LA-UR-00-3895

Los Alamos
NATIONAL LABORATORY
Technical Report

Los Alamos Neutron Science Center, LANSCE-1
Accelerator Physics and Engineering Group

To/MS: Distribution
From/MS: Christopher K. Allen, H817
Phone/Fax: 5-5241/5-2904
Symbol: LANSCE-1:99-163
Date: December, 1999
Email: ckallen@lanl.gov

SUBJECT: A SOFTWARE ENGINEERING APPROACH TO PARTICLE BEAM SIMULATION

The design of a particle beam simulation system for accelerator applications is presented from a software architectural standpoint. We are concerned with the structure of the software system and not the details of the physics and algorithms used within the system. However, most of the structural components and relationships are formulated directly from the physical problem domain. Techniques and technologies that enhance the robustness and upgradability of software are presented with specific attention to particle beam simulation. With these techniques, the architecture of a highly adaptable and easily maintained software simulation system is presented.

1 Introduction

In the current state of the art, software design and development differs very little from hardware design and development. Large software systems should be modular, consistent, easy to upgrade and based on accepted standards. In order to meet these criteria and the constantly changing requirement of the user, the *architecture* of large software systems deserves explicit concern. Here we cover the basic issues in the development of particle beam simulators from this architectural viewpoint. We point out the tried and true characteristics of well-designed systems and how to conform to these standards when implementing particle-beam simulators. We also discuss available software techniques and technologies that enable construction of robust software and how the accelerator community can exploit these resources.

1.1 Programming Paradigms

Typically, beam simulation codes are designed from an algorithmic, or procedural, perspective. This is the traditional paradigm for implementing software before the advent of object-oriented technology. In the algorithmic environment, the foundation is the procedure, or subroutine. The focus of the developers is thus on algorithms and program control. The data and algorithms are distinct entities and it is necessary to control which algorithms operate on which data and in what order. Although there is nothing inherently wrong with this design paradigm, it typically yields brittle code. When requirements change, and they always do, or when the system grows, and it always does, software built on this paradigm is very difficult to maintain. The choice of modeling perspective has a profound influence upon how the problem is attacked and the solution is formulated [2].

Modern software development is based upon an object-oriented perspective. The fundamental building block is the *object*. A software object is a code component that implements some abstraction from the problem domain or solution domain. Every object

has an identity (it can be uniquely identified), a state (an associated data set) and a behavior (it presents a set of operations on itself or other objects). These fundamental properties are necessary to make the software object consistent in form with a hardware object. We shall elucidate these notations further in the sequel.

1.2 Software Architecture and Development

As a whole, a large software system should consist of independent components that fit together nicely to form the whole. Individual components should have well-defined interfaces so that options or upgrades may be "plugged in" when so desired. A good analogy for this idea is the hardware design of a computer. If a computer requires a larger hard drive, it is simply connected to the appropriate internal interface perhaps replacing an old one. The hard drive is an autonomous component having a well-defined interface and protocol by which it communicates. For persons responsible for maintaining the computer, upon removing the cover it is easy to identify the individual components and also to replace and upgrade them. Likewise for the software system, responsible persons (i.e., developers) would prefer a similar situation.

On the lower level, the design philosophy for individual software components has also changed tremendously in the past years. The need for "tight code" has all but disappeared in favor of clear, concise program code that is easy to understand and maintain [6]. One should not code for detailed machine optimizations but, rather, tasks should be performed in a manner that is most clear and straightforward. In most situations it is best to "ignore" the physical machine in lieu of an abstract, generalized machine that has no specific structure[1]. This idea has been taken to the extreme in the Java environment, where programs run on a *virtual machine* that is universal across hardware and operating system platforms. Because hardware technology typically advances much more rapidly than software technology, one's effort is best spent writing clear understandable code rather than optimizing code performance. This is especially true when considering the amount of time spent post-production on upgrades and specific user modifications.

Particle beam simulation system attempt to imitate complicated real-world situations, accelerators. With a rigorous design methodology and a solid foundation such software systems can adapt smoothly to evolving design specifications and user demands. They should be designed with extensibility in mind. There are a number of software tools and technologies currently available that facilitate these goals. For example, object-oriented computer languages and component software development tools combine to provide an excellent framework for a robust implementation. The combination produces a software design environment that closely parallels that of modern hardware design.

The investment required for a good software design should not be overlooked. Forethought in this phase will produce systems capable of adapting naturally to changing requirements.

[1] Obviously this philosophy does not apply when developing for specialized machines such as Blue Mountain. However, even for these machines Application Program Interfaces (API's) are now available which present the software developer with an abstracted parallel environment (e.g. MPI).

2 Software Technology and Particle Beam Simulation

Here we present some software technologies, both new and old, that warrant particular attention to the design and development of particle simulation systems. Although much of this technology has been part of the computer scientist's toolbox for some time, it is seldom seen in the accelerator community. Consequently, we provide a brief overview for each technology then demonstrate its application to particle beam simulation.

2.1 The Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a recent development in the ability to model software systems [2]. It is a graphical, systematic method to model and represent the architecture and operations of complex software. Like any other language, it has grammar and syntax, however, its vocabulary consists of sets of associated diagrams. The diagrams may contain varying levels of detail, but all explicitly indicate the implementation of a software system built from objects. With the use of the UML, software may be designed and tested on paper before any code is actually written. Once the architecture of the system is agreed upon, the language, productivity tools, and platform may then be chosen which best suits the implementation. Most all the illustrations in this paper are written in the UML.

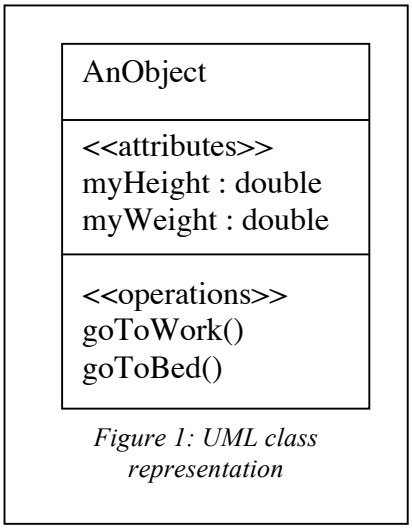

Figure 1: UML class representation

The UML provides the "blue prints" for application architectures. It is programming language independent and platform independent. Thus, like mechanical drawings, the UML provides a conceptual model of the application that may be understood by a wide audience. Figure 1 shows the UML representation for a class, or object. A class is represented by a rectangle with three internal compartments. The first is the class name, in our case AnObject, the second contains the attributes, or data, for the class, while the third compartment contains the operations of the class. Also shown in the figure are some adornments, in guillemets, to further identify intent.

2.2 Preliminaries: Data Records, Dynamic Allocation, and Pointers

We introduce some fundamental programming concepts that are independent of programming paradigms. These techniques improve robustness and enhance code clarity.

Data records are the predecessors of modern day objects. As the name implies, data records contain data (whereas objects contain both data and operations). Data records organize, categorize, and store program data in logical blocks consistent with the problem domain. That is, data is organized into units which "make sense" in the problem domain. For example, consider our problem domain of a particle accelerator. Here, a logical organization for particle simulation data would be records containing the phase states of the individual particles. Figure 2 shows the design of a data record called Particle. We see that it consists of eight attributes: the particle's transverse phase coordinates *x, xp, y, yp*, the particles charge and mass given by *chrg* and *mass*, and the particles phase and energy given by *phi* and *W*. Also shown in the figure are two implementations of the data record, one in FORTRAN and one in C.

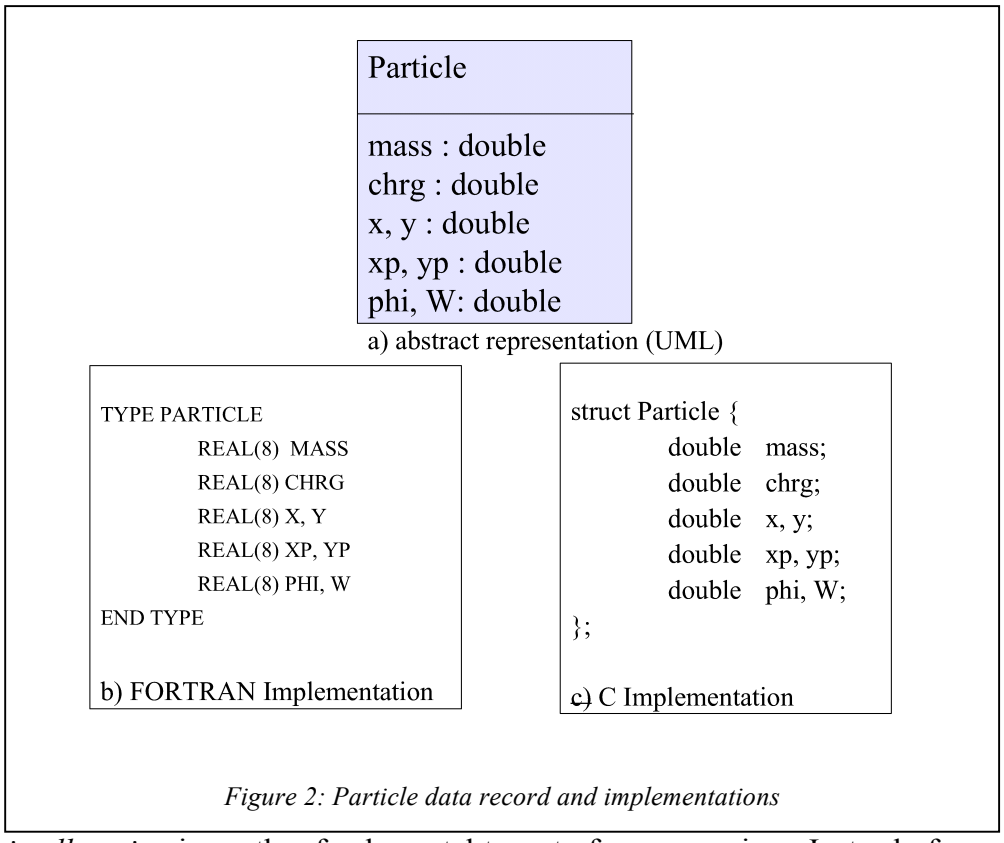

Figure 2: Particle data record and implementations

Dynamic allocation is another fundamental tenant of programming. Instead of reserving storage at compile time (before the program runs), it is best to allocate storage dynamically as needed. Storage should be allocated during run time according to the application requirements, rather than reserving an arbitrary amount of storage *a priori*. In this manner the amount of data that can be processed is limited only by the machine and not by some arbitrary limit determined at compile time. If the data set changes set no recompilation is necessary. We simply ask the operating system for more memory resources, as we need it. This idea is illustrated in Excerpt 1, where we implement the dynamic allocation of an array of the Particle data records. We assume the existence of an

unformatted data file, named "Particles.dat", where the first entry is the number of particle records to follow. Once that value is read, the array of records is allocated, and the data is loaded as a block. After the program is finished with the data array, the memory should be released with a DEALLOCATE statement.

```
TYPE(Particle), ALLOCATABLE   :: arrParticles(:)    ! the ensemble data
INTEGER                       :: cntParticles       ! number of particles in ensemble


! Open file and get number of particle coordinate sets
OPEN(UNIT=2, STATUS='OLD', FORM='UNFORMATTED', FILE='Particles.dat')
READ(2, *) cntParticles

! Allocate record array and read in coordinates
ALLOCATE(arrParticles(cntParticles))

READ(2, *) arrParticles
CLOSE(2)
```

Excerpt 1: dynamic allocation with FORTRAN 90

The final fundamental programming concept we cover is that of *pointers*. Pointers are software entities that point to data in core memory. In particular, they can point to dynamically allocated data records. Thus, program data is managed by accessing it through pointers (known as *dereferencing*) rather than direct declarations of specific data records. Specifically pointers contain memory addresses, however, one may loosely consider a pointer as being similar to an array index. The index i "points" to the i^{th} element in the array A; thus, that element is dereferenced using the syntax $A(i)$. In the case of a true pointer, the array would be all of core memory.

The actual value held by a pointer is the machine address of a memory location. This situation is depicted in Figure 3. Thus, the pointer "points" to a location in core memory. Although this physical picture is accurate, it is usually better to conceptualize the pointer as pointing to a "data object" that exists in the process domain of the program. This idea is also shown in the figure. The later viewpoint is that of the object-oriented paradigm and leads naturally to the notion of a software object. Finally, we make the aside remark that for security reasons the Java language environment does not support pointers. Instead of pointers Java implements *references*, which like pointer also provides for object indirection, however, core memory is not directly accessible.

Pointers are most often used when building data structures. For example, the array is one form of data structure, one where the internal data is referenced via an index. (In Excerpt 1 the underlying data structure is an array.) Other data structures require the use of pointers to reference their data records. In the next section, we discuss several different types of data structures.

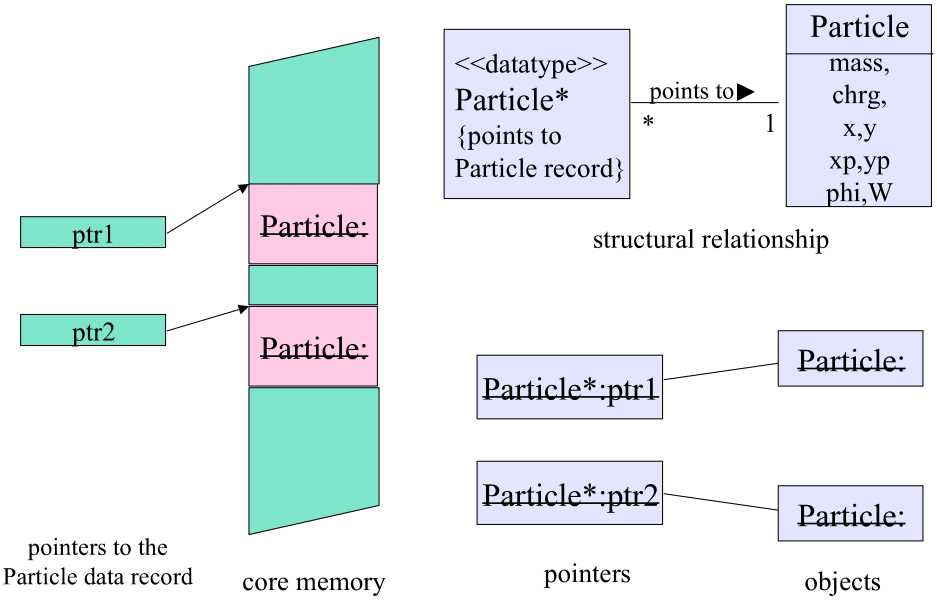

Figure 3: pointers

2.3 Data Structures

Traditionally, one of the basic concerns of computer science is the study of *data structures*. Here one considers the form of the program data along with its primary method of storage and access. Proper choice of data structures will greatly improve program clarity and manageability; moreover, using the proper data structure has the potential to drastically improve program performance. Depending upon the application, data structures can be very sophisticated. Typically, however, most programs require only a few fundamental data structures.

We are primarily interested in data structures that contain instances of basic data records, in our case the Particle record. These structures are usually referred to as *collections* or *containers*. For example, consider the *linked list* depicted in Figure 4. There we see one possible container for an ensemble of particles. As before, each data record contains the position, velocity, charge and mass of a particle object. In addition, however, the data record must also maintains a pointer to the next particle record in the list. All the particle records are thus linked together forming a chain, or list, of data records. The linked-list container is convenient in that it is not necessary to know *a priori* the number of elements the list will contain. Typically, new records are pushed onto the head or tail of the list. To iterate through the list, we simply follow the trail of pointers, stopping once the NULL value is encountered.

Not only can data records be allocated and added to the list dynamically, but they can also be deallocated dynamically. This is convenient whenever a particle is lost (e.g., it

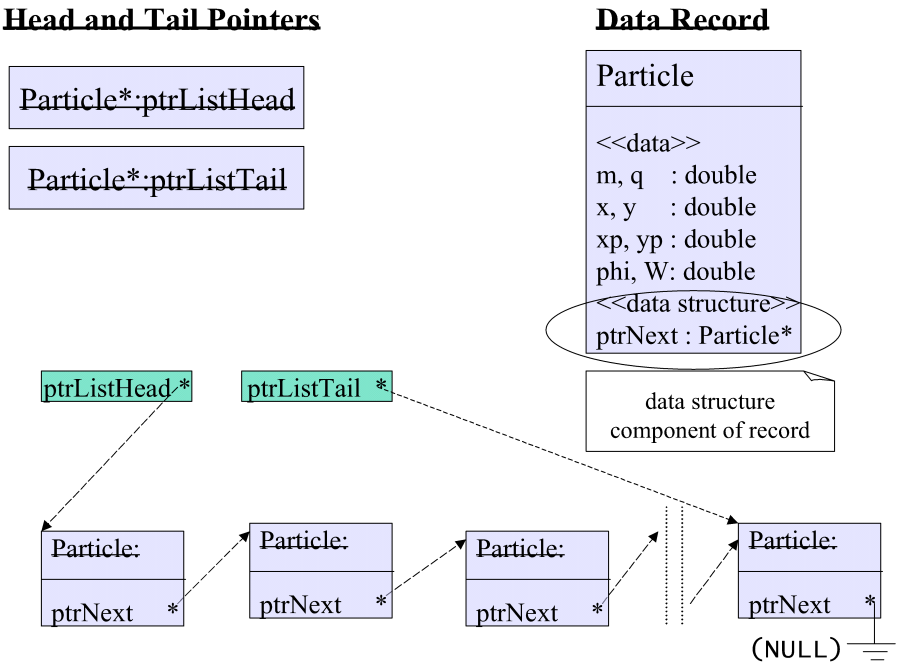

Figure 4: linked list particle ensemble

hits a structure in the beamline). If the particle is lost during a simulation, it is simply removed from the list and its memory resources are returned to core memory. If the ensemble were to be stored as an array, as typically done in FORTRAN programs, the index of the lost particle must somehow be flagged as lost, or the entire array must be repacked. This condition indicates that the array is not the natural data structure for the ensemble. To further this point, note that typically in particle simulations we only iterate through the particles in an ensemble. The array index is used only as a tool for iterating through the collection; it does not belong to the problem domain.

2.4 Object-Oriented Programming

In the past ten years, there has been a lot of attention and emphasis on object-oriented programming, which has all but lived up to its early expectations. It did not revolutionize code reusability, nor did it revolutionize software packaging, distribution and installation, as expected. However, it did completely change the way the industry views software design and modeling paradigms. It gives architects and developers a way to create software systems that mimic modern hardware systems. In this way the development of object-oriented programming can be compared to the discovery of penicillin. The new medicine was initially considered a miracle drug and generated copious publicity. However, as we now know the discovery really signified a new, and more fruitful, approach to the treatment and control of infection. By moving from a procedural design paradigm to an object-oriented one, the ability to maintain and upgrade existing application is drastically improved.

2.4.1 The Object

Object-oriented programming allows for the realization of abstractions that occur naturally in the problem domain. In our case it makes sense to implement objects representing particles, ion sources, quadrupole magnets, diagnostic equipment, etc. The

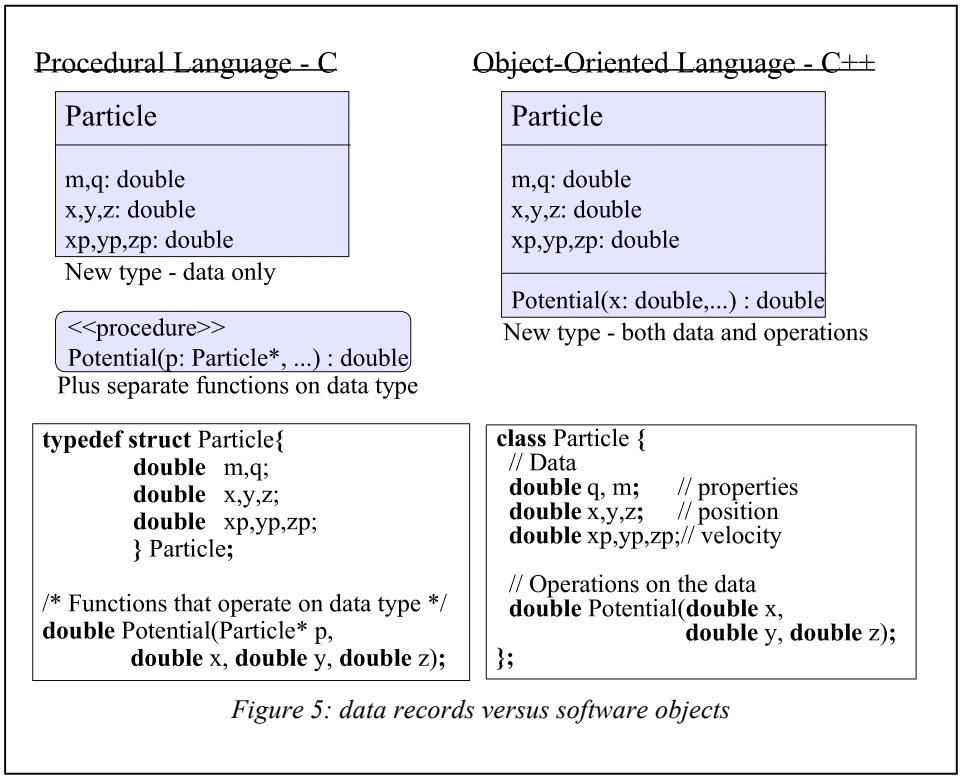

Figure 5: data records versus software objects

software object makes it possible to implement an accelerator simulator in much the same way we view the actual accelerator. This situation is extremely useful when code development and upgrades are implemented by a group of developers with an accelerator background.

Software objects have both attributes (data contained in the object) and operations (functions or procedures which may be called by internal and/or external events). This

gives the object the ability to exhibit identity, states and/or conditions, and behavior; a situation that is analogous to physical objects. From a language perspective, objects are essentially the progeny of data records. Within the data record, adding functions, as well as data, yields the object language-construct. The functions contained in the record are intrinsically aware of the data contained therein. Thus, the scoping of data is simplified; functions already know on which data they are to operate.

For us the most important aspect of object-oriented programming is the ability to create and support objects that have meaning to accelerator physics. This idea is illustrated in Figure 5, where we reconsider our Particle data record. The procedural approach and object-oriented approach are contrasted by implementing the record in both C (procedural) and C++ (object-oriented). In C, it is possible to create new data types from data records using the typedef keyword. However, the records contain only data, and no operations on the internal data. As seen in the figure, functions are separate entities and must be told explicitly on which data to operate. A true object contains both data and operations on the data. C++ adds language support for objects with the keyword class so that both data and functions may be included in the record, forming the complete object.

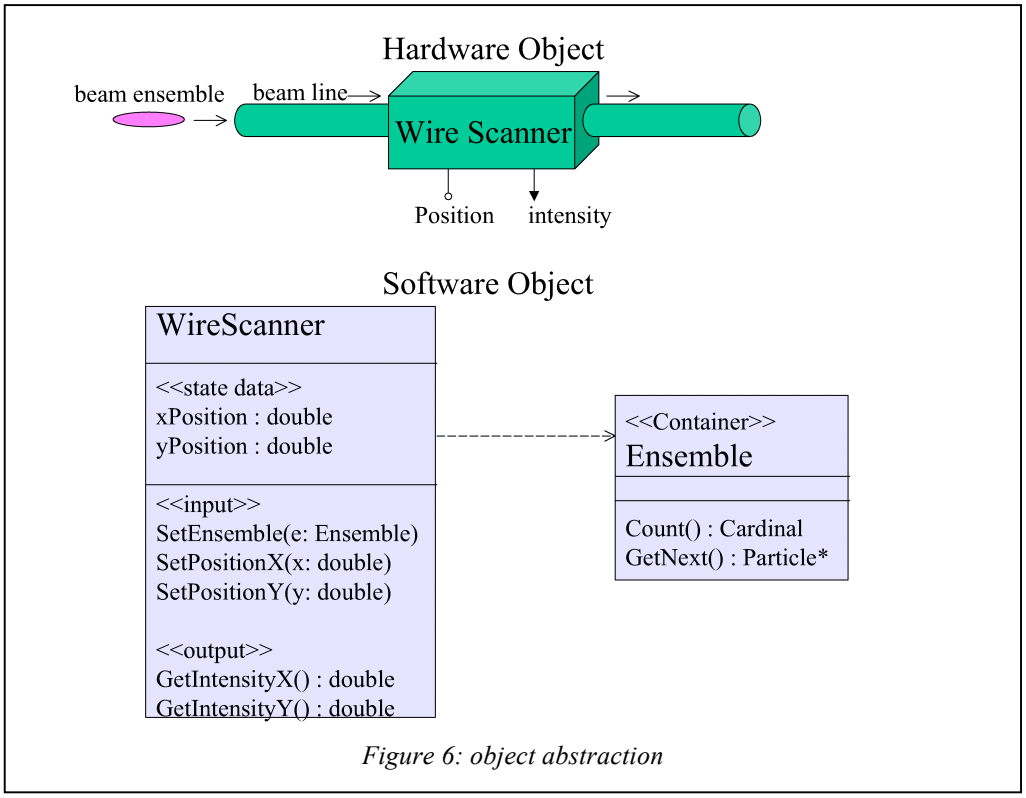

Figure 6: object abstraction

The additional capability of object-oriented languages may not seem significant at first; it is certainly possible to accomplish required tasks without it. However, the resulting design environment has now changed tremendously. Now we can create useful new data types, such as complex numbers, vectors, matrices, tensors, etc. Moreover, we can build objects with physical significance, such as particle ensembles, ion sources, drift-tube linacs, signal processors and diagnostic equipment. Moreover, these objects may be implemented in much the same way hardware is implemented. For example, consider the

situation shown in Figure 6 where a piece of real-world hardware, a wire scanner is being modeled. The wire scanner has input (the beam), a state control parameter (the wire positions), and output (the beam profile intensity). (Obviously there is more detail to consider, yet, these properties capture the essence of the object.) Referring to the software implementation, the object presents itself in much the same way as we view the actual hardware. It has state attributes, the current position of the wires, and it gives us an interface in which to move the wire positions and collect the intensity data. Notice that the specifics of the implementation are not shown in either case. As users of this object, we are unconcerned with these details and should not be burdened with them. That task falls solely upon the developers of the object, both hardware and software. Once developed, the object is a self-contained entity presenting a clear picture of its intended use.

2.4.2 Objects as Language Extensions

Another advantage of an object-oriented language is that is essentially supports its own extensibility. It enables the creation of software objects and allows them to be treated as ordinary language constructs. To illustrate, consider the (procedural) programming language C. It contains the following list of intrinsic data types (or atoms: bool, char, int, unsigned, float, double. Because these types are intrinsic to the language, the language also supports the usual compliment of operations on each data type. We can add int's, subtract int's and multiply int's using the respective operators +, -, and * provided by the language. The operators recognize the data type and understand how to perform their respective operations. Thus the system (int; +,-,*) is complete and closed, forming the data type int. This notion of a set of elements and all its operations is exactly the paradigm that object-oriented programming languages embody.

```
typedef double  Real;

class R3 {
        Real    x, y, z;    // coordinates

        // Assignment
        R3      operator=(R3 vec);

        // Vector algebraic operations
        R3      operator+(R3 vec);
        R3      operator-(R3 vec);
        R3      operator*(Real scalar);
};

⋮
R3      ptA, ptB, ptC;      // 3 points
⋮
ptA = ptB + ptC;
```

Excerpt 2: vector algebra in C++

A familiar object example might come from the addition of vectors. The C++ language includes the ability to redefine (or *overload*) the operators +, - and * on objects. Thus, we can create vector objects that are used syntactically just as the intrinsic scalar data types. Obviously, this has a clear advantage in terms of program clarity, a very important quality when debugging or working with code written by other developers. The idea is illustrated Excerpt 2, where we define objects of type R3, representing real 3-vectors (the type definition Real is meant to enforce this notion). The excerpt also illustrates how to add two vectors to obtain a third. This object is very convenient for working with position vectors in Euclidean space.

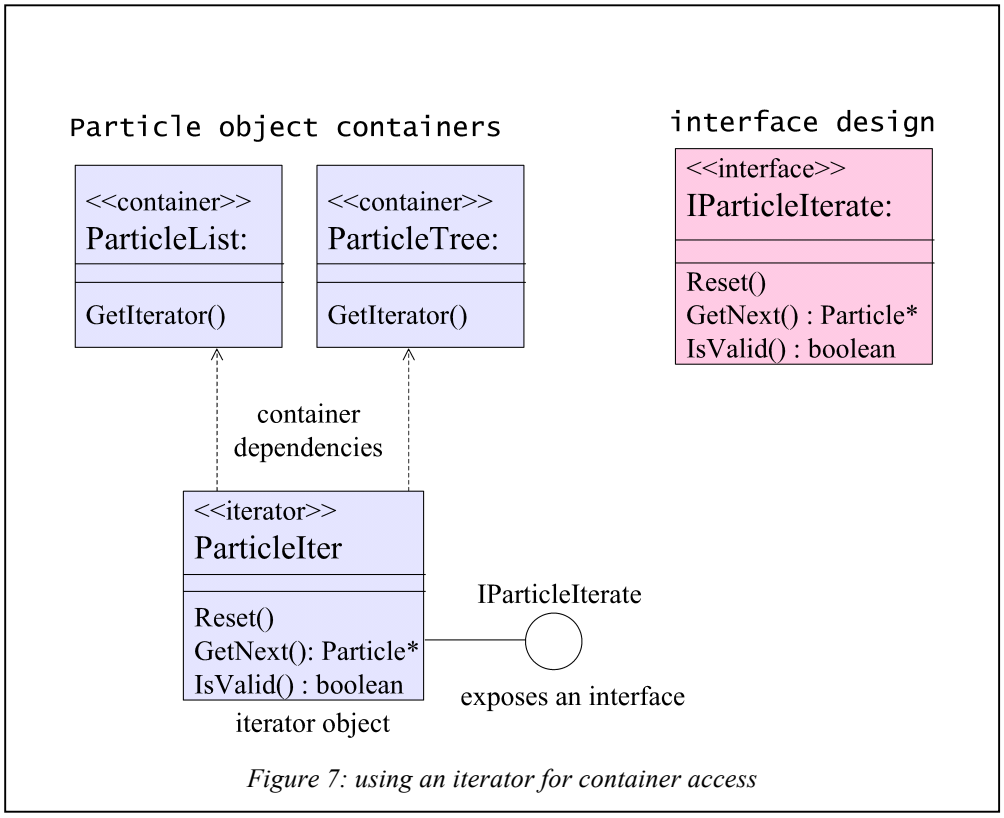

Figure 7: using an iterator for container access

2.4.3 Collection Iterators

Finally, we present another construction possible with object-oriented programming, that of a collection *iterator*. An iterator is used to iterate through collections of objects in a container-type data structure. The iterator understands the container structure and its access, yet presents a "formless" view of the container to the outside environment. If the fundamental structure of the container is changed, accessing elements remains consistent. Only the iterator must be modified.

In reality iterators typically understand how to access elements of several different types of containers. Thus, the elements may actually be contained in many different data structures but the iterator presents a consistent method for accessing them. This idea is demonstrated in Figure 7 depicting an iterator for particle-ensemble data structures. There we see that `Particle` objects are contained in two different types of data structures, a linked-list and a tree (yet to be discussed). The iterator `ParticleIter` maintains a consistent way of accessing elements regardless of the underlying container structure. The iterator `ParticleIter` has three operations, `Reset()`, `GetNext()`, and `IsValid()`. These functions reset the iterator to the (arbitrary) first element, retrieve the next element, and check whether we have reached the last element in the container. The job of the iterator is to provide access to every element in the container. It hides all structural properties of the container.

We point out that the current Standard C++ Library, `std`, has a variety of container classes capable of maintaining user-defined objects. These containers include vectors, lists, stacks, and maps. Yet, all these different data structures may be accessed using the class `iterator`, also in the library. The situation provides a quick and convenient method for making performance modifications. It may be that for a particular situation one type of

data structure provides faster iteration than for others. When that case occurs, simply switch the underlying container. This action can be done at run time.

2.5 Software Interfaces

Presenting a consistent method of communication leads to the notion of a *software interface*, or simply, *interface*. Iterators embody this idea. The iterator presents a consistent method of accessing containers; structure is irrelevant as far as the user is concerned. A software interface is a technique to strictly enforce such consistency within the software domain. That is, they present the capability of establishing well-defined methods of communication between software components. They represent a contract between software objects. Any object that supports an interface agrees to provide that communication schema to other components, forever. Software interfaces are analogous to hardware interfaces, such as PCI card slots, RS232 ports, BNS connectors, and standardized telephone jacks. For example, any telephone presenting a standard jack knows how to communicate with the PBX.

Structurally, interfaces are similar to objects in that they contain functions. However, they do not contain data. Therefore, they have no state or condition. They are simply meant as a means of communication between objects. An example interface IParticleIterate is shown in Figure 7. The structure of the interface is shown in the right hand side of the figure, while inspecting the iterator ParticleIter we see that it actually exposes this interface. The interface contains three member functions the ParticleIter objects needs to do its job. This example may seem somewhat trite at first, however, later we shall see that this is a mechanism for building software components discussed in the sequel.

Interfaces are also intended to enforce general situations. Many different types of objects can provide the same interface. For example, we can create a Quadrupole object used to focus ensembles of Particle objects. Although very different from a Particle object, both have electromagnetic properties. Thus, both may present the same interface used to display electromagnetic behavior, say IEmFields. Be aware, however, the interface is similar to, but not the same as, object inheritance. Unlike an object, once it is defined an interface should never change. Just as with hardware interfaces, this static condition is necessary to maintain backward compatibility. Consequently, the design of a software interface warrants careful consideration. If in the future it becomes necessary to upgrade an interface, than a completely new interface should be specified. For example, to upgrade the interface IEmFields we must also move to a completely new identifier, such as IEmFields2. This parallels the hardware interface upgrades seen in industry, like moving from SCSI to Wide SCSI, and then to Ultra Wide SCSI.

Before moving on, we point out two modern software technologies built primarily on the software interface. The JavaBeans™ of the Java environment is a component architecture built from prescribed interfaces [7]. Objects that conform to these interfaces are known as Java Beans. Java Beans are thus self-contained software components that may be invoked, inspected and run by applications without prior compilation. That is, they automatically fit into applications understanding the JavaBeans architecture. Microsoft's COM (for Component Object Model) software technology is now the basis for much of the Windows operating system and is entirely based on the concept of interfaces [8]. For example, all ActiveX™ components perform basic communication via

a set of pre-defined interfaces [3]. The particular behavior of an ActiveX component is defined by an additional set of interfaces that ship with the object. Typically, ActiveX components also provide a special interface that allows any outside software system to determine the component's operation and use it at run time (this capability is known as *automation*). A nice feature of the COM standard is that it is language independent. Thus, software may be written in the language best suited to the application. The drawback of COM is that it is currently available only on MS Windows platforms. The JavaBeans technology is platform independent. However, Java Bean components must be written in the Java language.

2.6 Software Components

Software applications originally consisted of a single, monolithic, binary file. If changes to the application where required, the source code was updated then recompiled into the new binary file. The entire application had to be rebuilt and redistributed. Application users had to wait for developers to modify, compile, and reship the newest version. This process is painfully slow and costly considering the advancing pace of current software technology. To offer an analogy, suppose you wished to upgrade the sound system in your automobile. If you owned a monolithic car then the only possible option would be to sell the old car and purchase the latest model, in the hopes that the manufacturer included a better sound system. Obviously this is a drastic action, as it is with large, sophisticated software systems.

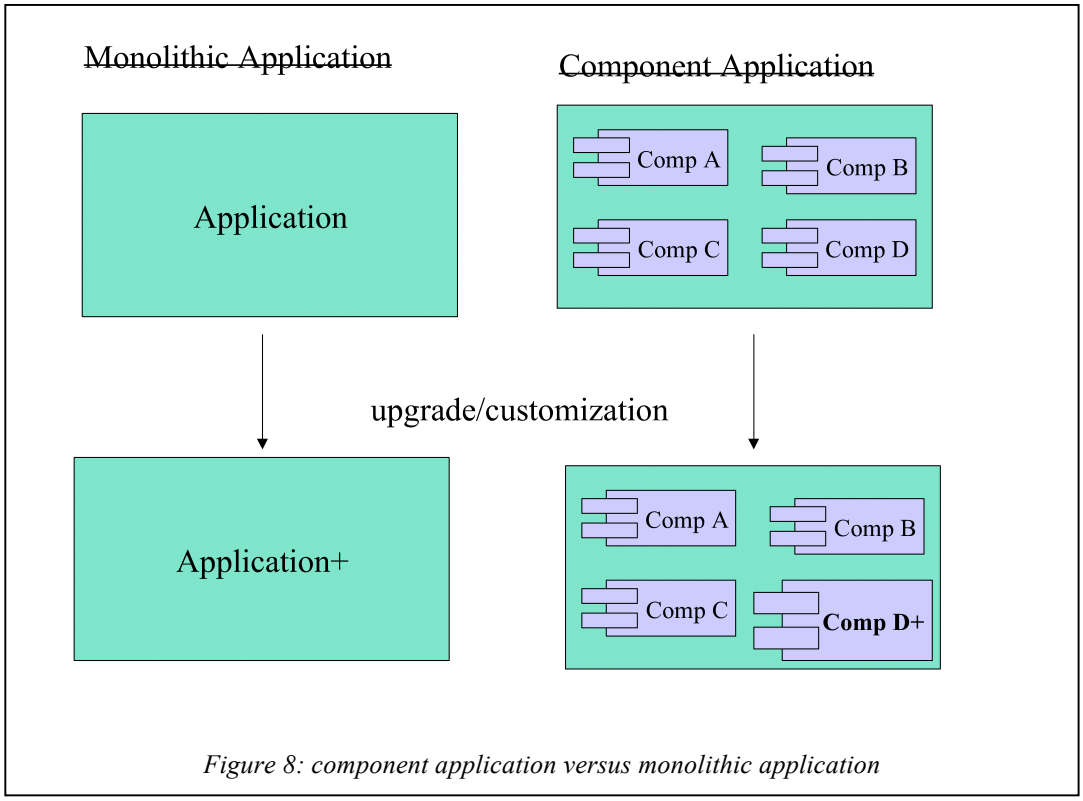

Figure 8: component application versus monolithic application

Software components provide a modular architecture for software systems. They allow for on-site modifications and upgrades as well as providing better reliability and lower costs. If an upgrade for a software system is desired, one need only acquire the specific

component, which may even be provided by third party vendors. Moreover, component systems can be tailored toward individual needs. These ideas are represented graphically in Figure 8. The situation is again analogous to purchasing a new sound system for your automobile. Typically, you can find high quality, after-market systems from third party manufacturers much cheaper than from the car's original equipment manufacturer (OEM). Moreover, users can upgrade the system on-site.

Software *interfaces* are the instrument by which component software systems are assembled. Interfaces are static entities and any component providing a specific interface will always be able to connect to a software system supporting that interface. The implementation details are irrelevant as far as the system is concerned, and the operation and task of the total system is unknown to the component. Also, the component may be upgraded or modified at any time without disrupting the total system, so long as everything abides by the interface.

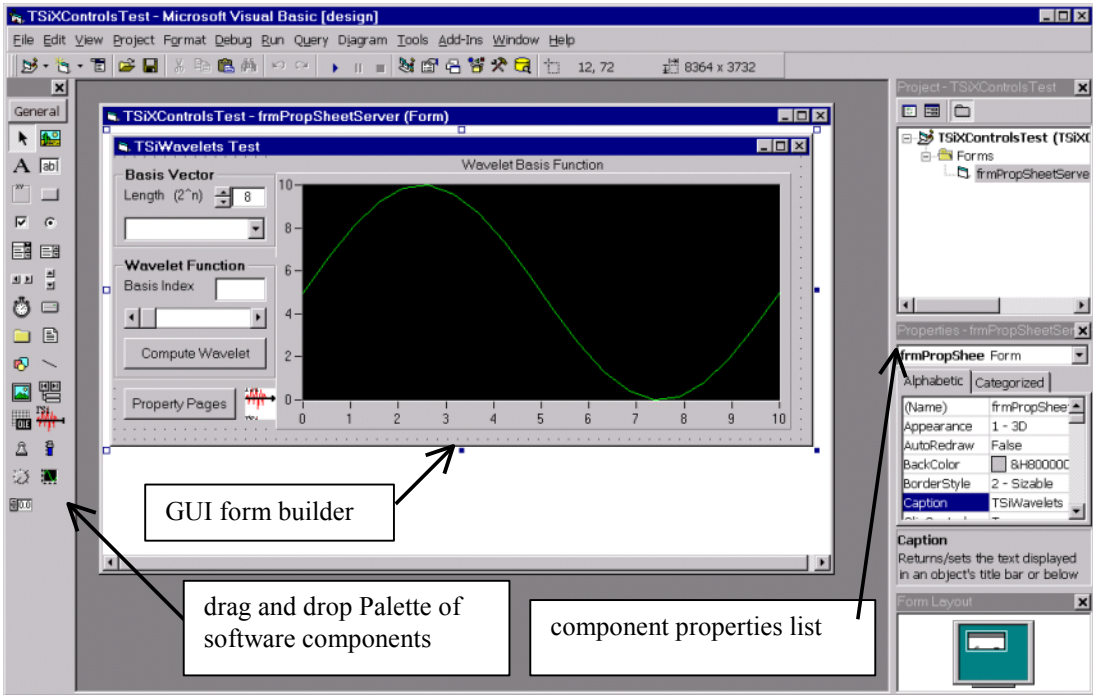

Figure 9: Visual Basic RAD environment

Recently, there has been an explosion of third party vendors for software components. As a consequence, there are a wide variety of system components available at low cost. This is especially true for ActiveX™ controls, which are very convenient and popular forms of software components for MS Windows platforms. Using a Rapid Application Development (RAD) tool such as Visual Basic, complex software systems can be built quickly and easily from ActiveX controls. Currently there exist off-the-shelf ActiveX controls for signal analysis, data acquisition, hardware control, scientific visualization, and more.

As an example of rapid application development from components, we refer to Figure 9. This is a picture of the Visual Basic™ RAD environment. As seen in the figure, we have a palette from which we may select software components, using point and click on the icons, then drag and drop them into the application. The GUI (Graphical User Interface)

of the application is built up from individual forms that allow the user to modify the system parameters. There is a drag-and-drop form builder from which to build up this interface using edit boxes, list boxes, buttons, switches, and other components that allow the user point and click interaction with the system. The forms may also contain charts, spreadsheets, or other methods of data visualization, these features being dependent upon whatever software components the developer has available to him or her. Creation of the application is primarily a drag and drop operation with a minimal amount of coding.

2.7 Visualization

Visualization has become an important research tool in recent years. This is a technology where beam simulation can find rich application. Accelerator physicists must contend with enormous data sets; graphical representations of the data are a convenient way to cope with this fact. The various forms of visual inspection may elucidate patterns and correlation in data sets not otherwise discovered.

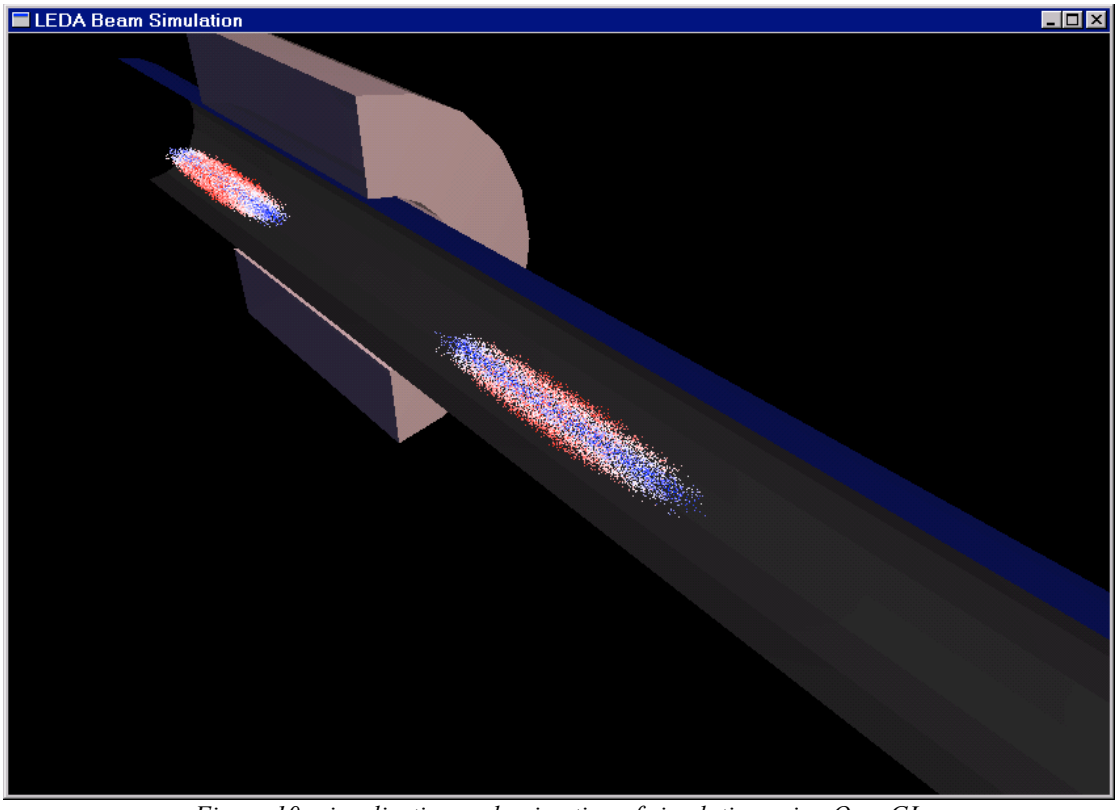

Figure 10: visualization and animation of simulation using OpenGL

There have been significant advances in visualization capabilities, both in hardware and in software. For three dimensional visualization and animation, the low-level graphics library OpenGL has become an industry standard [11]. It was originally developed by Silicon Graphics Incorporated (SGI) is now available on most platforms, including MS Windows and Linux. Most high-level visualization libraries and applications are now built on top of OpenGL. This graphics library is designed to recognize and use any specialized graphics hardware on the host platform. Other, high-level, visualization libraries built over OpenGL include OpenInventor (also by SGI) and the Visualization Toolkit (VTK) [10].

Figure 10 shows one frame in a three-dimensional dynamics animation of a particle simulation using the OpenGL graphics library. In the figure, two bunches can be seen travelling through a periodic lattice, one magnet is visible. The particles in the beam are color-mapped: red particles have large transverse velocities, blue particles have small transverse velocities and white ones have moderate velocities.

For visualization using plots and charts there are a number of stand-alone applications that take raw data sets and produce graphs. These include applications such as Excel, Axum, and PsiPlot as well as programming and data environments such as Mathematica and Matlab. Existing component graphing packages may be utilized in RAD environments. One of the more popular packages is Olectra Chart, which ships as a set of ActiveX components. For creating static three-dimensional models there are a number of applications that employ the Virtual Reality Markup Language (VRML). This is a standardized language for describing three-dimensional scenes based on the notion of a *scene graph*.

3 An Architecture for Particle Beam Simulation

This section describes a possible architecture for a particle beam simulation system based on all the software techniques and technologies discussed previously. It is a system built on components communicating through interfaces. Each component is also designed according to similar principles. That is, they are as modular as possible with distinct sub-components and internal interfaces.

3.1 The Ensemble Container

We have already considered two possible structures for the container of particle objects, the array and the linked list. We have also mentioned that the array is not a particularly good choice, we prefer a container where particles can be added and removed without repairing the data structure. In this section, we discuss tree-type data structure, since their structure can be exploited in the space-charge calculations of particle beam simulations.

A possible implementation for a particle ensemble is a tree data structure. Figure 11 shows a particle ensemble implemented as a *binary tree*. In a binary tree, each data record is a node in the tree. Each node maintains pointers to two child nodes (right and left, for example). These child nodes contain, in turn, children of their own. Branches that contain no data are terminated with a null pointer. In this case, the data record is very similar to that of the linked list, with the addition of an extra pointer. Note, however, from the figures that the two topologies are very different.

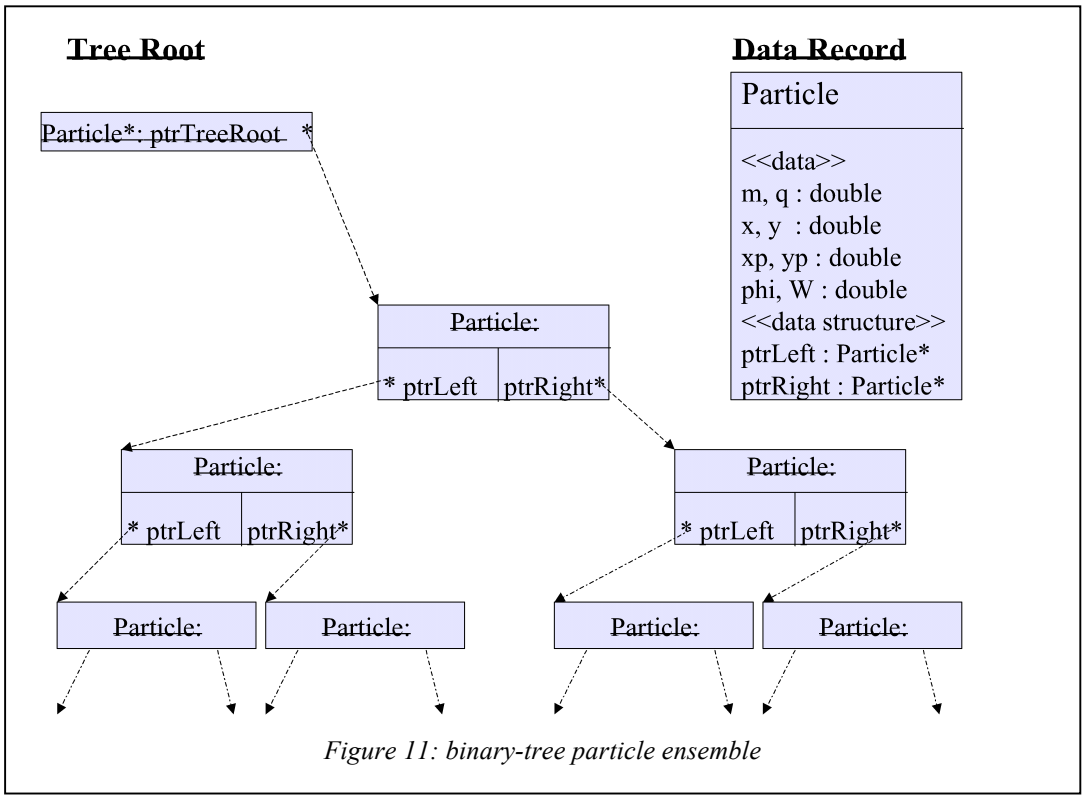

Figure 11: binary-tree particle ensemble

The binary-tree data structure has a self-similar topology. Such situations are highly suited to recursive iteration. For example, the entire tree could be searched with a single call to a recursive function such as that listed in Excerpt 3. In the code excerpt the boolean function TestFunc() is a function that returns true when called on the target particle. The recursive function TreeSearch() checks for this condition on each particle record, returning the particle if it passes TestFunc(). If the test fails, TreeSearch() uses recursion by calling itself on the left branch then the right branch. If the result of one of these calls is a Particle record, it is the desired particle. This particle is passed up the calling chain to the

original function call. Thus, to start the tree search, one need only call TreeSearch() on the root node of the tree.

```
Particle* TreeSearch(Particle* ptrNode)
{
        Particle*       ptrChild;  /* child node ptr */

/* Check if current particle (ptrNode) is desired particle */
        if ( true == TestFunc(ptrNode) ) return ptrNode;

        /* Does the left branch contain the desired particle? */
        ptrChild = TreeSearch(ptrNode->ptrLeft);
        if (ptrLeft != NULL) return ptrChild;

        /* Does the right branch contain the desired particle? */
        ptrChild = TreeSearch(ptrNode->ptrRight);
        if (ptrRight != NULL) return ptrChild;

/* Desired particle was not found underneath this node */
        return NULL;
};
```

Excerpt 3: binary tree search algorithm in C

Tree structures already have had success in particle simulations. Using an appropriately structured tree, the number of space charge calculations has been reduced from N^2 to $N\log N$ [5]. This feat uses the fact that particle dynamics depend upon the detailed field structure only for nearby particles. Particles more distant than the Debye length create the so-called "collective fields". (Particle-In-Cell codes only use this collective field, typically they do not consider the Coulomb collisions from nearby particles.) To exploit this physical feature, tree codes create data structures where a nearby node represents a nearby particle. The more distant two particles are in Euclidean space, the more distant they are in the tree structure. Thus, for remote nodes (particles) it is only necessary to consider their collective field.

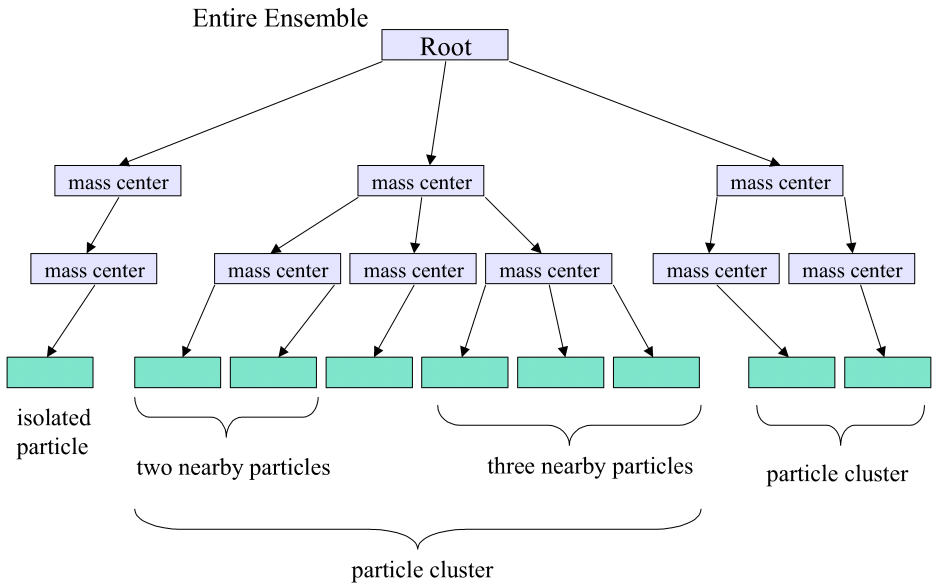

Figure 12: tree structure for particle tree code

Figure 12 depicts the structure of a typical tree used in particle simulation. This tree is different from the binary tree in that nodes may have multiple branches (rather than just two - implemented possibly with a linked list). In addition, particle records are stored only on the terminal nodes of the tree (called the tree *leaves*). The tree is constructed so adjacent particles are children of the same node. As one moves *up* the tree (from the leaves) the particle sets underneath the nodes become increasingly distant. After the tree is constructed, one computes the center of mass for a node and all its children (subset of particles). When determining the effects from distant particle clusters one assumes the field of one macro particle, positioned at the center of mass and having the aggregate charge. If more field structure is required from the clusters then one may also add in the fields of the dipole moment, quadrupole moment, etc. Obviously, the difficult part here is building the data structure. Once that is accomplished, the space charge computations are more-or-less trivial.

3.2 Interfaces

Here we lay out some of the important interfaces needed to assemble the components of the particle beam simulator. Since the components of a well-designed software system all use interfaces to communicate, they represent the glue that holds the application together. As already mentioned, the design of interfaces requires careful consideration. Once designed, they should never change.

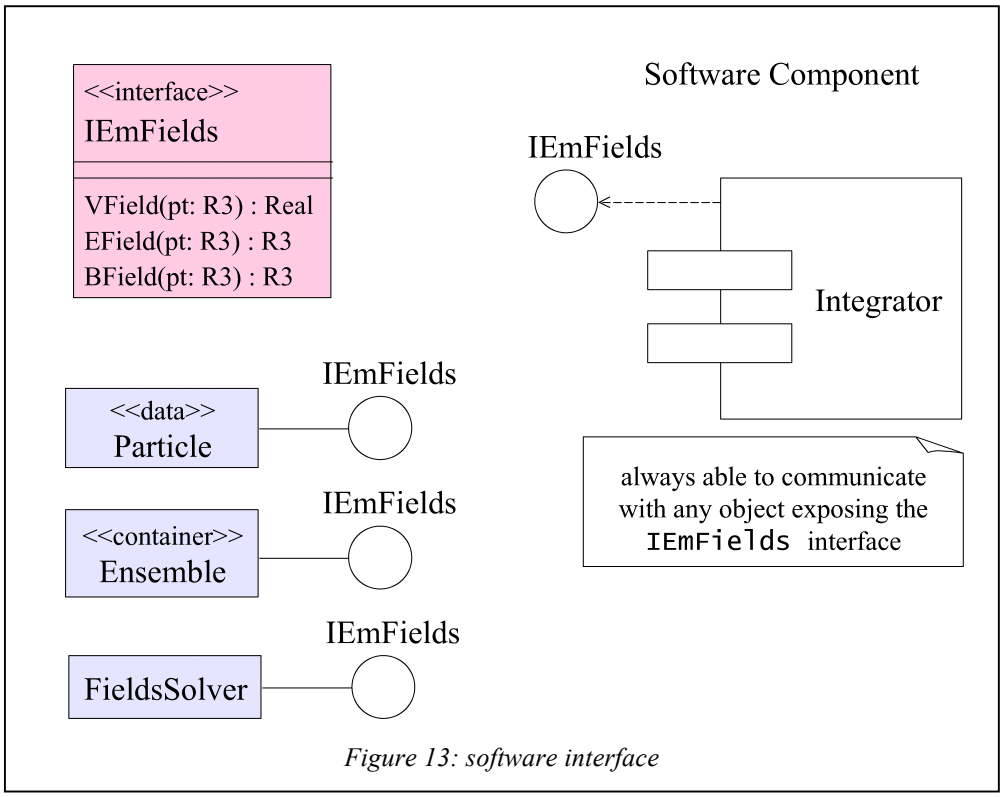

Figure 13: software interface

The schematic of a software interface used for electromagnetic behavior is shown in Figure 13. There we have defined an interface called IEmFields. This interface is seen to consist of three functions, VField(), EField and BField(), each taking an argument of type R3, representing a coordinate set in 3-space. The intent here to enforce the condition that any

software object with electromagnetic properties must express a potential field, an electric field and a magnetic field, respectively. This situation fits naturally into our notion of a physical object that exhibits electromagnetic behavior. Again, we point out that such design considerations are extremely good for maintaining clarity and modularity. Referring back to the figure, we have now equipped our Particle object with the IEmFields interface. This action informs the software domain that the Particle object conforms to the IEmFields standard for expressing electromagnetic behavior. Any Particle object must present the functions VField(), EField and BField(). Thus, our particles belong to the wide category of objects with electromagnetic properties.

Also shown in the figure are two more objects that exhibit the IEmFields interface, Ensemble and FieldsSolver. Since any ensemble of Particle objects must also have electromagnetic properties, the Ensemble object should also present the interface. The FieldsSolver object explicitly solves for electromagnetic fields, therefore conforming to the IEmFields interface is natural. The software component Integrator is shown in the figure to connect to an IEmFields interface. The component computes particle trajectories in the presence of an electromagnetic field. To the integrator, the source of the fields is irrelevant. The computations are carried out in the same manner irrespective. Thus, the trajectory calculations are not bound to any particular source of electromagnetic fields. The source may be an Ensemble object, a FieldsSolver object, or simply a Particle object. As long as an electromagnetic source presents the IEmFields interface, the Integrator component can compute trajectories. Note that in an accelerator simulation system, the object FieldsSolver might implement a particle-in-cell solution to the electromagnetic fields of an Ensemble object. In this case, the Ensemble object would actually support its IEmFields interface using a FieldsSolver object!

```
interface IEmFields {
        Real    VField(R3 pt, Real t);
        R3      EField(R3 pt, Real t);
        R3      BField(R3 pt, Real t);
};


class Particle implements IEmFields {
  private Real    m,q;
  private Real    x,y,z;
  private Real    xp,yp,zp;

  public Real VField(R3 pt, Real t)
  {
  ⋮
  };

        ⋮
};
```

Excerpt 5: Java IEmFields interface

```
class IEmFields {
        Real    VField(R3& pt, Real t) = 0;
        R3      EField(R3& pt, Real t) = 0;
        R3      BField(R3& pt, Real t) = 0;
};

class Particle : public IEmFields {
  private:
        Real    m,q;
        Real    x,y,z;
        Real    xp,yp,zp;
  public:
        Real    VField(R3& pt, Real t);
        R3      EField(R3& pt, Real t);
        R3      BField(R3& pt, Real t);
};

Real Particle::Vfield(R3& pt,
                Real t)
{
  ⋮
};
⋮
```

Excerpt 5: C++ IEmFields interface

Some computer languages, such as Java, provide direct support for software interfaces. For others, such as C++, interfaces can be enforced through appropriate constructs. Excerpt 5 shows a Java implementation for the IEmFields interface. Notice in the class

declaration for Particle the implements keyword explicitly enforces implementation of the interface. The three functions of the IEmFields interface are thus guaranteed to be a part of the Particle class. Also in Excerpt 5 we see the C++ implementation of an interface. The interface is defined via an abstract base class (containing only operations) that we identify as IEmFields. The three functions of the interface are made pure virtual (enforced by the = 0 qualifier). Thus, they must be defined in any instantiable subclass of IEmFields. For the Particle class, the interface is enforced through inheritance from the abstract base IEmFields. Unfortunately, since IEmFields is actually a class definition and inheritance is used to create the interface, it is not explicitly clear that our intent here is to express an interface.

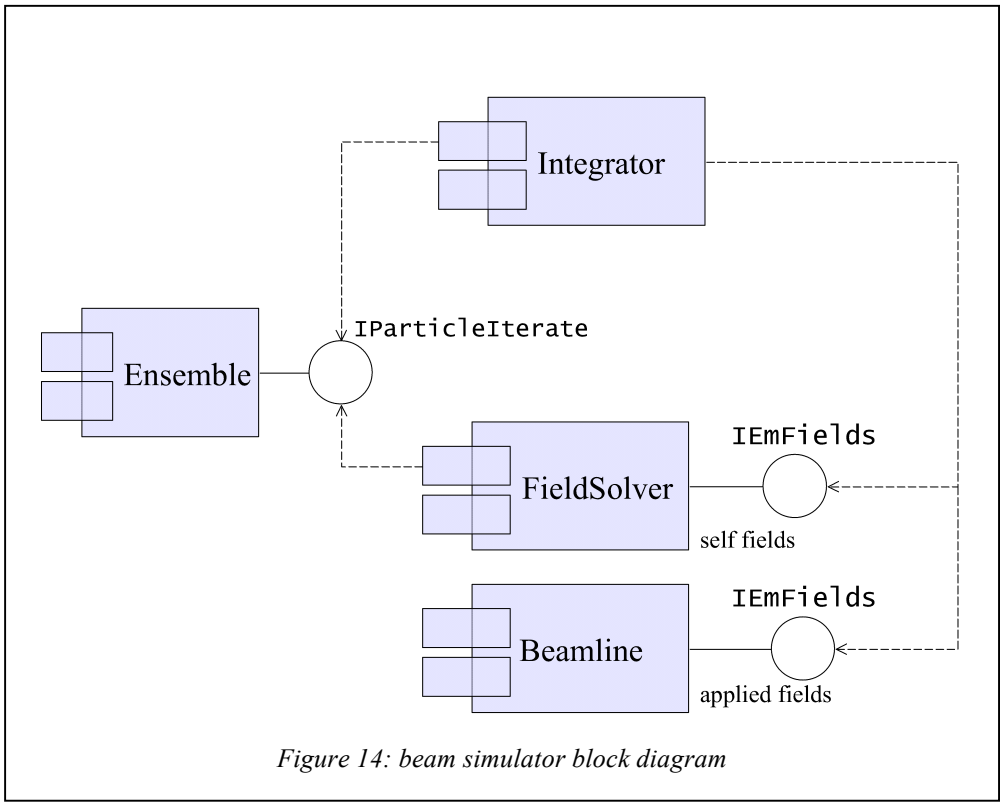

Figure 14: beam simulator block diagram

3.3 Components

The structure of a particle beam simulator naturally conforms to four major components. An addition component for data processing and visualization could also be attached via proscribed interfaces. Referring to Figure 14, the major components of the simulator are the following: 1) the particle ensemble, 2) the beam line modeler (i.e., the particle source, transport and accelerator systems), 3) the field solver and 4) the integrator. The particle ensemble component has been covered previously. It is some container of Particle objects. The beamline modeler is also a container; it contains objects representing beamline elements. The idea here is that the user should be able to create a beam channel by selecting and specifying individual beamline elements (preferably using point-and-click and/or drag-and-drop). The beamline elements are represented as software objects.

The function of the integrator is to advance the particle phase states in time. From the collective fields provided by the field solver component, it computes each particle's

position and velocity from the Lorentz force equations or some other method. This component is actually rather small in comparison to the others. However, it is important in that the actual integration methods be separate from the rest of the system. Since the performance of the various integration techniques varies from situation to situation, we should be able to plug in different integration modules on the fly (during run time).

The field solver is traditionally associated with the grid and grid cells of a PIC code, where the electromagnetic field values from the particle ensemble are interpolated and/or approximated. Note, however, the actual implementation of the field solver is hidden from the rest of the simulation system. All that is required from this component is the electromagnetic field values for any point in space. Therefore, the fields solver exposes the IEmFields interface to the outside world from which to acquire these values. Yet, within the module any form of field calculation is possible, we are not restricted to particle-in-cell schemes.

In Figure 14 it is important to see that the components are connected through the well-defined interfaces IEmFields and IparticleIterate (see Figure 7). The only deviation from this paradigm is that the integrator component must understand how to modify the coordinates of the Particle objects. In this case, the creation of a new interface may be over-engineering. This is the only place where the Particle object's data is actually accessed and modified (the one exception being a visualization component not shown). The Particle class is itself well defined. Since it will probably never change, relying on a consistent Particle object should present no future compatibility problems.

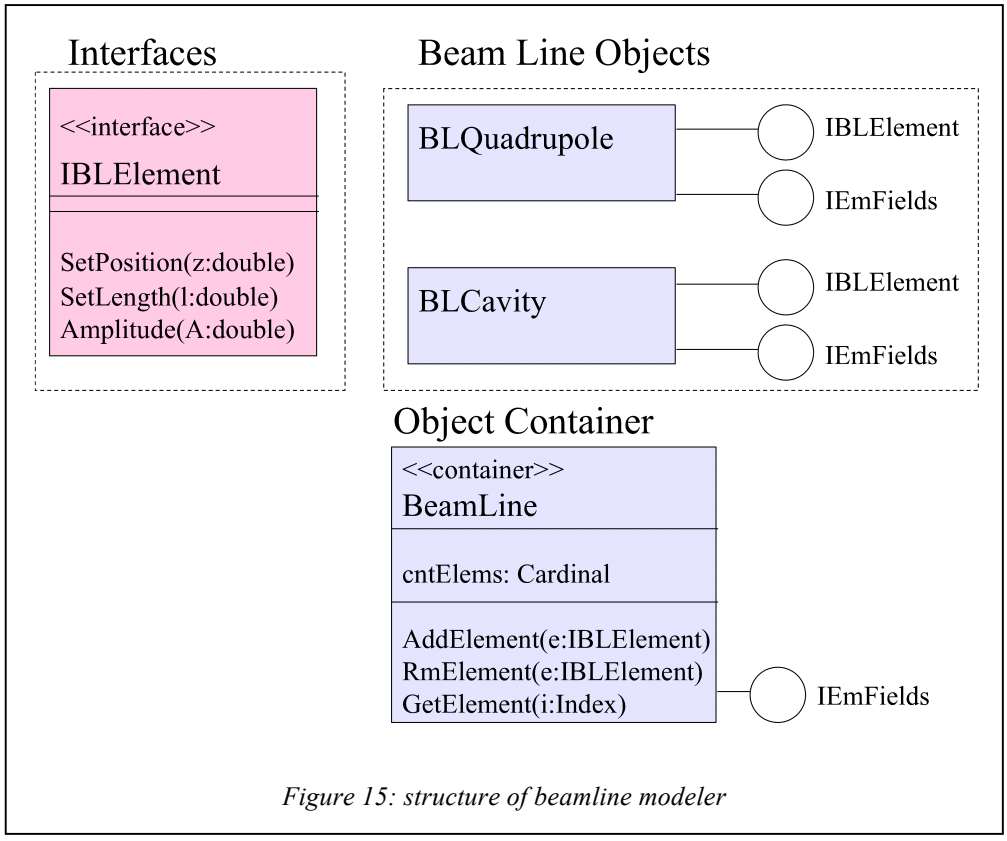

Figure 15: structure of beamline modeler

3.4 Components of Components

The individual components of the simulator are designed using the same philosophy as the overall system. Thus, the software components themselves are built up from software components. As an example, we consider some internal structure of the beamline modeler.

Referring to Figure 15 we see that the beamline modeler component defines internal interfaces and has components of its own. The interface IBLElement specifies the common properties associated with all elements in a beamline. In the figure, we see that this includes an axial position, an element length, and an amplitude of some type. All beamline elements are expected to present this interface. Two example beamline objects, BLQuadrupole and BLCavity are shown to do this in the figure. They also present the interface IEmFields because beamline elements are not useful unless they have electromagnetic properties. Presenting this type of uniform structure for beamline elements allows for the straightforward use of optimizers and other mathematical programming objects. For example, an optimizer could be used to tweak parameters of a subset of beamline elements until a particular condition is maximized [1].

In Figure 15, the object BeamLine represents the entire beamline of an accelerator. However, it is simply a container of IBLElement interface objects. Thus, BeamLine contains operations for adding and removing IBLElement objects as well as retrieving specific elements. The container should also provide operations for arranging the IBLElement objects, since accelerator components have a particular order. Thus, an appropriate container structure for the BeamLine object might be a linked list, or possibly a dynamic array.

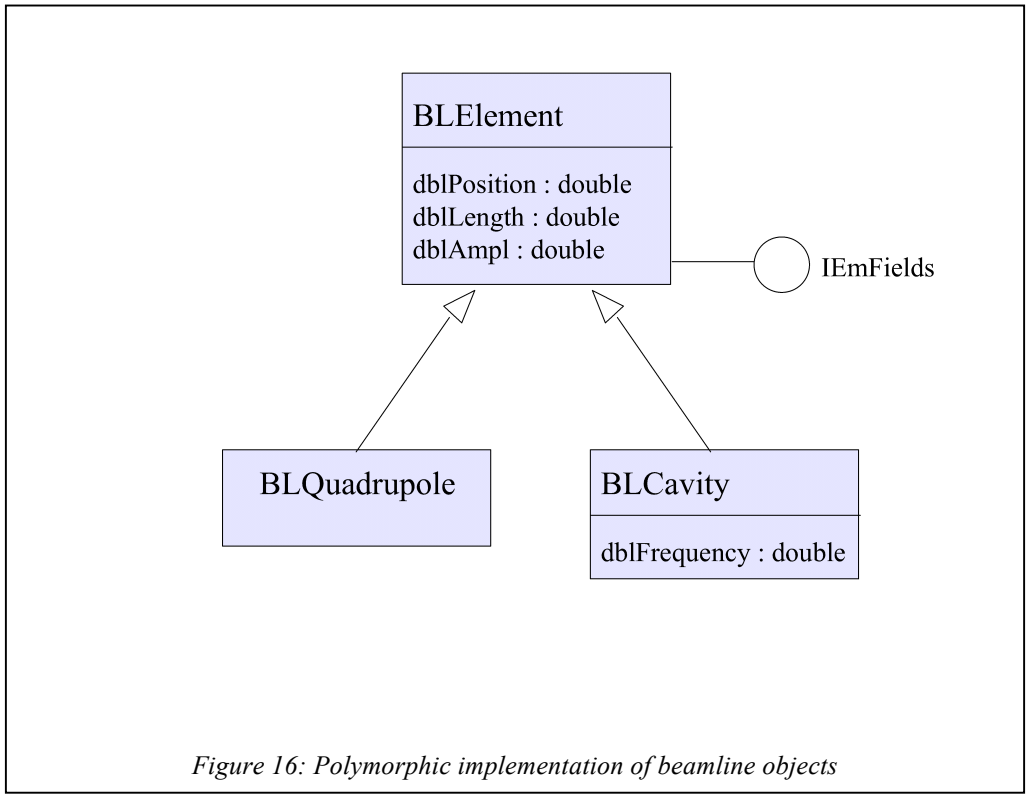

Figure 16: Polymorphic implementation of beamline objects

We point out that it is possible to implement the uniform properties of beam line elements without interfaces. The alternative technique involves the use of *inheritance* and *polymorphism*; this idea is depicted in Figure 16. In this case, we would implement a base class, say BLElement, which contains all attributes and operations common to beamline elements. The specific beamline elements are then derived from the base class, redefining their specific behavior and possibly adding attributes and operations particular to themselves. However, the common operations and attributes contained in the base class BLElement are the only ones useful to an optimizer. Requiring the optimizer to understand the structure of every possible beamline element is unrealistic and, therefore, of questionable design.

3.5 Component Operations

The operations within a component, that is the activity actually performed, can be modeled using UML activity diagrams. For example, we consider the operation of the Integrator component. The actual implementations of the integrator can vary greatly. It can be as simple as direct numerical integration of the equations of motion, say using Runge-Kutta techniques. Alternatively, it may use modern symplectic-integration techniques, or perhaps symplectic maps.

In Figure 17 we have the UML activity diagram for the integrator component. The integrator employs a split-operator technique to advance the particle states. Transfer maps for the space charge and the machine fields are computed separately, rather than computing a single map for the combined fields. The separate maps are then composed to simulate the action of both fields on the particle ensemble. In this manner, we only update either map whenever necessary, instead of continually rebuild the complete map whenever one of the update conditions is met.

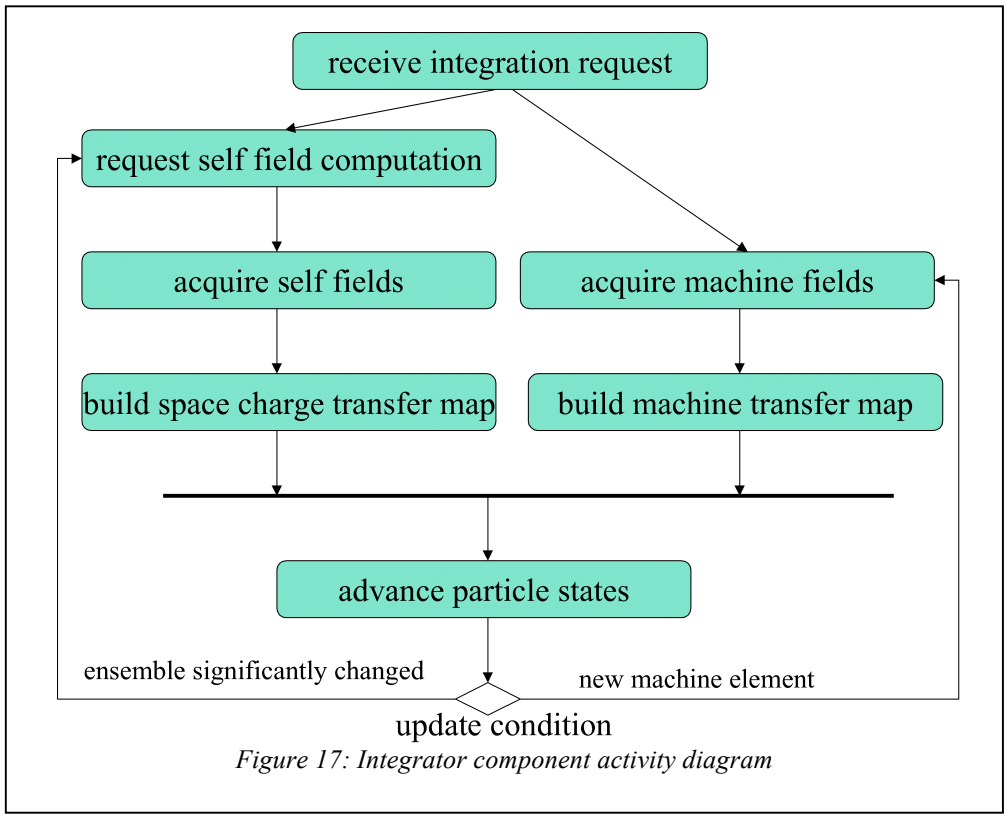

Figure 17: Integrator component activity diagram

In the activity diagram, the transitions on the left side concern the self-field calculations while those on the right represent the accelerator field computations. Both activities may be concurrent but are synchronized before the particle states are advances, this condition is represented by the thick horizontal bar. For example, say a new beamline element is encountered during the integration yet the particle ensemble does not significantly change its configuration. In that case, the fields of the new element are acquired and a new transfer map computed for the element. However, nothing is done on the left side of the diagram, that is, we simply wait at the synchronization bar with the same space-charge

transfer map. The reduced computational load is the motivation for the split operator here.

3.6 Component Object Packaging and Deployment

Here we briefly outline the topic of packaging and deployment. We include it because this is an important topic concerning upgradability and software distribution. With a well-conceived internal architecture, the task of packaging and deployment is greatly simplified. The components of our system can be packaged and distributed as individual software entities.

A convenient method for packaging software components is the use of dynamic link libraries (DLLs). The notion of a procedure library should be a familiar one, the FORTRAN libraries LINPACK, EISPACK, and the IMSL libraries are in wide distribution. Dynamic link libraries differ from static libraries in that they are bound (linked) at run time. If an application is bound at compile time, one large executable is the result. However, dynamic binding allows the application to be subdivided into component DLLs. If a procedure call to an external library is made at run time, the operating system locates the DLL, loads it into memory, and then executes the function call. Moreover, DLLs can contain data, that is, they can have a state. Thus, they themselves are concrete software objects.

Microsoft's ActiveX technology is a natural extension of dynamic link libraries. As mentioned, they are true software components with a well-defined interface and state. Thus, on Windows platforms a convenient way to package the above components is to compile them into ActiveX objects and/or controls. Once they are packaged as ActiveX objects, a variety of Rapid Application Development (RAD) tools may be used to create the actual beam simulators. Such tools include Visual Basic, J++ and even web page authoring tools such as Visual InterDev and FrontPage, since ActiveX objects are recognized by Microsoft's Internet Explorer. If the components were implemented in Java, the analogous procedure would be to create Enterprise JavaBeans in order to package the system as software components. In that scenario the components would be portable across platforms and not restricted to the MS Windows environment. One may assume that eventually the Java virtual machine will be efficient enough to practically implement CPU intensive software.

4 Conclusion

A particle beam simulator is typically a large, complex software system. When implementing these applications a great deal of time and effort can be saved by considering the architecture of the system, before any code is written. A well conceived design lends itself well to future upgrades and enhancements. This condition is extremely import since upgrades, modifications and re-specifications, integration, and refactoring (changing the implementation details) are usually where most of the development time is spent, not in the original implementation phase.

The ideas presented here are guidelines for the implementation of robust particle simulators. Accurate particle-simulation techniques have already been developed, are described in the literature, and are currently in use for accelerator design [9]. However, adapting these codes for new applications, or simply modifying them, can be extremely

time consuming. By employing these proven simulation techniques, along with the software techniques covered herein would yield a robust system that could be easily adapted to serve a varied compliment of accelerator applications.

REFERENECES:

[1] C.K. Allen and M. Reiser, "Optimal Transport of Particle Beams" *Nucl. Inst. and Meth. A*, Vol. 384, pp. 322-332 , Jan. 1997.

[2] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modeling Language User Guide* (Addison-Wesley, Reading, MA, 1999).

[3] K. Brockschmidt, *Inside OLE*, Second Edition (Microsoft Press, Redmond, WA, 1995).

[4] T. H. Cormen, C.E. Leiserson and R.L. Rivest, *An Introduction to Algorithms* (MIT Press, Cambridge, MA, 1990), Chapt 11.

[5] L. Hernquist, "Performance Characteristics of Tree Codes", *Ap. J.* (*Astrophysical Journal Supplement Series*) Vol. 64, No. 4, pp. 715-734.

[6] B.W. Kernigan and R. Pike, *The Practice of Programming* (Addison-Wesley, Reading, MA, 1999).

[7] P. Niemeyer and J. Peck, *Exploring Java* (O'Reilly, Sebastopol, CA, 1997), Chapt 18.

[8] D. Rogerson, *Inside COM: Microsoft's Component Object Model* (Microsoft Press, Redmond, WA, 1997).

[9] R. Ryne (Editor), *AIP Conference Proceedings 297, Computational Accelerator Physics* (AIP Press, New York, 1993).

[10] W.J. Schroeder, K.M. Martin and W.E. Lorensen, *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics (Second Edition)* (Prentice-Hall PTR, Upper Saddle River, NJ, 1998), Chapt. 2.

[11] M. Woo, J. Neider and T. Davis, OpenGL Programming Guide, Second Edition (Addision-Wesley, Reading, MA, 1997).

[12] J. Zabczyk, *Mathematical Control Theory: An Introduction* (Birkhäuser, Boston, MA, 1992), Chapt 3.

CKA

Distribution: