UAL 3: ASPECT-ORIENTED APPROACH

N. Malitsky, Brookhaven National Laboratory, U.S.A.
R. Talman, Cornell University, U.S.A.

Abstract

The paper presents the next step in the evolution of the Unified Accelerator Libraries (UAL). The existing version is base on an object-oriented framework addressing a broad spectrum of offline modeling applications ranging from configurable efficient integrators to full-scale realistic beam dynamics studies encompassing multiple physical effects. Accumulated experience with various projects has validated the UAL framework and outlined its relationship with the new aspect-oriented paradigm (AOP). As a result, the UAL3 version is designed to develop and apply the AOP Conceptual Reference Module in the context of the accelerator computational physics domain.

BACKGROUND

The design and operation of modern accelerators such as nuclear colliders and synchrotron light sources requires sophisticated, flexible and powerful modeling software. On the one hand, the complex problems that have to be studied require non-standard modeling techniques, such as tracking two beams, dealing with complex alignment tolerances for triplet assemblies, analysis of various insertion devices, etc. On the other hand, large accelerators are becoming international collaborative efforts, resulting in the consolidation of various programs in the unified environment aiming to facilitate the development and sharing of the most effective algorithms and approaches. Moreover, stringent parameters of modern high-intensity machine impose new expectations on beam dynamics studies and usually require the combination of several physical effects and processes.

The central part of this modeling environment is an internal representation of the accelerator system. The accelerator is a complex device combining many elements of different physical types with heterogeneous attributes, all organized in a nested hierarchical structure. The complexity of this organization prompts a variety of project-specific views and implementations of accelerator descriptions. Recently, the new version of the Accelerator Description eXchange Format (ADXF [1]) was introduced to provide a uniform, complete, and extensible model in the definition of the accelerator state. Its concepts are derived from experience with numerous accelerator applications and the generalization of several lattice formats, such as MAD/SIF, SMF, SXF, and others. A significant feature distinguishing this model from earlier models is its segregation of element positioning information from position-independent element properties, such as magnet strengths. The model is built from five main entities (see Figure 1):

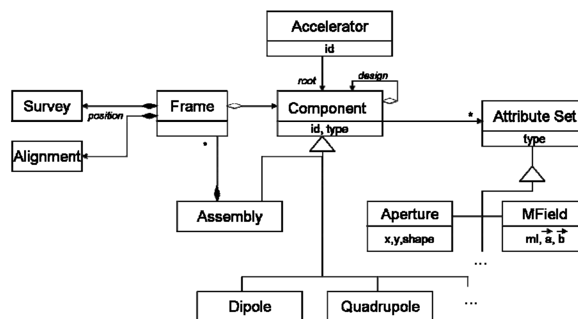

Figure1: ADXF 2.0 accelerator model.

- An accelerator is any accelerator component selected by the user
- An accelerator component is a node in the accelerator tree organization. There are many different types of lattice components (e.g. dipole, quadrupole, sector, etc). But all of them have the same structure: name and open collection of attribute sets. A component may have a reference to its design component.
- An accelerator component assembly is a named sector or composite element with a sequence of frames with installed accelerator components and insertions.
- An accelerator component frame is a layout of installed component. It contains a relative position, misalignments, and a reference to an associated component, sector or accelerator element.
- An accelerator component attribute set is a container of attributes relevant to the single physical effect of feature (e.g. magnetic field, aperture, etc.)

In contrast with other approaches, this accelerator model does not contain any application-specific attributes and serves as a joint point among various processing algorithms. This separation of propagating algorithms from accelerator elements embodies the main architectural principle of the Unified Accelerator Libraries (UAL) environment, enabling one to apply a variety of different simulation approaches to the same accelerator lattice. In the present version, the element-algorithm associations are built by the UAL propagator framework using Accelerator Propagator Description Format (APDF[2]). One can consider the APDF files as compliments to the conventional MAD or ADXF lattice input files. For example, the following APDF file represents the original TEAPOT element-by-element tracking engine:

```
<link algorithm=”TEAPOT::DriftTracker”
    types=”Default” />
<link algorithm=”TEAPOT::DriftTracker”
    types=”Marker|Drift” />
```

```
<link algoritm="TEAPOT::DipoleTracker"
    types="Sbend" />
<link algorithm="TEAPOT::MltTracker"
   types="Quadrupole|Sextupole|Multipole" />
<link algorithm="TEAPOT::MltTracker"
    types="[VH]kicker|Kicker" />
<link algorithm="TEAPOT::RfCavityTracker"
    Types="RfCavity" />
<link algorithm="TEAPOT::DriftTracker"
    types="[VH]monitor|Monitor" />
```

According to this description, the UAL propagator builder associates the different MAD types of accelerator elements (e.g. Quadrupole) with the corresponding classes of the TEAPOT propagators (e.g. TEAPOT::MltTracker). In the next example, the original TEAPOT simulation model is extended with the application-oriented class MIA::BPM that collects turn-by-turn BPM data for the subsequent SVD analysis:

```
…
<link algorithm="MIA::BPM"
 types="[VH]monitor|Monitor" />
```

As illustrated by these two examples, the APDF-based approach promulgates the rapid development of new applications and facilitates extensions of the original modules. Recently, similar issues have been addressed by the Aspect-Oriented Programming (AOP [3]) paradigm aiming to define a formal generic approach for integrating crosscutting extensions (concerns) into object-oriented software. From its perspective, the connection of propagators with accelerator elements can be considered as the model-specific concerns which crosscut through the entire accelerator structure. As a result, the UAL 3 aims to explore, develop, and apply the AOP Conceptual Reference Model [4] in the context of accelerator modeling applications.

The remainder of this paper is structured as follows. Section 2 presents the object-oriented Visitor-based solution [5] for adding crosscutting propagators to the accelerator structure. Section 3 considers the aspect-oriented variant based on Wu and colleagues' procedure [6]. Section 4 introduced the Mutable Class approach [7] implemented in the initial version of the UAL framework. Finally, section 5 introduces the design of the UAL 3 version based on the language-neutral implementation of the AOP model.

VISITOR APPROACH

Hierarchical heterogeneous trees represent the natural models of many software applications, such as the abstract syntax tree (AST) of compiler systems, the scene graphs of visualization toolkits, and others. Processing of these models in the object-oriented domain is addressed by the Visitor pattern [5]. This pattern groups the different types of heterogeneous structure-oriented operations into separate classes and provides a consistent mechanism for their interchange. In the context of accelerator modeling software, the Visitor pattern facilitates the development of algorithm-specific scenarios by separating the different types of processing algorithms. The corresponding structure diagram is shown in Figure 2.

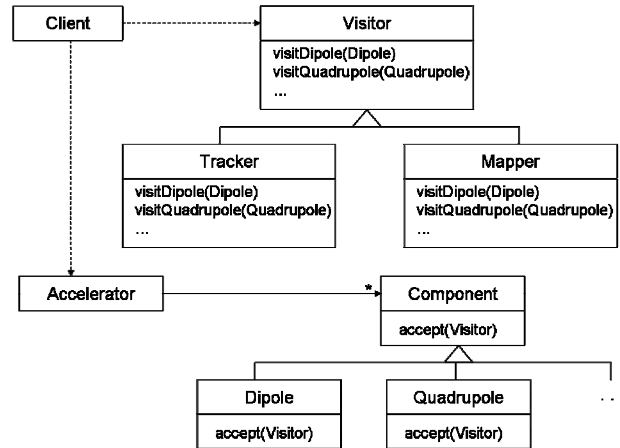

Figure 2: Diagram of the Visitor Pattern [5] in the context of the accelerator modeling environment.

According to the diagram, traversal algorithms are implemented with the two-dimensional collection of classes and visitors, differentiated by traversal categories and types of the processed objects. The visitors from the same traversal categories are encompassed into new classes, Tracker or Mapper, for processing the entire model structure. Each element of the model structure is algorithm-free and has a general *accept* method for passing itself to the appropriate visitor according to the double-dispatch mechanism. The Visitor pattern however introduces a serious limitation. It freezes existing class hierarchies and prevents any extensions of the processed tree structure. In particular, adding a new type of the accelerator element would require editing all visitor classes.

There have been several attempts aimed to resolve the problem of the original Visitor pattern. Within the object-oriented domain, the Acyclic Visitor pattern [8] suggested the most consistent alternative approach for breaking the dependency cycle with multiple inheritance. Inheritance however is a static mechanism and results in strong coupling between components.

ASPECT-ORIENTED APPROACH

Recently, a new Aspect-Oriented Programming (AOP) paradigm has introduced several ideas addressing the extensibility issues of object-oriented software. AOP originated from several related ideas, eventually becoming a consolidated core of many similar paradigms, including adaptive programming, composition filters, multidimensional separation of concerns, subject-oriented programming, and others.

The original term was introduced by Gregor Kiczales and colleagues in their report at a European Conference on Object-Oriented Programming [3]. Based on the

analysis of several applications, the report identified functional properties crosscutting the basic system's structure and suggested that they represented some new concept which was orthogonal to the conventional component-oriented view of the software design. Since such properties crosscut the system's basic functionality they could not be cleanly encapsulated in the existing programming languages. To address this problem, the authors suggested the AOP-based composite implementation consisting of three parts: the conventional component program, the aspect program implementing crosscutting concerns and an aspect weaver for combining both component and aspect modules.

Subsequent studies and aspect mining have confirmed the original motivation and identified crosscutting concerns in different applications. The scope of them is quite broad and expanding - security, logging, persistence, debugging, distribution and others. The Visitor-like accelerator modeling applications appear to fall very naturally in this category since they deal with many AOP concepts. Particularly, the Visitor pattern separates processing operations from processed data structures and combines related operations into the Visitor subclass. In AOP, these operations can be considered as crosscutting behavior of associated tree nodes and can therefore be represented by the corresponding construct. For example, the application of aspect-orientation to the processing of heterogeneous abstract syntax trees (AST) is discussed by Wu and colleagues [6].

Similar to the accelerator modeling environment, AST serves as an internal and intermediate representation of the source program during the different phases of the compilation process including context checking, optimization, and code generation. Depending on the applied algorithms, each compilation phase introduces an additional set of requirements. Moreover, a set of the compiler algorithms is not fixed and can vary according to the complexity of the programming languages and their implementations. Adhering to the Visitor pattern as a strategic direction, Wu and colleagues consistently developed its aspect-oriented version for working with AST. The suggested aspect-oriented approach was implemented in the AspectJ programming language and proof tested in a case study of the proprietary RelationJava compiler.

Following Wu and colleagues' approach, the procedure for connecting accelerator modeling algorithms with the accelerator structure would consist of three steps. First, the concrete visitor classes, Tracker and Mapper, should be rewritten as the corresponding abstract aspects with the common helper routines. In the second step, every *visit* method for each type of the accelerator elements should be implemented as the *propagate* method in the corresponding type-specific and algorithm-specific aspects (see Figure 3). These fine-grained aspects of each category of algorithms would be derived from the basic aspects, Tracker or Mapper, and inherited its helper routines.

```
abstract  aspect Tracker {
   public void  AcceleratorComponent
                      propagate(Bunch bunch) {
         …
    }
}
aspect DipoleTracker extends Tracker {
    public void Dipole.propagate(Bunch bunch){
        …
     }
}
```

Figure 3: The Tracker and DipoleTracker aspects

The third step would introduce the LossCollector aspect which allowed to extend the original algorithms with the additional behavior based on the AspectJ dynamic joint point model (see Figure 4).

```
aspect LossCollector {

  poincut checkAperture(AcceleratorComponent ac) :
        target(ac) && call (* *.propagate(Bunch));

after(AcceleratorComponent ac): checkAperture(ac) {
          ….
     }
}
```

Figure 4: The LossCollector aspect

This application however exposes an important issue associated with the run-time behavior of applying aspects and its comparison to the traditional plug-in mechanism, an issue that is especially important in the multi-stage scenarios. According to the AspectJ specification, inter-type declarations, which allow one to add members and methods across multiple classes, operate statically, at compile-time. On the other hand, the object-oriented patterns, like Visitor, can be transparently managed in run-time.

MUTABLE CLASS APPROACH

In the present UAL framework, the limitation of the Visitor pattern was solved by another object-oriented pattern, Mutable Class [7]. Mutable Class was derived from a combination of two design patterns: Type Object [8] and Strategy [5]. The Type Object encapsulated the common class data in a singleton of the additional class, the so called Type Class or Class Type. In the Mutable Class variant, this singleton also maintains the behavior of the class objects. A Strategy encapsulates the implementation of this behavior into separate classes and provides the mechanism for their interchange. The Mutable Class model is consistent with the UML semantics and can be considered as the implementation of the Instance-Class-Operation association [9]: "An operation is *owned* by a class and may be invoked in the context of objects that are instances of that class." The Strategy pattern does not change this ownership and only extends it with the flexible and dynamic features. The corresponding structure diagram is shown in Figure 5.

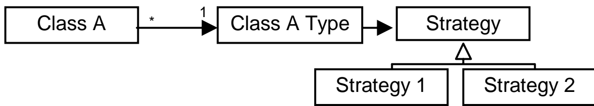

Figure 5: Structure of the Mutable Class pattern.

To manage the behavior of complex structures, the Class Type and Strategy instances can be grouped in Registries. Figure 6 shows a class diagram that illustrates relationships among the different Mutable Class components in the case of the accelerator heterogeneous model.

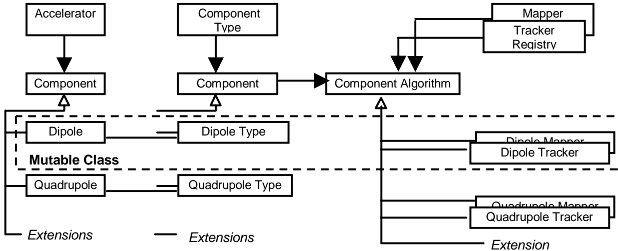

Figure 6: Structure of the Mutable Class-based framework for traversing heterogeneous accelerator structures.

According to the Mutable Class approach, each node of the accelerator structure is associated with the corresponding class type which maintains a pointer to the Component Algorithm instance. The accelerator traversing procedure does not access this instance directly and delegates the request via the *propagate* method. Drawing an analogy with the Visitor pattern, the Mutable Class approach replaces the Visitor run-time selection mechanism with prior binding. According to the aspect-oriented terminology, the Class Type serves as a joint point between the extent of the accelerator nodes and the woven algorithm. On the other hand, this weaving procedure is not limited by compile-time as in the case of aspect-oriented approaches and preserves the run-time behavior of the Visitor pattern. As a result, the Mutable Class represents a composite solution combining the advantageous features of both the Visitor pattern and the aspect-oriented weaving procedure.

UAL3 COMPOSITE APPROACH

The proposed UAL 3 framework is logically derived from the Mutable Class pattern. In this approach, the mutability concept is transferred from a class to individual objects and correspondingly the Mutable Class is transformed into the Mutable Object (see Figure 7).

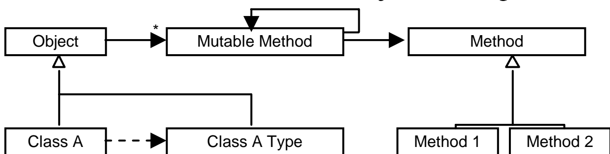

Figure 7: Structure of the Mutable Object pattern.

According to the Mutable Object pattern, each object may contain a collection of Mutable Methods. Like Mutable Class, Mutable Method is designed after the Strategy pattern and maintains a reference to the actual method, an instance of the Method class. In addition to this reference, Mutable Method can propagate the request to its delegate. By default, the Mutable Object pattern implements the Mutable Class scenario. In this case, only the Class Type singleton (e.g. ClassAType) allocates a collection of Mutable Methods. Then objects of the corresponding class (e.g. ClassA) delegate their requests to this singleton.

By adding the AOP Conceptual Reference Model [4], the Concern Composition package can select a collection of objects and assign their delegates to the Behavioral Advices of the Joint Points (see Figure 8).

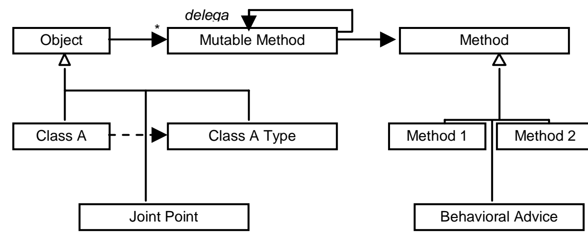

Figure 8: Integration of the AOP Conceptual Reference Model with the Mutable Object pattern.

This scheme also provides the implicit partial support of structural extensions that can be associated with the corresponding instances of the Method classes. The collection of Mutable Methods is designed after a virtual method table, a mechanism used in object-oriented languages for supporting run-time method binding. In UAL, it will be implemented in the framework layer. In principle, the mutability feature can be naturally added into the method declaration of the programming languages. As a result, this approach will also resolve the eligibility issues associated with the AOP-based changes of the original source code.

REFERENCES

[1] N.Malitsky and R.Talman, “Accelerator Description Formats,” ICAP’06

[2] N.Malitsky, T.Satogata, and R.Talman, “Configurable UAL-Based Modeling Engine for Online Accelerator Studies,” PAC’ 03

[3] G. Kiczales et al., “Aspect-Oriented Programming, “ ECOOP’ 97

[4] A. Schauerhuber et al., “A Survey on Aspect-Oriented Modeling Approaches,” Technical Report, Vienna University of Technology, 2006.

[5] E.Gamma, R.Helm, R.Johnson, and J.Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison-Wesley Professional, 1995.

[6] X. Wu et al., “Separation of concerns in compiler development using aspect-orientation,“ ACM Symposium of Applied Computing, 2006

[7] N.Malitsky and R.Talman,.”Framework of Unified Accelerator Libraries,” ICAP 98

[8] R.Martin, D.Riehle, and F.Buschmann, “Pattern Language of Program Design 3,” Addison-Wesley, 1997

[9] OMG, “Unified Modeling Language, Infrastructure, V2.1.2,“ OMG document number: formal /2007-11-04