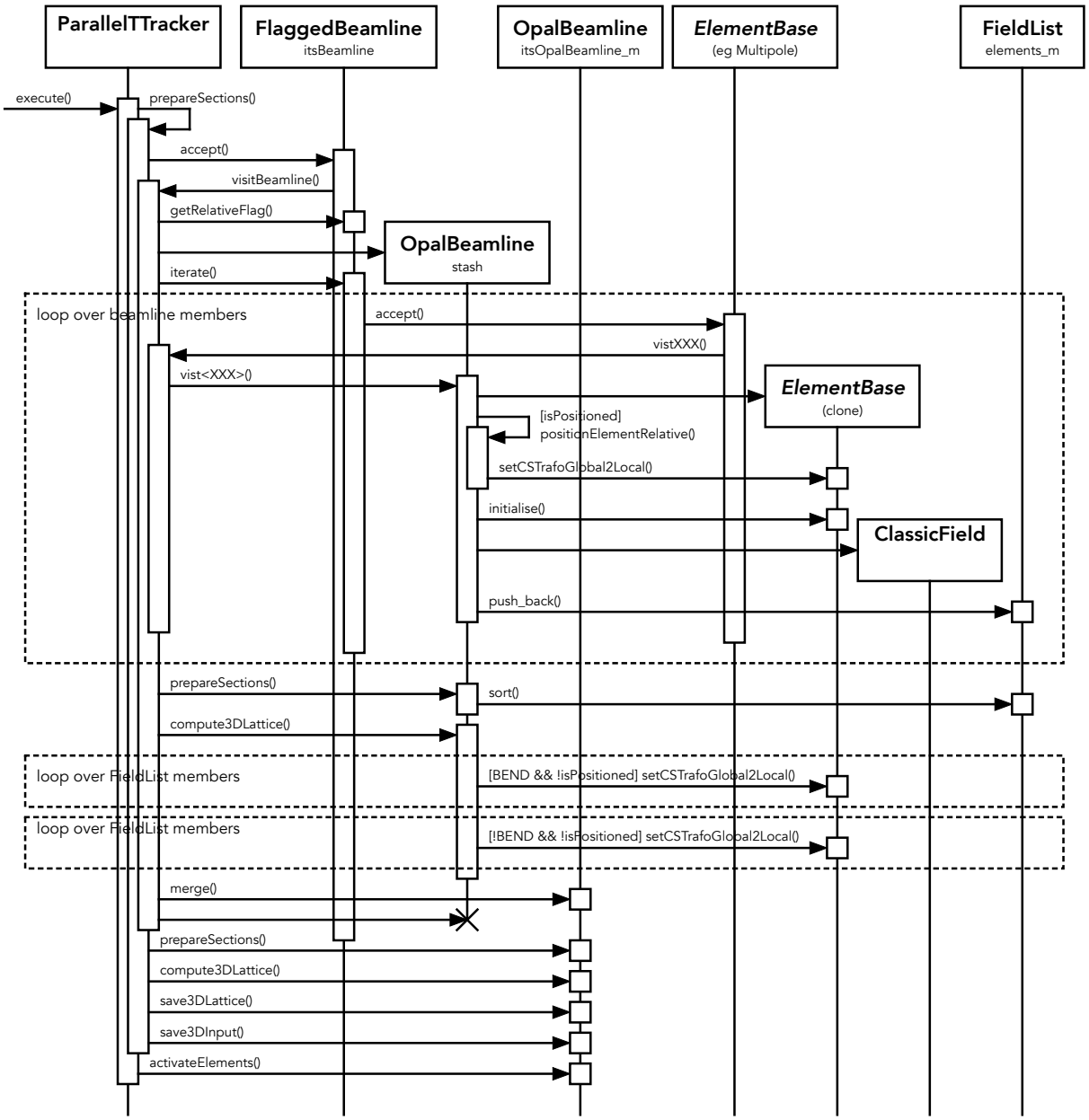
Opal-t supports two layout schemes; beamline relative positioning and a method that can be described as offset longitudinal. A class diagram is below.



Of these two, the beamline relative positioning scheme is the easiest to understand. It is triggered for an element whenever one of the ORIGIN, ORIENTATION, X, Y, Z, THETA or PHI (but not PSI) attributes is provided. The user interface base class function OpalElement::update() performs the necessary functions, setting a coordinate transform object and marking the element's position as fixed (to control the interaction with the main layout code, of which more later). The handling of PSI is different as it is also used in the offset longitudinal scheme with bending elements to rotate the beam out of the horizontal plane.

The offset longitudinal scheme uses the ELEMEDGE attribute to define the longitudinal offset (the s coordinate) from the reference position and orientation defined by the LINE statement. The orientation of the element is the end orientation of the previous element (or the orientation of the LINE if this is the first element). A nested LINE statement that provides position and

orientation attributes overrides the current position and orientation for its elements (indicated by a flag being set).



The layout action begins with the call to the ParallelTTracker::excute() function, as shown above. After the visitor pattern accept/visit sequence there is some code that stashes the itsOpalBeamline_m and replaces it with a new one if the LINE specified a position and orientation. This implements the nested LINE functionality mentioned previously.

The members of this LINE are now iterated through using the visitor pattern. Elements are cloned. Those that are marked as already positioned (by the beamline relative positioning) have their coordinate transform object adjusted by the position and orientation specified by the LINE statement. All the cloned elements are then added to the FieldList. This is then sorted by their ELEMEDGE value (or 0 if no ELEMEDGE specified) with alphabetical order as the fallback.

Next the OpalBeamline::compute3DLattice function is called. This is quite a complex function that consists of two loops each with initialisation. The sketch above has hidden most of the complexity. Each loop keeps track of a prior path length and a current coordinate transform. The first loop handles SBEND, RBEND and RBEND3D elements that have not already been positioned. Various attributes of these elements (including PSI) are used to create a coordinate system transform object that is given to the element and defines its local coordinate system. The second loop then handles all the non SBEND, RBEND and RBEND3D elements assigning their coordinate system transform objects. It also sets the positioned flag on all the elements.

Some observations

- Why are the bend elements split out into a separate loop in the compute3DLattice function?
- The bend elements are detected by explicit tests of the element type enumeration which then controls the functionality applied. The Multipole element can cause bends but it is not detected by this code. In general, tests like this should be avoided to make it easier to add elements.
- Nested LINE statements will cause recursive calls to the ParallelTTracker::visitBeamline function. This will in turn cause elements to be scanned by OpalBeamline::compute3DLattice more than once. One purpose of the positioned flag is to avoid this causing problems.

The TBeamline<T> template

The FlaggedBeamline instantiation of the TBeamline<T> template is used to represent the LINE statements in the input file. One of its base classes is std::list<T> template which is used to store the list of elements making up the beamline.

Using the standard template library container classes as base classes is advised against due to their destructors not being virtual, which would lead to incomplete destruction of the derived class if it were deleted using a pointer to the std::list<T> base class. As currently used in Opal there's no problem, it is still regarded as generally unsafe though.

There is also a conceptual argument. Is a beamline better thought of as containing a list of elements rather than being a kind of list of elements?

For safety, the std::list<T> should be moved from being a base class to being a member.

The visitor pattern

Iteration through the list of elements stored in the OpalBeamline object is currently performed using the visitor pattern. The aim of this pattern is to allow the easy extension of the operations you wish to perform on the list of elements without having to modify all the elements. It does this by defining an abstract interface with a visit function for each element type. Objects derived from this type are then used by the single iterate function to apply an operation to the elements through the accept/visit sequence. This separates the operation code for each element type into a single class.

The pattern is useful if the set of operations is expected to grow with time. The downside is that it makes it harder to add element types. It is ideal in the situation where there are a small

number of element types and a large and growing set of operations. However, in this case we have the exact opposite, a large and growing number of element types (the current count is around 33, many of which have been added over time) but only 5 or 6 operations. It is designed for operation extensibility, but we require type extensibility.

We have ended up with a very wide abstract visitor API that has to be modified with further virtual functions for each new element. The classes derived from the visitor API have god-like control over a large number of element types; a severe breakdown of the normal object-oriented encapsulation principles. This pattern is inappropriate here and should be removed.

Proposal

Layout Schemes

The following three layout schemes will be supported:

1. Offset longitudinal (s coordinate). Elements are placed with their start positions in the longitudinal direction defined by the ELEMEDGE value which is relative to the LINE's start position. Bend elements affect the orientation of following elements. The orientation of the bend axis is defined by the PSI attribute. Drift elements are optional. Elements are easy to overlap.
2. Relative longitudinal (s coordinate). A 'Lego block' scheme where elements are laid out one after the other with the next element's start position and orientation set to the previous element's end position and orientation. The L, (LENGTH) attribute and the element's geometry define an element's end position relative to its start. Drift elements are required. Elements may be overlapped by using nested LINE statements.
3. Beamline offset positioning. The position and orientation are defined by the X, Y, Z, THETA, PHI, PSI attributes which are interpreted as offsets from the current LINE statement's start position and orientation. Drift elements are optional, and elements are easy to overlap.

Other Aims

The implementation rework will also address these aims:

- Support the three layout schemes in a unified manner. In old Opal the implementation is scattered, beamline offset is partially handled by the user interface objects and partially by OpalBeamline, offset longitudinal by OpalBeamline and relative longitudinal by Ring.
- Remove the visitor pattern from the layout implementation. This will have the side effect of simplifying the tracker inheritance tree.
- Remove the numerous other places where encapsulation is violated. Detection of the bend elements by testing the element type against RBEND, SBEND, RBEND3D is quite common. There is also a test for SOURCE in one place.
- Handle nested LINE statements in a better manner than stashing whole OpalBeamline objects. A stack containing just the necessary position and orientation information would be sufficient and avoids the merge process currently required.

- Old Opal iterates through a LINE's elements three times to perform the offset longitudinal layout scheme. It is not clear why this is necessary. Reduce the number of iterations.
- Remove the std::list from the base class list of Beamline<T>, reimplement as a member.
- Rationalise the Beamline class hierarchy.

Input file syntax modifications

The LINE statement is used to group elements together. Lines may contain other lines. The same element (or line) may be included more than once. This is the same behaviour as old Opal.

The existing ORIGIN and ORIENTATION (or the X,Y,Z,THETA,PHI,PSI) attributes specify in global coordinates the start position and orientation of the first element in the line. In the top-level line, these default to zero, in nested lines the default is the enclosing LINE's values.

Add CLOSED and CLOSED_TOLERANCE attributes to the LINE statement. The presence of CLOSED=TRUE introduces a test of the layout of the line statement: the global coordinates of the end of the last element must match the start of the first element to within the CLOSED_TOLERANCE value (with a suitable default).

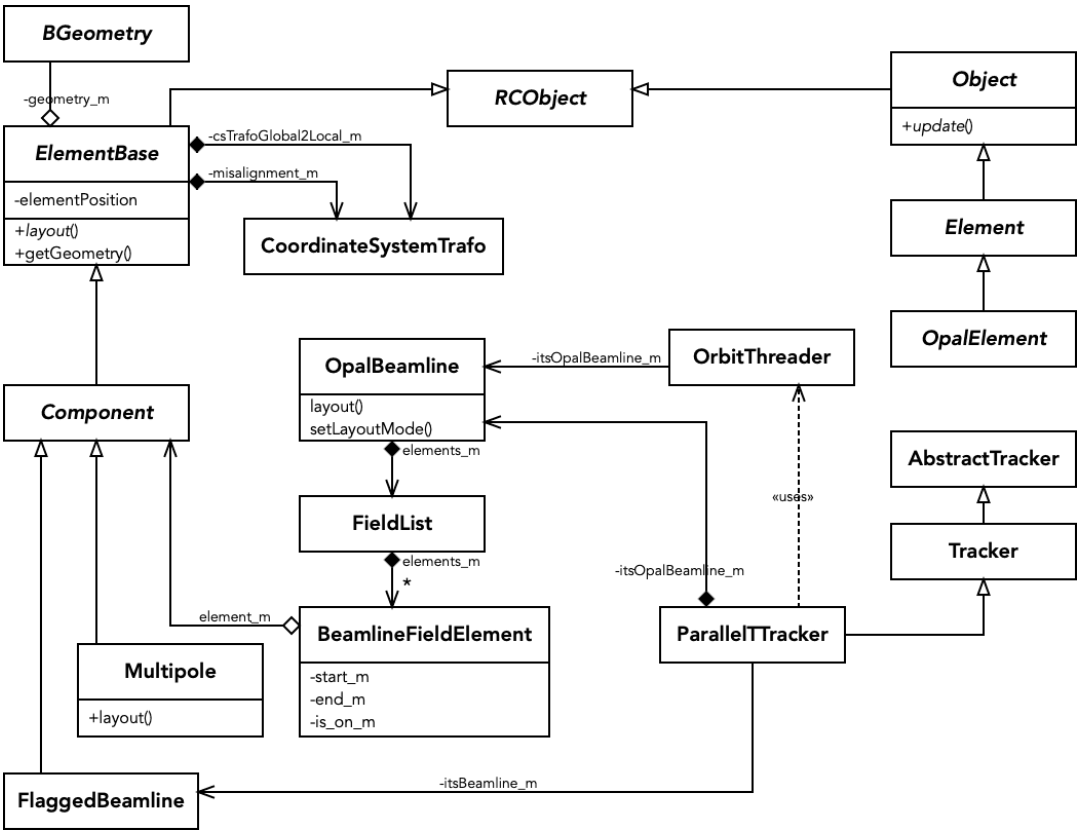
Add a LAYOUT_MODE attribute to the LINE statement that can have one of these three values:

1. OFFSET_S. Switches to the offset longitudinal scheme using ELEMEDGE. Drift elements may be used to define the aperture but are not required otherwise.
2. RELATIVE_S. Switches to the relative longitudinal scheme (does not use ELEMEGDE). Drift elements are required to separate elements.
3. BEAMLINE_OFFSET. Switches to the beamline offset coordinate layout scheme using X, Y, Z, THETA, PHI, PSI. Drift elements may be used to define the aperture but are not required otherwise.

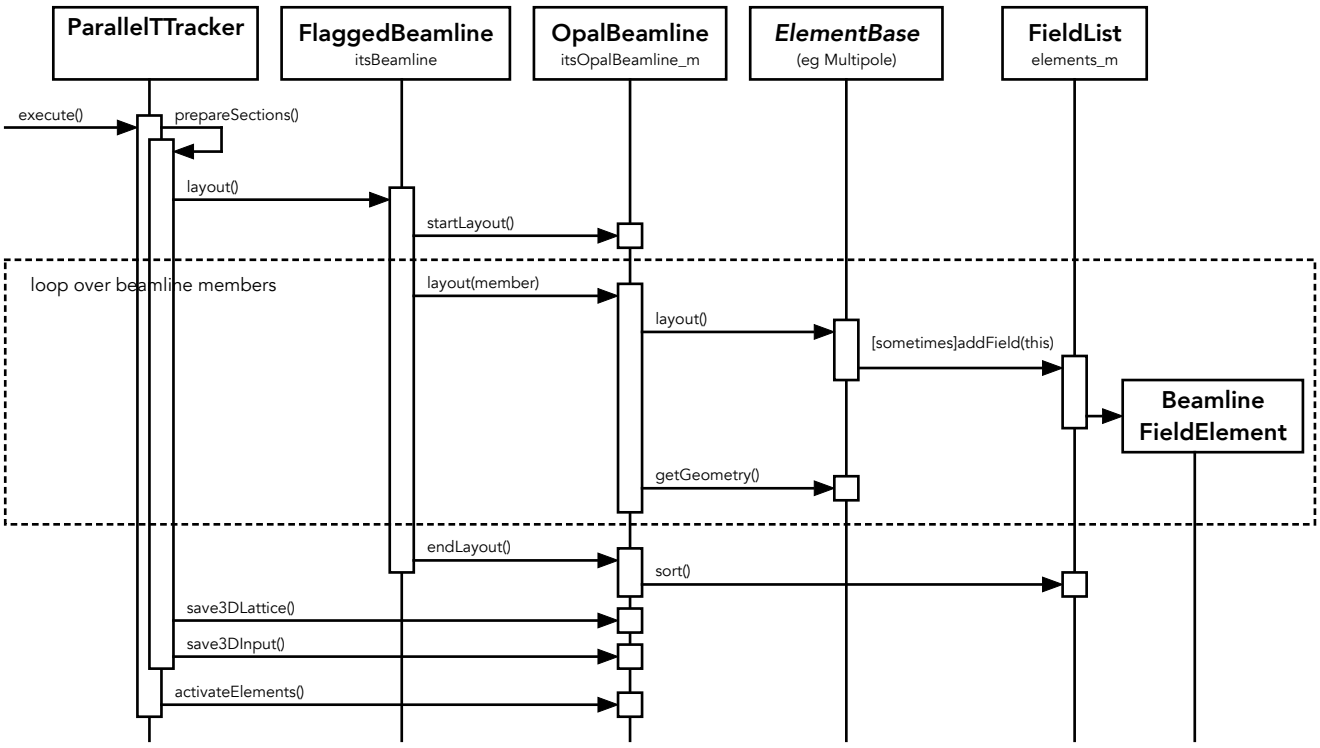
Remove the RINGDEFINITION element.

Implementation sketch

The proposed class structure is shown below.



It is based closely on the class structure associated with the original ParallelTTracker but it will be operated in a fashion that combines the functionality of both ParallelTTracker and ParallelCyclotronTracker. The layout operation should look like this:



It is not clear why there needs to be three loops over all the elements in old Opal, so in this version there is just a single loop that is not implemented using the visitor pattern. The

FlaggedBeamline object (which corresponds to the top level LINE statement) starts the layout by initialising OpalBeamline's current position and orientation. It then iterates through the elements asking OpalBeamline to lay it out element by calling the layout virtual function on it. Many of the elements will represent some kind of field and will add themselves to the FieldList. The element's geometry object is then used to determine the element's end position which is used to update the beamline's current position ready for the next element.

A nested LIST statement is represented by an instance of FlaggedBeamline appearing in the element list. This will cause a recursive call to its layout() function. The OpalBeamline::startLayout and endLayout functions will handle the save and restore of position and orientation information for the nesting.

The LINE's layout mode controls how the OpalBeamline object calculates the coordinate transform object that is given to each element to define its start position. The layout logic for the beamline relative mode will be moved to here from the user interface objects. The user interface objects will only be responsible for validating attributes and passing them to the elements, all the layout logic will be in the OpalBeamline.

The elements are responsible for adding themselves to the field list and performing any other type specific operations that may be necessary. This is where any special functionality in the removed visit functions will be moved to.

The geometry object is used to define how the end position relates to the start position; most elements will use a simple linear geometry but bends with use one of various arc geometries. Markers and other zero length elements will use a null geometry object.

There is an argument for a change to the OpalBeamline class name. This class is not a user interface class and it is not derived from OpalElement. Probably the only reason for the Opal prefix is to distinguish it from the Beamline class. Maybe it should be called BeamlineLayout or something like that instead?

I'm sure this will turn out to be a little over simple, but that's the concept.