THE FRAMEWORK OF UNIFIED ACCELERATOR LIBRARIES

Nikolay Malitsky and Richard Talman
Newman Laboratory, Cornell University, Ithaca, NY 14853

Abstract

An overview of the framework of Unified Accelerator Libraries (UAL) is presented. The UAL framework is a necessary and logical step in the UAL evolution. It intends to offer a standard-based infrastructure that facilitates compatible and independent implementation of accelerator applications and the development of interoperable Accelerator Simulation Facilities. The paper explain the two major framework components: a uniform mechanism for assembly and reuse of independently developed accelerator algorithms and a uniform infrastructure for optimization and correction approaches.

1 RATIONALE

Object-oriented class libraries have provided an important advance in the simulation of accelerator performance but their effective exploitation will rely on the environment or "framework" in which they are employed. In comparison with class libraries, frameworks offer additional benefits: strong "wired-in" interconnection between classes, customization mechanisms, and default behavior and architectural guidance for applications. The strong infrastructure significantly eases the implementation, management, and deployment of critical domain solutions, decreases the amount of standard code that a developer has to program, test, and debug, and frees the scientist from software design to apply his or her expertise on project-specific applications.

The Unified Accelerator Libraries (UAL[1]) have been designed as an open highly flexible environment built from diverse loosely-coupled components. This approach has facilitated the development of new accelerator applications and the selection of the most effective accelerator physics algorithms, analysis and design patterns. The accumulated experience in developing UAL applications, emerging new component-oriented technologies and enterprise-level distributed frameworks and systems have enabled the implementation of the UAL framework, its analysis and design patterns. The UAL framework is a necessary and logical step in the UAL evolution. It is simultaneously a product of the UAL environment and a tool for the development of a UAL-based Accelerator Simulation Facility. The next sections explain in detail the two major framework components: a uniform mechanism for assembly and reuse of independently developed accelerator algorithms and a uniform infrastructure for optimization and correction approaches.

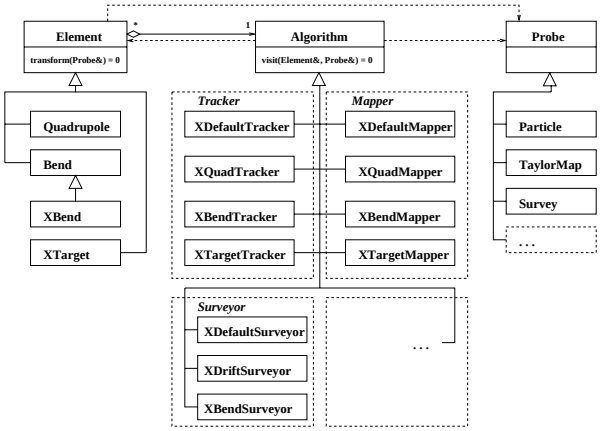

Figure 1: Structure of the Element-Algorithm-Probe analysis pattern.

2 UNIFORM MECHANISM FOR ASSEMBLY AND REUSE OF INDEPENDENTLY DEVELOPED ALGORITHMS

The main architectural principle of the Unified Accelerator Libraries is a separation of physical entities and mathematical abstractions from algorithms. The diverse accelerator algorithms are implemented as classes that share data via Common Accelerator Objects (Particle, Twiss, Element, and others). This distributed open architecture has promoted the introduction of the Element-Algorithm-Probe analysis pattern (see Fig. 1).

According to the Element-Algorithm-Probe analysis pattern, Common Accelerator Objects are divided into two categories (Element and Probe), and accelerator algorithms are implemented as separate interchangeable classes [1]. This structure is built after the Strategy design pattern [2] that lets one vary algorithms independently of their context, making them easier to switch, understand, and extend. The Strategy pattern provides many benefits by offering an alternative to subclassing and conditional statements for selecting desired behavior. However, it does not determine a mechanism for selecting algorithms relevant to the concrete element type. There is the Visitor design pattern [2] that declares the $visit$ operation for each type of the concrete element and lets one define a new operation without affecting class hierarchies. A Visitor gathers related operations and makes adding new operations easy. However, it prevents subclassing new element types and freezes existing element class hierarchies. Despite the simplicity and

[1]Here we are introducing a (unconventional) view of the particles or beams as "probing" the accelerator structure.

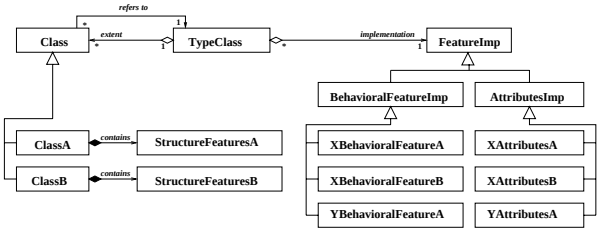

Figure 2: Structure of the Mutable Class design pattern.

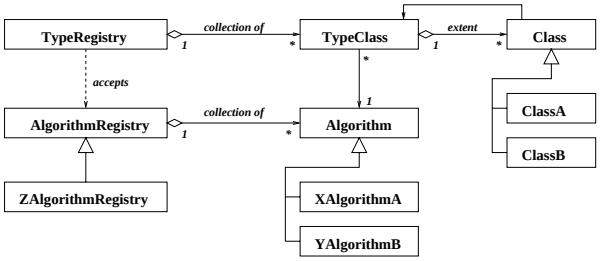

Figure 3: Structure of the Mutable Class Type Registry.

power of the Visitor patter, this drawback makes it unacceptable for open scientific applications. To address this problem we have derived the Mutable Class design pattern (see Fig. 2).

A Mutable Class is a combination of two design patterns, a Type Object [3] and a Bridge [2]. A Type Object divides a class into two parts, a type class and instance class, and an object's behavior is delegated to its type. A Bridge decouples an abstraction from its implementation so that the latter can be varied independently. This structure offers an alternative approach to subclassing and results in a highly flexible system configurable at run-time. On the other side, the Mutable Class model is well defined and can be considered as the physical implementation of the Class structural element described in the following Unified Modeling Language (UML) Semantics[4]:

"A Class defines the data structure of Objects, although some Classes may be abstract, i.e no Objects can be created directly from them. Each Object instantiated from a Class contains its own set of values corresponding to the StructureFeatures declared in the full descriptor. Objects do not contain values corresponding to BehavioralFeatures or class-scope Attributes; all Objects of a Class share the definition of the BehavioralFeatures from the Class, and they all have access to the single value stored for each class-scope attribute".

The TypeClass instances can be grouped together in the uniform collection and be controlled by the separate object, a Type Registry (or Finder). Registry services have been included in many complex design patterns and industrial frameworks. They deal with issues related to querying operations on managed objects. The Mutable Class Type Registry extends its functionality by providing the additional mechanism for dispatching and binding algorithms with the corresponding TypeClass instances (see Fig. 3):

An AlgorithmRegistry is preferable to a Visitor for open configurable systems. One can consider it as an alternative approach to the Acyclic Visitor pattern [3] that intends to break the dependency cycle of the original Visitor pattern using multiple inheritance. Inheritance is a static mechanism and results in strong coupling between components. The Mutable Class pattern offers a different approach based on a powerful dynamic composition technique and dividing a visiting process into two sequential steps:

1. dispatching and binding algorithms with the TypeClass instances; (in the UAL framework, it corresponds to binding accelerator algorithms with the twenty to thirty ElementType instances).

2. performing operations on elements of the complex structure; (for accelerator applications, the number of operations is determined by the lattice size and the number of repetitions and ranges from 500 to infinity.)

Implementation of the UAL framework based on the Element-Algorithm-Probe analysis pattern and the Mutable Class design pattern will create a universal mechanism for sharing, mixing, replacing, versioning, and extending diverse accelerator algorithms and will support the following actual applications:

- inclusion of new element types or physical effects (*combined function magnet (FNAL Recycler), superposition of solenoid and quadrupole fields (CESR), parasitic beam-beam effects (LHC, CESR)*);

- uniform access to accelerator data from diverse algorithms (*conventional tracking, spin, space charge, etc.*);

- optimal single accelerator representation for lattice design and performance modeling (*optimal combination of "thick" and "thin" elements*).

The proposed structure is intended to facilitate the extensions and maintenance of existing UAL components. For examples, the TEAPOT algorithms for different element types are represented by protected class methods and are selected by the engine according to the set of element attributes. In the Element-Algorithm-Probe framework, the TEAPOT monolithic engine will be split into an extendible collection of tracking algorithms. This promotes the (independent) development of new algorithms for propagating different accelerator "probes", such as Spin, Space Charge, and others.

3 UNIFORM INFRASTRUCTURE FOR OPTIMIZATION AND CORRECTION APPROACHES

The accelerator life cycle has several successive stages: design, construction, commissioning, and operation. In accelerator terminology, optimization procedures correspond to the early design phase, and correction is associated with

accelerator commissioning and operation. Different applications of these approaches result in the principle distinctions in their structure and algorithms. Optimization modules have been implemented as parts of accelerator simulation programs and have access to all kinds of information that is unavailable from the real accelerator. Usually, all data exchanges are hidden inside the simulation code "input language". The correction modules depend on the accelerator environment (hardware, operating system, *etc.*) and perform actual compensation procedures based only on detector measurements. Despite these differences, both approaches can be described by the same formula: *adjust correction elements in such a way as to minimize the deviation from required behavior measured in detector elements.* The UAL framework intends to use this formula for building a uniform infrastructure for optimization and correction approaches. A single environment for developing simulation and operational software would significantly facilitate its implementation, improve the quality and reliability of algorithms, and promote the exchange and integration of diverse approaches.

TEAPOT [5] is an operational code that was targeted to implement a universal operational algorithm [6] based on the internal accelerator simulator. This approach and these algorithms were successfully reused in the working prototype of the High Level Control Code for the SSC Low Energy Booster [7]. However, the constraints of the FORTRAN and C languages in which the original TEAPOT versions were written, prevented the direct integration of the TEAPOT libraries with operational control systems. Now, modern object-oriented technologies promote the modular structure of accelerator codes. A typical accelerator application can be represented by several cooperating components (Accelerator Simulator, Control, Interface, *etc.*) that communicate via Common Accelerator Objects (Detectors, Adjusters, *etc.*) (See Fig. 4). The UAL framework will provide the services for building these objects and the reliable infrastructure and mechanism to support their operation. In the present C++ version of the TEAPOT program, all correction algorithms are represented by separate class-services. Each service has direct access to the lattice elements and internally selects families of adjusters and detectors. In the new scheme, TEAPOT algorithms will interact with UAL systems via self-defined external Adjusters and Detectors instances and will be organized as replaceable modules of the Control component. This distributed architecture of the UAL environment will reflect the general structure of the accelerator control system and will make straightforward the development of an off-line Accelerator Simulation Facility and the conceptual model of an online Accelerator Simulation Facility (See Fig. 4).

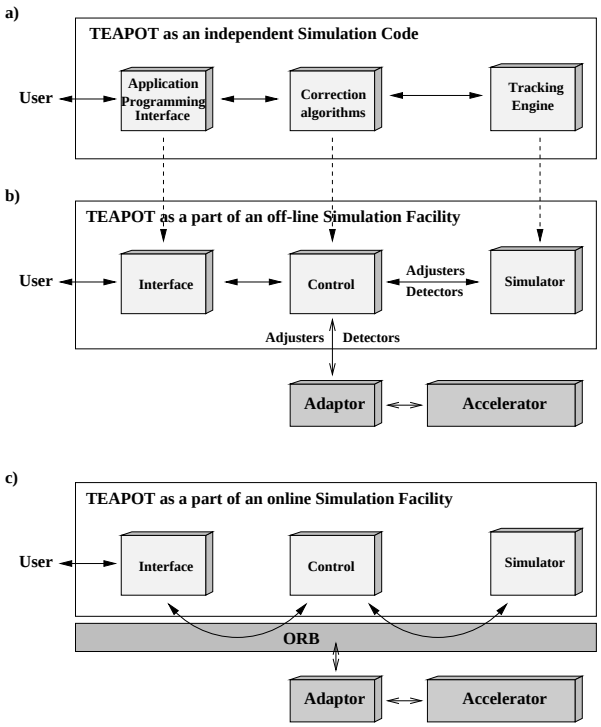

Figure 4: a) An independent Simulation Code vs. b) an off-line Simulation Facility vs. c) an online Simulation Facility (An off-line simulation facility will be the application of the UAL framework).

4 REFERENCES

[1] N.Malitsky and R.Talman: *Unified Accelerator Libraries*, AIP 391, Williamsburg, 1996

[2] E.Gamma, R.Helm, R.Johnson, and J.Vlissides: *Design Patterns: Elements of Reusable Software Architecture*. Addison-Wisley, 1995.

[3] *Pattern Languages of Program Design 3*, edited by R.Martin, D.Riehle, F.Buschmann, Addison-Wisley, 1997.

[4] *UML Semantics*. OMG document number: ad/97-08-04.

[5] L.Schachinger and R.Talman: *Teapot: A Thin-Element Accelerator Program for Optics and Tracking*, Particle Accelerators, **22**, 35(1987).

[6] R.Talman: *A Universal Algorithm for Accelerator Correction*, AIP 255, 1991.

[7] G.Bourianoff, A.Reshetov, N.Malitsky: *Object-Oriented Approach for the Design of the Simulation Facility of the SSC*, SSCL-677, 1994.