THE ELETTRA OBJECT-ORIENTED FRAMEWORK FOR HIGH LEVEL SOFTWARE DEVELOPMENT

C. Scafuri, Sincrotrone Trieste

Abstract

The new framework developed at ELETTRA is used to build model based accelerator control programs. The framework is fully object-oriented and has been designed following the UML methodology. The functions and responsibilities of the system are distributed among different modules, forming a hierarchy of layers of increasing level of abstraction. The system is fully distributed and is based on the CORBA distributed object model. All the design is based on accepted standards, so that it can be ported to different platforms. The parts of the software depending on specific features of the underlying environment are encapsulated and hidden by means of wrapper objects. The full architecture and design processes are presented in detail.

1 DEFINITION OF TERMS AND GOALS

A framework is “a set of co-operating classes that make up a reusable design for a specific class of software” [1]. It is not a closed solution, but rather a set of Object-Oriented (O-O) tools that can be used for writing programs dealing with some specific problem domain. It is evident that a framework must be based on a conceptual model of the problem domain. On the other hand it will impose its conventions to the programs based on its classes.

With High Level Software (HLS), we mean the programs used to monitor and control the beam dynamics view of the accelerator, transfer line or storage ring. That is, programs used to calculate and modify the accelerator physics parameters of the machine: for example tunes, Twiss parameters, transfer matrixes etc. HLS programs are also known as model based control programs. In this category of programs the actual readings obtained from the control system of the machine - usually available in engineering units - are transformed into quantities relevant to the accelerator physicist by means of a mathematical model of the accelerator. Model based controls are an established tool for accelerators [2][3][4] and already used at ELETTRA [5].

With this framework we want to give our physicists the means to write HLS programs exploiting O-O programming techniques, languages (C++) and tools from start to end. The unrestricted use of O-O programming techniques will reduce the development and maintenance costs of our HLS programs. The O-O programming approach will also help in case of developments of the underlying control system and, most important, for following the continuos improvements and developments of our accelerators.

Among the other HLS programs, we foresee an online model of the accelerator. The online model will be used as a machine physics server and will be accessed remotely by means of the CORBA technology. These requirements give us constraints and guidelines for some of the architectural choices.

Although HLS programs are also used for simulations, they are usually limited to the linearised or first order model [6] of the beam dynamics, leaving higher order models to other specialised tools [7]. The first release of the HLS framework directly supports the linearised model.

The framework does not pretend to be usable for writing machine design programs or to be applicable to any type of particle accelerator. We assume that we are dealing with electron accelerators of a synchrotron radiation facility.

2 DESIGN OF THE OBJECT MODEL

2.1 General Considerations

In order to design a good framework we must formulate an object model describing the most important aspects of the domain and supporting the most important and frequent operations needed by client programs. Since the framework is not the final program, it will not address all the possible algorithms or calculations which may be needed. It will provide the tools to write those algorithms and calculations. An important criterion we follow is that the HLS framework is not meant to be a general-purpose interface for controlling the accelerator. The framework will be concerned with the quantities directly related to the beam dynamics description; its interface to the general control system will thus be concerned only with these quantities. It will make no provision for fault diagnostics, start up procedures and other similar tasks.

The first step of the design of the object model, writing the requirements and specifications, is one of the hardest and most critical for the entire project. In order to manage this phase we adopted, although rather informally, the Use Case methodology [8][9][10] which is part of the Unified Modeling Language-Rational Unified Process (UML/RUP) methodology [13][14].

2.2 Use Cases

The Use Cases are the result of a set of interviews with beam dynamics and machine operations experts of our institute. The Use Cases must cover a typical and comprehensive set of the operations that will be performed by HLS programs. As a starting point to figure out the scenario of the framework, we have selected the following Use Cases:

- Calculate the machine optics parameters
- Get the actual machine physics parameters of a set of "elements"
- Transition between two different machine optics
- Scale the machine setting for a new beam energy
- Calculate various sensitivity matrixes to be used by correction algorithms
- Check the effects of changing some parameters on a simulated machine (do a what-if analysis)
- Check the injection conditions by tracking a group of particles for some turns in the storage ring

Although simple, these Use Cases sum up some of the most typical tasks of HLS programs used in our laboratory.

2.3 General Concepts and Ideas

From the analysis of the use cases and from the literature about accelerator beam optics [7][11][12], we derived a group of ideas or concepts for the design the framework.

The particle beam is guided and shaped by a series of "elements" acting on it by means of electromagnetic fields. These elements are laid down along the ideal beam path, which is formed by straight lines and arcs of circumference corresponding to bending magnets. A point of the "beam" is determined by its abscissa $\underline{s}$. The absolute position and orientation of an element can be obtained by traversing the various elements and accumulating displacements and rotations. We assume that the elements are laid down with accuracy, but it should be possible to take into account some small misalignments (survey errors). Each electromagnetic element is characterised by a set of physical (e.g. **B**, *l*, and *n* for a combined function magnet) and geometrical characteristics. There can be some constraints on the freedom to choose the working point of some of the elements, the typical case is that of a string of magnets connected to the same power supply. In short, the framework must offer the tools to assemble the model of the machine. The bare model of the machine gives its geometrical description (fig.1).

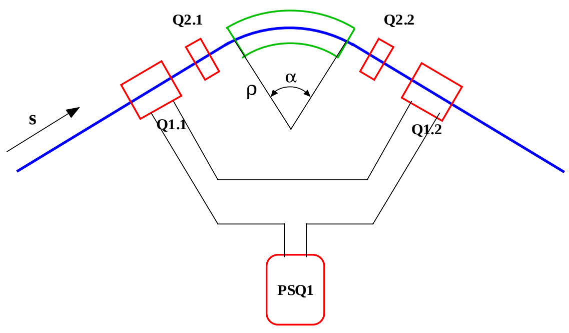

Figure 1. Bare machine model.

The readings of real or simulated control system variables are transformed by means of some algorithm into pertinent physical variables and then into "optical" parameters (e.g. focal length for a quadrupole) for each electromagnetic element. Calculations are for the great majority performed following a rather repetitive schema: start with an initial value, step through each electromagnetic element of a given set and transform the value according to the characteristics of the element (fig. 2).

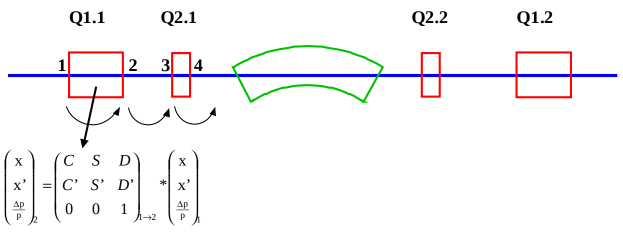
Q1.1
Q2.1
Q2.2
Q1.2
1
2
3
4
$\begin{pmatrix} x \\ x' \\ \frac{\Delta p}{p} \end{pmatrix}_2 = \begin{pmatrix} C & S & D \\ C' & S' & D' \\ 0 & 0 & 1 \end{pmatrix}_{1\to 2} * \begin{pmatrix} x \\ x' \\ \frac{\Delta p}{p} \end{pmatrix}_1$

Figure 2: typical calculation scheme

Client programs must have the possibility to use a "live" machine model, with the possibility of changing the machine settings, or a "model" machine without these capabilities.

As we foresee to write an on-line accelerator model server, we include also a set of CORBA interfaces in the framework in order to make it usable within a distributed object-programming environment.

2.4 The Object Model

From these ideas, we have derived the object model of the framework. The class hierarchy is divided into three main areas: electromagnetic elements, devices and machine. Electromagnetic elements handle the machine geometry and beam optics parameters; they are the fundamental building blocks of the actual machine model. Devices handle the entire interaction with the physical devices via the underlying control system. Machine is the assembly of elements and devices plus some global properties and functions. The model is presented in the form of diagrams drawn following UML conventions.

3 ELECTROMAGNETIC ELEMENT

3.1 EMelement

The Electromagnetic Element (EMelement) class includes the fundamental building blocks needed for machine modelling (fig. 3). All the EMelement objects share some common characteristics: name, geometric information, survey errors and matrixes for linear optics calculations. A sequence of EMelement objects also defines implicitly the layout of the machine and the geometry of the reference orbit.

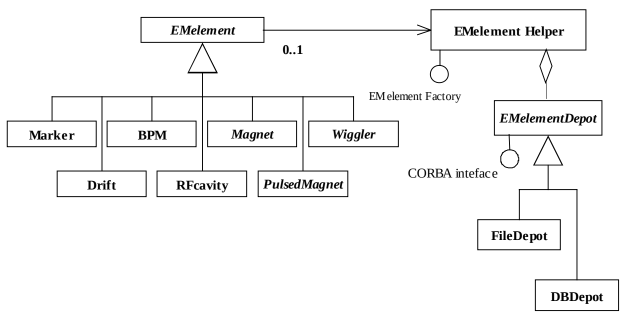

Figure 3: EMelement class diagram.

Objects of this class can also handle a reference to an EMelement Helper object. The EMelement Helper is a class used mainly during the building of the machine model. It is a composite object containing a reference to a repository (EMelement Depot) from which it is possible to retrieve the data needed to initialise correctly an EMelement. The EMelement Helper is also a *factory* object [1] that can be used to create automatically a fully configured EMelement. It is realised as a *singleton* [1], thus eliminating the need to have global variables in the final applications. The EMelement Depot is defined as an interface accessible also through CORBA. The actual realisation of EMelement Depot can be in the form of File Depot, taking all the needed parameters from a configuration file, or in the form of DB Depot, taking all the needed parameters from a relational database. The DB Depot is typically utilised as a CORBA service.
The EMelement class is specialised into a series of components. We will examine in detail the Magnet class, because it is the most important and is a good prototype for the remaining classes.

3.2 The Magnet Class

The Magnet class is an abstract class. It is further specialised into the various types of magnets used in our machine. All Magnet objects have a reference to one Current Controller objects, reflecting the situation that each "real" magnet is connected to one power supply (at least for our machines). All Magnet Objects are composite objects, each containing a specific instance of a MagField object (fig. 4).

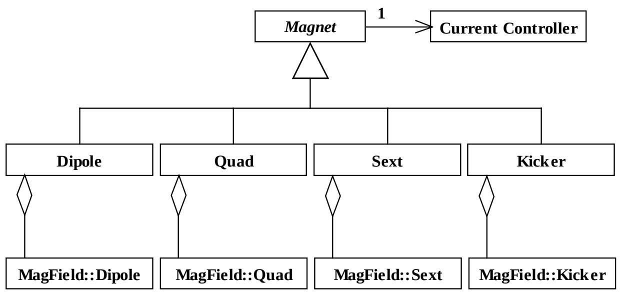

Figure 4: the Magnet class.

The main task of Magnets is to handle the updating the various matrixes and optical parameters from the values obtained from MagField and Current Controller.

3.2 The MagField Class

The MagField Class (fig. 5) handles the conversions from current to magnetic field components: $B_X=f(I)$. It also handles the inverse conversion from main magnetic field component to current: $I=g(B)$. The conversions are carried out with an interpolation algorithm, specialised according to the particular class of the magnet. In order to initialise correctly the coefficients or tables needed by the algorithm or to build automatically MagField objects, we can use a MagField Helper class. This class follows closely the design pattern of the EMelement Helper class to which the reader can refer.

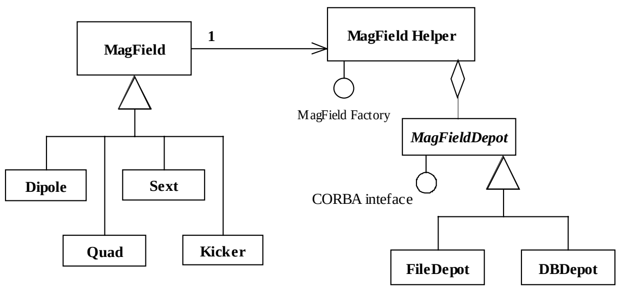

Figure 5: the MagField class.

4 DEVICE

4.1 Devices and the Control System

The abstract class Device is the basis of all the classes that encapsulate the interaction with the Control System of our machines (fig. 6). The Device class is designed to give access only to the parameters relevant to machine physics, not as a replacement of the general Control System facilities. In this way, we minimise the dependency of the Device class from the underlying Control System implementation. Eventual modifications or enhancements of the Control System will have a small and localised impact on the HLS framework.

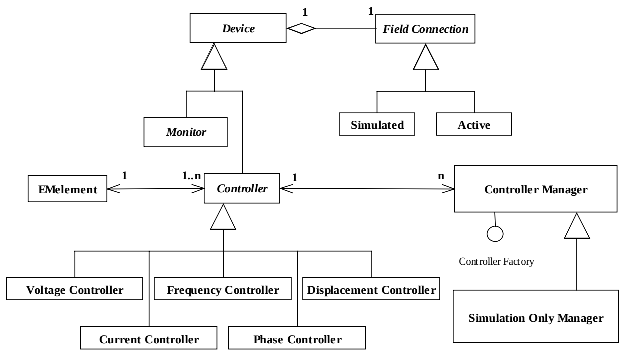

Figure 6: the Device class hierarchy (partial view).

4.2 Field Connection

The Device class is designed as a composite object, always containing a Field Connection object. This abstract class defines the interface for the exchange of values with the field. It defines two operating modes: active and simulated. Active connections manage true interactions with the field, that it is an active connection can modify the live parameters of the machine. Simulated connections cannot modify the live machine; modification requests are faked by using the requested set point as the value for the successive readings. Simulated connections can however read the actual values from the field.

Field Connection is specialised into two concrete classes: Active and Simulated. The Active class implements the full field access functionality and the simulation mode functionality. The Simulated class implements only the simulated connections. The Simulated class is used for safety reasons by HLS programs that are not allowed to modify the machine settings, for example simulators or operator training programs.

4.3 Controller and Controller Manager

The Controller class models a device capable of reading and setting values on the machine. Each controller is associated with one or more EMelement objects, while each EMelement is associated to one Controller. These associations can be traversed in both directions. This feature mimics the situation in which a single device controls a family of elements; the typical case is a power supply connected in series to a string of magnets. From another point of view, we can say that these associations express the constraints on the degree of freedoms available for controlling the machine. Controllers and EMelements also provide a set of tools to navigate among the web of associations. The associations are implemented by means of object references and standard C++ associative containers.

The Controller class is finally specialised into controllers specific to the actual quantities: Voltage, Current, Frequency, etc.

Each Controller is also associated to a Controller Manager. A Controller Manager performs global management of Controller objects, for example the switching between active mode and simulation mode. The Controller Manager handles a list of all the Controllers and acts as a factory object for Controllers. A specialisation of Controller Manager, Simulation Only Manager, has factory methods that return Controller objects with a field connection object of the Simulated specialisation.

4.4 Monitor and Monitor Manager

The Monitor class models a device capable of reading values on the machine. It is typically used to describe beam instrumentation objects. A Monitor is associated with a single EMelement. The Monitor Manager is analogous the Controller Manager, the reader can refer to it for details. The Monitor class has a single specialisation, Position Monitor.

5 MACHINE

5.1 Machine, Line and Ring Classes

Objects of class Machine are the repositories where all the components of the model of a machine are assembled (fig. 6).

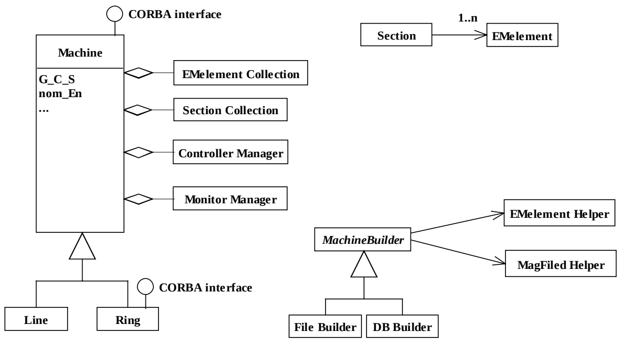

Figure 7: Machine and other auxiliary classes.

A Machine object provides also the tools (iterators) to navigate the web of elements. The Section class (fig. 7) is an auxiliary container used to build some logical grouping of elements and facilitate the navigation.

Each Machine object manages a group of global properties of the model, for example nominal beam energy, global co-ordinate system (GCS), etc. The GCS is the absolute position and orientation of the reference point (s=0) of the machine. The GCS is useful when we have to assemble different machines and, for example, must check that the beam extracted from one machine can be injected into the next.

A CORBA interface is defined for using Machine objects in distributed applications.
The Line and Ring subclasses add to the basic Machine class the specific methods to calculate beam dynamics properties of that type of machine, (e.g. equilibrium emittance, radiated power, etc.). The two subclasses add also some specific geometric properties (e.g. injection and extraction points, location of light ports, etc. for a Ring). The CORBA interfaces for Line and Ring are derived (subclassed) from the CORBA interface of Machine.

5.2 Auxiliary Classes

Two auxiliary classes are designed to work together with the Machine class: Section and Machine Builder.
Section is just a named container of EMelement objects. It provides some logical grouping and facilitates the traversal of the machine model. For periodic machines, it may be used to assemble a period of the lattice.
The Machine Builder is a factory of Machine (Line, Ring) objects. Actual Machine objects are built starting from a description stored in a configuration file (File Builder specialisation) or from a database description (DB Builder specialisation). Machine Builder objects use the services of the various Helper classes already described. Builder objects will be the preferred tools to assemble working machine models in HLS programs.

6 CONCLUSIONS

Machine physicists using the new HLS framework have to deal with a set of entities and concepts that are very close to their mental image of the problem. As a further benefit, they can exploit the existing and continuously evolving O-O tools and libraries such as the ANSI C++ Standard Template Library [15] or the catalogues of well documented "design patterns" [1]. All the programming efforts are directed toward the solution of the real problems, with fewer efforts wasted for solving trivial tasks. The practical benefits are faster developing times and easier maintenance for the new HLS programs.
The use of UML/RUP methodologies and diagrams for the design of the framework has proven to be effective and useful.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley Longman 1995.
[2] A. Akiyama et al. "Integration of the Modeling Program with Accelerator Control Systems in TRISTAN", Proc. ICALEPCS 95, Chicago 1995.
[3] S. Kuznetsov, "C++ Library for Accelerator Control and On-line Modeling", Proc. ICALEPCS 95, Chicago 1995.
[4] B.A. Bowling et al. "The Use of ARTEMIS With High-Level Applications", Proc. ICALEPCS 97, Beijing 1997.
[5] M. Plesko, "An Approach to Portable Machine Physics Applications", Proc. ICALEPCS 95, Chicago 1995.
[6] J. Rossbach, P. Schmüser, "Basic course on accelerator optics", CERN 94-01, Vol. I.
[7] F. C. Iselin, "The MAD Program User's Reference Manual", CERN/SL/90-13.
[8] G. Chiozzi, "Use Cases for Requirements Capture and Tracing", this conference.
[9] G. Schneider, J. P. Winters, "Applying Use Cases", Addison Wesley Longman 1998.
[10] A. Jaaksi "Our Cases with Use Cases", Journal of Object-Oriented Programming, Vol. 10, No. 9, Feb. 1998.
[11] F. Willeke, G. Ripken, "Methods of Beam Optics", DESY 88-114, Aug. 1988.
[12] R. V. Servranckx, K. L. Brown, "Circular Machine Design Techniques and Tools", SLAC-PUB-3942, Apr. 1986.
[13] M. Fowler "UML Distilled: Applying the Standard Object Modeling Language", Addison-Wesley Longman 1997.
[14] J. Rumbaugh, I. Jacobson, G. Booch "The Unified Modeling Language Reference Manual", Addison-Wesley Longman 1998.
[15] D. R. Musser, A. Saini "STL Tutorial and Reference Guide", Addison-Wesley Longman 1998.