2016

Mutable Class Design Pattern

Nikolay Malitsky
Nova Southeastern University, nmalitsky@gmail.com

Mutable Class Design Pattern

by

Nikolay Malitsky

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

Graduate School of Computer and Information Sciences
Nova Southeastern University

2016

We hereby certify that this dissertation, submitted by Nikolay Malitsky, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

_____ _____

Michael J. Laszlo, Ph.D. Date
Chairperson of Dissertation Committee

_____ _____

Francisco J. Mitropoulos, Ph.D. Date
Dissertation Committee Member

_____ _____

Amon B. Seagull, Ph.D. Date
Dissertation Committee Member

Approved:

_____ _____

Amon B. Seagull, Ph.D. Date
Interim Dean, College of Engineering and Computing

College of Engineering and Computing
Nova Southeastern University

2016

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Mutable Class Design Pattern

by
Nikolay D. Malitsky
December 2015

The dissertation proposes, presents and analyzes a new design pattern, the Mutable Class pattern, to support the processing of large-scale heterogeneous data models with multiple families of algorithms. Handling data-algorithm associations represents an important topic across a variety of application domains. As a result, it has been addressed by multiple approaches, including the Visitor pattern and the aspect-oriented programming (AOP) paradigm. Existing solutions, however, bring additional constraints and issues. For example, the Visitor pattern freezes the class hierarchies of application models and the AOP-based projects, such as Spring AOP, introduce significant overhead for processing large-scale models with fine-grain objects. The Mutable Class pattern addresses the limitations of these solutions by providing an alternative approach designed after the Class model of the UML specification. Technically, it extends a data model class with a class mutator supporting the interchangeability of operations.

Design patterns represent reusable solutions to recurring problems. According to the design pattern methodology, the definition of these solutions encompasses multiple topics, such as the problem and applicability, structure, collaborations among participants, consequences, implementation aspects, and relation with other patterns. The dissertation provides a formal description of the Mutable Class pattern for processing heterogeneous tree-based models and elaborates on it with a comprehensive analysis in the context of several applications and alternative solutions. Particularly, the commonality of the problem and reusability of this approach is demonstrated and evaluated within two application domains: computational accelerator physics and compiler construction. Furthermore, as a core part of the Unified Accelerator Library (UAL) framework, the scalability boundary of the pattern has been challenged and explored with different categories of application architectures and computational infrastructures including distributed three-tier systems.

The Mutable Class pattern targets a common problem arising from software engineering: the evolution of type systems and associated algorithms. Future research includes applying this design pattern in other contexts, such as heterogeneous information networks and large-scale processing platforms, and examining variations and alternative design patterns for solving related classes of problems.

Acknowledgements

I have been fortunate to work with a number of bright and talented people that inspired and influenced this research. First and foremost, I am pleased to acknowledge Richard Talman. His original accelerator physics algorithms and applications triggered the introduction of our common project, Unified Accelerator Libraries (UAL), and were the primary driving factors of these studies towards the invention of the Mutable Class approach. Next, I am very grateful to Christoph Iselin, a leader of the CLASSIC collaboration. The competition between CLASSIC and UAL projects represented a productive and energetic environment that generated and assessed numerous alternative solutions. Moreover, the success with computational accelerator physics applications led me to the PhD program aiming to generalize the Mutable Class approach as the corresponding design pattern.

Within the PhD program, I was advantageous to meet my adviser Michael Laszlo. His gentle comments and keen questions helped enhance the original solution and sharpen its description from multiple perspectives. I am also pleased to express my appreciation to the committee members, Francisco Mitropoulos and Amon Seagull, for their suggestions helping to strengthen this work.

Finally, I must thank my coauthors: Michael Blaskiewicz, George Bourianoff, Rama Calaga, Peter Cameron, Richard Casella, Keith Lally, Nicholas D'Imperio, Don Dohan, Alexei Fedotov, Valery Fine, Colwyn Gulliford, Niranjan Hasabnis, Fanglei Lin, Alfredo Luccio, Thomas Pelaia, Sheng Peng, Fulvia Pilat, Igor Pinaev, Vadim Ptitsyn, Alexander Reshetov, Todd Satogata, Frank Schmidt, Yannis Semertzidis, Kunal Shroff, Steven Tepikian, Nanbor Wang, Jie Wei, and Yiton Yan. This work would not have been possible without their contributions.

Table of Contents

Abstract iii
Acknowledgments iv
List of Tables vi
List of Figures vii
List of Listings x

Chapters

1. Introduction 1
Background 1
Problem Statement 2
Dissertation Goal 4
Research Questions 5
Barriers and Issues 6
Definition of Terms 7
List of Acronyms 9
Summary 10

2. Review of Literature 12
Visitor Approach and Extensions 15
Aspect-Oriented Approach 24
Examples of the Heterogeneous Tree-Based Applications 31
Large-Scale Graph Data Processing 45
Summary 50

3. Methodology 51
Mutable Class Approach 51
Mutable Class Pattern 57
Quality Factor Assessment 75
Summary 77

4. Results 79
Computational Accelerator Physics 79
Compiler Construction 103
Summary 111

5. Conclusion, Recommendations, and Summary 112
Conclusions 112
Recommendations 115
Summary 119

Appendices 124
A. ISO/IEC 25010 Product Quality Model 125

References 127

List of Tables

Tables

1. Assessment of quality attributes for existing and proposed approaches 76

2. Accelerator programs used in the SNS project 89

List of Figures

Figures

1. Game world model designed after the Composite pattern 13

2. Two-dimensional collection of methods associated with the model-visitor interactions 13

3. Graph-oriented extension of the Composite model 14

4. Visitor pattern in the context of the virtual world example 15

5. Interaction diagram of the Visitor pattern 16

6. Extended Type Visitor pattern 17

7. Generic Visitor pattern 18

8. Dynamic Dispatch Visitor pattern 19

9. Reflective Visitor pattern 20

10. Normal Form Visitor pattern 22

11. Acyclic Visitor pattern 23

12. AOP-based Normal Form Visitor pattern 27

13. Interaction diagram of the Interceptor pattern 30

14. AOP Proxy 31

15. Girder Assembly 33

16. ADXF accelerator model 34

17. Three-dimensional view of accelerator physics algorithms 35

18. Example of the heterogeneous AST model 38

19. Three-dimensional view of compiler-compiler algorithms 39

20. Shape nodes of the Open Inventor scene graph model 42

21. Two-dimensional view of scene graph algorithms 44

22. Example of the four-layer metamodel hierarchy 52

23. Streamlined diagram of the UML Class metaclass 53

24. Mutable Class approach 54

25. Structure of the Strategy pattern 55

26. Structure of the Mutable Class model 55

27. Class diagram of the Mutable Class configuration approach 56

28. Component-strategy associations of the imaginary game application 58

29. Mutable Class-based structure of the imaginary game application 59

30. Class diagram of the Mutable Class pattern 62

31. Registering a mutator and operations of the ComponentA class 63

32. Interactions for binding data processing algorithms 64

33. Interactions for data processing of the ComponentA instance 64

34. Mutable Class pattern in the context of accelerator algorithms (Figure 17) 80

35. UAL-based approach for adding new devices 82

36. RHIC application 83

37. CESR application 84

38. MADX-UAL suite 86

39. SNS application 87

40. SNS benchmark infrastructure 89

41. Blow-up of beam profile due to skew-quadrupole sum resonance in the presence of space charge: blue color (in the middle) - no space charge, no errors; yellow color - space charge, no errors; red color - space charge, expected errors and quadrupole tilt (0.2 mrad); green color – space charge, expected errors and quadrupole tilt (1 mrad) 91

42. Tune spreads for working points (6.23, 6.20) and (6.4, 6.3), respectively 92

43. Loss curves for working points (6.23, 6.20) and (6.4, 6.3), respectively 92

44. Implementation of the Element-Algorithm-Probe analysis pattern 94

45. Integration of the TEAPOT and SPINK algorithms based on the combination of the Mutable Class and Decorator patterns 95

46. The configuration of the UAL propagator based on the SXF and APDF files 96

47. Class diagram of the Mutable Group variant of the Mutable Class pattern 98

48. Object diagram of the Mutable Group variant of the Mutable Class pattern 98

49. Typical three-tier high level application environment 100

50. Virtual Accelerator server 102

51. Mutable Class-based structure of compiler algorithms (Figure 19) 104

52. The three steps of the JastAdd compilation process: (a) building a parser, (b) building the AST classes, (c) compiling a source program 106

53. The delegation scheme of the algorithm invocation in the AST program node based on the Mutable Class approach 109

54. Knowledge discovery process model 117

55. Spark-based integrated platform 118

List of Listings

Listings

1. Mountain Observer aspect 28

2. Vertex API and PageRank implemented in Pregel 47

3. Framework layer of the Mutable Class pattern 66

4. GenericMutator class template 67

5. GenericOperation class template 67

6. Linker of component mutators and registry of observers 68

7. MutatorInitializer and ObserverInitializer 68

8. Region and RegionMutator classes 69

9. Mountain and MountainMutator classes 69

10. RegionObserver class 70

11. MountainObserver class 70

12. Main program 71

13. Observer class 71

14. MountainObserver class with the Observer state 72

15. X and XMutator classes 73

16. XObserver class 74

17. Main program using the new component X 74

18. APDF description of the TEAPOT tracking engine 97

19. APDF description of the Model Independent Analysis (MIA) propagator 97

20. Fast TEAPOT 99

21. Extract of the PrettyPrint.jadd file with the PrettyPrint aspect 107

22. Examples illustrating the run-time weaving mechanism of the Mutable Class approach 110

Chapter 1

Introduction

Background

The design of modern software systems represents a complex task that must consider numerous multi-scale multi-domain requirements, technologies, and perspectives. Software engineering addresses this task by providing a growing collection of design patterns – reusable solutions that were invented, presented, evaluated, and proven in previous projects. There are several categories of these design patterns, such as creational, structural, and behavioral (Gamma, Helm, Johnson, and Vlissides, 1995). Many of them are developed to encapsulate or decouple related concepts. For example, the Bridge pattern decouples an abstraction from its implementation; the Strategy pattern encapsulates the implementation of the object behavior into separate classes, etc. The choice of the appropriate concept or the combination thereof, is usually a tradeoff determined by the project requirements. As a result, the collection of design patterns is dynamic and follows the changes in software technologies and applications.

The scope, effectiveness, and aptitude of the software applications directly depend on the quality and capability of the data models describing and implementing the entities of the application domain and their relationships. The application models may vary in many ways: size, number of data types, and complexity of data collections. Among the most universal and sophisticated data structures used in modern applications are heterogeneous graphs. These allow the description of collections of heterogeneous entities connected by an arbitrary number of pairwise relationships. To represent hierarchical structures, graphs commonly take the special

form of trees. As a result, heterogeneous graphs became the natural data models of a variety of application domains, such as the abstract syntax tree (AST) of compiler systems, the scene graphs of visualization toolkits, biological and social networks, etc.

The natural accommodation of the application data, however, covers only one of the software requirements. Another important and usually contradictory aspect is associated with providing an efficient approach for data processing with different algorithms. For example, the AST serves as both an internal and intermediate representation of the source program during the different phases of the compilation process including context checking, optimization, and code generation. Designed after the Interpreter pattern, the AST heterogeneous model maps the language grammar into the corresponding hierarchical object-oriented structure designed to capture program semantics. Depending on the applied algorithms, each compilation phase introduces an additional set of requirements. Moreover, a set of compiler algorithms is not fixed and can vary according to the complexity of the target language as well as the function to be performed. As a result, the choice of the AST model is not only determined by the structure of the source language, but rather is a tradeoff among the various objectives of the processing algorithms. Recently, this topic is especially emphasized by modern graph-based processing applications (Sun and Han, 2012) bringing a variety of data-mining and machine learning algorithms.

Problem Statement

Processing the heterogeneous models in the object-oriented approach is addressed by the Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1995). 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 the compiler system, for example,

the Visitor pattern facilitates the development of the AST-based modules by separating the different types of AST processing algorithms. The Visitor pattern, however, introduces a serious limitation by freezing the existing class hierarchies and preventing any extensions of the processed tree structure with new types. There were several attempts aiming to resolve the problem of the original Visitor pattern. Within the object-oriented paradigm, the Acyclic Visitor Pattern (Martin, Riehle, and Buschmann, 1997) suggested the most consistent alternative approach by breaking the dependency cycle with multiple inheritance. But moving strong coupling between components from the framework to the application layer does not fully resolve the problem. In addition, the design of the Visitor pattern is tailored to traversal scenarios and requires the reconsideration or further development of this approach in the context of a node-centric computational model implemented by the modern large-scale graph processing systems such as Google's Pregel (Malawics et al., 2010). In this model, nodes of application structures compute algorithms in parallel and communicate directly with one another by sending messages along outgoing edges. The model addresses influential algorithms, such as Page Rank (Page, Brin, Motwanl, and Winograd, 1998) and Shortest Paths (Gross and Yellen, 2005) for processing homogeneous information networks and needs to be extended for supporting heterogeneous applications.

An aspect-oriented programming (AOP) paradigm brought several ideas addressing similar issues. This paradigm introduced a new concept, aspect, associated with the crosscutting functional properties of the object-oriented applications and defined the corresponding pointcut-advice model for integrating object-oriented and aspect modules. In the context of the dissertation problem, processing operations can be considered as crosscutting behavior of heterogeneous models and therefore be interchanged using the AOP configuration approach.

However in the present variant, this approach is defined only on the preprocessor level using annotation directives, deferring the implementation issues to the AOP-based projects, such as Spring AOP (Walls, 2013).

Dissertation Goal

The thesis proposes, presents, and analyzes a new design pattern, called the Mutable Class pattern, to support the processing of large-scale heterogeneous data models with multiple families of algorithms. The idea of the Mutable Class pattern was initially introduced in the framework of the Unified Accelerator Libraries (Malitsky and Talman, 1998) for building dynamic associations among heterogeneous physical devices and modeling algorithms. From the conceptual perspective, the Mutable Class approach was designed after the Class model of the UML specification (OMG, 2011). Technically, it extends a data model class with is a singleton that maintains the behavior of the class objects based on the Strategy pattern (Gamma, Helm, Johnson, and Vlissides, 1995). A Strategy encapsulates the implementation of this behavior into separate classes and provides the mechanism for their interchange. Later, the solution (i.e., Mutable Class pattern) was successfully validated in the context of the JastAdd metacompiler construction system (Malitsky, 2008).

The goal of this thesis is to formalize and validate the Mutable Class approach through analysis of different applications. In accordance with the software engineering methodology, the analysis of each domain starts with the description of the corresponding use case and associated projects. This analysis is followed by an examination of the strong and weak features of the existing approaches and their comparison with the new prototype or extension based on the Mutable Class.

Research Questions

The thesis addresses the following three questions:

- *Can the Mutable Class approach be formalized as a new design pattern for processing heterogeneous tree-based models?* Design patterns represent reusable solutions to recurring problems. According to the design pattern methodology (Gamma, Helm, Johnson, and Vlissides, 1995), the definition of these solutions encompasses multiple topics, such as the problem and applicability, structure, collaborations among participants, consequences, implementation aspects, and relation with other patterns. The thesis provides this formal description of the Mutable Class pattern for processing heterogeneous tree-based models and elaborates it with a comprehensive analysis of a sample code and implementation aspects. In addition, the dissertation includes the quality factor assessment of the Mutable Class pattern and comparison with the current approaches based on the Visitor pattern and Aspect-Oriented Programming paradigm.
- *Is the Mutable Class approach generic and can be reused in multiple application domains?* The commonality of the problem and reusability of this design pattern is demonstrated within several application domains. Initially, the Mutable Class pattern was derived and explored in the context of computational accelerator physics applications. Next, it was applied to a compiler construction project. Third, the analysis of one of major scientific visualization toolkits revealed a proprietary mechanism that was closely related with the Mutable Class approach for processing type-specific algorithms. Finally, these studies are complemented with an overview of a new category of heterogeneous

models, known as heterogeneous information networks (Sun and Han, 2012), using heterogeneous graphs.

- *How scalable is the Mutable Class approach from the perspective of the application architecture and computational infrastructure?* The original catalog of design patterns (Gamma, Helm, Johnson, and Vlissides, 1995) is considered in the context of three classes of software: application programs, toolkits, and frameworks. According to the previous Unified Accelerator Library (UAL) applications, the Mutable Class approach provides an architectural solution addressing all three categories of software. Being part of the framework layer, it identified the structure of data-algorithm associations across multiple layers of the UAL application toolkit. Simultaneously, the same approach defined a consistent mechanism for developing third-party extensions and building project-specific applications. Eventually, the UAL applications were deployed on parallel clusters and three-tier distributed infrastructure. The thesis aims to present a comprehensive analysis of these use cases and explore the corresponding technical solutions and scalability issues in the context of other application toolkits.

Barriers and Issues

The Mutable Class pattern is designed to provide a general architectural solution addressing the multi-layer structure of application toolkits across multiple application domains. As a result, the scope of this pattern represents its major challenge.

The initial version of the Mutable Class approach was developed within the C++ framework of the Unified Accelerator Libraries (Malitsky and Talman, 1998) after refactoring and integration of two major accelerator programs, TEAPOT (Schachinger and Talman, 1987) and

ZLIB (Yan and Yan, 1990), originally written in the FORTRAN programming language. Eventually, the UAL toolkit encompassed nine accelerator libraries and various extensions covering different simulation topics (Malitsky et al, 1999; Lin et al., 2009). Such integration was driven by demand for complex beam dynamic studies including a combination of several effects and dynamic processes. In turn, these composite studies revealed the necessity of the UAL parallel extension (D'Imperio et al., 2006). The transition to the high performance computing environment demonstrated new capabilities of the Mutable Class-based framework allowing to mix together sequential and parallel algorithms. Eventually, the UAL framework was deployed on the three-tier distributed infrastructure for developing model-based control systems (Malitsky et al., 2010).

Consideration of the problems and technical solutions associated with the UAL multi-scale projects is the first topic of the thesis study. Two other application domains, compiler construction and 3D computer graphics, introduce the additional challenge requiring reverse engineering and integration of the Mutable Class approach with existing application toolkits, such as the metacompiler construction system JastAdd (Hedin and Magnusson, 2003; Soderberg et al., 2013) and the 3D graphics toolkit Open Inventor (Wernecke er al., 1994, Heck, 2010).

Definition of Terms

- *Abstract syntax tree*: compiler's internal hierarchical representation of a program.
- *Accelerator lattice:* hierarchical representation of a particle accelerator model composed from heterogeneous physical devices like magnets, radiofrequency cavities, and others.

- *Agent-oriented programming*: programming paradigm designed after the concept of software agents exhibiting different aspects of the artificial intelligence behavior, such as autonomy, reactivity, learning, social ability, and others.
- *Aspect:* unit of modularization in the aspect-oriented programming paradigm. It combines crosscutting extensions of the conventional program with well-defined places for their insertion.
- *Aspect-oriented programming*: programming paradigm addressing crosscutting properties of conventional programs.
- *Big Data*: collection of data sets or streaming data characterized by the new level of four dimensions: volume, velocity, variety, and veracity.
- *Clipping*: process of removing polygon parts that lie outside a view frustum.
- *Composite pattern*: software approach for building a tree structure of heterogeneous objects.
- *Culling*: process of checking visibility of scene objects within a view frustum.
- *Design pattern*: reusable solution for recurring software design problems.
- *Fourth Paradigm:* data-intensive shift in science exploration.
- *High-order Taylor map:* computational representation of an accelerator sector as a non-linear transformation of particle coordinates.
- *Heterogeneous graph*: collection of nodes connected by edges, where node and edges are of different types.
- *Heterogeneous tree*: hierarchical collection of nodes with different types, where each node may have a value and a collection of other nodes.
- *Metamodel*: specification of methodology for creating models.

- *Proxy pattern*: software approach for extending object functionality by creating a placeholder of the original object.
- *Registry pattern*: software approach suggesting a dedicated class for maintaining and querying a dynamic collection of managed objects.
- *Rendering*: process of creating an image from a model.
- *Strategy pattern*: software approach encapsulating the implementation of algorithms into separating classes and making them interchangeable.
- *Scene graph*: hierarchical and spatial representation of a graphical scene.
- *Visitor pattern*: software approach that groups the different types of heterogeneous structure-oriented operations into separate classes and provides a consistent mechanism for their interchange.
- *Weaving*: procedure for composing aspects of the aspect-oriented programming paradigm with components of the conventional program.

List of Acronyms

- ADXF: Accelerator Description Exchange Format
- AOP: Aspect-Oriented Programming
- AST: Abstract Syntax Tree
- CCM: CORBA Component Model
- COM: Component Object Model
- CORBA: Common Object Request Broker Architecture
- DCOM: Distributed Component Object Model
- DSL: Domain Specific Language

- EJB: Enterprise JavaBeans
- IR: Intermediate Representation
- MOF: Meta-Object Facility
- NF: Normal Form
- OMG: Object Management Group
- UML: Unified Modeling Language
- UAL: Unified Accelerator Libraries

Summary

This chapter introduces the dissertation topic, the Mutable Class pattern, and overviews the context of the addressed problem, major ideas of the proposed solution, anticipated issues and consequences. The Mutable Class pattern is proposed to provide a reusable solution for processing large-scale heterogeneous models with different families of algorithms. This task is important in the context of multiple application domains, such as computational accelerator physics, compiler construction, 3D computer graphics, and heterogeneous information networks. As a result, it has been addressed by multiple approaches including the Visitor design pattern and the aspect-oriented programming paradigm. However each of these predecessors comes with drawbacks. The Mutable Class pattern aims to overcome the limitations of these solutions by providing an alternative approach designed after the Class model of the UML specification. The pattern extends a data model class with a singleton that maintains the behavior of the class objects based on the Strategy pattern.

The scope of the dissertation topic is outlined by three research questions. The first question is formulated after the design pattern methodology and focuses on reusability aspects of the

Mutable Class pattern in the context of three heterogeneous tree-based application models. The second question addresses the scalability of this approach from the perspective of the application architecture and computational infrastructures. The final question challenges the Mutable Class pattern with the new category of large-scale applications, so called heterogeneous information networks. These questions introduce multiple technical issues, most of them associated with the scope of the Mutable Class applications. As a result, consideration of these questions aims to solidify the proposed pattern and identify its application boundaries.

Chapter 2

Review of Literature

Handling data-procedure associations is one of the major topics in programming languages and software design. For example, the object-oriented paradigm superseded procedural programming by explicitly combining data members and associated methods into reusable units of programming logic. However, for large-scale software projects, data models need to change their behavior in the face of different application requirements and environment states. As a result, finding an optimal solution represents a complex decision based on the analysis of multiple technical approaches implemented in the context of the different application domains.

Many existing application models can be described by large-scale hierarchical trees of heterogeneous elements. As an illustration, this chapter will consider a virtual world consisting of two types of components: Plains and Mountains. These world components can be grouped into bigger areas, called Regions, forming a hierarchical model. Furthermore, the world is not static and evolves from version to version by adding new components such as seas, forests, and cities. To simplify the example, these extensions are represented by single component X. Following the design pattern methodology, the described world can be implemented based on the Composite pattern (Gamma, Helm, Johnson, and Vlissides, 1995). As shown in Figure 1, Region represents a composite node that can include other Regions and leaf nodes, such as Mountain, Plain and X.

Once the world is built it has to be explored. In computer science, such world exploration can be performed by a traversal process that subsequently visits each node of the hierarchical

model in some particular order. Usually, the visiting scenarios are defined after the application model. As a result, the visitation procedure cannot be fully captured in the model, prompting the definition of additional classes, such as Visitor. Many different types of visitors are possible, depending of the application. For example, the world can be explored by either an observer or a settler. Moreover, their behavior depends on the types of the world locations, in our case, plains and mountains. In computer programs, these place-visitor interactions can be implemented with a two-dimensional collection of corresponding methods as shown in Figure 2.

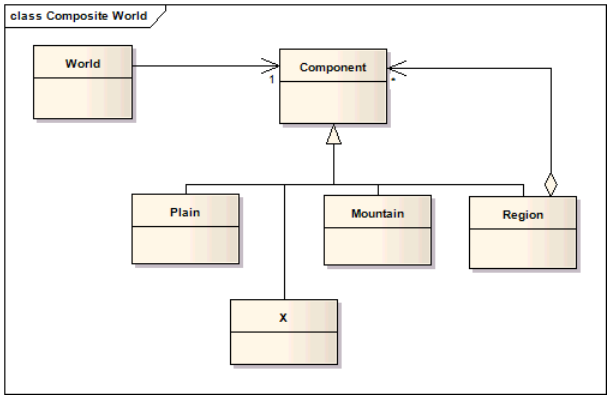

Figure 1: Game world model designed after the Composite pattern

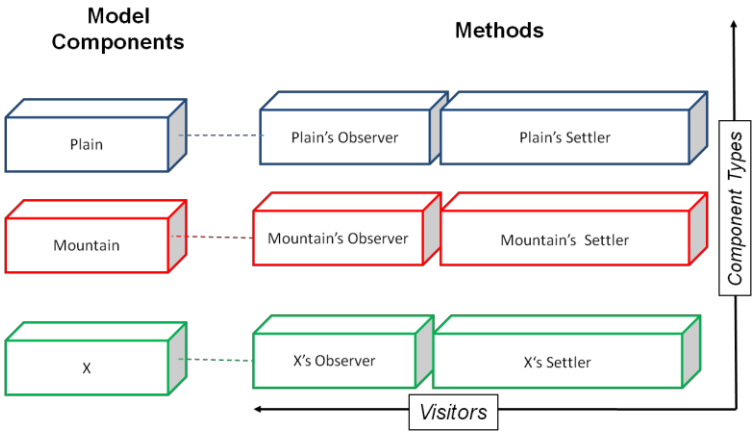

Figure 2: Two-dimensional collection of methods associated with the model-visitor interactions

While traversing through the world model, methods have to be selected according to runtime types of places and visitors. Major object-oriented programming languages, however, only

support a single dispatch mechanism provided by a virtual function. This issue has been addressed by the dedicated Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1995), implementing a double-dispatch approach based on the combination of object-oriented techniques. Yet, the Visitor pattern was also not ideal, introducing a principal constraint for adding new types of model components, such as X in our example. This limitation triggered the development of numerous extensions of the original variant (Pati & Hill, 2010) providing partial enhancements in the context of different applications. Finally, the same problem was addressed by the Aspect-Oriented Programming (AOP) paradigm (Wu et. al, 2005). Recently, this topic became especially important with the development of large-scale graph applications. In comparison with hierarchical trees, heterogeneous graphs introduce two additional aspects. First, they extend the Composite model with a new association, allowing links between leaf nodes as shown in Figure 3. Second, large-scale graph applications bring node-centric algorithms in addition to traversal procedures.

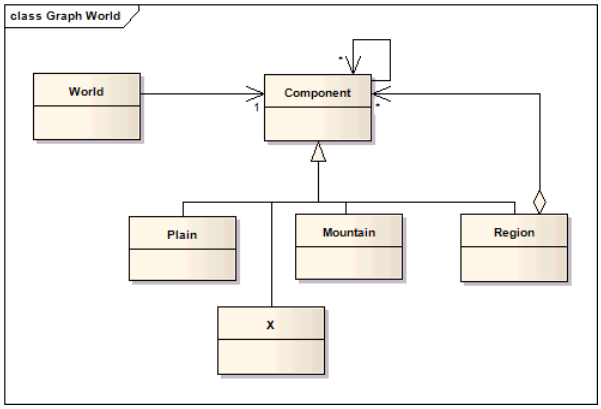

Figure 3: Graph-oriented extension of the Composite model

The rest of the chapter is structured as follows. The first section provides an overview of the original Visitor pattern and its extensions. It is followed by the section describing the aspect-orient approach. The third section considers the problem of processing trees in the context of three application domains: computational accelerator physics, compiler construction, and 3D

computer graphics. The final section presents a new direction associated with the development and processing of large-scale heterogeneous information networks.

Visitor Approach and Extensions

The Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1995) is a well-known technique that allows the application of different types of operations on a collection of heterogeneous objects. For example, in the context of our virtual world example, the pattern facilitates development of different types of visitors, such as observers or settlers, without modifying the world model. The corresponding structure diagram is shown in Figure 4.

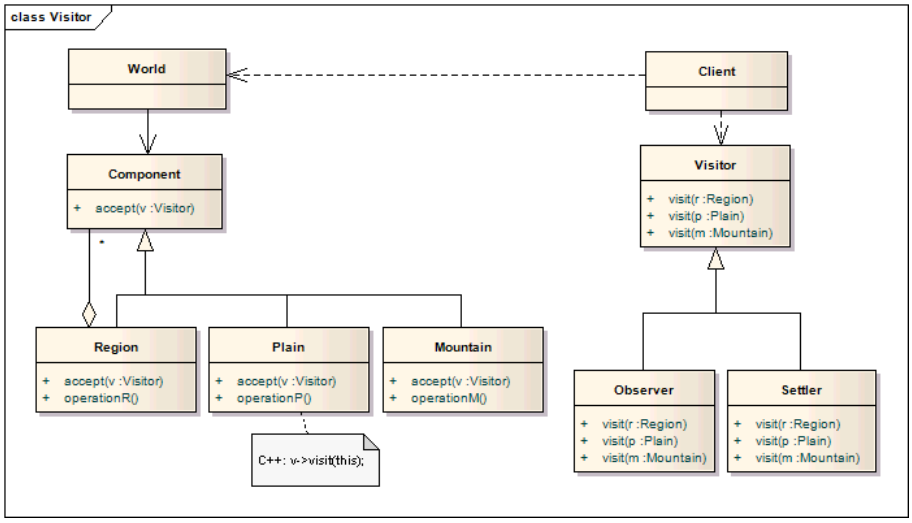

Figure 4: Visitor Pattern in the context of the virtual world example

According to the diagram, the Visitor pattern defines two class hierarchies associated with multiple types of model components and multiple types of visitor classes. Traversal algorithms are implemented with the two-dimensional collection of methods, differentiated by traversal categories and the types of processed objects. Methods from the same traversal categories are encompassed into the corresponding Visitor classes, Observer and Settler, for processing the entire model structure. Each element of the model structure is algorithm-free and is responsible

for implementing a virtual method *accept()* by passing itself to the appropriate visitor method. Collaborations between the model components and visitors are illustrated by the interaction diagram of Figure 5.

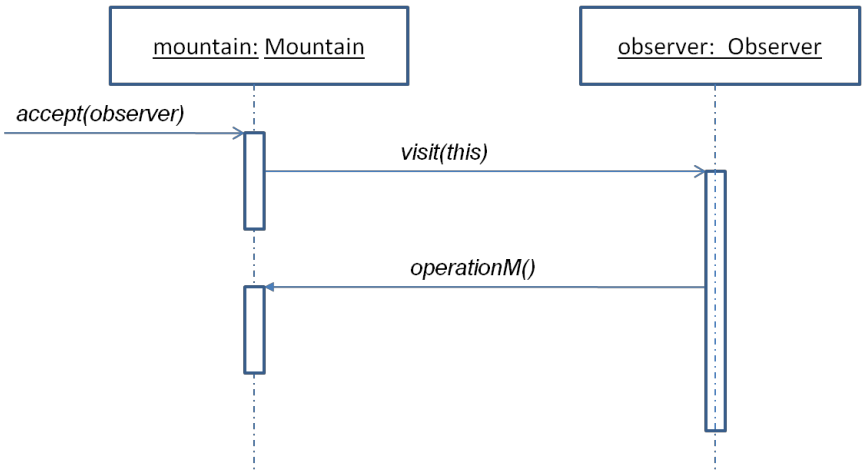

Figure 5: Interaction diagram of the Visitor pattern

The coupling between methods of the Visitor participant and the associated concrete Components of the model allows the implementation of double-dispatch behavior in conventional single-dispatch object-oriented languages such as C++ and Java. In the context of the Visitor pattern, the double-dispatch mechanism provides a convenient and efficient approach for adding any number of new Visitor types for the existing application model. This coupling, however, introduces a serious limitation. It freezes existing class hierarchies of the world model. Particularly, adding the new world component X would require editing all visitor classes by adding to each class a *visit(x: X)* method. From a more general perspective, the Visitor pattern violates the dependency inversion principle (Martin, 1996), which requires the independence of the abstract layer from its specializations. In the case of the Visitor pattern, the methods of the abstract Visitor class are dependent on concrete classes, Plain and Mountain, of the world model.

The double-dispatch mechanism of the Visitor pattern plays an instrumental role in a wide range of actual applications. As a result, limitations of the original approach generated numerous

extensions. The remainder of this section provides a brief overview of the most common solutions (Pati and Hill, 2010).

Extended Type Visitor Pattern

The Extended Type Visitor pattern has been developed in the context of application toolkits like the SableCC object-oriented compiler (Gagnon and Hendren, 1998). According to this approach, the application is divided into the toolkit layer and third-party extension, as shown in Figure 6.

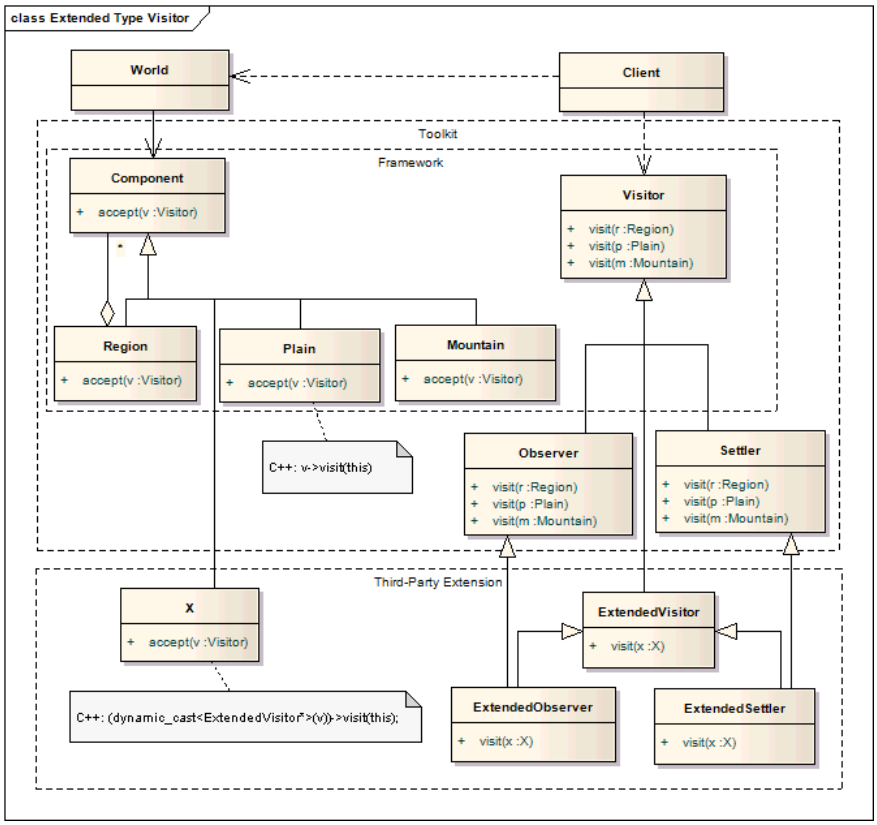

Figure 6: Extended Type Visitor pattern

The toolkit layer is designed after the original version of the Visitor pattern. Following the object-oriented methodology, model components and visitors of the application extension are derived from the corresponding framework classes. To deal with the new model components, the

extended visitor adds new methods, such as *visit(x: X)*. Since the *accept()* method takes the type of the framework visitor, changes of the extended visitor interface require dynamic type casting in the new model components. Despite the explicit definition of the third-party extension layer, the pattern does not provide a solution for managing multiple third-party extensions inside of one composite application. As a result, the pattern just propagates the extensibility issues of the original Visitor pattern from the toolkit to the application layer.

Generic Visitor Pattern

The Generic Visitor pattern represents another toolkit-oriented approach introduced and developed in several papers (Vlissides, 1999; Visser, 2001) and application toolkits such as OpenSceneGraph (Martz, 2007). The pattern addresses the extensibility issue of the Visitor pattern by adding a generic method *visitAny()* in the Visitor class as shown in Figure 7.

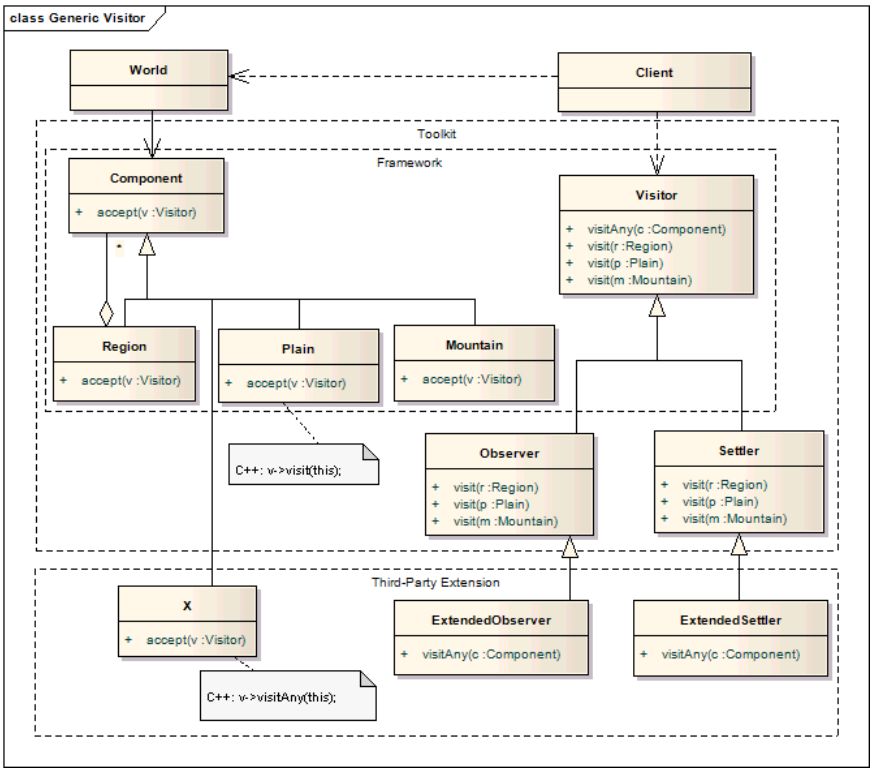

Figure 7: Generic Visitor pattern

The combination of the *visitAny()* method with the Visitor pattern provides a consolidated hybrid interface for supporting both predefined and user-specific subsets of model components. The generic interface is a well-known technique for dealing with heterogeneous data types. Its implementation, however, needs some sort of reflection mechanism that is not universally supported by all major programming languages, for example C++.

Dynamic Dispatcher Visitor Pattern

Dynamic Dispatcher Visitor (Buttner et al., 2004) can be considered as an unconventional generic variant of the Visitor pattern. It solves the extensibility problem of the original pattern by eliminating the *accept()* method of the model components and moving the dispatching operation into the *dispatch()* method of a new class Dispatcher as shown in Figure 8. The *dispatch()* method serves as a generic interface for selecting the most appropriate visitor method based on a particular type of model component.

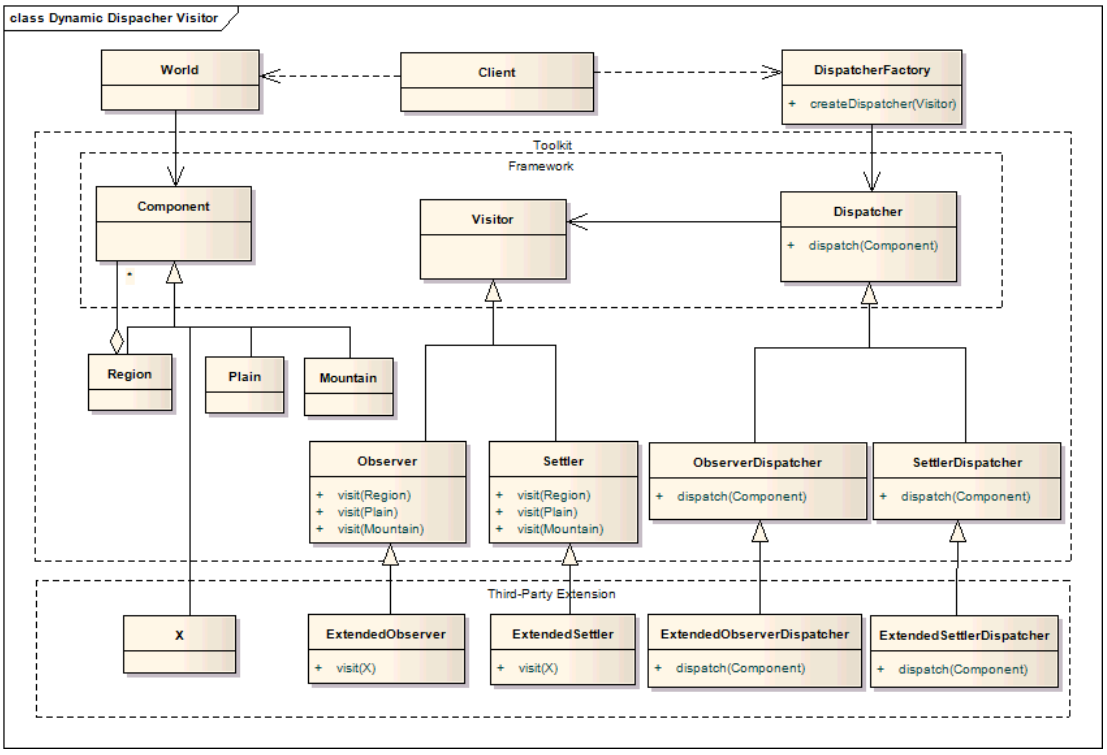

Figure 8: Dynamic Dispatch Visitor pattern

As a generic approach, this pattern introduces an issue similar to that of the Generic Visitor pattern of not providing an explicit implementation solution of the generic method *dispatch()* and simply propagating the problem.

Reflective Visitor Pattern

One of the solutions for implementing a generic interface is offered by the Reflective Visitor (Mai and Champlain, 2001). Figure 9 illustrates the structure of this pattern. It comes with two major changes to the original Visitor pattern. First, like the Dynamic Dispatcher Visitor, the pattern breaks cyclic dependencies between visitors and model components by eliminating the *accept()* method. In this case, the dispatching operation is moved into the Visitor classes. Second, the pattern applies the reflection mechanism provided by several programming languages such as Java and C#. Reflection is used for selecting the appropriate *visit()* methods with respect to the type of the model component.

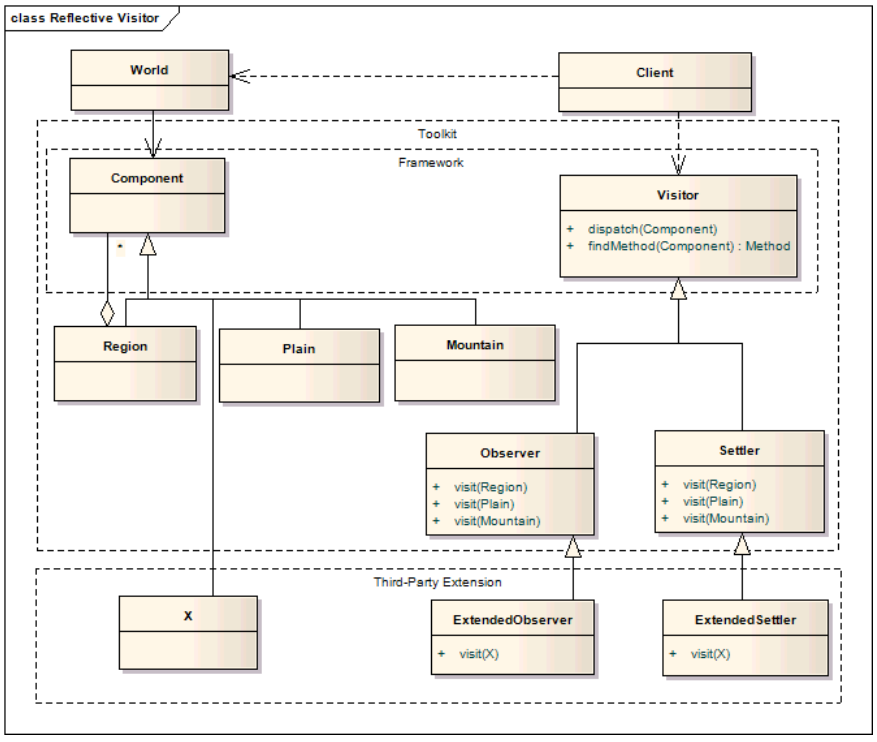

Figure 9: Reflective Visitor pattern

A similar approach has been developed by Palsberg and Jay (1998) in the context of the Walkabout Class Visitor pattern. The authors suggest a compact reflection algorithm that can be implemented in the top Visitor class and shared by all visitors. The reflection-based invocation of the visitor methods, however, induces a significant performance overhead in comparison with direct access. This issue has been tackled by two patterns, Runabout (Grothoff, 2003) and Sprintabout (Forax, Duris and Roussel, 2005). Runabout replaces the reflection-based *lookup()* method with a hybrid approach based on the dynamic code map, the Java reflection API and Java class-loading mechanism. According to the pattern, the constructor of the Visitor class scans all available *visit()* methods using reflection, generates on-the-fly the corresponding wrapper classes for each method, and dynamically loads these classes into the Virtual Machine using Java class-loading mechanism. Finally, instances of these classes are created and stored in a dynamic code map, providing efficient lookup access. In contrast to the Runabout, the Sprintabout pattern builds a single class for all methods.

In addition to the performance advantage, replacing the Java reflection interface with a dynamic map introduces a language-neutral approach for developing generic visitors. One such generic visitor is the Normal Form Visitor pattern that will be considered in the next subsection.

Normal Form Visitor Pattern

The Normal Form Visitor pattern is named after the corresponding database normalization technique that has been applied by Xiao-Peng and Yuan-Wei (2010). To solve the cyclic dependence of the original Visitor pattern, the authors consider requirements of the third normal form (3NF) and then break all the transitive dependencies among the pattern classes, like:

- Base Visitor → Derived Visitor
- Base Component → Derived Component

- Base Visitor → Base Component
- Derived Component → Base Visitor

In this set of the non-transitive dependencies, the last one, Derived Component → Base Visitor, clearly violated the dependency inversion principle. To fix it, the authors split relationships between visitors and components using the Factory pattern:

- Base Visitor → Base Visitor Factory
- Base Visitor Factory → Base Component

The corresponding product of these transformations is shown in Figure 10.

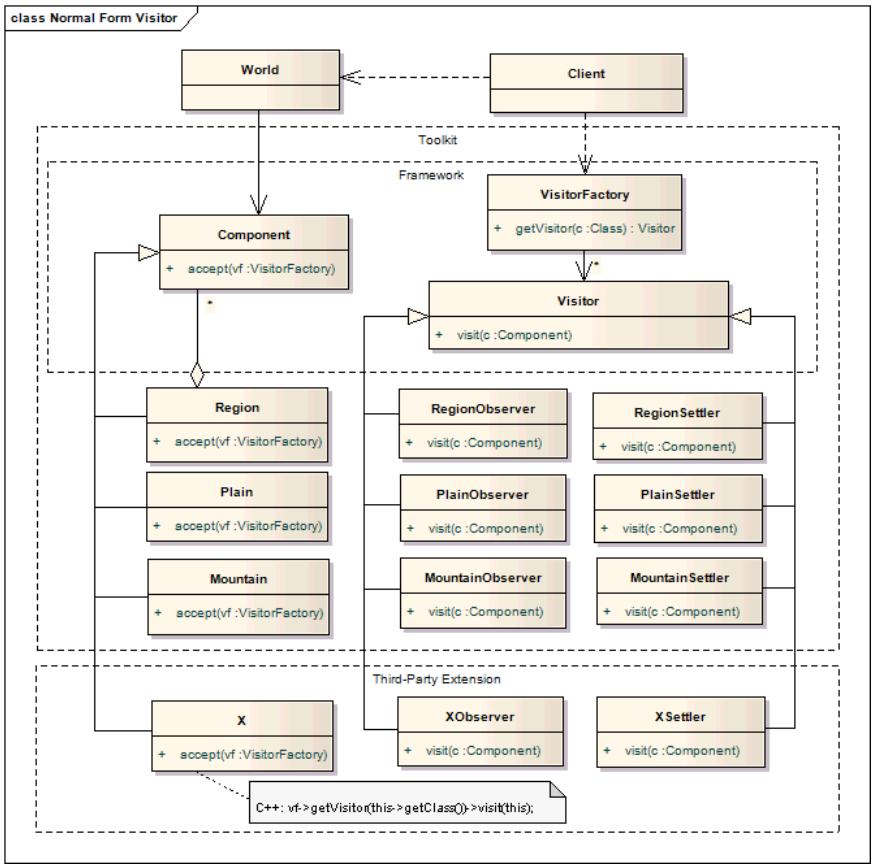

Figure 10: Normal Form Visitor pattern

According to the diagram, the Factory pattern separates the *visit()* methods into multiple wrapper classes. These changes also make the pattern consistent with the atomicity requirements of the

First Normal Form (1NF) and the interface segregation principle (Martin, 1998). In the context of the visitor patterns, Normal Form Visitor is very similar to the Runabout pattern, which it modifies with the explicit language-neutral definition of the wrapper classes. Additionally, Normal Form Visitor can be considered as a dynamic variant of the Acyclic Visitor pattern that is a subject of the next subsection.

Acyclic Visitor Pattern

The solution of using multiple wrapper classes originates from the Acyclic Visitor pattern (Martin, R., Riehle, D., and Buschmann F, 1997) shown in Figure 11.

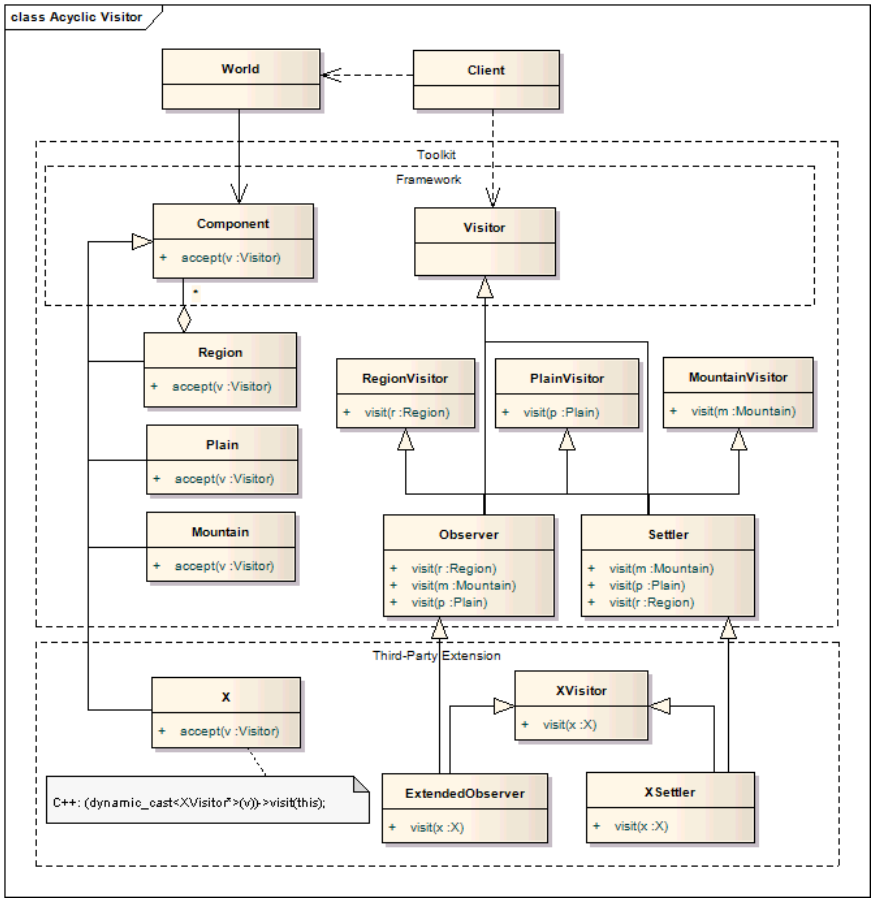

Figure 11: Acyclic Visitor pattern

In contrast with Runabout and Normal Form Visitor, the pattern suggested an alternative approach for composing these classes using multiple inheritance. In this case, the base class

Visitor does not have any member functions and is used as a marker interface (Bloch, 2008) in the *accept()* methods of the model structure. The *visit()* methods are defined corresponding to the component-specific abstract visitors: RegionVisitor, PlainVisitor and MountainVisitor. The actual visitor class Observer is derived from the basic class (to be accepted by components) and implements interfaces of abstract visitors. Similar to Runabout and Normal Form Visitor, this scheme breaks the dependency cycle of the original pattern. Multiple inheritance, however, is a static mechanism and results in strong coupling between components.

Visitor Combinator Pattern

In comparison with the Visitor extensions described in the previous subsections, Visitor Combinator (Visser, 2001) addresses orthogonal issues of the original Visitor pattern associated with lack of traversal control and resistance to combinations. For solving these limitations, the pattern suggested a set of reusable classes called *visitor combinators* implementing the basic traversal strategies, such as identity, sequence, choice and others. The different visitor combinators can then be combined to construct complex strategies and enhance traversal control. The theoretical formalism of this direction has been thoroughly developed by Oliveira (2007) in the context of the Scala programming language.

Aspect-Oriented Approach

The Aspect-Oriented Programming (AOP) paradigm originated from several related ideas, eventually becoming a consolidated core of many similar paradigms, including adaptive programming (Lieberherr, 1996), composition filters (Aksit, Bergmans, and Vupal, 1992), multi-dimensional separation of concerns (Ossher and Tarr, 1999), and subject-oriented programming (Harrison and Ossher, 1993). The original term was introduced by Gregor Kiczales and his colleagues in their report at a European Conference on Object-Oriented Programming (Kiczales,

et al., 1997). Based on the analysis of several applications, the report identified functional properties crosscutting a basic system's structure. One of the examples was a communication property of remote method invocation in a distributed document processing system. The complete list of these features is quite broad, spanning over security, logging, persistence, debugging, and others. Since such properties crosscut a 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 object-oriented code, the aspect code implementing cross-cut properties and aspect weaver metaprogramming mechanism for integrating both conventional and aspect modules.

Consideration of the data processing applications, and, particularly, the Visitor use cases in the aspect-oriented context, appears quite naturally since they are associated with many of the common issues and techniques. For example, 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 therefore represented by the corresponding construct (Wu et al., 2005, 2006). On the other hand, unconventional concepts of the AOP approach introduced an alternate angle to the problem and, as a result, triggered the development of new solutions. The rest of this section provides an overview and comparison of two AOP implementations: AspectJ (Laddad, 2003) and Spring AOP (Walls, 2013). This review is preceded with a brief introduction of the AOP model in the context of the Visitor pattern.

AOP model and Visitor pattern

AOP is a relatively young and iteratively evolving paradigm. After the publication of the original paper (Kiczales, et al., 1997), the AOP programming concepts gradually stretched into

different areas of software engineering. In turn, this development generated a variety of extensions of the AOP concepts entailing different interpretations and terminologies. To address this problem, the AOSD-Europe project consolidated aspect-oriented dialects into the Aspect-Oriented Software Development ontology (Berg, Conejero, and Chitchyan, 2005). Later, this work was elaborated on by Schauerhuber and colleagues (2006) into a conceptual reference model that was eventually revised (Wimmer, et al, 2011) based on the thorough analysis of multiple modeling approaches. The aspect-oriented ontology and reference model highlight the following major concepts:

- **Component**: element of the conventional (not-aspect-oriented) program
- **Crosscutting Concern**: structural and behavioral changes that have to be inserted across heterogeneous components of the conventional program
- **Joint point**: well-defined place in the structure or execution flow of the conventional program for attaching the implementation of a crosscutting concern
- **Pointcut**: selector of joint points
- **Aspect**: unit of modularization combining crosscutting concerns with pointcuts
- **Weaving**: procedure of composing aspects with components of the conventional program

As shown in Figure 12, semantics of these concepts can be naturally demonstrated in the context of the Normal Form Visitor pattern. The corresponding UML diagram follows a common stereotype-based notation used in many aspect-oriented modeling approaches (Wimmer et al., 2011). According to the diagram, the Visitor pattern is divided into two parts. The components of the data model form the conventional object-oriented part. The aspect part is composed from the Visitor classes representing the crosscutting concerns. The *accept()* method of the Component interface is completely decoupled from the Visitor classes and serves as a joint point for

attaching the Visitor operations. The Visitor classes are transformed into aspect units, thus augmenting the *visit()* methods with the *interceptAccept* pointcuts. In comparison with the Normal Form Visitor pattern, this diagram does not use the Visitor Factory for connecting the Component and Visitor objects. In the aspect-oriented approach, this role is performed by the weaving mechanism which is a subject of the AOP implementations, such as AspectJ and Spring AOP.

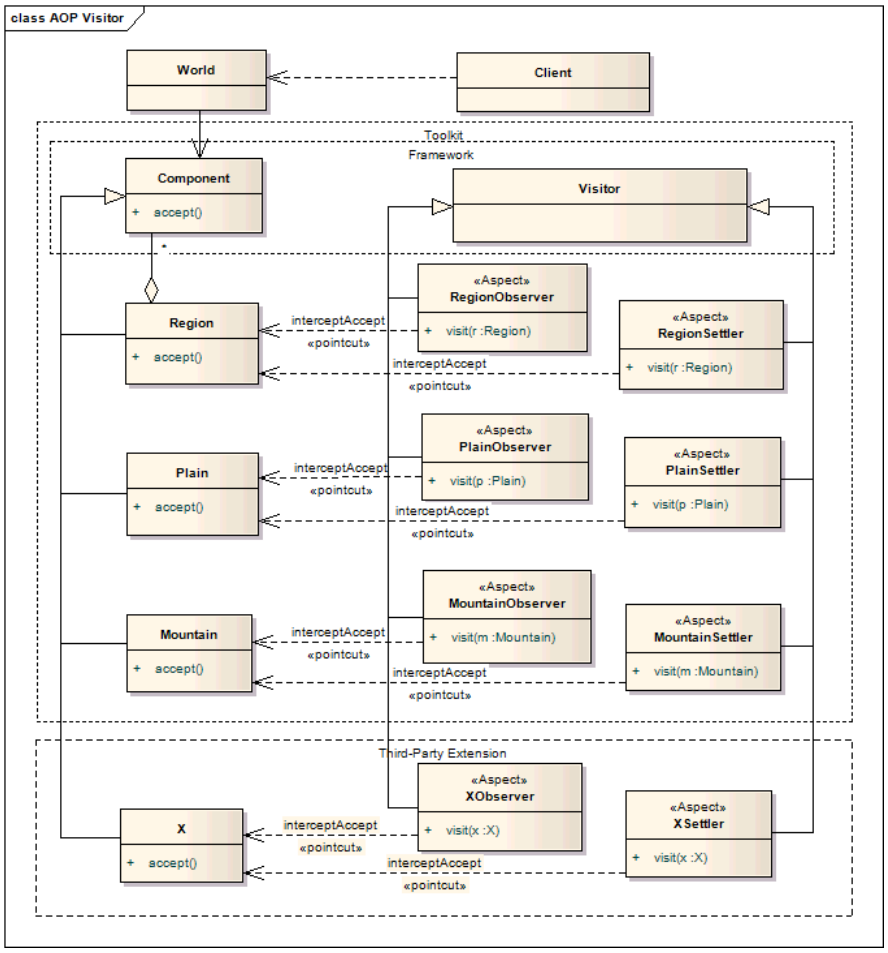

Figure 12: AOP-based Normal Form Visitor pattern

AspectJ

AspectJ (Laddad, 2003) is a general-purpose aspect-oriented Java extension developed by the authors of AOP for validating, developing and endorsing the new programming paradigm. In

2002, it was transferred to an openly-developed Eclipse project. In AspectJ, the aspect is a programming unit written after the conventional Java class using an annotation-based style of aspect declarations. Listing 1 illustrates the implementation of the MountainObserver aspect of Figure 12.

```
@Aspect
public class MountainObserver implement Visitor {

   public void visit(Mountain m) {
      // implementation of the visit method
   }

  @Pointcut ("call(void Mountain.accept()) && target(m)")
   public void interceptAccept(Mountain m) {}

  @Around("interceptAccept(m)")
  public void invokeVisit(Mountain m) {
      visit(m);
  }

}
```

Listing 1: Mountain Observer aspect

In this example, the MountainObserver aspect implements the *visit()* method for processing the Mountain data and adds two methods annotated with the pointcut and advice declarations. The *interceptAccept()* pointcut picks out joint points associated with the *accept()* methods of the Mountain class. AspectJ supports eleven different kinds of joint points such as method call, method execution, and construction call. Each joint point potentially has access to three objects of the contextual state: the currently executing object, the target object, and an array of arguments. The *interceptAccept()* pointcut, particularly, takes a target object *m* which is an instance of the Mountain class. The pointcut, however, does not call the aspect code and needs to be augmented with the corresponding *invokeVisit()* method called advice. In AspectJ, advice can be bound with pointcuts with three relationships: before, after, and around. In accordance with

the Around annotation, the *invokeVisit()* method traps the execution of the joint point and runs instead of the *accept()* method of the Mountain class.

The above example is implemented after the AspectJ dynamic joint model that does not change the interface of the conventional object-oriented components. In addition, AspectJ supports another variant, called introduction, allowing extension of the original classes of the conventional programs with inter-type declarations. In both variants, the weaving of changes is implemented at compile time using the AspectJ compiler that merges the aspect-oriented extensions directly into the byte code. This approach exposes an important issue associated 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 dynamic scenarios.

Spring AOP

Spring AOP is a Java aspect-oriented framework implemented as part of the Spring project (Walls, 2013). This approach is built around the Spring proprietary container-based architecture using the inversion of control (IoC) mechanism for configuring and managing Java objects. Inversion of control is an umbrella term associated with various techniques for building dynamic dependencies among objects. The corresponding techniques are usually related with several design patterns. The Spring AOP framework particularly leverages from two patterns, Interceptor and Proxy.

Figure 13 shows the interaction diagram of the Interceptor pattern (Schmidt et al., 2000). The diagram explains the collaborations between two major participants, Framework and Interceptor. In the context of the AOP model, Framework represents the conventional object-oriented program and Interceptor corresponds to the crosscutting concern construct of the aspect module. According to the diagram, the application instantiates a concrete interceptor and registers it with

a dispatcher. The framework subsequently receives an interception event, creates the associated context object and notifies the appropriate dispatcher about the occurrence of the event. As mentioned in the AspectJ subsection, the context object may contain the executing object, the target object, and an array of arguments. Following the framework request, the dispatcher selects the related interceptors and invokes their callback methods, passing the context object as an argument. Finally, interceptors process the content of the context object and return results to the framework.

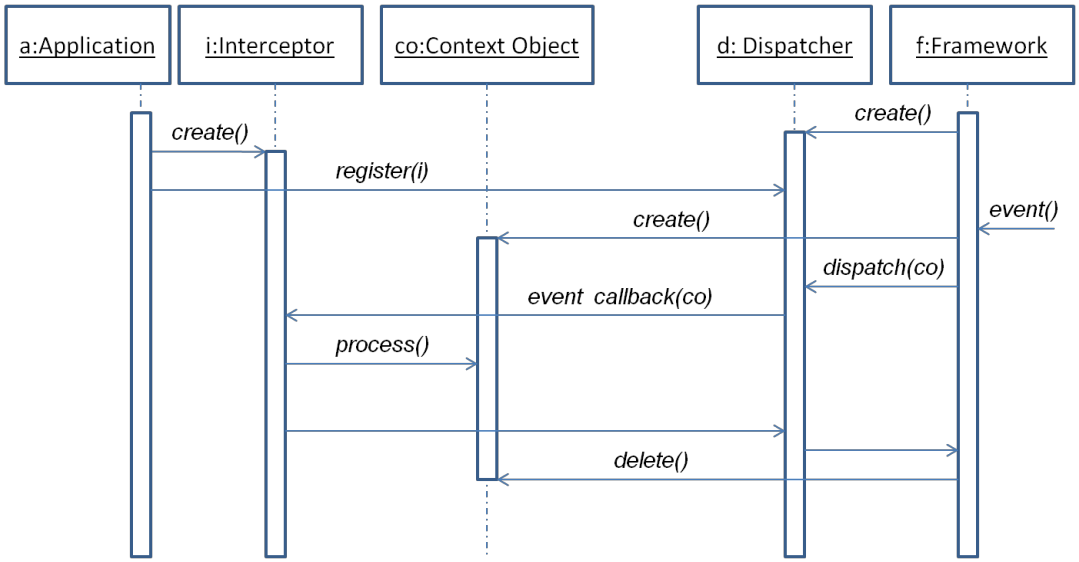

Figure 13: Interaction diagram of the Interceptor pattern

The Interceptor pattern leaves the implementation choice of the Dispatcher service to the developer. In Spring AOP, this task is solved after the Proxy pattern, using standard J2SE dynamic proxies. Proxy is one of the design patterns presented in the famous book of Gamma, Helm, Johnson, and Vlissides (1996). The pattern is used to create a placeholder of the original object for extending its functionality without changing its interface. In the context of the AOP

framework, it allows interception of the call of the original method and delegates this call to the dispatcher of interceptors. The corresponding structure is shown in Figure 14.

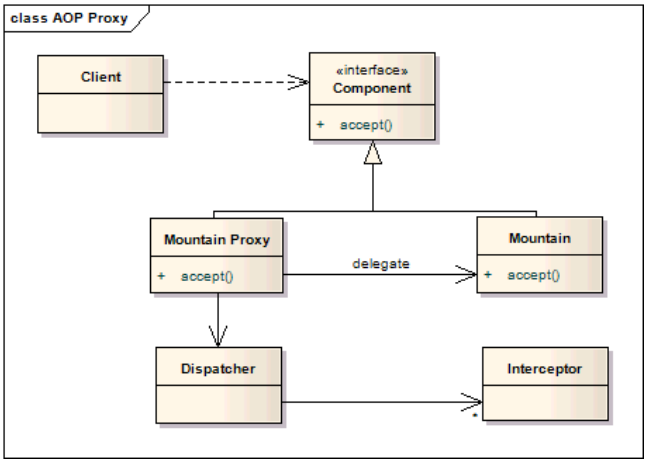

Figure 14: AOP Proxy

In Spring, the AOP proxies are generated at compile time following the AspectJ annotations. The weaving mechanism, however, is performed at run-time. As a result, Spring AOP resolves limitations of the AspectJ approach for the enterprise applications. On the other hand, the dependence of the AOP framework on the IoC container architecture introduces a significant overhead, preventing its application to fine-grain objects such as model components of our example. To address the corresponding AOP applications, Spring AOP provides a hybrid approach by integrating the AspectJ compiler.

Examples of the Heterogeneous Tree-Based Applications

The next subsections consider the problem of processing trees in the context of three application domains: computational accelerator physics, compiler construction, and 3D computer graphics. All these applications demonstrate the importance of heterogeneous types of hierarchical structures where the nodes might have different sets of properties. For example, in

the case of compiler construction, each node represents a programming language construct that can be a whole program or a tiny assignment statement. Such heterogeneous models bring up the main question: how to develop the efficient mechanisms for supporting interchangeable collections of type-oriented algorithms. Resolving this and other related questions will have an immediate practical value and create a basis for building future graph-based applications.

Computational Accelerator Physics

The design and operation of modern accelerators, such as the nuclear colliders or synchrotron light sources, requires sophisticated, flexible and powerful modeling software. On the one hand, the complex problems that need to be studied require non-standard modeling techniques, such as tracking two beams, dealing with complex alignment tolerances for triplet assemblies, analyzing various insertion devices, etc. On the other hand, large accelerators are becoming international collaborative efforts, resulting in the consolidation of various programs into a unified environment aiming to facilitate the development and sharing of the most effective algorithms and approaches. Moreover, stringent parameters of modern high-intensity machines 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 and heterogeneous attributes, all organized in a nested hierarchical structure. For example, Figure 15 shows one of the girder assembles designed for the storage ring of the new National Synchrotron Light Source (NSLS-II). This particular girder hosts several magnets of different types, such as dipole correctors (red), quadrupoles (yellow) and sextupoles (orange). There are many other types of assemblies and each usually addresses one dedicated task. Similar to the

pattern-based approach, the accelerator physicist connects the different types of primitive assemblies into higher level functional units, such as cells or sectors, with the well-defined properties. And finally, the project-specific configuration of cells and sectors forms the entire accelerator lattice design.

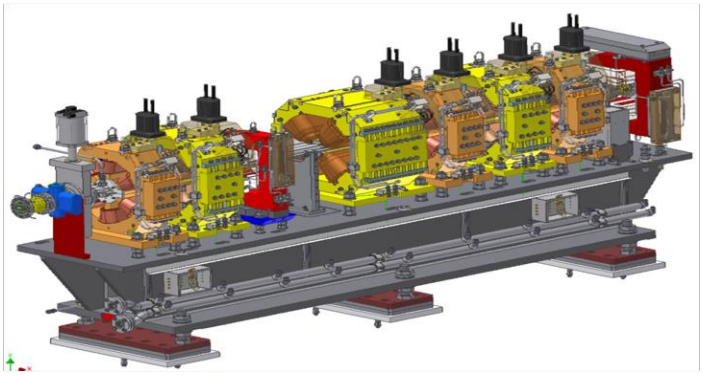

Figure 15: Girder Assembly

The complexity and heterogeneity of this organization prompted a variety of project-specific views and implementations of accelerator descriptions. The Accelerator Description Exchange Format (Malitsky and Talman, 2006) represents one of the most complete and extensible accelerator models addressing different types of accelerator computational tasks. The model is built after the modified variant of the Composite pattern (Gamma, Helm, Johnson, and Vlissides, 1995) including three major participants (see Figure 16):

- **Component**: a node in the accelerator tree organization. There are many different types of lattice components (e.g., Dipole, Quadrupole, etc.) implemented with the corresponding subclasses.
- **Assembly**: a named sector or composite elements with a sequence of frames with installed accelerator components and insertions.
- **Frame:** a layout of installed component. It contains a relative position, misalignments, and a reference to an associated component, sector or accelerator element.

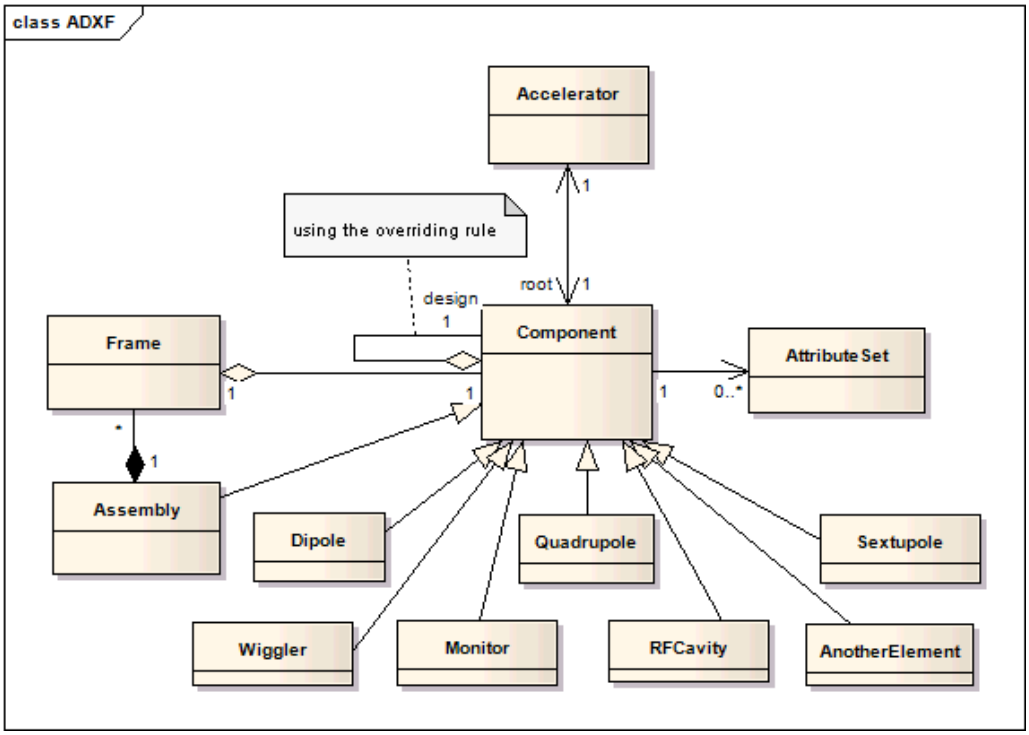

Figure 16: ADXF accelerator model

The accelerator description however represents only raw data which has to be processed in order to extract the different features or observables characterizing the accelerator performance. The list of these features is long including 6D particle coordinates of all particles in a bunch, linear lattice functions, geometrical and momentum aberrations, high-order Taylor maps, pseudo Hamiltonians and others. As a result, the accelerator physicist usually has to deal with algorithms that vary along three orthogonal dimensions as shown in Figure 17. First, algorithms can be grouped according to propagating features. Second, different types of accelerator elements require individual approaches. Finally, each feature-specific and element-specific procedure can be implemented in many ways. For example, in particle tracking applications, algorithms vary from the most efficient matrix-based approaches to the most accurate brute-force direct integrators of equations of motion.

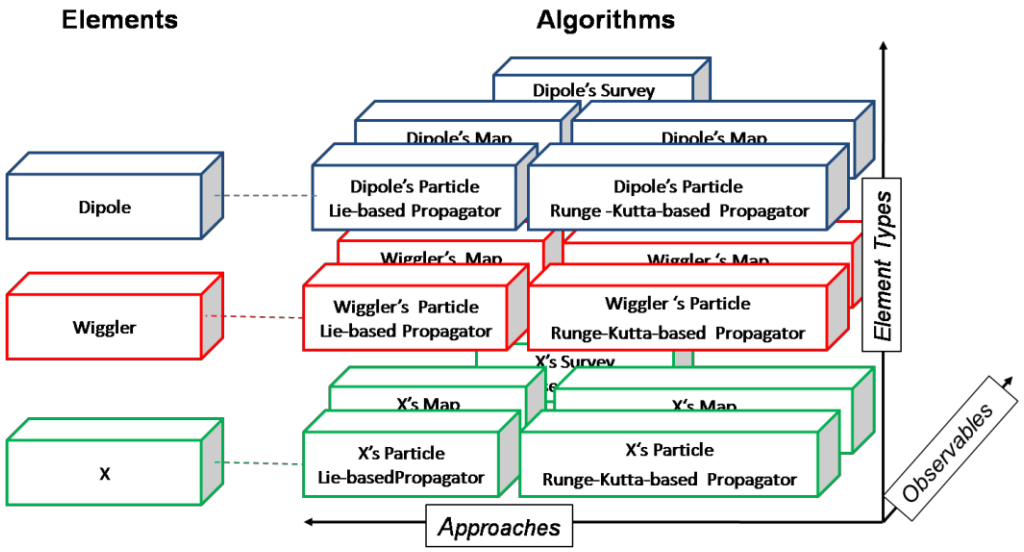

Figure 17: Three-dimensional view of accelerator physics algorithms

Ideally, all these processing algorithms should be combined and available in the common research and development environment. Their integration however introduces a serious problem in designing the universal accelerator structure and multi-purpose plug-in framework. As a result, in the early history of accelerator simulation, this problem generated a huge compendium of single task-oriented accelerator codes (Los Alamos Accelerator Control Group, 1987). An important step in their coordination occurred at a workshop for the standardization of the accelerator input format based on the MAD input language (Carey and Iselin, 1984). A common accelerator input format addressed immediate requests of the multi-team international projects. But it did not resolve the principal problems of modern accelerator computational tasks. Their solution was dependent on the development of an open and configurable simulation environment addressing two major requirements: a generic description of existing and future accelerator projects and a universal mechanism for processing accelerator heterogeneous structures with the interchangeable collections of accelerator algorithms and approaches.

As a result, in 1995, several developers of accelerator programs formed collaborations in order to start two independent projects: CLASSIC (Class Library for Accelerator System Simulation and Control) and UAL (Unified Accelerator Libraries). Both projects addressed similar goals but used different approaches. The CLASSIC project (Iselin, 1996) aimed to refactor and consolidate the existing FORTRAN programs using the Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1995). This pattern however brought a strategic limitation into the software framework complicating the integration of new types of accelerator elements and physical effects. Such elements were essential research topics in new accelerator projects. In a few years, this CLASSIC collaboration was canceled.

Facing the same problem, the UAL project suggested replacing the Visitor pattern with the new framework based on the Mutable Class concept (Malitsky and Talman, 1998) described in the Methodology chapter. This framework had been successfully employed in several major accelerator projects significantly extending the scope of initial applications and computer environments. For example, the same approach was perfectly deployed on parallel clusters for simulating the time-consuming complex scenarios requiring the combination of parallel and conventional algorithms. The comprehensive analysis of different use cases is presented in the Results chapter.

Compiler Construction

Starting with the invention of high-level programming languages in the 1950s, the construction of compilers is now one of the oldest fields of computer science. Since their introduction, compilers evolved into the large algorithm-rich systems that have to deal with complex multi-step chains of operations: lexical analysis, parsing, semantic analysis, optimization and code generation. The results of each step are maintained in an intermediate representation (IR) which can be processed multiple times before emitting the target program.

The structure of the IR depends on many factors like the complexity of the programming language and requirements of processing algorithms. Moreover, many compiler systems, so called compiler-compilers, represent configurable automatic builders constructing compilers from language grammas described in an extended BNF notation. Such systems are especially important for supporting domain-specific languages (DSL) tailored to specific tasks including descriptions of domain models, complex query languages, configuration file formats, state notation languages, network protocols, and many others.

The language structure is determined by its context-free grammar: a set of production rules for defining and connecting the language elements. The production rules of major modern general-purpose and domain-specific languages are defined in the form of $A \rightarrow \gamma$, where the left-hand side A consists of a single nonterminal symbol and the right side γ is a finite sequence of terminals and nonterminals. This grammar and associated semantics naturally suggest the hierarchical organization of the compiler's intermediate representation, called Abstract Syntax Tree (AST). In the compiler construction, the different variants of intermediate representations can be divided into two major categories: homogeneous and heterogeneous ASTs.

The homogeneous AST has only a single node type. This design facilitates the development of generic frameworks and has been implemented in many popular parser generators, for example, ANTLR (Parr and Quong, 1995; Parr, 2013). Being part of a framework, the single node type has to encapsulate only the basic data required by all applications. This common dataset includes an identifier, a type field, a reference to a parent node, and a collection of children. Such simplicity of the homogeneous structure has both strengths and weaknesses. On the one hand, it eases the development, maintenance, and documentation of the AST objects. On the other hand, such a

universal organization clashes with the heterogeneous nature of programming languages encompassing various types of concepts.

A heterogeneous AST is built after the Interpreter pattern mapping the language grammar into the corresponding hierarchical object-oriented structure as shown in Figure 18. This approach is taken by extensible compiler systems, for example, JastAdd (Hedin and Magnusson, 2003; Hedin, 2010). There are many variants of the heterogeneous ASTs. In general, the internal nodes represent a programming language construct and their children implement alternatives. Such organization better captures the program semantics. As a consequence, heterogeneous ASTs allow developers to make superior code and to more effectively apply a full spectrum of the powerful techniques offered by object-oriented methodology: built-in type system, polymorphism, inheritance, etc.

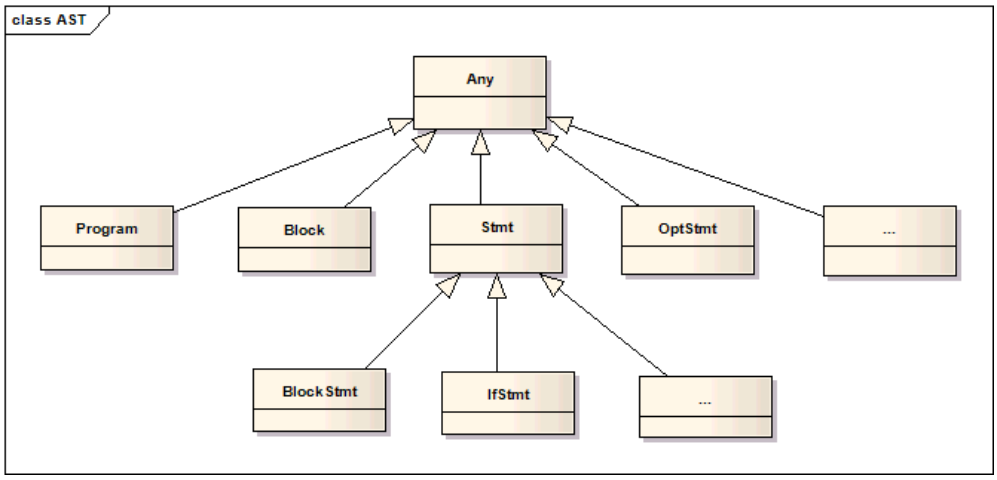

Figure 18: Example of the heterogeneous AST model (Hedin and Magnusson, 2003)

As discussed earlier, the AST serves as an intermediate representation for the different operations of the multi-phase compiler scenarios. Following a common scheme, the associated algorithms of the meta-compiler system can be presented in two-dimensional view as shown in Figure 19. First, algorithms can be grouped according to the compiler's phases, such as the

semantic analysis, optimization or code generation. Secondly, algorithms vary for different types of the AST nodes.

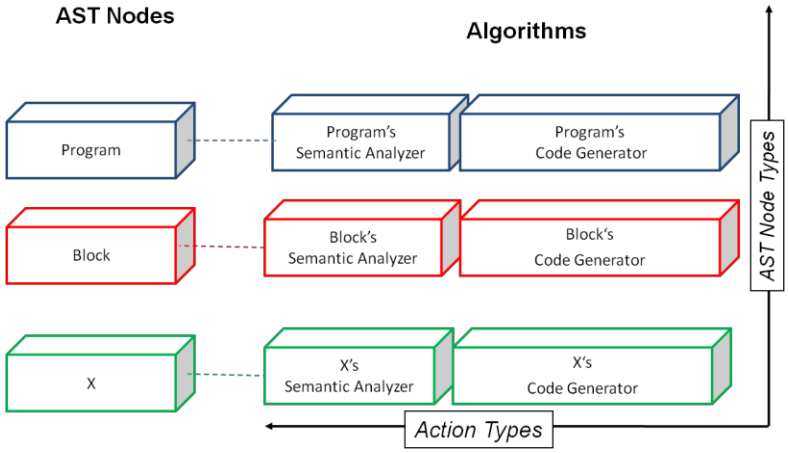

Figure 19: Two-dimensional view of compiler-compiler algorithms

Similar to the computational accelerator physics domain, the interoperability between the compiler internal representations and diverse algorithms introduced a design dilemma. In the case of homogeneous structures, the Visitor pattern provides an adequate solution, allowing to group algorithms according the compiler's phase and programming language. The simplicity of the homogeneous structure however does not come free and eventually results in the complexity of processing algorithms. As a result, the multi-language compiler-compilers systems lean towards heterogeneous AST structures. In this case, the Visitor pattern does not address extensibility requirements and developers need to find alternative solutions. For example, the JastAdd configurable metacompiler construction system (Hedin and Magnusson, 2003) suggested a composite approach combining the object-oriented mechanism with the proprietary declarative implementation of the aspect-oriented concepts. The JastAdd extension mechanism however is static and does not resolve the run-time issues associated with the aspect-oriented approach. Initially separated in the different files, the JastAdd aspects are eventually merged and

disappear into the huge monolithic AST classes preventing its run-time interchange and extension.

As will be shown in the Methodology chapter, the Mutable Class represents a similar solution combining the advantageous features of both the Visitor pattern and aspect-oriented weaving approach. On the one hand, it resolves the Visitor dependency cycle by adding the preliminary step for weaving algorithms with the processed structure. On the other hand, this weaving procedure is not limited by compile-time as in the case of the JastAdd original approach and preserves the run-time behavior of the Visitor pattern. To demonstrate the advantage of this approach, the Mutable Class pattern was integrated with the JastAdd framework. The corresponding application has been presented at the OOPSLA conference (Malitsky, 2008) and is thoroughly described in the Results chapter.

3D Computer Graphics

3D computer graphics is probably one of the most prominent domains of computer science. Its applications dramatically changed traditional multimedia and technical resources with the materialization of new concepts, like virtual reality, and the introduction of new ways of visualizing our world. Capturing and presenting the beauty and richness of a 3D environment onto a 2D computer screen is a complex procedure involving multiple tasks and algorithms implemented in multi-component toolkits.

The graphics software stack interacts with the graphics hardware via a low-level API designed around the rendering pipeline, processing graphic elements into a video display frame buffer. OpenGL (Shreiner et al., 2013) represents one of the most popular rendering API and pipeline specifications and is implemented on many hardware platforms and in multiple programming languages. In the OpenGL framework, a variety of geometries are specified via a small set of geometric primitives based on points, lines, triangles, quadrilaterals, and polygons.

In turn, the description of the OpenGL geometric primitives is based on one generic representation consisting of a vertex array and a set of state variables. Each vertex has three-dimensional coordinates and can be explicitly assigned an RGBA color and normal vector. The state variables complement the vertex data with other information such as geometric transformation, material components, drawing style, and lighting model. The choice of this elementary graphics description has been determined by the performance requirements of the pipeline algorithms. At the same time, such a low-level approach significantly complicates the description of the 3D world objects requiring the explicit definition of vertex coordinates and associated parameters or writing numerous object-specific extensions. This gap between the complexity of the 3D graphics applications and a low-level hardware-oriented interface is addressed by the higher level frameworks, such as Open Inventor (Wernecke et al., 1994; Heck, 2010) , OpenSceneGraph (Wang and Qian, 2012), and Three.js (Dirksen, 2013) based on the scene graph models.

The scene graph is an object-oriented tree data structure providing the application-oriented spatial representation of a graphical scene. From the perspective of the design pattern approach, scene graph nodes can be considered as a composite adapter processing and delegating drawing requests of high-level graphics applications to a low-level rendering pipeline. The implementation and hierarchical organization of these nodes vary for the different toolkits. For example, the Open Inventor model is built from the nodes of multiple types including group nodes (e.g., separator or switch), shapes (e.g., sphere and quad mesh), lights and cameras (e.g., perspective or orthographic), property nodes (e.g., material or texture), engines creating the dynamic interdependencies among nodes, transformation nodes, and sensors responding to the

graph changes. The core of this model is based on a hierarchy of shape nodes that can be divided into four categories (see Fig 20):

- object-oriented wrappers of the OpenGL primitives: points, lines, and polygons
- simple shapes: cone, cube, sphere, and cylinder
- non-uniform rational B-spline (NURBS) curves and surfaces
- 2D and 3D text objects

To manage the variety of types in a consistent way, the Open Inventor specification directly follows the standard object-oriented model and represents each node type with the corresponding class. As a result, the collection of the node-specific classes is derived from a common base class SoNode and can be combined together.

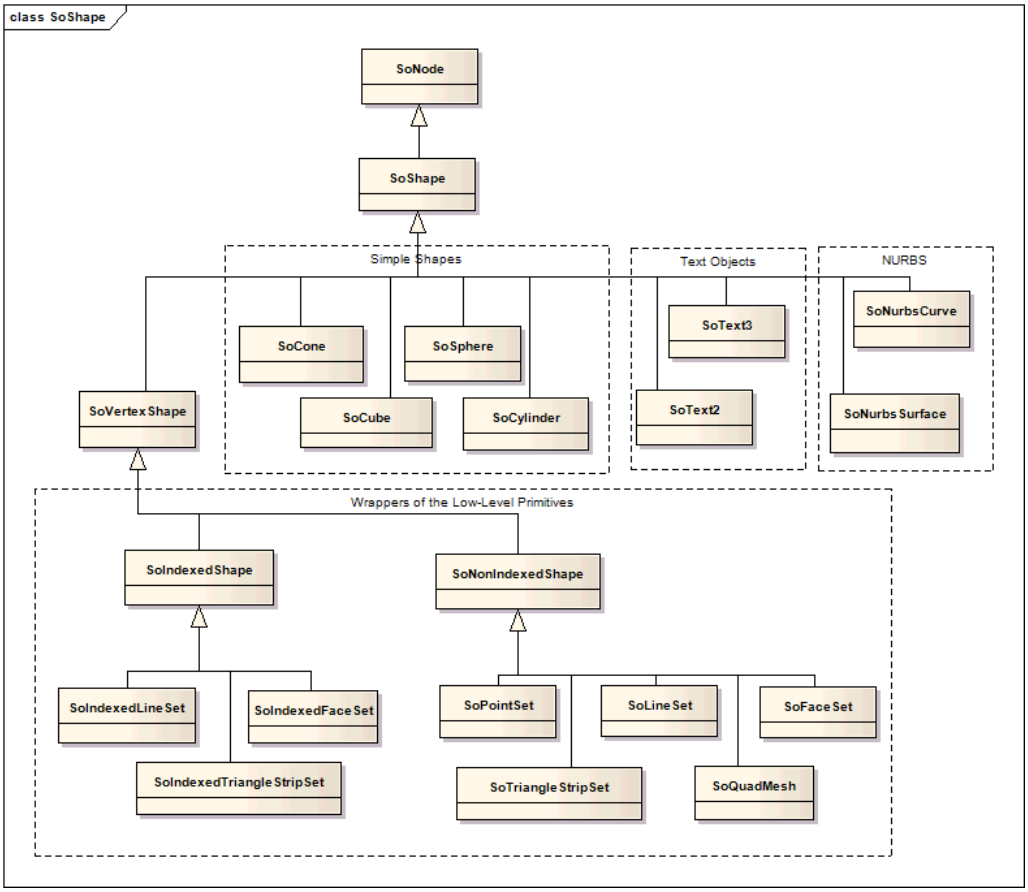

Figure 20: Shape nodes of the Open Inventor scene graph model

The OpenSceneGraph model applies another approach, using the divide and conquer technique and the "has-a" relationship to connect the single dedicated node class Geode with a collection of geometry objects composed from the separate hierarchy of the Drawable classes. Despite some design differences, both toolkits support the comparable catalogs of the geometry types and provide a consistent object-oriented mechanism for developing new extensions.

The object-oriented structure of the scene graph model not only adapts and extends the low-level API but also augments the pipeline processing architecture with a framework of the graph traversal methods. In the context of this framework, a pipeline-oriented sequence of the rendering commands is implemented as a product of the corresponding traversal procedure on the scene graph. The drawing traversal allows improvement of the performance of the rendering process by optimizing the management of the OpenGL state attributes and grouping of commands, called display lists, stored for later execution. Moreover, the same traversal technique can be applied to many other important tasks extending the capabilities of the rendering pipeline. In addition to rendering, the OpenSceneGraph specification emphasizes two major types of traversals: update and culling. The update traversal handles the dynamic modifications of the scene graphs prompted directly by either the applications or with callback functions assigned to nodes. Culling is a process of checking visibility of scene objects within a view frustum involving the 3D bounding box calculations and consideration of the opaque and translucent geometries. The collection of scene graph traversals is not limited by three tasks and may include other procedures or divided into sub-tasks, such as writing scene graphs into files, searching for nodes, or performing application-specific actions. Again, this many-to-many association between heterogeneous nodes and different traversal algorithms can be described with the two-dimensional view as shown in Figure 21. To deal with multiple traversal

algorithms, the Open Inventor and OpenSceneGraph toolkits encapsulated them into dedicated classes. Depending on the task, these classes can be generic or associated with the different types of scene nodes.

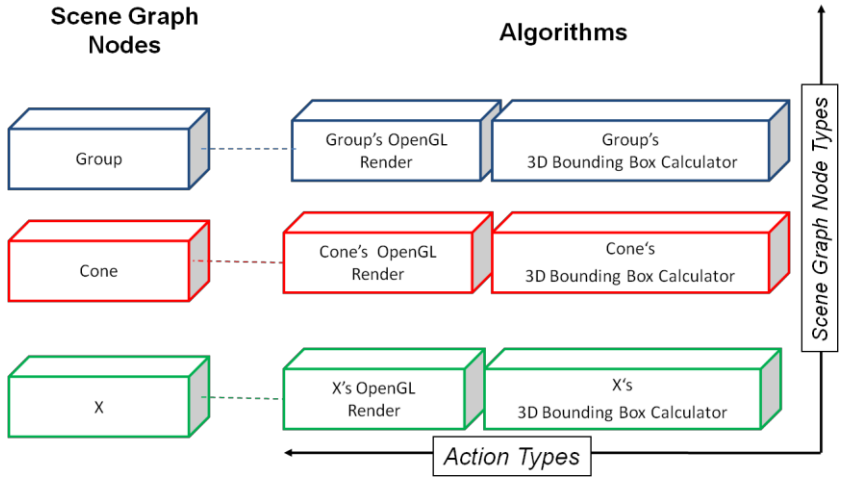

Figure 21: Two-dimensional view of scene graph algorithms

For bundling multiple processing algorithms together with the graph nodes, the Open Inventor team derived an original approach based on the combination of a proprietary run-time system and the virtual function mechanism. The approach is described in the context of the Inventor Toolmaker (Wernecke et al., 1994) providing guidance for building the Open Inventor extensions. According to this specification, each action maintains a list of node-specific static methods. When an action is applied, it obtains the type identifier from the processed node and dispatches the action request to the selected method. In the case of built-in actions, such as SoGLRenderAction or SoGenBoundingBox, all node-specific entries of the action method list are assigned a single static method that calls the corresponding built-in virtual function of the SoNode base class. Using virtual functions of the scene nodes allows the application of standard object-oriented dispatching mechanisms and facilitates the implementation of the action algorithms. A set of virtual functions, however, represents a part of the framework interface that

cannot be changed for the new extensions. To address the extensibility requirements, the OpenSceneGraph team developed one of the proprietary versions of the Generic Visitor pattern. (Vlissides, 1999; Visser, 2001). In Open Inventor, this extensibility issue is resolved with the consistent registration mechanism that is closely related with the Mutable Class approach. The Methodology chapter provides the quality factor assessment of the Mutable Class pattern and comparison with different versions of the Visitor pattern including Generic Visitor.

Large-Scale Graph Data Processing

The topic of heterogeneous data processing is becoming especially important in the context of a new field know as Big Data Science. In particular, this includes the analysis of large-scale graph data sets. This filed encompasses multiple application domains, for example, social networks, transportation routes, and protein-protein interaction networks. Graph theory itself, however, is not new, tracing its origins all the way back to the famous paper on Seven Bridges of Konigsberg written by L. Euler in 1736. Since that time, graph structures and algorithms have become instrumental for solving multitudes of practical problems in such domains as artificial intelligence and operations research, among others. Big Data Analysis, or the so-called “Fourth Paradigm” (Hey, Tansley, and Tolle, 2009), introduced a new conceptual landscape requiring the reconsideration and refactoring of existing technical solutions.

The initial landscape of the Big Data technologies was designed after Google's I/O stack, which included the Google File System (GFS), Bigtable distributed storage system, and the MapReduce processing framework. GFS (Ghemawat, Gobioff, and Leung, 2003) represented a large-scale fault tolerant distributed file system running on commodity computers. To address the Big Data requirements, the GFS developers relaxed the POSIX interface, reusing a plain

single-master architecture and focusing on the scalability, high-throughput, and fault tolerance issues. Bigtable (Chang, et al., 2006) extended GFS with a data storage layer. It resembled the architecture of parallel databases, but relaxed the relational data model to confront the scalability requirements and to support a soft schema of web-related semi-structured data. The resulting architecture was built around a sparse distributed multi-dimensional sorted map with keys and values represented by uninterpreted strings. For processing this large-scale distributed data, the Google team introduced a new parallel programming framework inspired by two Lisp primitives, *map* and *reduce*, giving the apt name to this approach, MapReduce (Dean and Ghemawat, 2004). The model was designed around the communication-free, so called embarrassingly parallel, use case that split the computer-intensive tasks into the parallel *map* functions that processed requests and generated intermediate key/value pairs. The *reduce* function then received an intermediate key with its set of values and merged them together. Such a simplified approach now provides a reliable and scalable solution for many web-oriented data processing systems including Dremel (Melnik, et al., 2010), further representing Google's influential technology for executing queries over nested data.

The embarrassingly parallel model of the MapReduce processing framework, however, could not address the requirements of all algorithms, and even became an obstacle for many machine learning and graph-based applications. One of the major limitations was associated with the missing support for the interactive processes. For example, algorithms like gradient descent, expectation-maximization, and belief propagation, iteratively refine the space of parameters until achieving some termination condition. Additionally, graph models typically involve more complex computational dependencies in the data than conventional MapReduce applications. Finally, the processing of graph algorithms leans towards an asynchronous model, exhibiting a

flexible and dynamic degree of parallelism. These and other shortcomings of the MapReduce parallel framework have been addressed by several teams bringing new computational models. We now consider the Pregel model developed by the Google team (Malewicz et al., 2010).

Pregel is a scalable graph processing system designed after the Bulk Synchronous Parallel (BSP) computation model. In accordance with BSP (Valiant, 1990), graph algorithms are expressed as a sequence of iterations called supersteps. Each superstep represents atomic units of parallel computations. Initially, all vertices are assigned an active status. During a superstep, each active vertex V runs the *compute()* user function that reads messages sent to V in the previous superstep, sends messages to other vertices, and modifies the state of V and its outgoing edges. The active vertex can deactivate itself by voting to halt and turn to an inactive state. To implement such a Pregel program, a developer needs to subclass the predefined Vertex template class and override the virtual *compute()* method. Listing 2 shows the C++ interface of the Vertex template and an example of the *compute()* method implementing the famous PageRank algorithm (Page, Brin, Motwanl, and Winograd, 1998).

```
template <typename VertexValue,
          typename EdgeValue,
          typename MessageValue>
class Vertex {
  public:
    virtual void Compute(MessageIterator* msg) = 0;

    const string& vertex_id() const;
    int64 superstep() const;

    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
                const MessageValue& message);

    void VoteHalt();
};
```

```
class PageRankVertex  : public Vertex<double, void, double> {
 public:
   virtual void Compute(MessageIterator* msgs) {

     if (superstep() >= 1) {
       double sum = 0;
       for (; !msgs->Done(); msgs->Next())
          sum += msgs->Value();
       *MutableValue() =  0.15 / NumVertices() + 0.85 * sum;
     }

     if (superstep() < 30) {
       const int64 n = GetOutEdgeIterator().size();
       SendMessageToAllNeighbors(GetValue() / n);
     } else {
       VoteToHalt();
     }
   }
};
```

Listing 2: Vertex API and PageRank implemented in Pregel (Malewicz et al., 2010)

The Vertex template class is parameterized with three template arguments defining the three value types associated with vertices, edges, and messages. The *compute()* method takes the inbound messages from the vertices, iterates over them and sums the associated values in order to calculate the rank of the assigned vertex. In the end of this superstep, the method sends messages through outgoing edges. In this particular example, the iteration process is finished after achieving superstep 30.

Current web-oriented algorithms, such as the PageRank and community detection methods, are usually based on homogeneous graphs built from generic nodes connected with links of the same relation type. Real-world models, however, represent far more complex graphs, consisting of heterogeneous vertices and edges. For example, a healthcare information system includes a set of object types, such as doctor, patient, disease, and treatment, and multiple types of relations among these objects. In the context of the Big Data analysis domain, such systems are known as heterogeneous information networks. Sun and Han in their manuscript (2012) provided a comprehensive comparison and overview of the corresponding data-mining algorithms. Particularly, the authors identified the following six categories:

- ranking-based clustering: collection of hybrid approaches allowing to cluster one type of object (e.g., venues) based on a proximity measure calculated from the ranking of other types of objects (e.g., authors) and links in the network;
- classification of heterogeneous information networks: generalized variants of the homogeneous applications extended with classes composed of multi-typed data (e.g., movies, directors, and actors) sharing a common topic (e.g., genre). Following the idea of ranking-based clustering, the accuracy of these algorithms can be further enhanced with ranking techniques;

- meta-path-based similarity search: methods for finding similarity in networks using meta-paths defined as composite relations between different types of objects (e.g., venue-paper-author-paper-venue and venue-paper-topic-paper-venue);
- meta-path-based relationship prediction: a category of supervised models for predicting relationships across heterogeneous typed objects;
- relation strength-aware clustering with incomplete attributes: collection of probabilistic clustering models taking into account heterogeneous links between objects and an incomplete attribute space (e.g., user-provided attributes);
- user-guided clustering via meta-path selection: composite clustering approaches using weighted meta-path combinations selected by supervised procedures.

The original paper of the Pregel framework does not provide an explicit solution for managing heterogeneous use cases. In fact, the generic interface of the Vertex class cannot adapt to a variety of scenarios and needs to be reconsidered from the perspective of the corresponding design patterns, such as Visitor. The brief analysis shows that the application of the Visitor pattern introduces two principal constraints. First, the pattern assumes the well-defined data model of the application domain. In the context of heterogeneous information networks, this requirement would hardly be acceptable. Second, the Visitor-based traversal approach clashes with the behavior of the Pregel processing model. As a result, this problem requires new solutions. In contrast with tree-based applications, processing heterogeneous graphs however represents a very new field accumulated a few research papers. Therefore, the dissertation overviews this category of applications to highlight the direction for future studies that will be discussed in the final chapter.

Summary

The chapter presents the problem of processing heterogeneous models addressed by the Mutable Class design pattern. This overview includes two major topics. The first topic is dedicated to the analysis of existing solutions, including the Visitor pattern, its various extensions, and the aspect-oriented approach. Each solution is explained with the corresponding class diagram and a brief description of its strong and weak features. The second topic considers four application domains introducing this problem. Three of them deal with heterogeneous tree-based models. These are computational accelerator physics, compiler construction, and 3D computer graphics. The fourth use case represents a new direction associated with the development and processing of large-scale heterogeneous information networks.

Chapter 3

Methodology

The chapter overviews the Mutable Class pattern designed to provide an extensible run-time solution for processing heterogeneous data models with multiple families of algorithms. The idea of the Mutable Class was initially introduced in the framework of the United Accelerator Libraries (Malitsky and Talman, 1998) for building dynamic associations among heterogeneous physical devices and modeling algorithms. Later, the approach was successfully validated in the context of the JastAdd metacompiler construction system (Malitsky, 2008). To present this approach, the chapter is broken into three sections. The first section introduces the conceptual model based on the UML specification. The second section provides a formal description of the Mutable Class according to the design pattern methodology (Gamma, Helm, Johnson, and Vlissides, 1995). The third section assesses and compares its quality characteristics with existing solutions described in the previous chapter.

Mutable Class Approach

The Mutable Class approach was designed after the Class model of the Unified Modeling Language (UML) specification. UML is a standard software engineering language developed from the consolidation of three major object-oriented methodologies: the Booch method (1991), object-modeling technique (Rumbaugh et al., 1990), and object-oriented software engineering (Jacobson, 1992). As a modeling specification, UML represents a metamodel for instantiating user-specific object-oriented models. To support the model-driven architecture (MDA) tools, the

Object Management Group (OMG) developed the Meta-Object Facility (MOF) augmenting the UML specification with a meta-metamodeling layer. Figure 22 shows the corresponding example from the UML specification (OMG, 2011) illustrating the UML four-layer metamodel hierarchy.

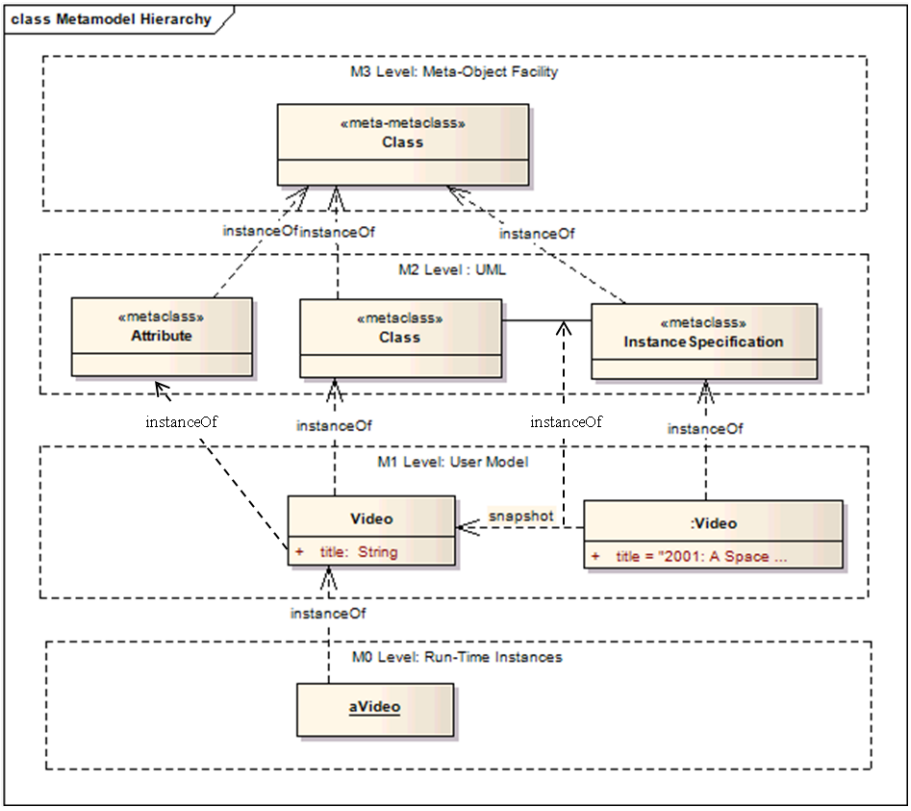

Figure 22: Example of the four-layer metamodel hierarchy (OMG, 2011)

According to this architecture, the UML metamodel elements, such as Attribute, Class, and InstanceSpecification, are instantiated from the Class meta-metaclass of the MOF layer and instantiated by the user model including the Video class and the :Video object. The bottom layer contains the run-time instances of the user model classes, such as aVideo.

The UML specification is organized into two parts: Infrastructure and Superstructure. The UML Infrastructure consists of abstract and common modeling elements that are reused by both the MOF and UML layers. In the case of MOF, the Infrastructure metaclasses are imported without changes, while in the case of UML these model elements are extended with the new

features. The UML Superstructure defines the structural, behavior, and auxiliary constructs used in the UML diagrams. All of these constructs are directly or indirectly based on the Kernel package encapsulating the core modeling concepts, including classes, associations, and packages. The streamlined diagram of the Class metaclass is shown in Figure 23.

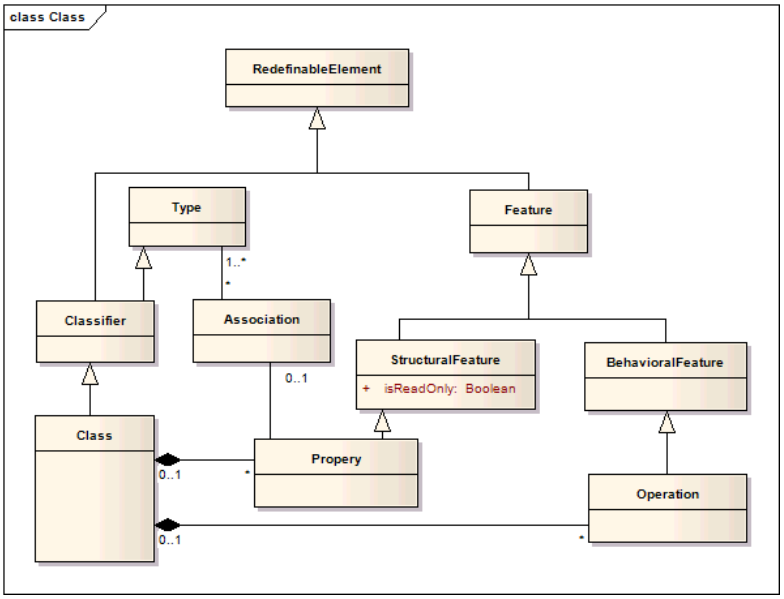

Figure 23: Streamlined diagram of the UML Class metaclass

According to the UAL specification (OMG, 2011):

> "Class is a kind of classifier whose features are attributes and operations. Attributes of a class are represented by instances of Property that are owned by the class. Some of these attributes represent the navigable ends of binary associations".

In addition to the structure definition, the UML Infrastructure provides two extensibility mechanisms specified in the Redefinitions and Changeabilities packages. The Redefinition package introduces an abstract metaclass RedefinableElement that is associated with capability to be redefined "more specifically or differently in the context of another classifier that specializes (directly or indirectly) the context classifier" (OMG, 2011). In Figure 23, RedefinableElement is specialized by Classifier, Property, and Operation. This platform-

independent semantics directly corresponds to the inheritance mechanism of the object-oriented programming languages, such as C++ and Java. The Changeabilities package specializes the StructuralFeature metaclass by adding an attribute defining if the value of this feature can be modified. For features representing the navigable ends of binary associations, the Changeabilities mechanism corresponds to the composition approach of object-oriented models including the component-oriented technologies, like COM, DCOM, CORBA Component Model (CCM), and Enterprise JavaBeans (EJB).

Inheritance and composition are two major approaches used in the object-oriented software design. According to the authors of the design pattern book (Gamma, Helm, Johnson, and Vlissides, 1995), the composition approach provides the preferred solution, improving the coupling and cohesion characteristics of software systems. The Changeabilities mechanism of the UML specification, however, does not support the interchangeability of operations. The Mutable Class approach addresses this issue on the user model level by instantiating a changeable operation as the ComponentOperation class and adding the ComponentMutator singleton that implements the Component-ComponentOperation binary association as shown in Figure 24.

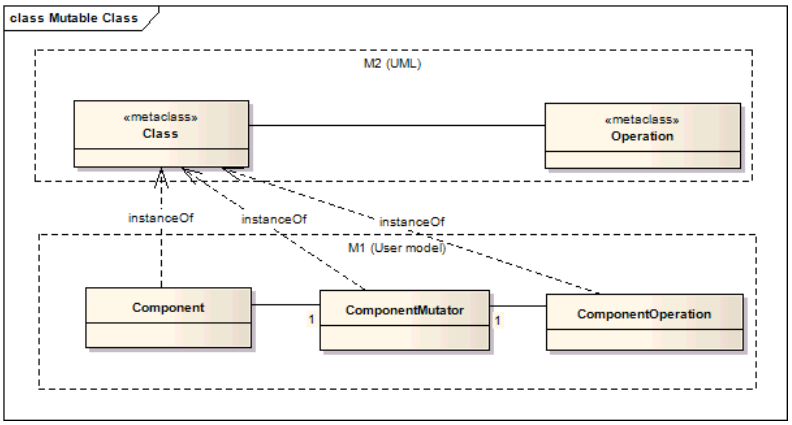

Figure 24: Mutable Class approach

The encapsulation of interchangeable operations into dedicated classes has been suggested by the Strategy pattern (Gamma, Helm, Johnson, and Vlissides, 1995). Figure 25 shows the structure of this pattern using the Mutable Class terminology.

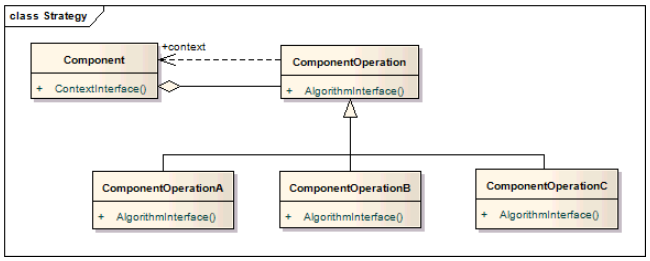

Figure 25: Structure of the Strategy pattern

The Strategy pattern contains two primary classes: Component and ComponentOperation. The Component object maintains a reference to the ComponentOperation instance and serves as the context of operation invocation. The Mutable Class approach assigns these two roles to separate participants according to the definition of the metaclass Operation of the UML specification (OMG, 2011): "An operation is owned by a class and may be invoked in the context of objects that are instances of that class". In this model (see Figure 26), the ComponentMutator singleton represents the extension of the Component class, maintains a reference of the ComponentOperation instance, and invokes it in the context of the Component object.

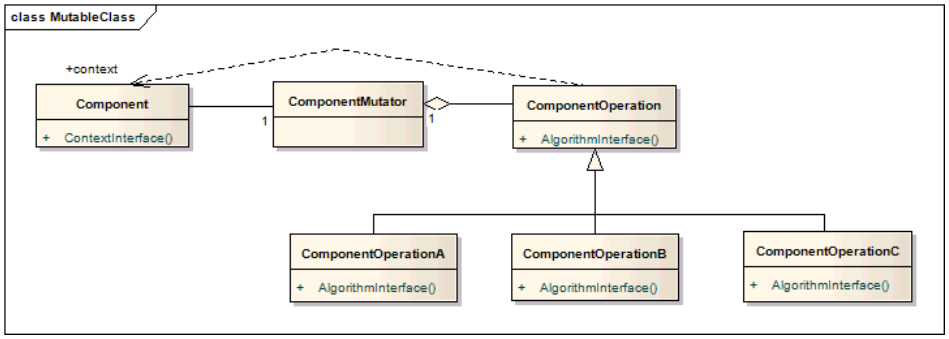

Figure 26: Structure of the Mutable Class model

To manage multiple Mutable Classes, they can be configured with the Component Mutator Linker as shown in Figure 27. This linker maintains a registry of mutators and provides a runtime configuration mechanism for binding them with registries of operations.

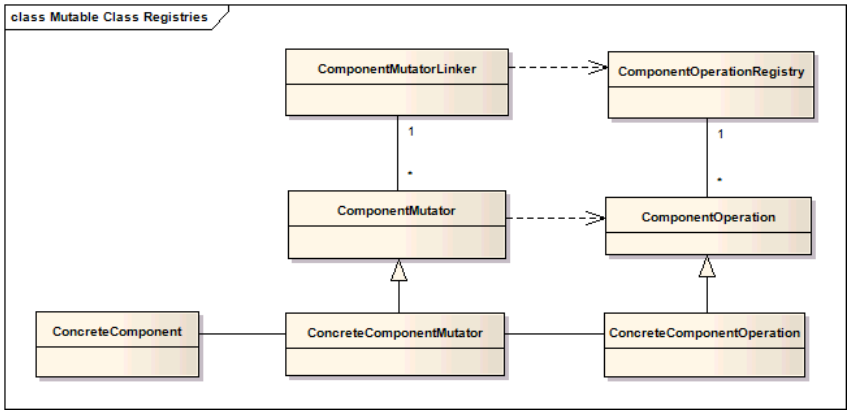

Figure 27: Class diagram of the Mutable Class configuration approach

The Mutable Class configuration approach is related to the aspect-oriented programming (AOP) paradigm, augmenting the inheritance and composition mechanisms with the weaving procedure for inserting structural and behavioral features across multiple classes of the object-oriented programs. These crosscutting changes are associated with different aspects of software systems, for example, persistence and logging. From the perspective of the AOP paradigm, the Mutable Class addresses the timing aspect associated with the evolution of software programs. On the system level, the timing aspect is already addressed by multiple technologies, like version control systems or update managers of operating systems. In software application, this topic is closely related to the agent-oriented programming paradigm, focusing on the development of the agent-based dynamic environments.

The new programming paradigms, however, introduce significant overhead associated with the broad scope of new concepts and the impedance mismatch with the existing object-oriented

programming languages. In contrast with these paradigms, the Mutable Class approach follows the incremental development procedure starting with a lightweight solution addressing the immediate applications. The next section presents the Mutable Class approach in the context of the design pattern for processing heterogeneous tree-based models.

Mutable Class Pattern

Intent

Provide an efficient run-time mechanism for processing large-scale heterogeneous models with multiple data processing algorithms.

Motivation

The development of the Mutable Class pattern was motivated by applications associated with data processing of large-scale heterogeneous models, such as computational accelerator physics models, abstract syntax trees, 3D scene graphs, and information networks. To facilitate the description of the addressed problems and corresponding solutions, the approach can be illustrated with a simplified example that uses an imaginary game world. In our case, this world is built by employing a combination of numerous components from a limited set of types such as Mountains. These components are then assembled and grouped into Regions, thus forming a hierarchical model. Once the world is generated, it needs to be inhabited and explored by multiple teams, such as observers and settlers, each using different strategies and missions. While a standard game edition then provides a basic set of model components and explorers, it can also grow by integrating multiple third-party extensions including new components such as forests and cities. In our example, they are represented by single component X. The corresponding class diagram of these component-strategy associations is shown in Figure 28.

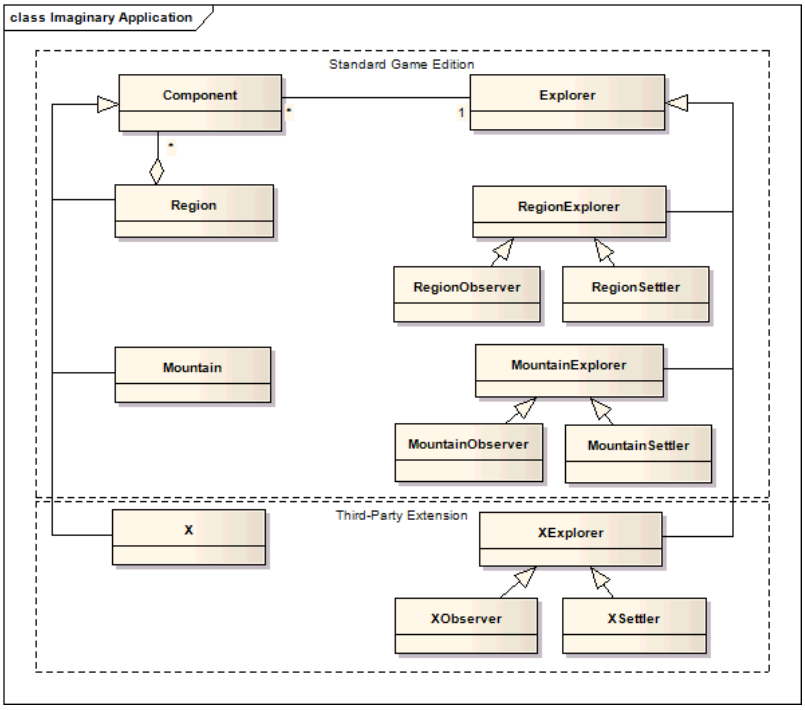

Figure 28: Component-strategy associations of the imaginary game application

The diagram follows the Strategy pattern (Gamma, Helm, Johnson, and Vlissides, 1995) separating object state and behavior, implementing each of them in dedicated classes. The Strategy pattern, however, does not define a mechanism for managing collections of these classes, nor for building associations between components and strategies. This issue has been addressed by the Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1995), which suggests combining component-specific operations, such as *visit(r: Region)* and *visit(m: Mountain)*, into the strategy-specific Visitor classes. This hard-coded approach, however, freezes class hierarchies of the application models, preventing new extensions. Particularly, adding the new world component X requires editing all visitor classes by adding to each class a *visit(x: X)* method. The Mutable Class approach resolves this extensibility limitation by splitting the Visitor hard-coded monolithic interface into fully decoupled component-specific singletons, so called class mutators, implementing component-operation associations (see Figure 29). A class mutator

connects a component-specific class with Strategy-based interchangeable operations. In the same time, it augments the Strategy pattern with an efficient mechanism for configuring component-operation associations of any number of components via a single instance. As a result, the mutable class model represents a triplet consisting of the component class, mutator and Strategy-based operation. As shown in Figure 29, mutable classes of the basic model and third-party extensions can be independently developed and combined together into the corresponding registries of mutators and operations. The Component Mutator Linker maintains a registry of mutators and provides a runtime configuration mechanism linking mutable class triplets for the selected registry of operations.

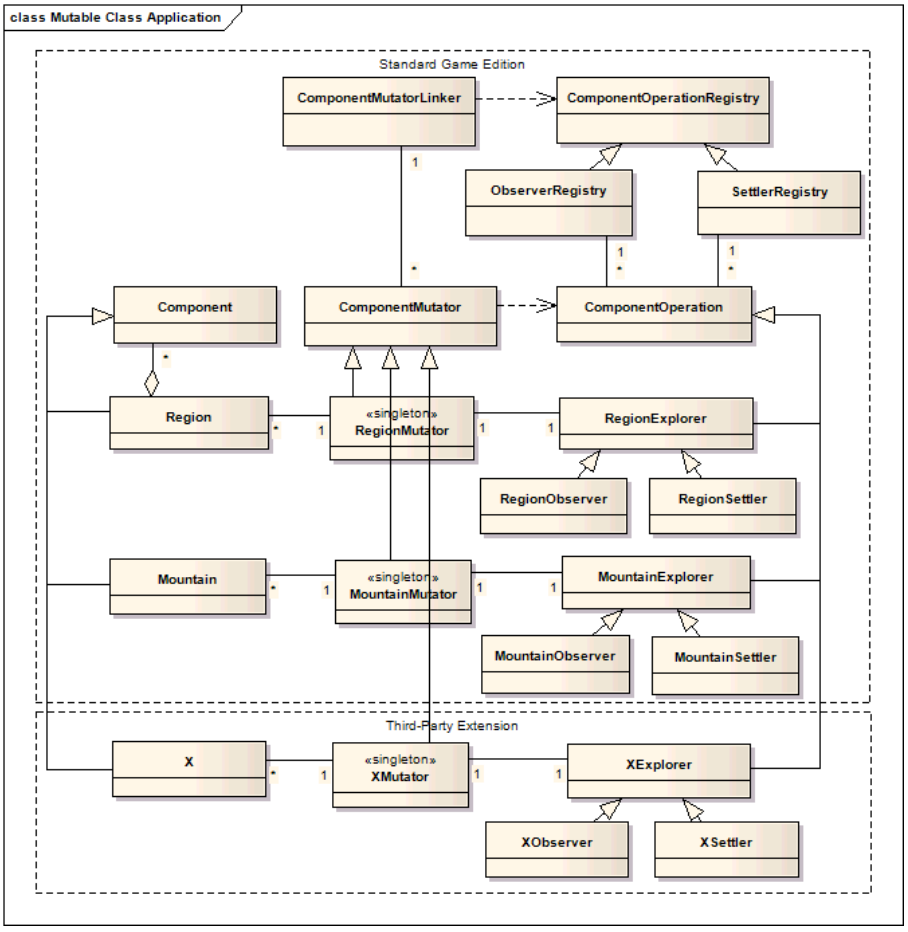

Figure 29: Mutable Class-based structure of the imaginary game application

Applicability

The Mutable Class pattern should be used when:

- a model structure represents a large-scale heterogeneous model that has to be processed with different operations depending on types of model components;
- processing operations can be developed and added after the definition of the model structure;
- a class hierarchy of the model structure is not fixed and can be extended with new component types;
- model-associated operations can be changed dynamically according to application-specific scenarios, for example, finite state machines or agent-oriented adaptable systems.

Structure and Participants

The Mutable Class pattern provides an architectural solution that addresses the multi-layer structure of composite applications. Figure 30 shows the corresponding class diagram. The diagram encompasses the following participants:

- **Component:** base class of the model components. It defines a common interface, including a method for processing components;
- **ComponentMutator:** common interface of the component type-specific mutators. It defines the *accept()* method for setting the component operation. In addition, it serves as Marker Interface (Grand, 1998) used by the registry of component type-specific mutators;
- **ComponentOperation:** common interface of the component type-specific operations. It serves as Marker Interface (Grand, 1998) used by the ComponentMutator interface and registries of component type-specific operations;

- **ComponentA and ComponentB:** concrete classes of model components providing access to heterogeneous component members and implementing the *process()* method by propagating its call to the corresponding component mutator;
- **ComponentA_Mutator and ComponentB_Mutator**: concrete classes of the component type-specific mutators. They maintain an operation shared by instances of concrete component classes, ComponentA and ComponentB. The operation can be defined using the *accept()* method of the Component interface. In addition, these classes introduce the component type-specific *process()* methods that serve as Proxies (Gamma, Helm, Johnson, and Vlissides, 1995) of their operations;
- **ComponentA_Operation and ComponentB_Operation**: interfaces of component type-specific operations defining the *process()* method that takes the instance of the concrete component class as an argument;
- **ComponentA_Operation1 and ComponentB_Operation1**: component type-specific operations of the Operation1 category;
- **ComponentA_Operation2 and ComponentB_Operation2**: component type-specific operations of the Operation2 category;
- **ComponentMutatorLinker:** linker of the component mutators. It maintains a registry of mutators and implements the *join()* method for binding mutators with operations using an operation registry as an argument;
- **ComponentOperationRegistry**: common interface of the operation registries. It serves as a Marker Interface (Grand, 1998) used by the linker of component mutators;
- **Operation1_Registry and Operation2_Registry**: registries of the component type-specific operations belonging to the Operation1 and Operation2 categories.

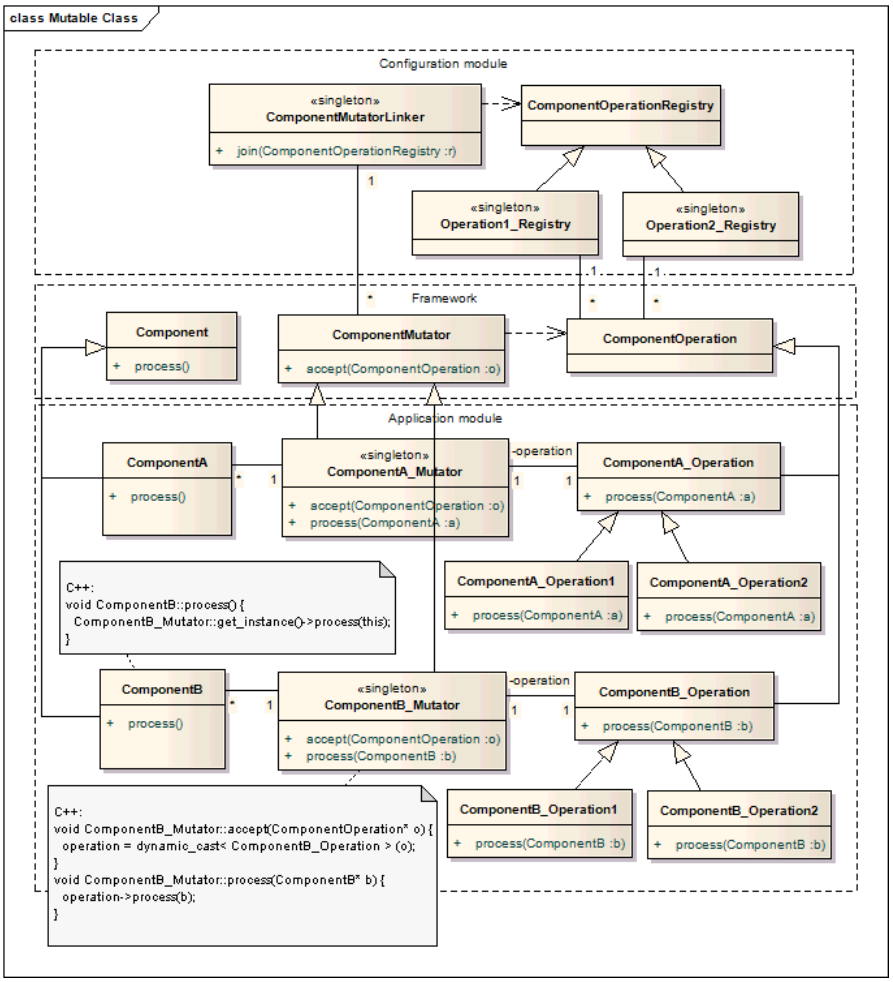

Figure 30: Class diagram of the Mutable Class pattern

Collaborations

In the Mutable Class pattern, the processing of the data model consists of three steps:

- registering component-specific mutators and operations;
- joining the mutator-operation associations;
- performing operations on elements of the heterogeneous hierarchical structure.

Following the design pattern description format, interactions among participants at each step can be explained with a corresponding sequence diagram. The first step is illustrated by Figure 31. As described in the Motivation subsection, mutable classes of the basic model and its extensions

can be developed and registered independently from each other. The registration process can be implemented with many approaches, for example, by using a dedicated configuration module of a high level application program or some static initializer of a component-specific library. Figure 31 shows the latter case. According to this diagram, the initializer program consequently creates a mutator singleton and operation instances and updates the corresponding registries.

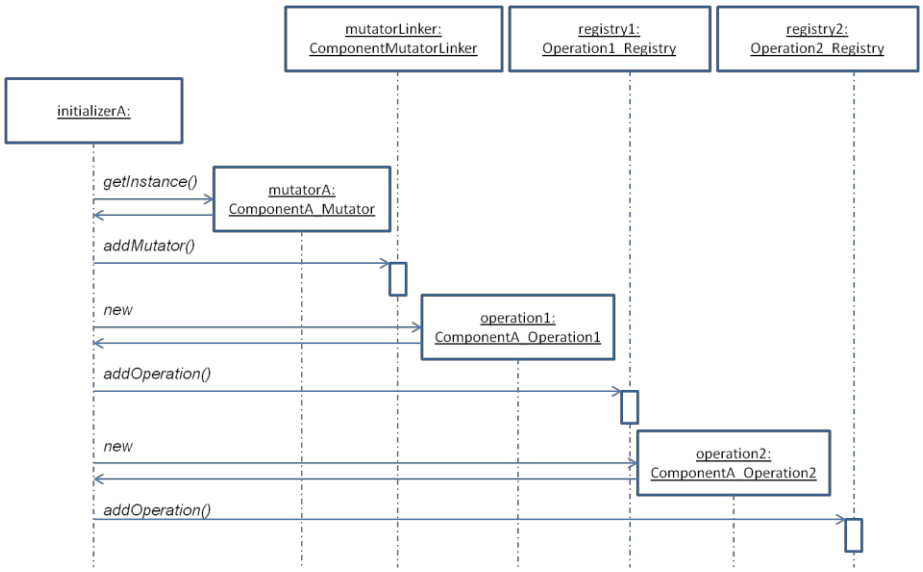

Figure 31: Registering a mutator and operations of the ComponentA class

The second step assigns or reassigns mutators with the new category of operations. As shown in Figure 32, this step is initiated by a client calling the *join()* method of the mutatorLinker object. This method takes an instance of the Operation1_Registry class as an argument and sequentially iterates through entries of pairs containing ids and component mutators. In this particular example, the first entry is associated with the ComponentA class. Thus, mutatorLinker takes the corresponding id, selects an operation, and binds it with the component mutator within the accept() method. As a result, the same operation object of the mutator singleton can be accessed by multiple components of the same type.

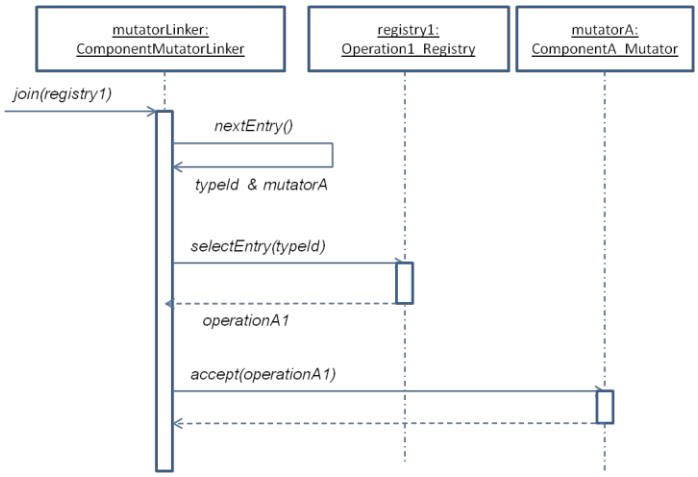

Figure 32: Interactions for binding data processing algorithms

The third step includes the processing of the ComponentA instance (see Figure 33). According to the Mutable Class pattern, the instance propagates the application request through the associated component mutator to the operation assigned in the previous step. The Mutable Class pattern does not address the instantiation of the ComponentA objects that can be created by conventional constructors as shown in Figure 33 or Factory patterns. This is consistent with the Mutable Class conceptual approach that is designed as an extension of the programming language type system. Similar to the Visitor pattern, it belongs to the category of behavioral patterns and aims to support the interchangeability of operations for already instantiated components.

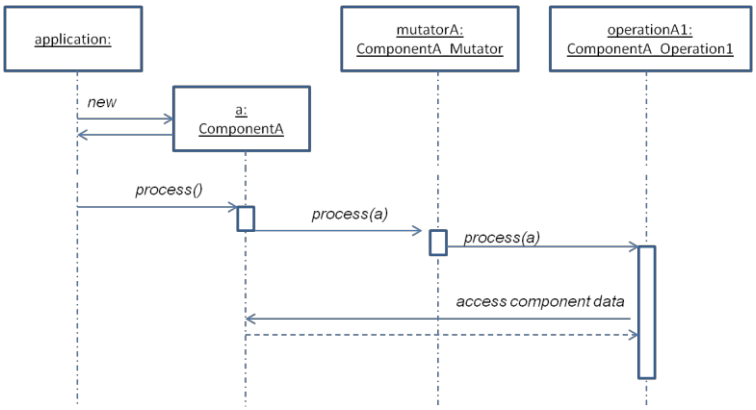

Figure 33: Interactions for data processing of the ComponentA instance

Consequences

The Mutable Class pattern consolidates the benefits provided by multiple design patterns and approaches:

- As the Strategy and Visitor patterns, the Mutable Class defines a consistent approach for managing families of related algorithms. These algorithms are separated from the data model and can be easily changed without affecting the model classes;
- The Mutable Class resolves the extensibility issue of the Visitor pattern by breaking its dependency cycle and adding the preliminary step for weaving algorithms with the processed structure. The weaving procedure is an essential part of the Aspect-Oriented Programming (AOP) paradigm, providing the mechanism for dealing with crosscutting concerns in the object-oriented languages;
- Compared to the AOP approach, the Mutable Class pattern introduces a light-weight object-oriented solution, avoiding the overhead associated with the new programming paradigm.

The Mutable Class pattern also presents a drawback of the associated design patterns:

- As the Strategy and Visitor patterns, the Mutable Class forces the data model classes to provide access to their internal states for data processing algorithms. This in turn may compromise their encapsulation.

Sample Code

As described in the Structure and Participants section, the Mutable Class pattern can be associated with the category of architectural patterns that specify a horizontal framework and multi-layer structure across different vertical application domains. On the framework layer, the Mutable Class is based on three primary concepts: Component, ComponentOperation, and

ComponentMutator. In C++, these concepts can be implemented with the corresponding abstract classes as shown in Listing 3.

```
class Component {
 public:
  virtual void process() = 0;
};

class ComponentOperation {
 public:
  virtual ~ComponentOperation() {}
};

typedef shared_ptr<ComponentOperation> ComponentOperationPtr;

class ComponentMutator {
 public:
  virtual void accept(ComponentOperationPtr op) = 0;
};
```

Listing 3: Framework layer of the Mutable Class pattern

Each abstract class defines a common interface implemented by hierarchy of the application domain classes. In this approach, the ComponentMutator-derived classes play a special role connecting model components with associated data processing operations. For many applications, this role is generic and does not depend on the application model. Then the structure of the ComponentMutator-derived classes can be implemented generically using one C++ class template GenericMutator<T> parameterized over hierarchy of the component-specific types (see Listing 4). Following the general definition of the Class concept, GenericMutator<T> is implemented after the Singleton pattern (Gamma, Helm, Johnson, and Vlissides, 1995) to maintain the T-specific operations shared by the extent of the T instances. According to this pattern, the singleton of GenericMutator<T> can be accessed with the static method *get_instance()*. In addition to the singleton interface, GenericMutator<T> implements the *accept()* method inherited from ComponentMutator and introduces the new method *process()*. Both methods are associated with the GenericOperation<T> interface defined in Listing 5. The *accept()* method assigns a pointer to the instance implementing this interface and the *process()*

method delegates a request to this instance. Downcasting in the *accept()* method allows to resolve the type safety issue of data processing interfaces.

```
template <class T>
class GenericMutator : public ComponentMutator {
 public:

  static GenericMutator<T>* get_instance();

  virtual void accept(ComponentOperationPtr op);

  void process(T* component);

 public:

  shared_ptr< GenericOperation<T> > operation;
};
```

```
template<class T>
GenericMutator<T>* GenericMutator<T>::get_instance(){
   static GenericMutator<T> singleton;
   return& singleton;
}

template<class T>
void GenericMutator<T>::process(T* component){
  operation->process(component);
}

template<class T>
void GenericMutator<T>::accept(ComponentOperationPtr op){
  operation =
    dynamic_pointer_cast< GenericOperation<T>,  ComponentOperation >(op);
}
```

Listing 4: GenericMutator class template

```
template <class T>
class GenericOperation : public ComponentOperation {
 public:

  virtual void process(T* component) = 0;
};
```

Listing 5: GenericOperation class template

The design with separated component-specific operations is closely related with the Normal Form Visitor approach (Xiao-Peng and Yuan-Wei, 2010). Formally, it addresses the interface segregation principle (Martin, 1998) aiming to enhance flexibility of the overall system. In the context of the Mutable Class applications, the approach facilitates the independent development of the third-party extensions, like the X component and associated operations, and mixed them together with other mutable classes using the registry-based configuration mechanism. According to the class diagram of Figure 30, the Mutable Class configuration framework consists of a mutator linker and registries of operations. Their implementation is shown in Listing 6.

```
class ComponentMutatorLinker  {
 public:

  static ComponentMutatorLinker* get_instance();

  void join(ComponentOperationRegistry* r);

  map<type_index, ComponentMutator*> mutators;
};

ComponentMutatorLinker* ComponentMutatorLinker::get_instance(){
  static ComponentMutatorLinker singleton;
  return &singleton;
}

void ComponentMutatorLinker::join(ComponentOperationRegistry* r){
  map<type_index, ComponentMutator*>::iterator it;
  for(it = mutators.begin(); it != mutators.end(); it++){
    it->second->accept(r->operations[it->first]);
  }
}
```

```
Class ComponentOperationRegistry {
  public:

    map<type_index, ComponentOperationPtr > operations;
};

class ObserverRegistry  :
    public ComponentOperationRegistry {
 public:

  static ObserverRegistry* get_instance();
};

ObserverRegistry* ObserverRegistry::get_instance(){
  static ObserverRegistry singleton;
  return &singleton;
}
```

Listing 6: Linker of component mutators and registry of observers

The registration procedure can be further automated with generic initializers, MutatorInitializer<T> and ObserverInitializer<T, TObserver> parameterized over hierarchy of the component-specific types and observers (see Listing 7). The initializers emulate the Java static initialization blocks allowing the independent registration of different mutable classes.

```
template <class T>
class MutatorInitializer {
 public:
   MutatorInitializer();
};

template<class T>
MutatorInitializer<T>::MutatorInitializer(){

  ComponentMutatorLinker* mutatorLinker =
     ComponentMutatorLiner::get_instance();

  mutatorLinker->mutators[typeid(T)] =
    GenericMutator<T>::get_instance();
}
```

```
template <class T, class TObserver>
class ObserverInitializer {
 public:
   ObserverInitializer();
};

template<class T, class TObserver>
ObserverInitializer<T, TObserver>:ObserverInitializer(){

  ObserverRegistry* observerRegistry =
     ObserverRegistry::get_instance();

  observerRegistry->operations[typeid(T)] =
    ComponentOperationPtr(new TObserver());
}
```

Listing 7: MutatorInitializer and ObserverInitializer

The MutatorInitializer<T>, ObserverInitializer<T, TObserver>, GenericMutator<T>, and GenericOperation<T> class templates establish a framework for registering and connecting the

component types and data processing algorithms. This framework is application-neutral and can be illustrated with the demonstration example from the Motivation section. Listing 8 and Listing 9 show the implementation of two example's components, Region and Mountain. Both components have a similar structure of interfaces consisting of common and specialized parts. The common interface includes the *process()* method inherited from the Component class. This method redirects a method call together with the component data to an actual operation implementation maintained by a singleton of the corresponding component mutator. The specialized part of the component interface provides access to the component-specific data, such as the Region collection of its components or the Mountain height. To simplify the demonstration example, setters and getters of the specialized interfaces have been replaced with the direct access to component members.

```
typedef shared_ptr<Component> ComponentPtr;

class Region: public Component {
 public:
  // Component API
  virtual void process();

  // Region API
   list<ComponentPtr> components;
};

typedef GenericMutator<Region> RegionMutator;
```

```
void Region::process() {
  RegionMutator::get_instance()->process(this);
}

MutatorInitializer<Region> regionMutatorInitializer;
```

Listing 8: Region and RegionMutator classes

```
class Mountain: public Component {
 public:
  // Component API
  virtual void process();

  // Mountain API
   double height;
};

typedef GenericMutator<Mountain> MountainMutator;
```

```
void Mountain::process() {
  MountainMutator::get_instance()->process(this);
}

MutatorInitializer<Mountain> mountainMutatorInitializer
```

Listing 9: Mountain and MountainMutator classes

Similar to the Visitor pattern, data processing operations of the Mutable Class are implemented independently from data structures. In contrast to the original version of the Visitor pattern, operations of different components, for example Region and Mountain, are fully decoupled as shown in Listing 10 and Listing 11.

```
class RegionObserver : public GenericOperation<Region> {
 public:

  virtual void process(Region* r);
};

void RegionObserver::process(Region* r){
  list<ComponentPtr>::iterator it;
  for(it = r->components.begin(); it != r->components.end(); it++){
    (*it)->process();
  }
}

ObserverInitializer<Region, RegionObserver> regionObserverInitializer;
```

Listing 10: RegionObserver class

```
class MountainObserver : public GenericOperation<Mountain> {
 public:

  virtual void process(Mountain* m);
};

void MountainObserver::process(Mountain* m){
  cout << typeid(Mountain).name() << ", height: " << m->height << endl;
}

ObserverInitializer<Mountain, MountainObserver> mountainObserverInitializer
```

Listing 11: MountainObserver class

Finally, the main program of Listing 12 demonstrates the major steps of the Mutable Class application. It starts with the construction of the application model. In this example, the model includes only one region and one mountain. In the next step, the linker of component mutators takes a collection of observers and connects each component type with the corresponding operation. The final step processes the model with connected observers.

```
main() {

 // Build the model

 Region model;

 model.components.push_back(ComponentPtr(new Mountain()));

 // Join component types with observers

  ComponentMutatorLinker* mutatorLinker = ComponentMutatorLinker::get_instance();
  ObserverRegistry* observerRegistry = ObserverRegistry::get_instance();

 mutatorLinker->join(observerRegistry);

 // Process observers on a model

 model.process();

 …

}
```

Listing 12: Main program

Implementation

The sample code presented in the previous section outlines a typical structure of the Mutable Class application highlighting several implementation topics:

- The structure of the Mutable Class application has several layers encompassing an application-neutral framework, application domain toolkit, third-party extensions, and high level configuration layer.
- Similar to the Visitor pattern, the Component objects serve as front ends for traversing a data model by providing the *process()* method. In Listing 3, this method does not have arguments suggesting that the intermediate results (e.g., the OpenGL rendering state) must be maintained outside of the data model. This case can be illustrated by adding the Observer class as shown in Listing 13.

```
class Observer {
 public:
  static double results;
};
```

Listing 13: Observer class

The static member of the Observer class represents results accumulated during the traversal of the model. To facilitate the access to the Observer state, it can be inherited by the component operations. Listing 14 shows the corresponding version of the MountainObserver class updating the Observer results according to some algorithm.

```
class MountainObserver : public GenericOperation<Mountain>, public Observer {
 public:

  virtual void process(Mountain* m);
};

void MountainObserver::process(Mountain* m){
  results += 1.2*m->height;
}
```

Listing 14: MountainObserver class with the Observer state

- As shown in Listing 8 and Listing 9, interfaces of different components and associated operations are fully decoupled. These examples illustrate a consistent procedure for developing data model components across multiple applications using toolkits and third-party extensions.
- The application model may use hierarchical relationships between component classes including the inheritance of members and methods. The Mutable Class pattern does not prevent such relationships. Moreover, the pattern supports the reuse of algorithms by ancestor components. This case is not unusual since some categories of algorithms may use only a common subset of component members defined in both the parent and descendant classes. In the Mutable Class approach, such the sparse associations can be handled by the Component Mutator Linker using, for example, the XML description.

Relation to the Visitor Pattern

The Mutable Class pattern has been developed as an alternative approach of the Visitor pattern providing a consistent mechanism for processing heterogeneous models with multiple

algorithms. The Visitor pattern addresses this task by combining component-specific methods into the algorithm-specific Visitor classes and implementing the run-time double-dispatch approach for binding these methods with corresponding components. The Visitor combined interface, however, freezes class hierarchies of application models and prevents the introduction of new component types. For example, adding the new component X would require to extend this interface with the new visit(x: X) method and to change all visitor classes. The Mutable Class pattern resolves this limitation of the Visitor pattern by replacing its monolithic interface with extendable registries of operations and introducing a run-time linking step connecting mutable classes with the selected registry of operations.

The development and integration of the new component types can be naturally illustrated with the sample code of the demonstration example. According to the Sample Code framework, the mutable class triplet of the new component X would include three classes: X, XMutator, and XObserver. As show in Listing 15 and Listing 16, their implementation follows the common procedure for developing the Region and Mountain classes. As a new component type, the specialized part of the X interface introduces a new member y that is accessed with the *process()* method of XObserver.

```
class X: public Component {
 public:
  // Component API
  virtual void process();

  // X API
   double y;
};

typedef GenericMutator<X> XMutator;
```

```
void X::process() {
   XMutator::get_instance()->process(this);
}

MutatorInitializer<X> xMutatorInitializer;
```

Listing 15: X and XMutator classes

```
class XObserver : public GenericOperation<X> {
 public:

  virtual void process(X* x);
};

void XObserver::process(X* x){
  cout << typeid(X).name() << ", y: " << x->y << endl;
}
ObserverInitializer<X, XObserver> xObserverInitializer
```

Listing 16: XObserver class

These classes can be packaged in a third-party library and linked with the X-aware application without affecting other libraries. The corresponding main program is shown in Listing 17. It differs from the original program of Listing 12 only in the construction of the application model including the X component. In the practical applications, the model is usually created with the Builder pattern (Gamma, Helm, Johnson, and Vlissides, 1995) based on another registry of component-specific instances with the Factory methods. This tiny example demonstrates the principal advantage over the Visitor pattern for processing extendable heterogeneous models.

```
main() {

  // Build the model

  Region model;

  model.components.push_back(ComponentPtr(new Mountain()));
  model.components.push_back(ComponentPtr(new X()));

  // Join component types with observers

   ComponentMutatorLinker* mutatorLinker = ComponentMutatorLinker::get_instance();
   ObserverRegistry* observerRegistry = ObserverRegistry::get_instance();

  mutatorLinker->join(observerRegistry);

  // Process observers on a model

  model.process();

 ....

}
```

Listing 17: Main program using the new component X

Quality Factor Assessment

Design patterns represent proven solutions distilled and elaborated from successful products. These solutions however rarely provide an unconditional cure, and have both positive and negative characteristics. Furthermore, the assessment of software design is a complex task dealing with many vague concerns. The ISO/IEC 25010 System and Software Quality Requirements and Evaluation standard (2011) aims to straighten the decision-making process by identifying a product quality model derived from the consolidation of several software metrics suites. The resulting quality model is based on eight quality characteristics: functional suitability, performance efficiency, compatibility, usability, reliability, security, maintainability, and portability. Since each of these characteristics covers too broad a topic, the ISO/IEC 25010 standard further divides them into the supporting sub-characteristics as shown in Appendix A.

The problem of heterogeneous model processing is primarily related to the maintainability sub-characteristics. The assessment of the aspect-oriented approach has added the performance efficiency topic dealing with the consideration of time behavior and resource utilization. Finally, to differentiate several existing solutions, such as the Reflection and Normal Form patterns, the analysis has included the portability characteristic. Table 1 summarizes and compares the corresponding quality attributes of existing approaches derived in this thesis from a thorough analysis of the literature on the Visitor pattern (Gamma, Helm, Johnson, and Vlissides, 1995; Martin, Riehle, and Buschmann, 1997; Vlissides, 1999; May and Champlain, 2001; Grothoff, 2003; Buttner et al., 2004; Forax, Duris and Roussel, 2005; Xiao-Peng and Yuan-Wei, 2010; Pati & Hill, 2013) and the AOP paradigm (Kiczales et al., 1997; Laddad, 2003; Wu et al., 2005; Walls, 2013).

Approaches	Performance efficiency	Maintainability			Portability
		Reusability	Modularity	Modifiability	
Visitor	high	n/a for multiple third-party extensions	low; strong coupling between model and visitor's interfaces	n/a for model extensions	language-independent
Extended Type Visitor	high	"	"	normal; based on inheritance	language-independent
Acyclic Visitor	high	limited; based on the C++ multiple inheritance	normal; based on inheritance	"	based on the C++ multiple inheritance
Generic Visitor	unclear; depends on an external reflection mechanism	unclear; depends on an external reflection mechanism	unclear; depends on an external reflection mechanism	unclear; depends on an external reflection mechanism	unclear; depends on an external reflection mechanism
Dynamic Dispatch Visitor	"	"	"	"	"
Reflective Visitor	depends on the approach: Walkabout: low Runabout: high Sprintabout: high	high	limited; coupling between application and configuration interfaces	high	based on the Java reflection API and class loading
Normal Form Visitor	high	high	"	high	language-independent
AspectJ	high	limited; only at compile time	high	high	based on the Java byte code
Spring AOP	low	high	high	high	language-independent
Mutable Class	high	high	high	high	language-independent

Table 1: Assessment of quality attributes for existing and proposed approaches

The evolution of the Visitor extensions follows two general rules suggested by the authors of the design pattern book (Gamma, Helm, Johnson, and Vlissides, 1995): association over aggregation and object composition over class inheritance. These good practice principles primarily address two software quality metrics, coupling and cohesion (Stevens, Myers, and Constantine, 1974), associated with the maintainability aspects. Following these rules, the Acyclic Visitor approach improved the original version of the Visitor pattern by breaking the monolithic interface aggregating loosely related methods into different components and

recombining them together using multiple inheritance. The Reflective and Normal Form Visitors replace inheritance with the composition approach. The Normal Form Visitor pattern still does not resolve coupling between application and configuration modules. This topic has been addressed by the pointcut-advice model and weaving procedure of the aspect-oriented programming (AOP) paradigm. The scope of this paradigm, however, introduces a significant challenge that leads to the limited bytecode-based solution of the AspectJ compiler or the resource utilization overhead of the Spring AOP container architecture.

As shown in Table 1, the Mutable Class represents an optimal approach combining the advantageous features of both the Visitor-based patterns and the AOP paradigm. This combination becomes especially important in the context of the new heterogeneous information network applications. Technically, the advantage of the Mutable Class pattern is achieved with the extra level of indirection that can be considered as another form of object composition (Gamma, Helm, Johnson, and Vlissides, 1995). The next chapter will validate and demonstrate the preliminary analysis of this chapter in the context of two application domains.

Summary

The chapter presents the conceptual approach and formal description of the Mutable Class pattern addressing the first research question of the dissertation. The approach is designed after the UML metamodel as an extension of the Class concept to support the interchangeability of operations. Technically, it extends the application class with a singleton that maintains the reference to the interchangeable operation designed after the Strategy pattern. Adherence to the UML metamodel level facilitates the generalization of the Mutable Class approach as the corresponding design pattern for actual applications. Particularly, the chapter deliberately

describes this solution in the context of processing the large-scale heterogeneous tree-based models. The description follows the formal design pattern format and covers multiple topics, such as intent, motivation, applicability, structure, collaboration among participants, and implementation aspects. Finally, the chapter assesses quality characteristics of the Mutable Class pattern according to the ISO/IEC 25010 standard and compares the proposed approach with existing solutions described in Review of Literature.

Chapter 4

Results

The chapter presents the application of the Mutable Class pattern to two application domains, computational accelerator physics and compiler construction.

Computational Accelerator Physics

A brief overview of computational accelerator physics was already presented in Chapter 2. Specifically, it introduced a three-dimensional view of accelerator physics algorithms. To facilitate their applications, in 1995, the Unified Accelerator Libraries (UAL) project (Malitsky and Talman, 1996) suggested an open architecture in which diverse computational algorithms were connected together via common accelerator objects such as Element, Bunch, Twiss, etc. The architecture immediately led to the consideration of new types of simulation studies involving combinations of conventional approaches and various extensions. The implementation of composite scenarios, however, required a consistent and efficient mechanism for managing many-to-many associations among simulation algorithms and heterogeneous elements of accelerator models. The dedicated analysis of existing design patterns did not identify an optimal solution that would address all requirements of the UAL simulation environment. Therefore, the new approach was derived after merging ideas of two design patterns, Strategy (Gamma, Helm, Johnson, and Vlissides, 1995) and Type Object (Martin, Riehle, and Buschmann, 1997). The Strategy pattern encapsulated the implementation of the behavior into separate classes and provided the mechanism for their interchange. Its structure was already discussed in Chapter 3 and shown in Figure 25. The Type Object encapsulated the common class data in a singleton of

the additional class, the so called Type Class or Class Type. Eventually, the structure of the new approach was refined after the Class model of the Unified Modeling Language specification and transformed into the Mutable Class pattern (see Figure 30). The corresponding instantiation of this pattern in the context of the accelerator physics domain is shown in Figure 34. It captures the element and approach dimensions of a three-dimensional view of accelerator algorithms (see Figure 17). The extension of the pattern with the Observable dimension will be considered later in this section.

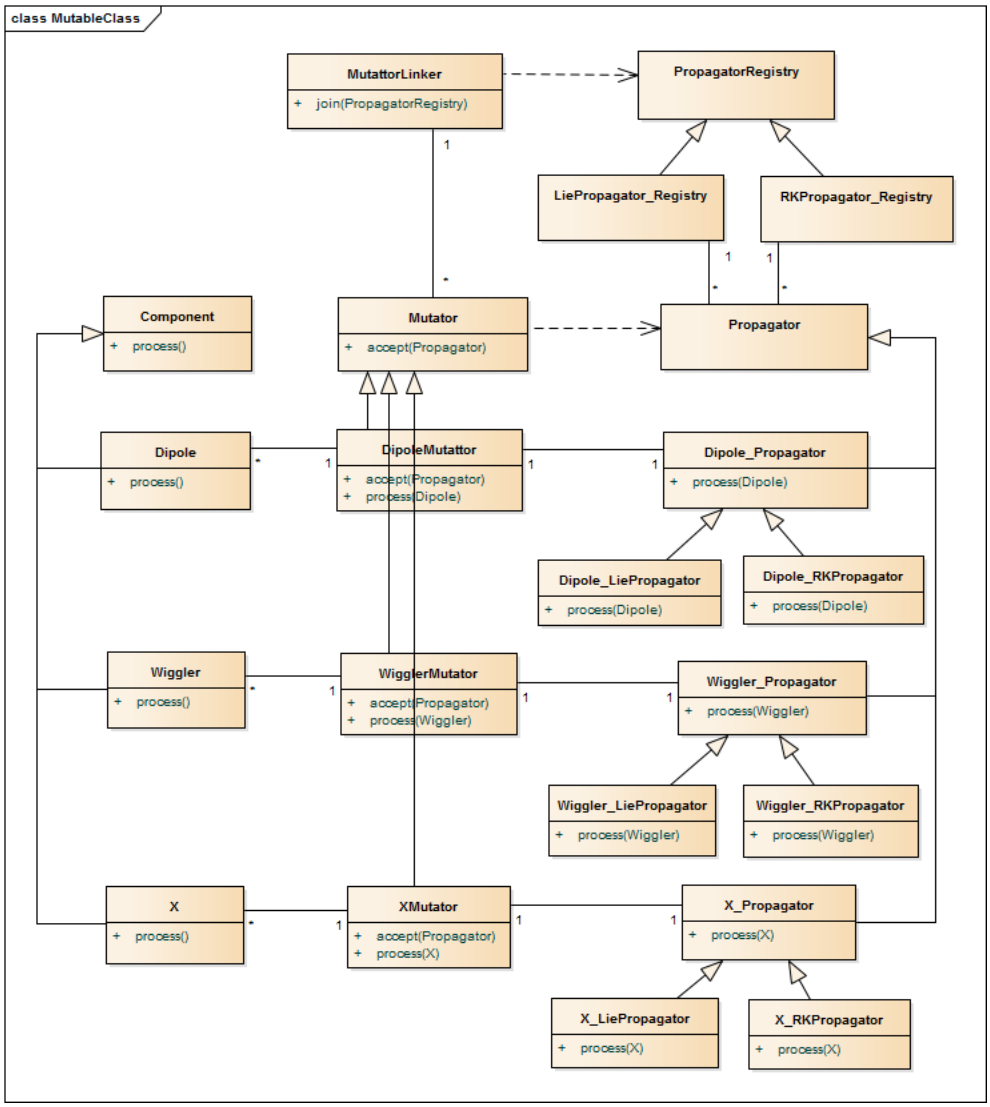

Figure 34: Mutable Class pattern in the context of accelerator algorithms (Figure 17)

The Mutable Class framework (Malitsky and Talman, 1998) boosted the development of the UAL applications. At the same time, new applications incrementally extended the scope of the new pattern by challenging it from different angles. The following subsections provide an overview of this consistent development starting with the analysis of new physical devices and concluding with large-scale model-based control systems.

Analysis of New Physical Devices

The construction of modern accelerator complexes is an expensive enterprise designed for new scientific mission studies aiming to assess theoretical hypotheses or to extend the horizons of existing experimental data. Scientific challenges lead to the design and consideration of new types of physical devices or more accurate treatment of high-order beam effects. In the context of existing accelerator programs, the implementation of new elements or effects introduced several issues associated with the changes of internal data structures for accommodating new sets of element and algorithm parameters. Moreover, in most cases, these sets were not well defined and changed according to different engineering designs and computational approaches. UAL addressed these requirements by proposing a generic solution based on the combination of the C++ propagation framework implemented after the Mutable Class approach (Malitsky and Talman, 1998) and Perl-based interface (Malitsky and Talman, 1996) supporting interactive insertions of project-specific extensions. Figure 35 shows the overall diagram of the UAL-based application. The main part of the UAL toolkit consisted of the accelerator model designed after the Standard Machine Format (Malitsky et al., 1995), the TEAPOT tracking algorithms (Schachinger and Talman, 1987) refactored after the Mutable Class pattern, and the UI::Shell Perl class providing a user-oriented interface to the C++ classes of the UAL components. Adding a new device required two extensions: implementation of the C++ library with the corresponding

mutable class and a new Perl class, Project::UI::Shell, with a few project-specific commands for accessing new attributes and inserting this device into the UAL environment. The approach was successfully applied to three different projects: Relativistic Heavy Ion Collider (RHIC) at Brookhaven National Laboratory (BNL), Cornel Electron-positron Storage Ring (CESR), and Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN). The following paragraphs provide a brief overview of corresponding applications.

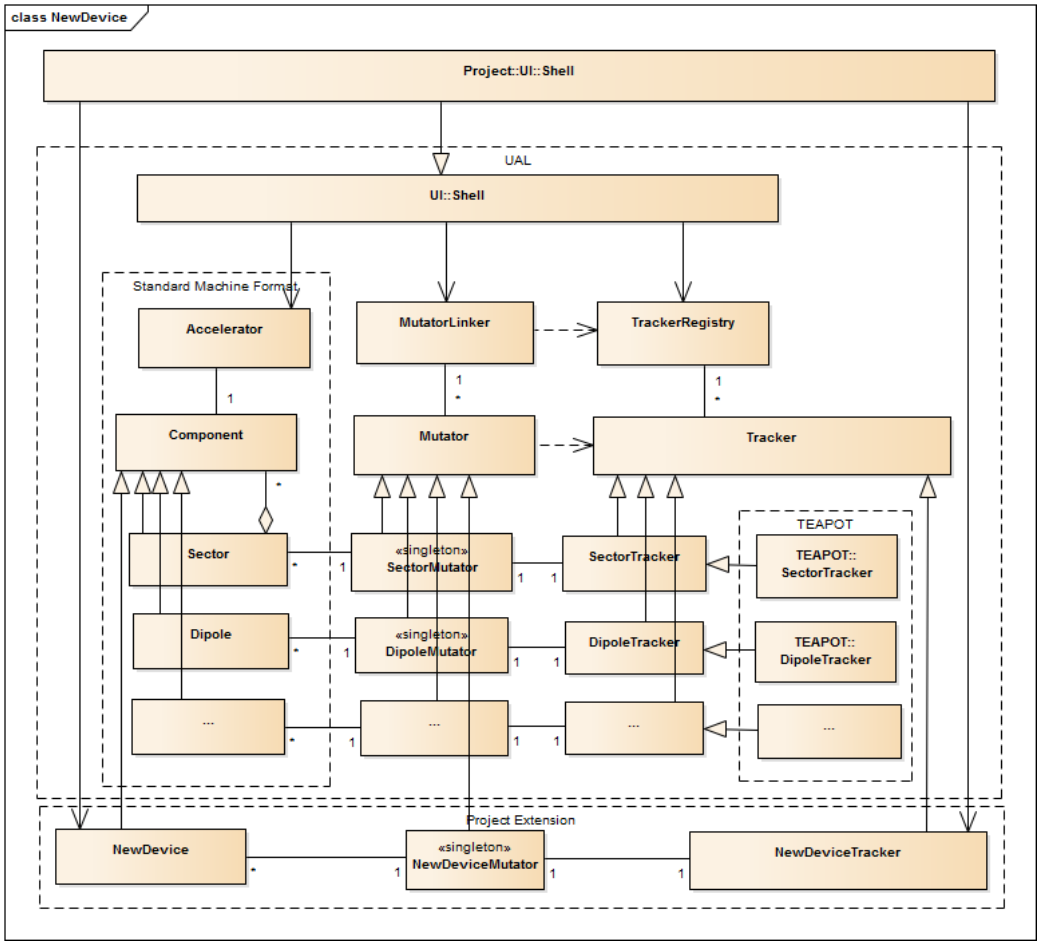

Figure 35: UAL-based approach for adding new devices

The RHIC accelerator complex is a chain of several accelerators, such as Electron Beam Ion Source, Linac, Booster, Alternating Gradient Synchrotron, and two rings of Relativistic Heavy Ion Collider built for the exploration of quark-gluon plasma and spin physics of protons. The

acceleration of polarized protons is achieved with special helical dipoles, called Siberian snakes. At the design phase, these devices were not supported by conventional accelerator codes and required dedicated research and development effort. Since the description of helical dipoles mismatched with other accelerator elements, they were implemented by the new Mutable Class based on the mapping approach using Taylor series. Figure 36 shows the corresponding application that can be elaborated by the Perl script.

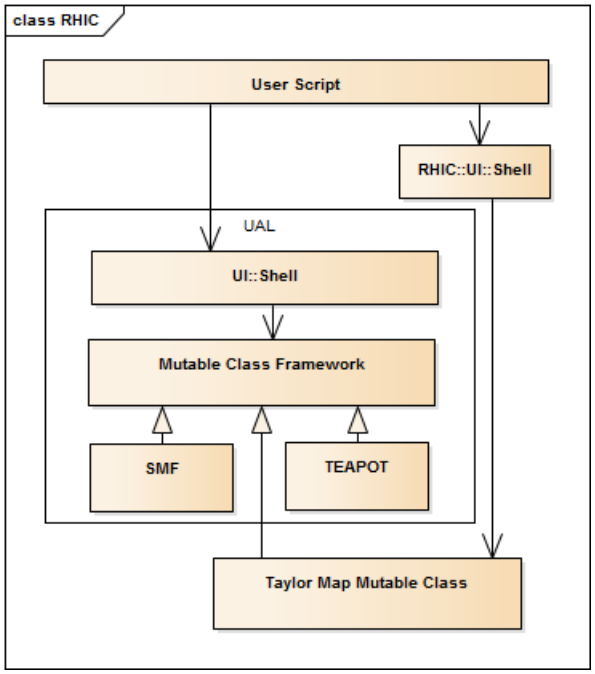

Figure 36: RHIC application

For the typical simulation scenario, the RHIC script began with the instantiation and initialization of a singleton of the Standard Machine Format (SMF) class representing an accelerator model in the UAL environment. The SMF model consisted of four layers comprising definitions of various parameters, elements, hierarchical view of accelerator, and flat sequence of elements with assigned individual magnetic fields and misalignment errors. In early version of the RHIC application, layers were initialized with a set of corresponding programs reading data

from distributed sources. As a result, building of the RHIC operational version was a laborious and error-prone procedure dealing with tens of thousands of elements and their individual characteristics. At this point, however, the accelerator model included only conventional elements and was thoroughly benchmarked with other accelerator programs. For the UAL applications, this was considered as a starting point for new studies. With the Mutable Class approach, the insertion of new elements required only a single method of the RHIC::UI::Shell interface that replaced conventional elements with Taylor maps of helical dipoles. Under the hood, the method reassigned element nodes of the accelerator models and associated with them the Mutable class of map-based algorithms. The approach was effectively used in the optimization of the RHIC design and operation studies (Pilat, F. et al, 1997 and 1999).

A similar approach was applied in beam dynamic studies at the Cornell Electron Storage Ring (CESR). Most of CESR elements were described by conventional attributes. But there were two element types, wiggler and element with the superimposed quadrupole and solenoid fields that required new extensions (see Figure 37).

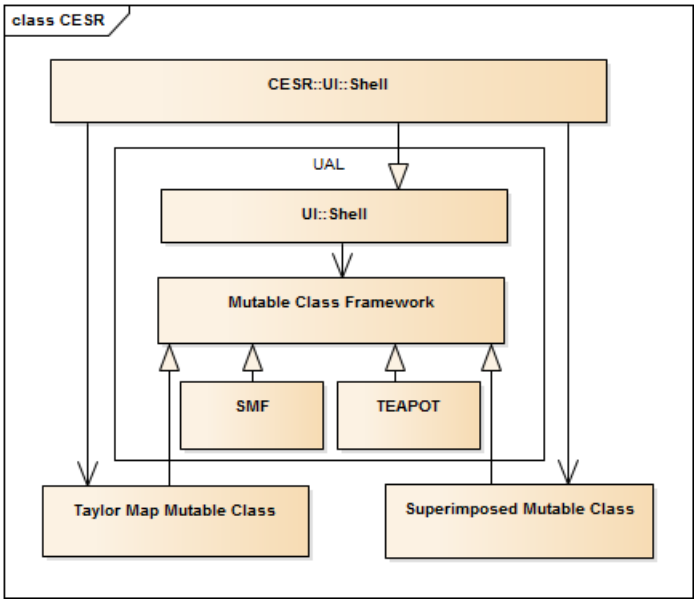

Figure 37: CESR application

As in the RHIC case with a helical dipole, wiggler's field satisfied Maxwell equations and was propagated with the Differential Algebra approach for producing corresponding Taylor maps. The CESR superimposed element represented a completely new type with unique project-specific parameters. Within the Mutable Class framework, both elements, however, were treated uniformly using the CESR-specific pairs of element-algorithm associations. As a result, the major effort was associated with the construction of the SMF object from distributed data sources used by local conventional programs. All components of the new simulation environment were glued together within the CESR::UI package that provided a uniform CESR-specific user interface to SMF data and UAL tracking, analysis, and fitting libraries. The CESR application (Malitsky and Pelaia, 1998) confirmed and generalized the RHIC-based approach and helped to further consolidate and refine the Mutable Class framework. Moreover, the approach was rapidly reused for the development and integration of new modules for simulating beam-beam effects in the context of the CESR upgrade (Koyama, Malitsky, and Talman, 1998).

The success of the RHIC and CESR applications lead to the extension of the US-LHC collaboration with a new direction focusing on the development of the modeling ecosystem based on the MAD and UAL software. MAD is an abbreviation of the Methodical Accelerator Design code developed by the CERN team (Carey and Iselin, 1984) and successfully applied for the design of most accelerator projects. UAL complements the MAD design and optimization capabilities with the Mutable Class framework for supporting new extensions. To exchange accelerator descriptions between two programs, the US-LHC collaboration introduced the Standard eXchange Format (SXF) that consolidated the joined software development and simulation studies (Pilat et al., 1998; Malitsky and Talman, 1998; Fisher, Pilat and Ptitsin, 1999). Soon, the format was accepted by many other teams and replaced project-specific formats in

various projects. Later, the SXF project was transformed into the MADX-UAL suite (Malitsky, et al., 2004) bringing together the MAD design algorithms and the UAL simulation libraries and project-specific extensions (see Figure 38). As a result, the SXF format significantly facilitated the application of the Mutable Class extensions to other facilities. Eventually, changes in computer technologies and new accelerator applications gradually accumulated a set of new requirements which resulted in the subsequent XML-based versions (Malitsky and Talman, 1998; Malitsky and Talman, 2006).

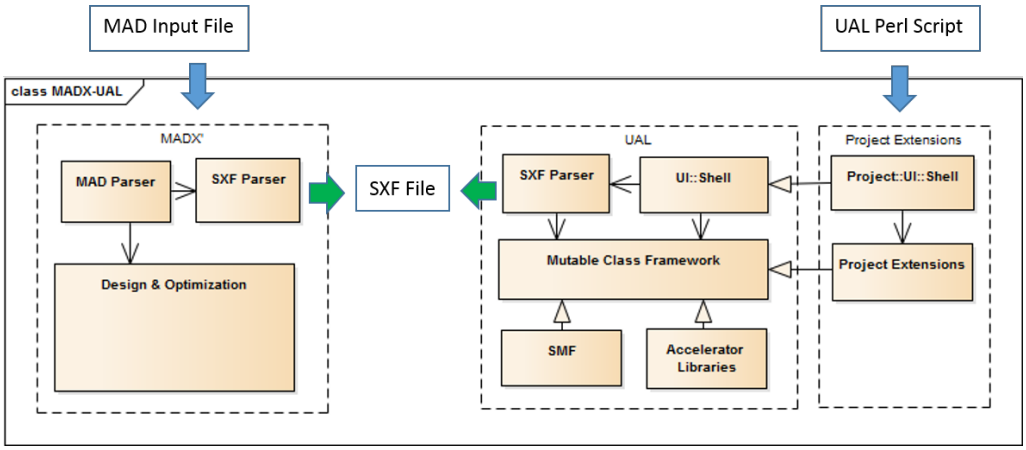

Figure 38: MADX-UAL suite

Integration of Multiple Effects

Designs and parameters of high-intensity machines, such as the Spallation Neutron Source (SNS) accumulator ring, imposed new expectations on the beam dynamics studies. One of the major scientific and technical challenges was the extremely strict requirement on uncontrolled beam loss at 10^{-4} level. In order to describe and analyze such low-level losses, one should closely reproduce all actual effects of a realistic machine. Some of them, such as field errors and misalignments were supported in general-purpose accelerator codes. Other effects, such as space charge and collimator surface grazing, were actual only for high intensity hadron rings and

distributed into a set of independent specialized programs. The mismatch among diverse data formats, units, and notations complicated the usage of these programs and increased the risk of errors and misinterpretations. Besides, the accurate simulation of the very low beam loss required the simultaneous consideration of several different effects in a single scenario. The Mutable Class framework and the Perl-based dynamic interface of the UAL open environment addressed all these tasks. As a result, for the SNS project, the UAL was extended with three accelerator libraries (see Figure 39): ACCSIM (Jones, 1997), ORBIT (Galambos et al., 1999) and AIM (Cameron, Fedotov, and Malitsky, 2002). The following paragraphs provide a brief overview of extended features: injection painting, collimator, space charge, and diagnostics.

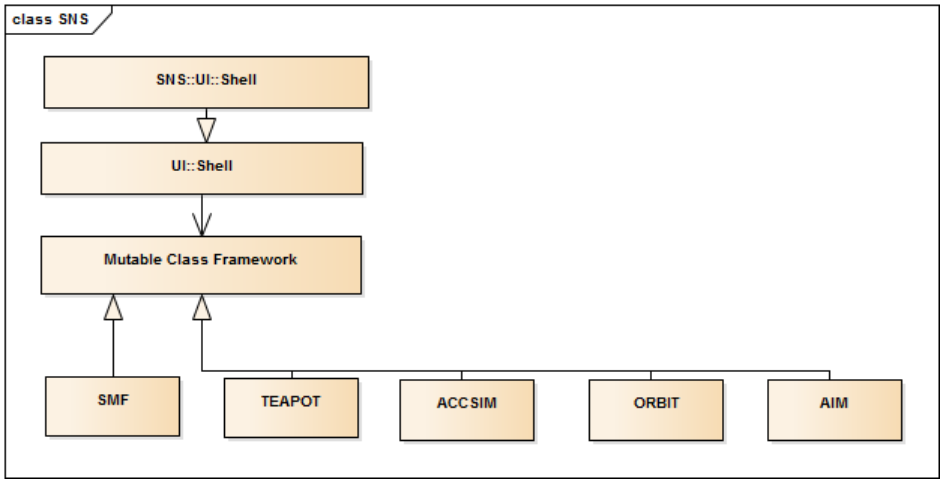

Figure 39: SNS application

Injection painting is a multi-turn injection procedure for filling a large phase space volume of beam distribution in order to reduce the space charge effect and to minimize the number of traversals through the injection foil. The ACCSIM code offered the most comprehensive approach for optimization and simulation of these dynamical processes. The control of the different scenarios however was hidden behind of the ACCSIM input language impeding the inclusion of new physical effects (field errors, misalignments, *etc*.). In UAL, all these dynamical

processes were implemented with the Perl interface that provided a direct access to the UAL packages via the configuration mechanism of the Mutable Class framework.

To protect the SNS ring from spreading up beam halo of the accumulated beam, it was equipped with the composite collimation system including four adjustable tantalum thin scrapers and three shielded long secondary collimators. The design of this system depended on many factors, such as an injection painting scheme, lattice parameters, and others. Then the simulation model had to be adaptable to an arbitrary combination of lattice and collimator variants. In general, it could be achieved by implementing the collimator system as an insertion device and splitting the one-turn tracking procedure into three steps: propagating particles (with a conventional programs) from the injection point to the collimator system, applying the collimator algorithms, and completing the turn by following particles back to the injection point. In the UAL environment, this scenario was implemented with a new Perl module complementing the injection painting procedure. Moreover, the UAL framework supported multiple representations of the collimator module. For example, this module could be implemented as a local adapter to the High Energy Physics shared libraries, such as GEANT 4. The integration of the accelerator and high energy physics software however introduced the significant overhead and was implemented later in the context of other projects (Fine, Malitsky, and Talman, 2006). For the SNS project, the ACCSIM approach was accepted as an optimal solution providing a necessary set of algorithms for particle-target interactions such as Landau and Bethe-Bloch energy loss distributions, Moliere multiple scattering, and nuclear interactions.

The major impact on halo growth and uncontrolled beam loss in the SNS ring was determined the space charge effect. Its implementation represented a difficult task involving the trade-off between the performance and accuracy of available algorithms. The Mutable Class framework of

the UAL environment addressed this issue by providing a uniform mechanism for selection and comparison of alternative approaches. The analysis of different algorithms suggested the "two-and-a-half" approximation of the ORBIT program representing the three dimensional space charge effect with a distributed collection of two-dimensional transverse kicks and one master node that updated a longitudinal beam distribution after each turn. To facilitate the implementation and employment of new modules, the SNS team developed a benchmark infrastructure shown in Figure 40 and Table 2. It inherited and generalized the previous methodology for the incremental development and analysis of the Mutable Class extensions and their subsequent integration into composite scenarios.

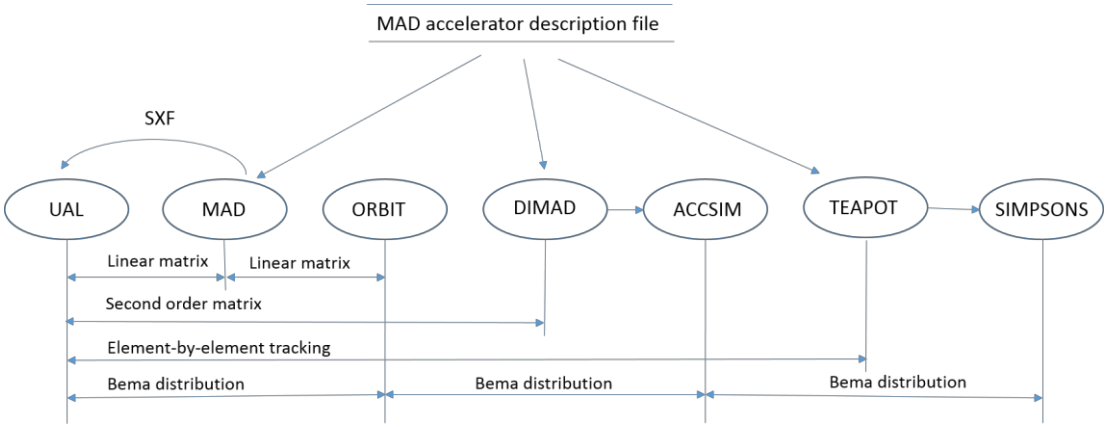

Figure 40: SNS benchmark infrastructure

Table 2: Accelerator programs used in the SNS project

Features	UAL	MAD	ORBIT	DIMAD	ACCSIM	TEAPOT	SIMPSONS
Interface	Perl API	MAD language	Super Code	dialect of MAD	DIMAD output	dialect of MAD	TEAPOT output
MAD standard elements	yes	yes	yes	yes	yes	yes	yes
design & optimization		yes		yes			
element errors & correction	yes	yes		yes		yes	
tracking	yes	yes	yes	yes	yes	yes	yes
mapping	any order	third order	linear order	second order	linear order	second order	
injection painting	yes		yes		yes		
collimation	yes				yes		
space charge	yes		yes		yes	yes	yes
instrumentation models	yes						

According to the UAL approach, the SNS simulation environment was organized as an additional package integrating together the UAL libraries SNS-specific extensions. Below, there is a list of some beam dynamics topics considered in the context of the SNS package:

- single-particle tasks, including such effects as kinematic non-linearity, non-linear tune-spread, dynamic aperture, and resonance driven diffusion maps (Fedotov et al., 2000; Papaphilippou, 2001)
- effect of space charge during transverse painting (Fedotov, Wei, and Gluckstern, 2001)
- optimization of painting bump functions
- combined tune spread due to the space charge, chromaticity and other nonlinearities (Fedotov et al, 2001)
- imperfection resonance crossing in the presence of space charge with corresponding choice of working points and intensity limitation (Malitsky et al, 2002)
- effect of ½ coherent resonance crossing in the presence of high-order resonances
- coherent resonance crossing of coupling resonances
- collective instability due to the transverse impedance (Fedotov et al., 2002)

An ability to study a complex combination of several effects provided scientists with the realistic model for beam losses and intensity limitation. For example, Figure 41 shows blow-up of beam profile due to skew-quadrupole sum resonance. In the absence of the space charge the strength of introduced skew-quadrupole component (tilt of 0.2 mrad) was not sufficient for particles to be trapped into the resonance. However, the space charge depressed the tunes, and some particles were trapped even for a relatively small skew-quadrupole components. Note that observed resonance was not the space-charge induced resonance since, in this case, it was driven by the skew-quadrupole field, and space charge played only a secondary role. Such resonance could be

corrected using the decoupling schemes. However, the vicinity of this dominant sum resonance made many working points less attractive. Additional problems with these working points surfaced when one included the effect of the quadrupole fringe fields. As a result, researchers observed a significant beam loss due to the combined effect of the space charge and fringe fields.

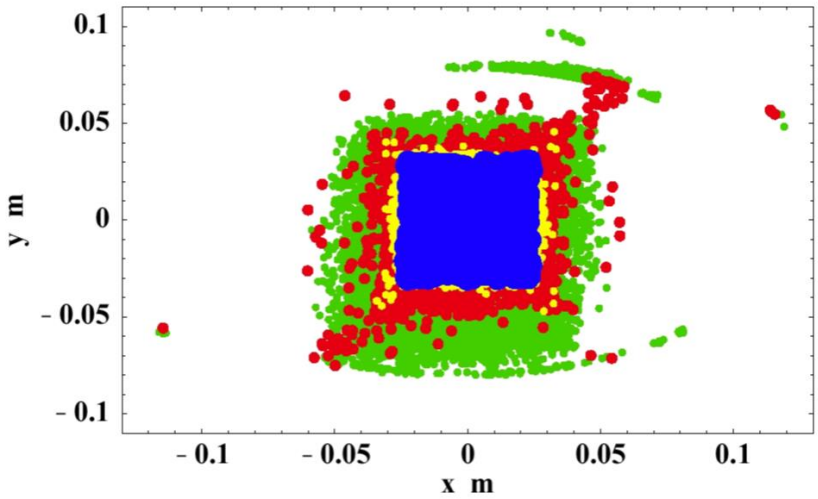

Figure 41: Blow-up of beam profile due to skew-quadrupole sum resonance in the presence of space charge: blue color (in the middle) - no space charge, no errors; yellow color - space charge, no errors; red color - space charge, expected errors and quadrupole tilt (0.2 mrad); green color – space charge, expected errors and quadrupole tilt (1 mrad).

Finding the best choice of working point became very challenging for the SNS due to its special characteristics of a very large tune spread mainly associated with the space charge, chromaticity and magnet fringe fields. Figures 42 and 43 show the tune spreads and corresponding resonance driven loss curves for two working points (6.23, 6.20) and (6.4, 6.3).

The imperfection errors were excited at a level slightly higher than expected to get a conservative estimate. The full 1060-turn injection was then performed for each of beam intensities with beam losses at the end of accumulation recorded for a specific acceptance. The working point (6.23, 6.20) was essentially free from resonance losses apart from some low loss due to the resonances above the working point and chromatic tune spread. For high beam intensities the tune was effectively depressed by space charge. The intensity limitation for this

working point was associated with the coherent beam response near the tune of 6.0. However, this limitation was due to the structure resonances and thus was very strict. For the working point (6.4, 6.3), the loss curve demonstrated impact of each individual resonance crossed during accumulation. The strong loss at low intensity was due to the sum sextupole resonance. Other loss peaks were due to the 3rd and 4th order resonances, which were crossed for higher beam intensity.

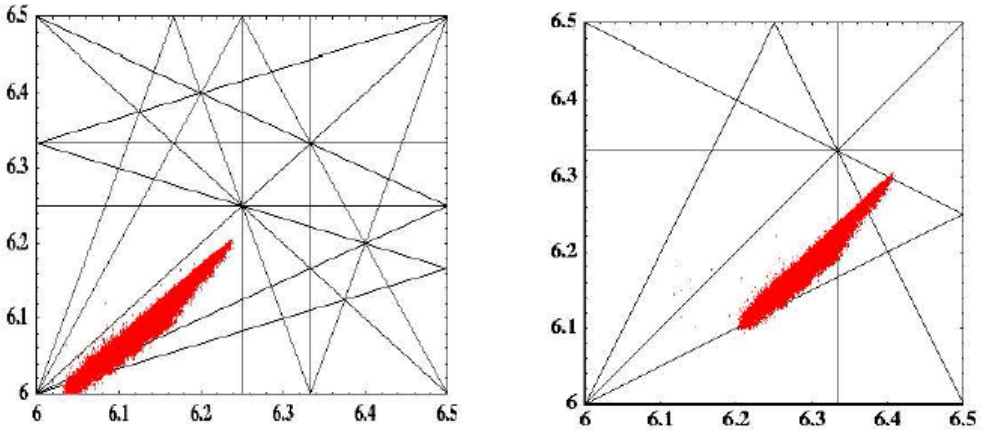

Figure 42: Tune spreads for working points (6.23, 6.20) and (6.4, 6.3), respectively.

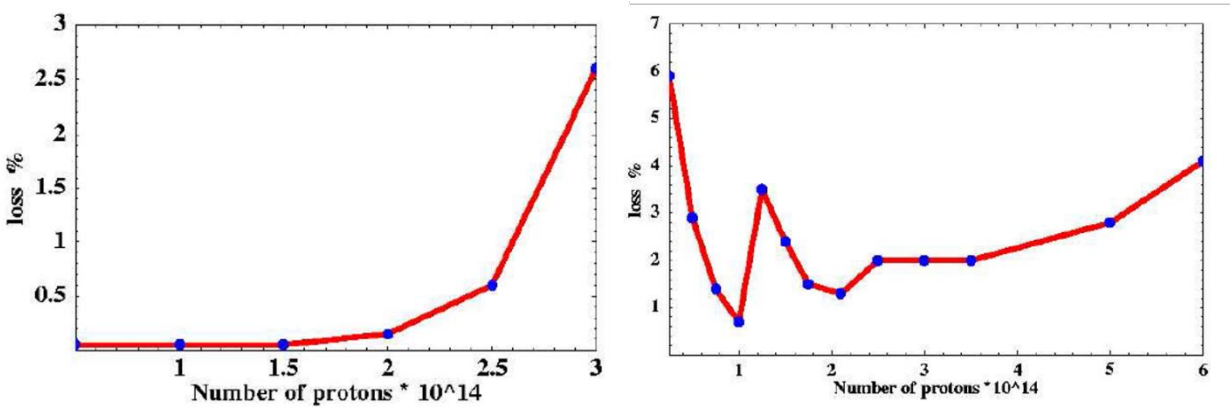

Figure 43: Loss curves for working points (6.23, 6.20) and (6.4, 6.3), respectively

Minimization of beam loss in the SNS ring was highly dependent on proper control of the tune footprint. In addition to the challenge of accurate measurement in the presence of large tune spread, large dynamic range was required to permit measurement through the accumulation

cycle. There were many possibilities for measuring coherent and incoherent tune and tune shift in the SNS Ring (Cameron, Fedotov, and Malitsky, 2002):

- coherent (dipole) tune/tune shift from impulse excitation
- incoherent tune from injection oscillations
- incoherent tune from Schottky
- incoherent tune from quadrupole mode oscillation
- incoherent tune from resonance crossing
- incoherent tune from Beam Transfer Function (BTF)

To facilitate the design of measurement systems, UAL was extended with the Accelerator Instrumentation Module (AIM) providing a set of diagnostics devices implemented after the Mutable Class framework.

Extending the Element-Algorithm association with the Probe dimension

Different use cases of the UAL open architecture were eventually generalized into the Element-Algorithm-Probe analysis pattern (Malitsky and Talman, 1998) introducing the three-dimensional view of accelerator algorithms. The pattern was inspired by the famous discussion around the quantum measurement problem involving interactions of macroscopic objects with microscopic world of particles. Following this measurement scenario, the pattern described simulation applications as interactions of probes with elements. From this perspective, probes represented any observable objects for which continuous evolution was meaningful and the evolution was caused by elements making up an application model. For example, in accelerator applications, probes can be 6D phase space coordinates of particles, lattice functions such as Twiss functions and dispersion functions, transfer matrices and nonlinear truncated power series, survey coordinates, wake fields, and others. The Probe objects were easily accommodated within

the Mutable Class pattern by adding the additional argument into the processing methods as shown in Figure 44.

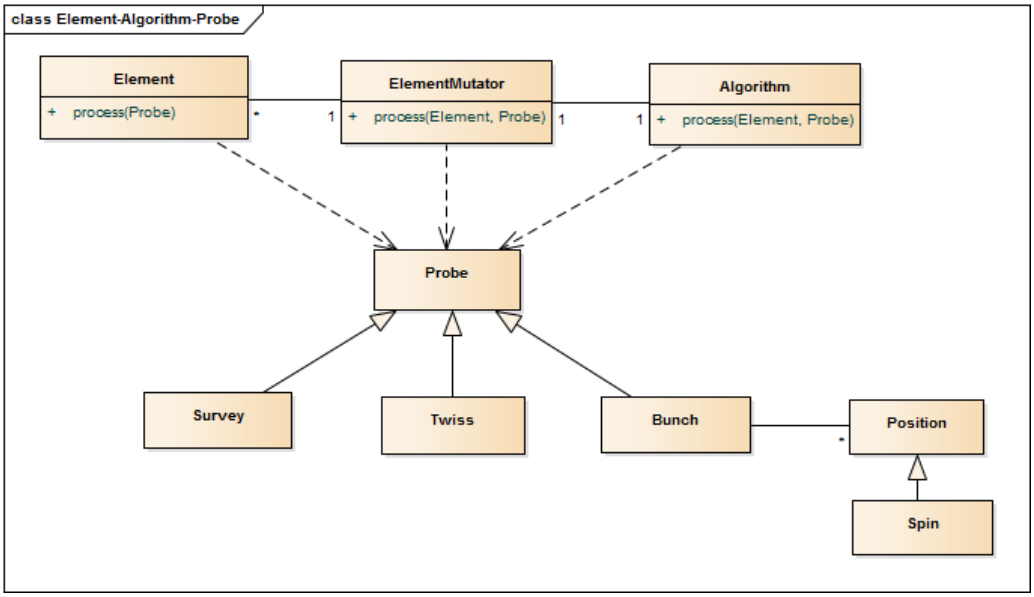

Figure 44: Implementation of the Element-Algorithm-Probe analysis pattern

Each Probe type started a new category of algorithms. In most cases, these types were independent. Therefore, corresponding algorithms were implemented by different accelerator libraries following the original version of the Mutable Class pattern. The Spin type however represented a composite case involving the simultaneous consideration of another type, Position of particles. Since the particle motion was already implemented in the TEAPOT library, the spin propagators were developed after the Decorator pattern (Gamma, Helm, Johnson, and Vlissides, 1995) augmenting the TEAPOT tracking algorithms with the SPINK approach as shown in Figure 45.

The SPINK program (Luccio, 1995) was originally written for the RHIC project at Brookhaven National Laboratory and employed for years to study the behavior of polarized protons in all stages of the accelerator complex. SPINK used a composite approach including the

second order maps of the orbital module and additional spin matrices rotating a spin in each accelerator element. This approach had the advantage of very high computational speed. However, the second order truncation of the orbit module introduced serious constraints for accurate long-turn simulation studies of the new High Energy Physics experiment aiming to measure an electric dipole moment (EDM) at unprecedented level of 10^{-29} e.cm. Therefore, the integration of the SPINK approach and TEAPOT symplectic tracking engine represented a natural and perfect solution and was implemented by the EDM team (Lin et al., 2009).

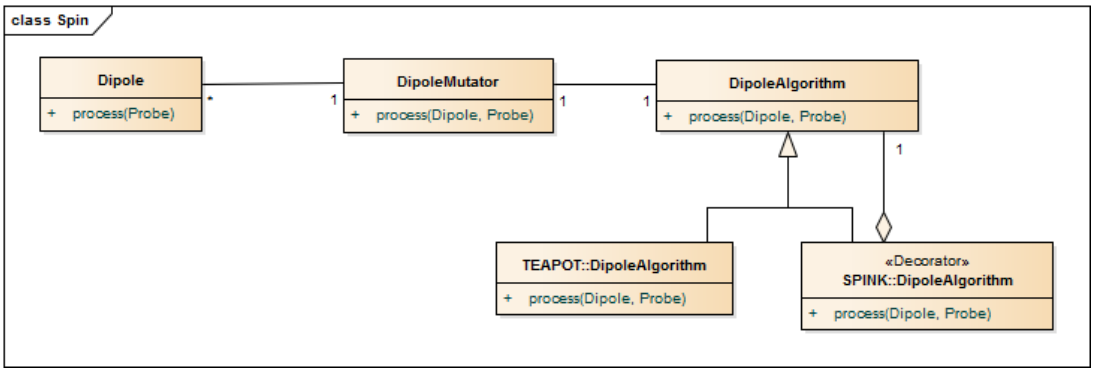

Figure 45: Integration of the TEAPOT and SPINK algorithms based on the combination of the Mutable Class and Decorator patterns

Accelerator Propagator Description Format

The variety and evolution of accelerator approaches suggested that an optimal program interface should be built as the combination of compact dynamic scripts and large well-structured input files containing the description of accelerator elements and computational algorithms. Initially, the UAL environment used only accelerator description files and the configuration of corresponding propagation algorithms was directly specified in user scripts. Eventually, the accumulated experience with multiple applications was transformed into the definition of a new specification, Accelerator Propagator Description Format (APDF),

complementing accelerator files (such as SXF or ADXF) with the description of accelerator-algorithm associations (see Figure 46).

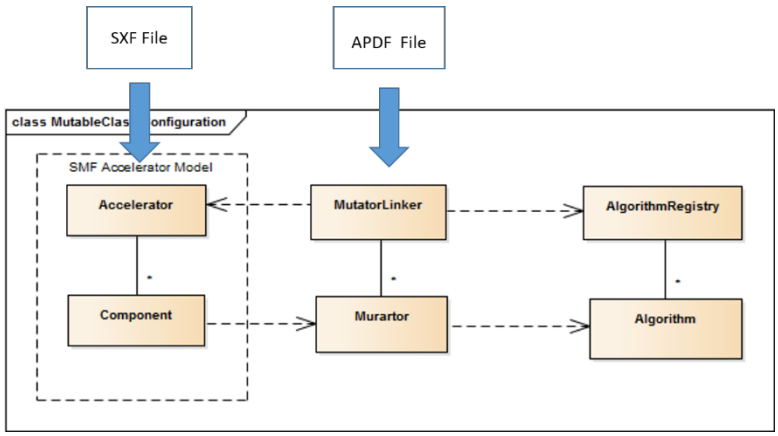

Figure 46: The configuration of the UAL propagator based on the SXF and APDF files

Technically, the APDF format formalized the Mutable Class configuration mechanism with the explicit specification. The structure of the APDF file (Malitsky and Talman, 2006) was designed around two XML elements: Propagator and Link. The Propagator represented a heterogeneous hierarchical structure of the Mutable Class instances maintaining element-algorithm associations. The Link statement defined these associations using the following attributes:

- types: regular expression for selecting accelerator nodes with specified element types, e.g., “quadrupole|sextupole”
- elements: regular expression for selecting accelerator nodes with specified design names, e.g., “q1|q2”
- algorithm: full class name of the associated propagator, e.g., “TEAPOT::MltTracker”

Despite the simplicity of the XML schema, the APDF description addressed the wide spectrum of applications ranging from small tasks to full-scale realistic beam dynamic studies encompassing heterogeneous algorithms and special effects. For example, Listing 18 shows the

APDF-based description of the full-scale TEAPOT tracking engine. The file includes a few lines binding the TEAPOT algorithms with element types.

```
<propagator id = “teapot” ring = “rhic” >
  <link algorithm = ”TEAPOT::DriftTracker” types = “default” />
  <link algorithm – “TEAPOT::SectorTracker” types = “sector” />
  <link algorithm =  “TEAPOT::DriftTracker” types = “marker|drift|[vh]monitor|monitor” />
  <link algorithm = “TEAPOT::DipoleTracker” types = “sbend” />
  <link algorithm = “TEAPOT::MltTracker” types = “quadrupole|sextupole|multipole|[vh]kicker|kicker” />
  <link algorithm = “TEAPOT::RFCavityTracker” types = “rfcavity” />
</propagator>
```

Listing 18: APDF description of the TEAPOT tracking engine

By changing one line and adding a new MIA::BPM propagator, the example can be transformed into the Model Independent Analysis (MIA) application for collecting turn-by-turn data from beam position monitors (BPMs) as shown in Listing 19:

```
<propagator id = “mia” ring = “rhic” >
  . . .
  <link algorithm =  “TEAPOT::DriftTracker” types = “marker|drift|[vh]monitor” />
  . . .
  <link algorithm = “MIA::BPM” types = monitor” />
</propagator>
```

Listing 19: APDF description of the Model Independent Analysis (MIA) propagator

In this example, MIA::BPM is an application-specific class that collects turn-by-turn data and writes them in some common container that is analyzed by the MIA prost-processing library.

Adherence to the conventional accelerator type system however introduced serious constraints for multiple applications. To resolve this issue, the APDF format added a more flexible mechanism for associating propagation algorithms with groups of elements using name-based regular expressions. From the general perspective, elements of these groups can be considered as instances of new transient types, Mutable Groups. The approach required the corresponding extension of the original Mutable Class pattern presented in Chapter 3.

Specifically, it moved element-mutator associations into the parent class as shown in Figure 47 and Figure 48.

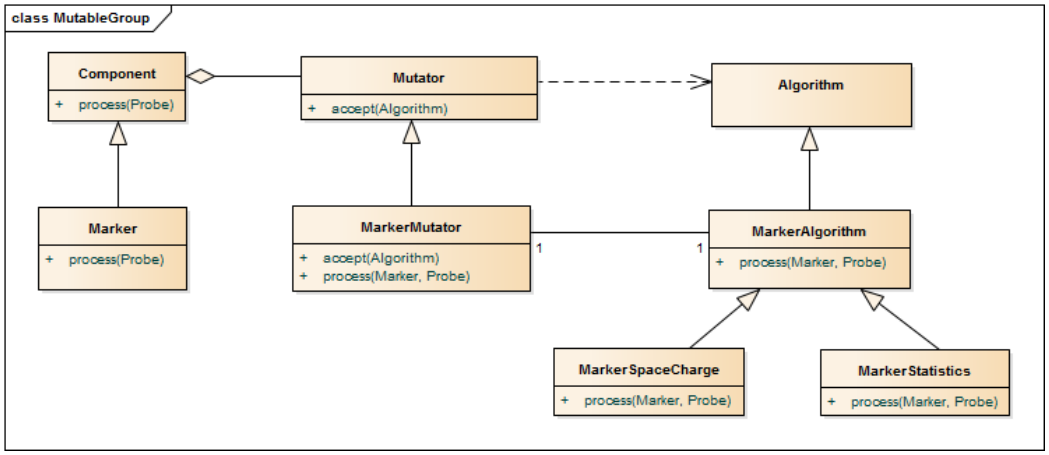

Figure 47: Class diagram of the Mutable Group variant of the Mutable Class pattern

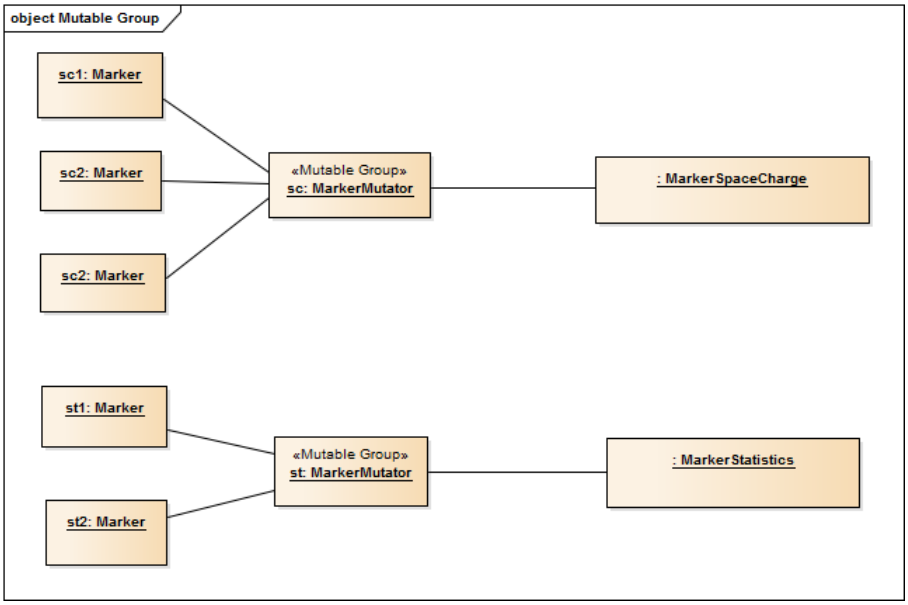

Figure 48: Object diagram of the Mutable Group variant of the Mutable Class pattern

In the new variant, elements (e.g., Markers) aggregated a pointer to a base class of element-specific mutators and downcasted it to the appropriate type (e.g., MarkerMutator) in the *process()* command. The Mutable Group extension complicated the configuration procedure

with the additional step for selecting groups of elements and building element-mutator associations. On the other hand, it significantly facilitated and generalized multiple applications by superimposing conventional elements with element-independent physical effects, such as space charge and beam-beam effects, or additional functionality, for example, measurement and connection with an interactive analysis and visualization toolkit (Fine, Malitsky, and Talman, 2006).

In addition, the name-based selection approach created a powerful platform for building efficient online modeling engines using the optimal combination of algorithms associated with different accelerator sectors (Malitsky, Satogata, and Talman, 2003). For example, chromatic effects are a typical accelerator feature modeled by many conventional element-by-element and differential algebra-based algorithms. The power of these approaches however significantly diminished their computation speed, tending to make them unacceptable for online applications. With the APDF configuration mechanism, an element-by-element offline engine can be optimized by representing regular "arc" sectors with linear matrices (see Listing 20).

```
<propagator id = "fast_teaport" ring = "rhic" >
  . . .
  <link algorithm =  "TEAPOT::MatrixTracker" elements  = "arc.*" />
  . . .
</propagator>
```

Listing 20: Fast TEAPOT

The same approach can be applied to other online applications for studying localized dominant effects (for example, interaction regions) or employing different approximations within the context of machine studies and operations. As a result, the combination of the Mutable Group framework together the SXF and APDF specifications created prerequisites for expanding the scope of the UAL off-line simulation environment towards online accelerator control systems.

Three-tier model-based control system

The modern accelerator complexes represent large billion-dollar-scale projects involving the design, manufacturing, and operation of a variety of engineering devices and systems, such as superconducting and warm high precision magnets, power supplies, RF, vacuum and cryogenic systems, diagnostics, equipment and personal protection systems, etc. The integration and control of these heterogeneous distributed facilities require advanced control systems. For example, the control system of the new National Synchrotron Light Source II (NSLS-II) project encompasses 150,000 physical I/O connections and 400,000 computed variables. To provide the comprehensive control and automation, this data has to be continuously monitored, correlated, archived, and processed in the different feedback systems and model-based high-level applications.

As in many industrial facilities (OMG, 2005), a typical accelerator control system is built after a three tier architecture illustrated in Figure 49.

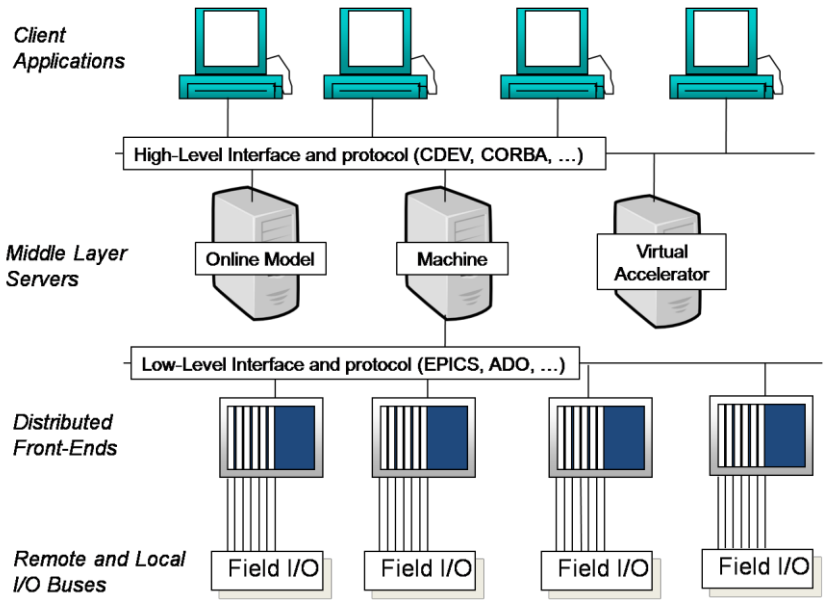

Figure 49: Typical three-tier high level application environment

In this environment, front end computers controlling physical devices form the bottom tier. Middle layer servers, such as Virtual Accelerator or Online Model, maintain common data structures and algorithms which are shared and used by an open collection of top tier thick and thin client applications.

Despite their common conceptual architecture, new accelerator projects routinely started with a new development of the model-based system, re-implementing a long list of proprietary and non-interoperable applications. This practice was determined by two associated problems: a lack of standard accelerator-oriented high-level middleware and, as a result, a lack of a middleware framework for hosting the different accelerator models and algorithms. This problem has been addressed by the EPICS-DDS project (Malitsky et al., 2009; 2010) extending the two-tier architecture of the Experimental Physics and Industrial Control System (EPICS) with the OMG Data Distribution Service middleware and the UAL framework.

DDS (OMG, 2015) is a new communication paradigm suitable for a range of computing environments, from small networked embedded systems to large-scale information backbones. At the core of DDS is the Data-Centric Publish-Subscribe (DCPS) standard API connecting applications running on heterogeneous platforms via a global distributed data space. Applications that want to share information with others can use this global data space to declare their intent to publish data that is categorized into one or more topics of interests to participants. Similar, applications that want to access topics of interests can also use this data space to declare their intent to become subscribers. The underlying DDS middleware propagates data samples written by publishers into the global data space, where it is disseminated to interested subscribers.

EPICS-DDS specialized the DDS topic-oriented approach in the context of accelerator model-based control systems. According to the EPICS-DDS uniform scenario, middle layers servers

maintained states of topics shared by other servers and high-level clients. The Machine server represented a central component of this facility. It maintained a state of magnet strengths. Other participants subscribed to the Machine server for synchronizing their containers. Particularly, the Online Model and Virtual Accelerator servers recalculated and updated their own states of the design optics and turn-by-turn beam data respectively. The UAL framework complemented this generic service-oriented interface with the consistent configuration mechanism for building project-specific computational engines. For example, Figure 50 shows a structure of the Virtual Accelerator (VA) server.

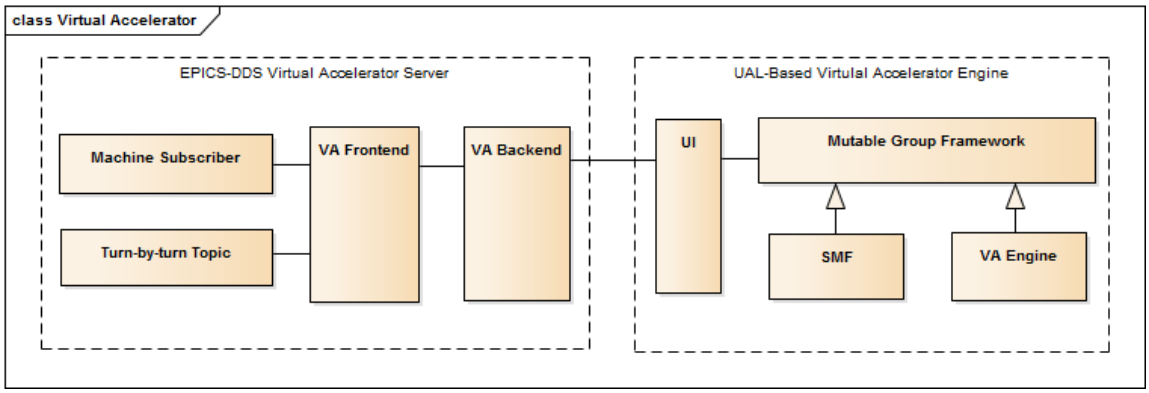

Figure 50: Virtual Accelerator server

A server front end provides a communication with the Machine server and transferred data updates to a corresponding computational backend consisting of the UAL accelerator model and propagator. The consistency among accelerator models of distributed servers are determined by common initialization data sources, such as accelerator exchange files (e.g., SXF or ADXF) or accelerator control databases. Similar to the UAL off-line applications, the accelerator model can be extended with new element types and the DDS communication protocol supports these extensions with dynamic self-described data types of the DDS Extensible and Dynamic Topic Types specification (OMG, 2014). The propagator part of the computational backend is server-specific and is configured with the APDF (accelerator propagator description format) files. As a

result, EPICS-DDS preserves the extensibility and flexibility of the Mutable Class framework in the context of large-scale model-based control systems.

Compiler Construction

Similar to computational accelerator physics studies, compiler construction relies on multiple collections of algorithms associated with the different phases of the compilation process, including context checking, optimization, and code generation. The connectivity of these phases is provided by an intermediate model, called Abstract Syntax Tree (AST), representing the source program. Chapter 2 overviewed two categories of the AST structures, homogeneous and heterogeneous, and discussed advantages of the latter approaches. The example of the heterogeneous AST model is shown in Figure 18. It maps programming language constructs, such as the if statement, into the corresponding data structures improving modularity and cohesion of compiler systems. As shown in Figure 19, the heterogeneous model adds a new dimension to a collection of compiler algorithms leading to their two-dimensional view. This type of system is addressed by the Mutable Class pattern (see Figure 30) and the corresponding instantiation of this pattern is shown in Figure 51.

The section considers the application of the Mutable Class pattern in the context of the JastAdd extensible compiler construction system (Hedin and Magnusson, 2003; Hedin, 2010). In contrast with alternate projects, JastAdd introduces an ideal platform for such studies. First, its highly configurable framework and the Mutable Class pattern are driven by the same conceptual objective that facilitates their comparison and integration. Second, JastAdd combines the object-oriented approach with the aspect-oriented weaving mechanism.

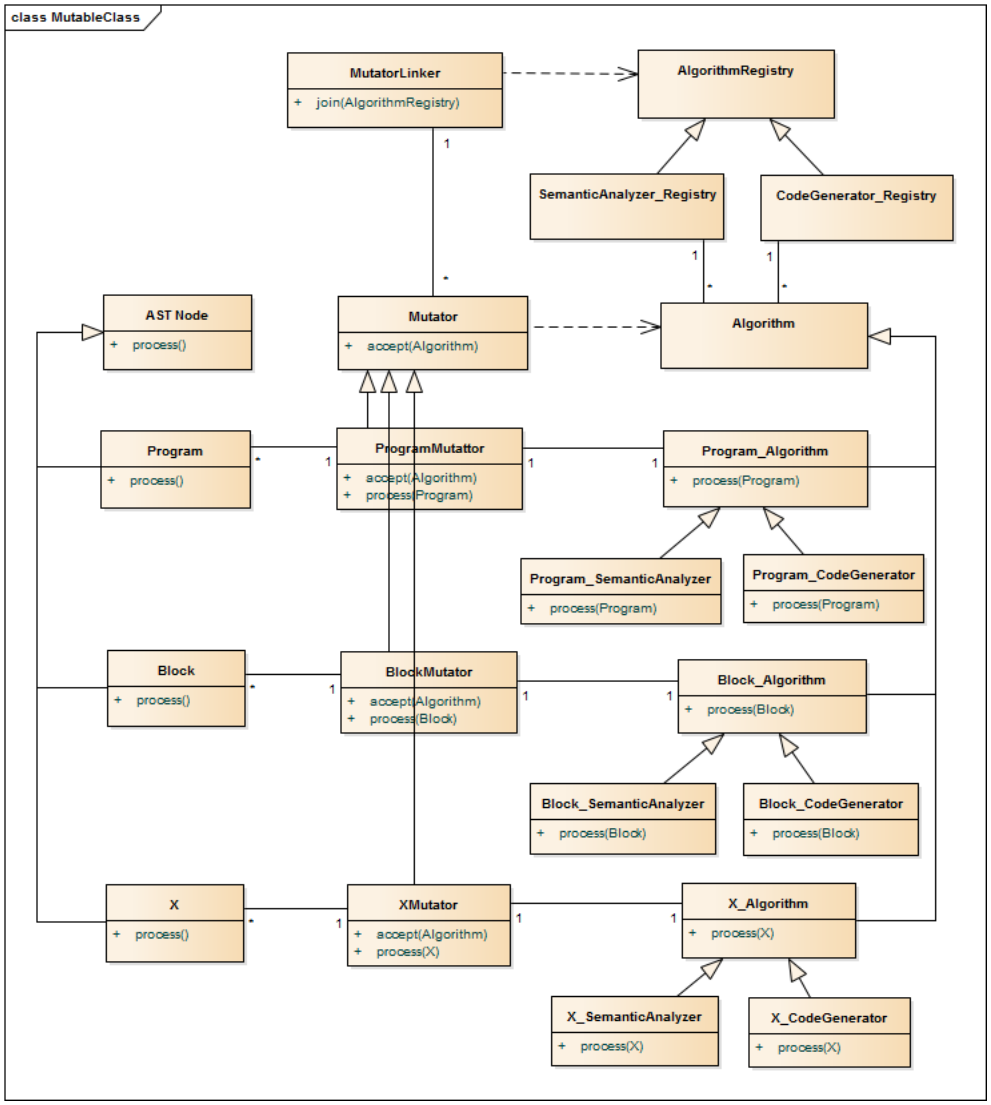

Figure 51: Mutable Class-based structure of compiler algorithms (Figure 19)

The impact of aspect-orientation on compiler development was thoroughly discussed by Wu and colleagues (Wu et al., 2006). Adhering to the Visitor pattern as a strategic direction, the authors consistently developed an aspect-oriented version based on the elaborated comparison of the pattern's object-oriented features and the AspectJ programming language concepts, such as inter-type declarations, pointcut-advice model, aspect field and methods, and aspect inheritance. The suggested aspect-oriented approach was proof tested in a case study of the proprietary

RelationJava compiler. The JastAdd compiler system generated an extensible Java compiler JastAddJ (Eman and Hedin, 2007) elevating the research applications to the next level.

The rest of this section is broken down into two parts. First, it gives a brief overview of the JastAdd framework and then introduces the new extension based on the Mutable Class pattern.

JastAdd Framework

JastAdd (Hedin and Magnusson, 2003; Hedin, 2010) is a configurable metacompiler construction system. For achieving a higher level of extensibility it is designed after a composite approach combining the object-oriented mechanism with the proprietary declarative implementation of the aspect-oriented concepts. The JastAdd framework and the generated compilers are implemented in the object-oriented language Java, but the language grammar and related processing algorithms are defined in a collection of the external text files. These files represent key components of the extensible mechanism in the JastAdd compilation process which is organized as a sequence of the file-processing steps (see Figure 52): generation of the parser according to the context-free grammar, translation of the abstract grammar file, building the AST classes from the integration of the declarative and imperative behaviors, and compiling a source program.

In JastAdd, a parser is generated with external tools, the usual choice being one of two open-source parser generators: JJTree and Beaver. Both tools work according to a similar scheme. They read files with the *context-free grammar* of the compiling language and generate a Java class that associates the grammar production rules with the construction of the AST nodes. This tree-building mechanism is integrated with the JastAdd framework by implementing the corresponding interface of the AST classes. The parser generators do not impose any constraints on the AST implementation and the actual structure of the abstract syntax tree is defined in the

additional *abstract grammar* .ast file.

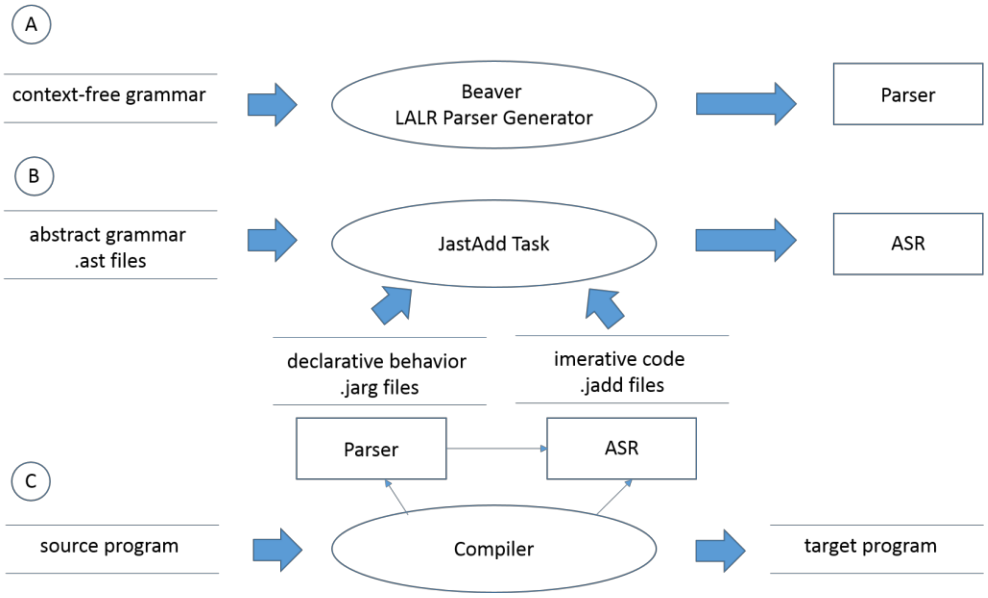

Figure 52: The three steps of the JastAdd compilation process: (a) building a parser, (b) building the AST classes, (c) compiling a source program

The context-free and abstract grammars outline only the backbone of the abstract syntax tree. Its implementation, behavior and extensions are defined in other external .jrag and .jadd files. Each file, .jrag and .jadd, represents the particular crosscutting functionality (or aspect) of the AST-oriented operations, such as name analysis, type checking, and others. During the generation of the AST classes, JastAdd processes all these files and inserts the fields and methods into the appropriate nodes. The two types of these files correspond to the two types of node behaviors: declarative and imperative.

The *declarative behavior* is specified in .jrag files and includes the inter-type declarations written in Reference Attributed Grammars (RAG). The RAG language uses a slightly extended and modified variant of Java semantics. Each class consists of a list of attribute declarations, method declarations, and equations. Attribute declarations are written like field declarations, but

with additional modifiers. In the resulting tree, all attributes of .jrag files are included in the corresponding AST nodes and complimented with the public access methods.

The .jadd files encapsulate the *imperative code* of the node-specific algorithms. In the context of the Visitor pattern, each file corresponds to the concrete Visitor addressing the particular task. The .jadd files use the conventional Java syntax and contain a list of visit-like methods associated with the different AST nodes. For example, Listing 21 shows an extract of the PrettyPrint.jadd file with a collection of the AST dumpTree methods.

```
aspect PrettyPrint {

  …

// dump the AST to standard output

public String Program.dumpTree() {
    StringBuffer s = new StringBuffer();
    for(Iterator iter = compilationUnitIterator(); iter.hasNext(); ) {
      CompilationUnit cu = (CompilationUnit)iter.next();
      if(cu.fromSource()) {
        s.append(cu.dumpTree());
      }
    }
    return s.toString();
  }

 public void ASTNode.dumpTree(StringBuffer s, int j) {
    for(int i = 0; i < j; i++) {
      s.append("  ");
    }
    s.append(dumpString() + "\n");
    for(int i = 0; i < getNumChild(); i++)
      getChild(i).dumpTree(s, j + 1);
  }
…
}
```

Listing 21: Extract of the PrettyPrint.jadd file with the PrettyPrint aspect.

In accordance with the aspect-oriented terminology, this file represents the JastAdd-based aspect for printing of the AST structure. Since JastAdd weaves the .jadd file into the AST classes, the implementation of the aspect methods takes into account the AST class hierarchy. The weaving process is scalable and can be simultaneously applied to many other aspects, such as type checking, code generation, and others.

The JastAdd system was applied to build the full-scale extensible Java compiler JastAddJ

(Ekman and Hedin, 2007). According to the benchmark results, it outperformed other extensible Java compilers, like Polyglot and JaCo and was only within a factor of three slower than Javac, a standard compiler in Sun JDK. Additionally, the implementation of the JastAddJ compiler demonstrated and confirmed the extensibility mechanism of the JastAdd system. This JastAdd extension mechanism however is static and does not resolve the same run-time issues associated with the aspect-oriented approach. Initially separated in the different files, the JastAdd aspects are eventually merged and disappear into the huge monolithic AST classes preventing its run-time interchange and extension.

Mutable Class-based JastAdd Extension

According to the Mutable Class approach, each node of the AST is associated with the corresponding class type which maintains a pointer to the AST Node Algorithm instance. The AST traversing procedure does not access this instance directly and delegates the request via the AST *process* method. Drawing an analogy with the Visitor pattern, the Mutable Class approach replaces the Visitor run-time selection mechanism with prior binding. Figure 53 illustrates this delegation scheme on the example of the AST Program class. The algorithm for processing objects of this class is already selected and connected with ProgramMutator by some external procedure.

The extra level of indirection in the Mutable Class approach brings flexibility to the overall framework. According to the aspect-oriented terminology, the Mutator serves as a joint point between the extent of the AST nodes and the woven algorithm. The advantage of this scheme is especially visible in multi-type models like the heterogeneous AST structures. Continuing the analogy with the Visitor pattern, one can consider the registry of algorithms as an extensible alternative variant of the Visitor classes.

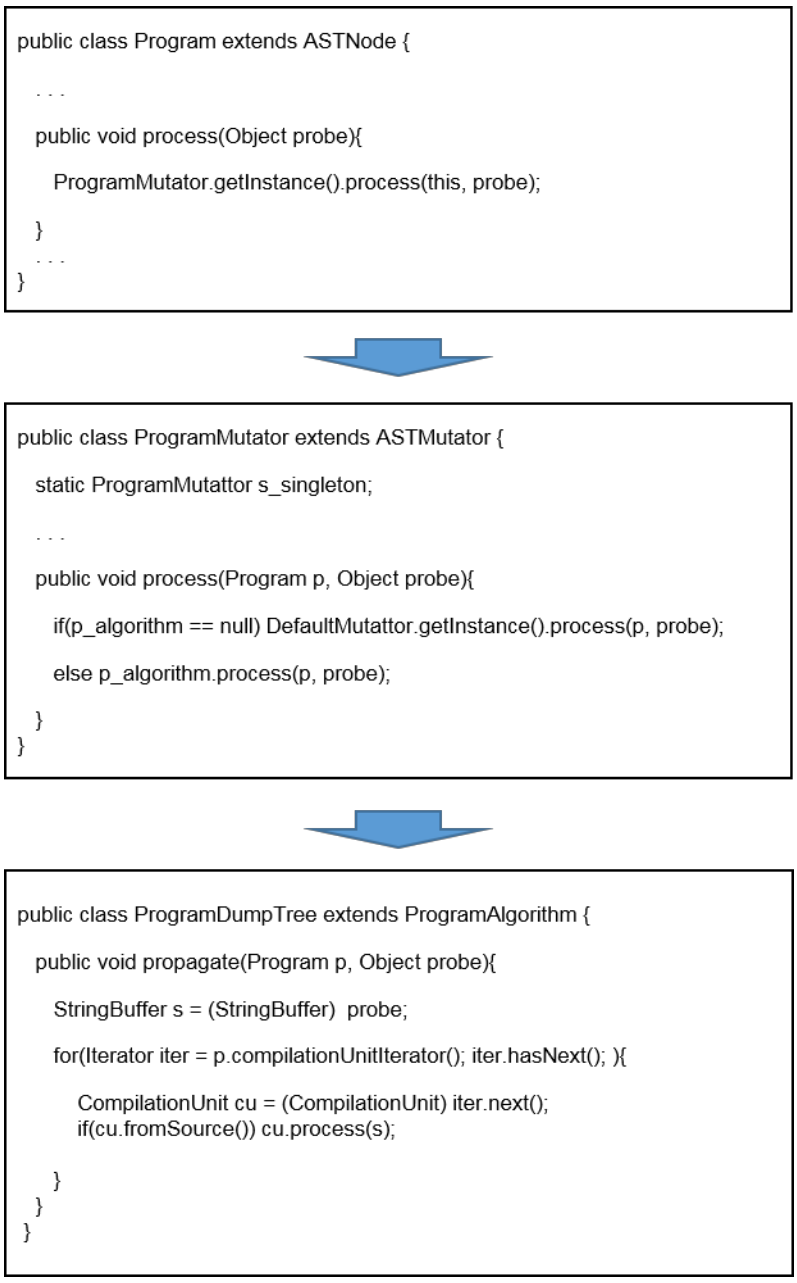

Figure 53: The delegation scheme of the algorithm invocation in the AST program node based on the Mutable Class approach.

The collection of algorithms can be dynamically changed in many different ways. Two of them are illustrated in Listing 22. In the first example, the algorithm of the MethodAccess node has been replaced with some local version. As shown in Figure 53, the Mutable Class delegation scheme does not require the implementation of all types in the algorithm hierarchy and the

original set of algorithms can be augmented with the new entries for more precise processing of particular nodes. In the same way, the collection of algorithms can be extended to support new AST classes because of the evolution of the language grammar and constructs.

```
// Example 1: Replacing algorithm of the MethodAccess node

dumptree.AlgorithmRegistry dtRegistry = dumptree.AlgorithmRegistry.getInstance();

// replace or add algorithm for the corresponding node type
dtRegistry.getAlgorithms().put("MethodAccess", new MethodAccessAlgorithm () {
    public void process(MethodAccess ma, Object probe){
        StringBuffer s = (StringBuffer) probe;
        s.append("\n *** ");
        s.append(ma.getClass().getName());
        s.append(" - New method access algorithm \n\n");
    }
});

// connect algorithms with the corresponding node types

MutatorLinker.getInstance().join(dtRegistry)

// define the propagated object (probe)
StringBuffer probe1 = new StringBuffer();

// propagate it through AST
program.process(probe1);

// postprocess the probe
System.out.println(probe1.toString());

// Example 2: Applying a new collection of algorithms

NewAlgorithmRegistry myRegistry = NewAlgorithmRegistry.getInstance();

// connect algorithms with the corresponding node types
MutatorLinker.getInstance().join(myRegistry);

// define the propagated object (probe)
int[] probe2 = new int[1];
probe2[0] = 0;

// propagate it through AST
treePrinter.program.process(probe2);

// postprocessed the probe
System.out.println("number of nodes = " + probe2[0]);
```

Listing 22: Two examples illustrating the run-time weaving mechanism of the Mutable Class approach.

The second example shows the application of the new collection of algorithms to the same AST structure. Rebinding of processing algorithms is done with the single method of MutatorLinker. As a result, the different phases of compilation procedure can be dynamically loaded and combined in the boundary of the common application.

The integration of the Mutable Class approach with the JastAdd framework was natural and did not require any changes in the existing classes (Malitsky, 2008). The JastAdd compilation

process was extended with two steps: implementation of the Mutable Class delegation scheme and refactoring the JastAdd files with the imperative code into the reusable collections of the corresponding algorithms. The Mutable Class delegation scheme (see Figure 53) is based on the AST Mutator classes and requires the additional *process* methods in the nodes of the AST structure. The hierarchical tree of AST Mutators was automatically generated from the abstract grammar .ast file. Following the JastAdd procedure, the propagate methods of the AST nodes were defined as the declarative behavior in file .jrag file and woven in the subsequent step. In the new scheme, files with the imperative code were not included in the weaving step and were instead replaced with the run-time libraries. As a result, the approach added run-time dynamics to the compiler implementation. First, it facilitated the interchange, comparison and composition of the third-party extensions. Second, it allowed the combination of different compiler phases into a single application.

Summary

The chapter addresses the second and third research questions of the dissertation. First, it demonstrates reusability of the Mutable Class pattern in the context of two application domains: computational accelerator physics and compiler construction. Moreover, the corresponding applications were implemented in two programming languages, C++ and Java. Second, these studies explore the scalability boundary of the pattern from the perspective of the application architecture and computational infrastructure. As shown in this chapter, the Mutable Class model became a core part of the Unified Accelerator Library (UAL) framework employed in various types of application programs and deployed on parallel clusters and three-tier distributed infrastructure.

Chapter 5

Conclusions, Recommendations, and Summary

Conclusions

The dissertation proposed a new approach, Mutable Class, for processing heterogeneous models and provided a comprehensive study addressing three research topics: formalization of this approach as a design pattern, validation of its reusability in the context of two application domains, and analysis of the scalability boundary of pattern-based applications including distributed three-tier systems.

After the first publication of the book "Design Patterns: Elements of Reusable Object-Oriented Software" (Gamma, Helm, Johnson, and Vlissides, 1995), the catalog of software design patterns has accumulated numerous solutions spanning multiple categories of software design topics. The integrity and consistency of this collection has been determined by a standard format. The dissertation followed the formal procedure and presented the Mutable Class pattern through a sequence of required sections: intent, motivation, applicability, structure, and others. The corresponding description clearly identified its relationship with the Visitor pattern, addressing the same intent and motivation. The Visitor pattern, however, introduced a serious limitation by freezing the class hierarchy of application. This limitation was explicitly recognized in the pattern specification (Gamma, Helm, Johnson, and Vlissides, 1995). Therefore, the dissertation thoroughly analyzed this issue in the context of dedicated extensions of the Visitor pattern and showed that it cannot be resolved within the Visitor framework. As a result, the Mutable Class pattern introduced a new approach based on the Class model of the UML specification (OMG,

2011). Technically, it augmented the Class model with the Strategy pattern, implementing the mutation mechanism for interchangeable operations. According to the standard outline of the pattern description, this composite approach was unambiguously expressed with the UML class and interaction diagrams, demonstrated with a sample code, and elaborated with implementation aspects.

The idea of the Mutable Class pattern was introduced in the context of the framework of Unified Accelerator Libraries (Malitsky and Talman, 1998), addressing actual applications of computational accelerator physics. UAL was designed to establish a universal platform for modeling existing and future accelerator projects with an open and configurable set of accelerator algorithms. The significant scope of this environment provided an excellent testbed for the incremental development and validation of the Mutable Class pattern. Moreover, this approach boosted the development of new types of simulation studies, such as insertion and analysis of new physical devices, integration of multiple effects, consideration of new categories of observables propagated by algorithms, and extension of algorithms for selected groups of heterogeneous elements. The corresponding applications were implemented in multiple accelerator projects and presented at various conferences and workshops.

The accumulated experience with accelerator tasks confirmed the extensibility solution of the Mutable Class pattern and encouraged further exploration within other application domains, such as 3D computer graphics and compiler construction. Open Inventor (Wernecke et al., 1994; Heck, 2010) is one of major scientific visualization toolkits establishing a de facto standard of the 3D scene graph model and application programming interface. Analysis of its source code revealed a proprietary mechanism that was closely related with the Mutable Class approach for processing type-specific algorithms. Therefore, the dissertation considered the implementation of

the Mutable Class pattern in the context of the compiler construction domain, particularly, the JastAdd metacompiler construction system (Hedin and Magnusson, 2003; Soderberg et al., 2013). In contract with the Open Inventor toolkit, JastAdd introduced a solution designed after the aspect-oriented programming (AOP) paradigm. According to this approach, algorithms and extensions of data models were defined as aspects and merged with object-oriented data models using an aspect-oriented compiler. The approach, however, was static leading to composite monolithic classes. The Mutable Class pattern resolved this issue by bringing the run-time mechanism for managing the JastAdd aspects. In addition, this project highlighted the relationship between the Mutable Class pattern and the AOP approach.

The final research topic was dedicated to the scalability analysis of the Mutable Class pattern. Being a core part of the UAL framework, the pattern was challenged in different projects and settings. The initial applications addressed immediate requirements of modern accelerator facilities, such as RHIC and LHC, for evaluating effects of new physical devices. These studies eventually accumulated major accelerator libraries and numerous proprietary algorithms into a common integrated environment. In turn, this environment triggered the development of realistic beam dynamic models encompassing multiple physical effects and dynamic multi-stage processes. The scale of studies, especially space charge simulations with millions of particles, required significant computational resources. The Mutable Class pattern addressed this demand by providing a flexible mechanism for mixing conventional and parallel algorithms associated with different types of elements of the same model. This approach was further developed for mixing simulation algorithms with subscribers of third-party visualization and analysis toolkits. The success and experience with simulation studies encouraged extending the scope of the UAL applications with the three-tier distributed accelerator control system. As a result, the

Mutable Class pattern was used as a common configuration framework for building the middle layer of model-based servers processing different algorithms triggered by operator's requests or changes in control devices.

Recommendations

The Mutable Class pattern has been developed as an alternative approach to the Visitor pattern to support the evolution and extensions of heterogeneous application models. As described in the Review of Literature, the Visitor ecosystem encompasses multiple application domains, such as compiler construction and 3D computer graphics. Therefore, it will be important to consider the Mutable Class pattern in the context of the next versions of existing Visitor-based toolkits or new Visitor-oriented projects.

This Visitor-to-Mutable Class transition will facilitate the consolidation of accumulated legacy third-party applications and bring a consistent mechanism for the development and configuration of new extensions. Technically, the Mutable Class framework can be integrated after the Adapter pattern (Gamma, Helm, Johnson, and Vlissides, 1995). The corresponding approach is comprehensively described in Results within the Mutable Class-based extension of the JastAdd metacompiler construction system. According to this example, the node interface of the original model needs to be extended with the *process ()* method associated with the Mutable Class pattern. Then, each heterogeneous node can be updated to implement this method or extended with the corresponding specializations. In the case of the JastAdd application, the development of many extensions was automated by reusing the JastAdd aspect-oriented compilation procedure. As a result, the Mutable Class pattern augmented the original static approach with the run-time mechanism for interchanging different compilation phases.

The relationship of the Mutable Class pattern with the Aspect Oriented Programming (AOP) paradigm highlighted another topic associated with the further development of the Mutable Class run-time mechanism with the AOP conceptual model, including the implementation of Aspects, Pointcuts and other major concepts. This direction will approach three tasks. First, it can significantly enhance the upgrade of the Visitor-based toolkits with a more consistent framework for managing run-time extensions. Next, it can facilitate composite studies described in the Results chapter in the context of computational accelerator physics projects. Moreover, corresponding integrated models can be further generalized for joining several intra- and inter-domain libraries or toolkits. Finally, the Mutable Class pattern can be considered for deriving a generic aspect-oriented reference model bringing run-time mutability to object-oriented applications.

Recently, the processing of heterogeneous models with multiple algorithms becomes especially actual in the context of large-scale data-intensive computational platforms driven by requirements of industrial and scientific applications. One of them, SciIO[1], was proposed to address several major research themes defined in the Working Group Report of the Accelerator Scientific Knowledge Discovery (ASKD) workshop (2013):

1. knowledge acquisition, management, and sharing
2. rapid knowledge-based response and decision making mechanisms
3. data and knowledge fusion
4. dynamic resource collection, discovery, allocation, and management
5. composition and execution of end-to-end scientific processes
6. human computer interaction
7. trust and attribution

[1] Malitsky, N. (2015). Assessment of the Spark Approach for NSLS-II. Computational Science Center Seminar, BNL

This list was compiled from 21 science drivers including high energy physics and light source facilities, materials genome, and others. The SciIO project aims to facilitate the steering of efforts by providing an integrated framework for developing and composing many-to-many associations between multiple processing algorithms and heterogeneous data sources. The proposed approach uses the integration concept from two angles: conceptual and technical. Within the conceptual view, the platform aims to provide a common data science environment for building a path from data to information to knowledge as shown in Figure 54.

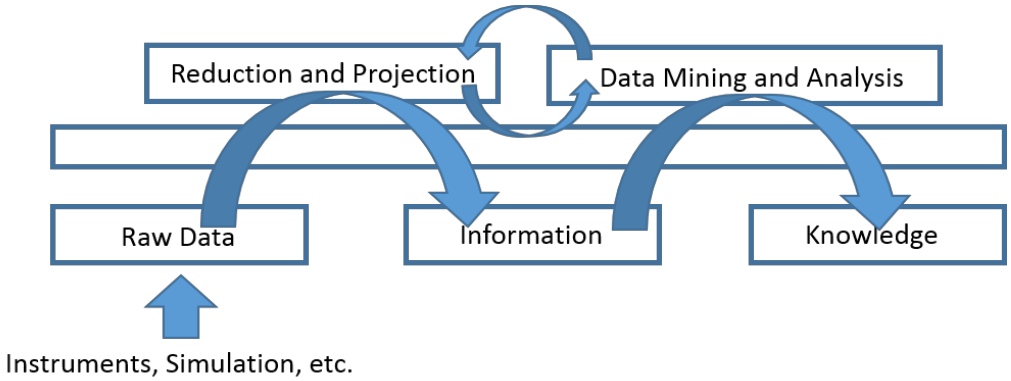

Figure 54: Knowledge discovery process model

The diagram only outlines an abstract sketch of the knowledge data discovery path without describing the complexity of this topic. In fact, there are a variety of different knowledge discovery process models in industrial (Mariscal, Marban, and Ferndez, 2010) and scientific (DOE ASCAC, 2013) application domains. The scope and scale of their implementation introduce a serious technical challenge and require significant resources. Therefore, the project endorses the integration approach built around a Spark programming model (Zaharia, 2013). In contrast with existing data management and analytics systems, this model provides a consistent framework for in-situ processing of various algorithms with a variety of data sources. For example, Spark already supports SQL engines, machine-learning techniques, graph-based algorithms and several relational and NoSQL databases. Therefore, the SciIO project proposes to

extend the Spark ecosystem with the heterogeneous data of experimental facilities and new types of algorithms for implementing different phases of the knowledge discovery path (see Figure 55). Recently, this approach has been included in the DOE SBIR proposal (Pazandak, 2015) that combines the scopes of scientific-oriented facilities and emerging Industrial Internet of Things (IIoT) applications.

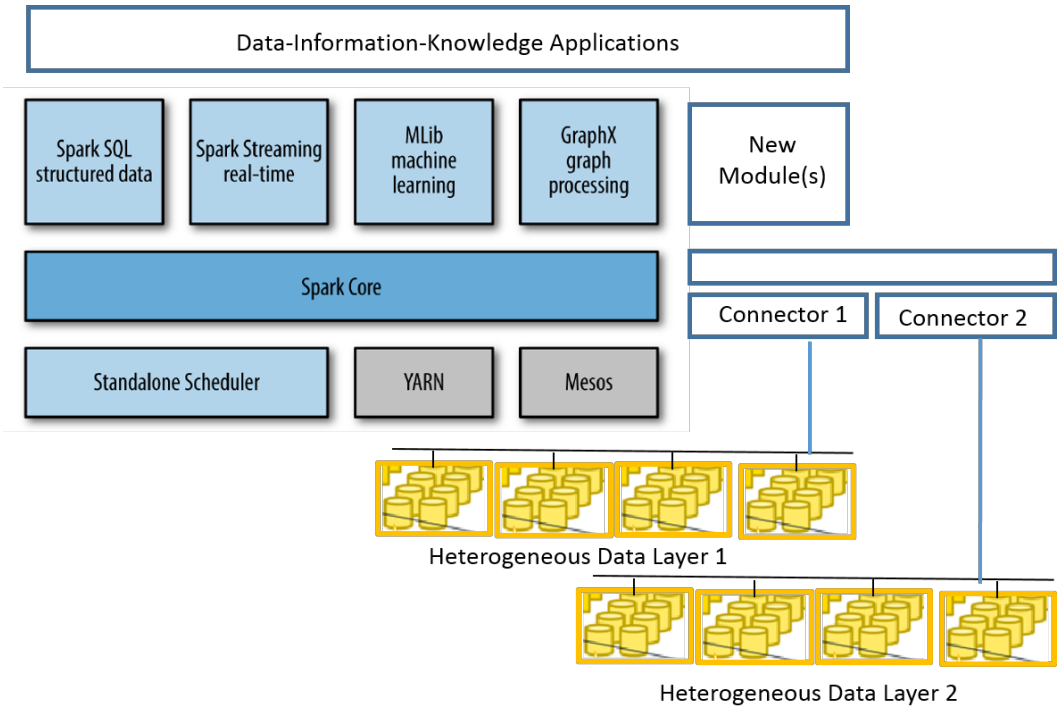

Figure 55: Spark-based integrated platform

In order to provide coverage of a broad set of applications, the design of the SciIO integrated platform is adhered to a generic data model of the latest version of the Hierarchical Data Format (HDF5) that has become a de facto standard for a wide range of application domains (HDF, 1997-2015). The HDF5 model is based on four primary concepts: multi-dimensional *datasets*, user-defined *datatypes*, *attributes* for containing metadata information, and *groups* for composing a collection of datasets into the hierarchical structures. As mentioned above, Spark already supports several important categories of applications including graph algorithms. For example, the GraphX module introduces the highly optimized implementation of the property graph based on the three distributed collections: VertexRDD, EdgeRDD, and EdgeTriplet. This

property graph model, however, does not provide an efficient implementation of hierarchical tree algorithms. Therefore, SciIO aims to add a new module with the TreeRDD collection of key-value pairs with values maintaining branches of heterogeneous trees.

According to the Visitor and Mutable Class patterns, the processing of heterogeneous trees with multiple families of algorithms relies on the run-time configuration mechanism that needs to be controlled from the Spark client application. This type of interface is not supported by the Spark programming model and requires the development of corresponding extension. Moreover, this requirement highlights a conceptual issue and its TreeRDD-based solution can be considered as a prototype for advancing the Spark programming model with the next level of flexibility expected by scientific-oriented applications. On the other hand, the Spark-based applications introduce another conceptual issue affecting the Mutable Class pattern. Specifically, new data-intensive applications consistently move from structured models towards semi-structured and unstructured datasets. As a result, corresponding run-time configuration mechanisms of model-algorithm associations require more flexible variants, like Mutable Group (see Figure 48), or new solutions.

Summary

The dissertation introduced a new design pattern, Mutable Class, to support the processing of large-scale heterogeneous data models with multiple families of algorithms. The pattern captures two fundamental concepts: heterogeneity (of data models) and mutability (of associated processes). As a result, it addresses multiple applications. Particularly, the dissertation explored this design pattern in several application domains, such as computational accelerator physics, compiler construction, and 3D computer graphics. The analysis showed that all these

applications can be considered from the perspective of a heterogeneous tree-based data model and a two-dimensional view of processing algorithms. The first dimension of this view is associated with the different tasks of algorithms. For example, in the context of compiler construction, it corresponds to the different phases of a compilation process, such as lexical analysis, parsing, semantic analysis, optimization and code generation. The second dimension is induced by the data types of heterogeneous application models. For example, in compiler construction, data types represent different nodes of Abstract Syntax Tree, such as a program, block, if statement, and others. According to the two-dimensional view of algorithms, data-algorithm associations need to be dynamically changed in complex multi-phase applications. This requirement is not explicitly supported by modern programming language models and represent an important target of multiple software engineering approaches.

The dissertation considered two major approaches to address this problem: the Visitor pattern and the aspect-oriented programming paradigm. The Visitor pattern slices a two-dimensional matrix of algorithms into type-specific collections of type-associated algorithms and implements these collections with separate classes. As a result, the pattern provides a consistent mechanism for interchanging type-specific algorithms. The approach however introduces a serious limitation by freezing the class hierarchies of application models. The aspect-oriented programming (AOP) paradigm brings new ideas addressing similar issues from a different perspective. Particularly, it augments the object-oriented model with a weaving mechanism for inserting structural and behavioral changes across heterogeneous components of conventional (not-aspect-oriented) programs. The dissertation analyzed two influential implementations of the AOP approach: AspectJ and Spring AOP. AspectJ is an original aspect-oriented Java extension developed by the authors of AOP to validate and endorse the new programming paradigm. This implementation,

however, is based on a compiler that merges the aspect-oriented declarations into the Java byte code. Spring AOP extends the AspectJ static approach with a run-time mechanism based on the Interceptor and Proxy patterns. The mechanism addresses enterprise-level applications, but introduces a significant overhead, preventing its integration in the context of fine-grain application models, such as abstract syntax trees or scene graphs.

The Mutable Class pattern represents a composite solution combining the best features of both the Visitor pattern and the AOP paradigm. Conceptually, it is designed as an extension of the object-oriented class model by adding the mutability concept. From this perspective, the pattern is related to the AOP paradigm, augmenting the inheritance and composition mechanisms with a weaving procedure for changing data-algorithm associations. On the other hand, this procedure does not introduce any overhead associated with the AOP paradigm and can be directly applied within the existing object-oriented applications and approaches. As a result, it preserves the run-time behavior of the Visitor pattern. Technically, the Mutable Class pattern replaces the Visitor monolithic interface with extendable registries of operations and adds a run-time linking step serving as a lightweight weaving mechanism for connecting objects of processed models with the selected registry of operations. This additional step is fully consistent with the design pattern methodology and can be considered as the extra level of indirection improving coupling and cohesion metrics of the object-oriented applications.

The dissertation provided a formal description of the Mutable Class pattern and evaluated its applicability and value in the context of two application domains: computational accelerator physics and compiler construction. Historically, the idea of the Mutable Class pattern was introduced for building an open simulation environment addressing multiple tasks of accelerator studies (Malitsky and Talman, 1998). As a result, the pattern became a core part of the Unified

Accelerator Libraries (UAL) framework that integrated major accelerator approaches, numerous extensions, and applied to several accelerator projects, such as Relativistic Heavy Ion Collider (RHIC) at Brookhaven National Laboratory (BNL), Cornell Electron-positron Storage Ring (CESR), Large Hadron Collider (LHC) at the European Organization for Nuclear Research (CERN), the Spallation Neutron Source (SNS) accumulator ring at the Oak Ridge National Laboratory, and others. The applications challenged and confirmed the approach within different contexts and infrastructures ranging from task-specific extensions to facility-wide online model-based control systems. Moreover, the Mutable Class pattern facilitated the development of a new direction in accelerator computational studies involving the integration of multiple physical effects.

Following the design pattern methodology, the assessment of the Mutable Class model required another vertical application domain for testing its generalization ability. Therefore, the dissertation extended the scope of the pattern analysis with the JastAdd extensible compiler construction system. For archiving a higher level of extensibility, JastAdd implemented its own variant of the aspect-oriented weaving mechanism and represented a principally new platform for these studies. Similar to the AspectJ compiler, the JastAdd extension mechanism was static, leading to the huge monolithic classes that merged multiple processing algorithms with the application model. The Mutable Class pattern enhanced this approach by replacing these classes with dynamic associations and providing run-time support of their interchange and composition with the third-party extensions.

The Mutable Class pattern targeted a fundamental topic of software engineering, the evolution of type systems and associated algorithms. In the spirit of the design pattern methodology, it highlighted the essence of a problem and provided the corresponding solution addressing

immediate practical applications. As a result, this approach and associated concepts can be, and need to be, further developed in the context of new tasks and technologies. The development of emerging technologies is driven by dramatic increases in multiple V's (Volume, Velocity, Variety, Value, and Veracity) of Big Data. Moreover, the Variety is becoming one of the most challenging requirements of new applications. This topic is directly related to major aspects of the Mutable Class pattern. New data models, such as heterogeneous information networks (Sun and Han, 2012), and large-scale computing platforms, like Spark (Zaharia, 2013), extend the context of this pattern and raise the demand for future studies.

Appendices

Appendix A: ISO/IEC 25010 Product Quality Model (2011)

Characteristics	Sub-characteristics	Definition
Function suitability		degree to which a product or system provides functions that meets stated and implied needs when used under specified conditions
	function completeness	degree to which the set of functions covers all the specified tasks and user objectives
	function correctness	degree to which a product or system provides the correct results with the needed degree of precision
	function appropriateness	degree to which the functions facilitates the accomplishment of specified tasks and objectives
Performance efficiency		performance relative to the amount of resources used under stated conditions
	time behavior	degree to which the response and processing times and throughput rates of a product or system, when performing its functions, meet requirements
	resource utilization	degree to which the amounts and types of resources used by a product or system, when performing its functions, meet requirements
	capacity	degree to which the maximum limits of a product or system parameter meet requirements
Compatibility		degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment
	co-existence	degree to which a product can perform its required functions efficiently while sharing a common environment and resources with other products, without detrimental impact on any other product
	interoperability	degree to which two or more systems, products or components can exchange information and use the information that has been exchanged
Usability		degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use
	appropriateness recognizability	degree to which users can recognize whether a product or system is appropriate for their needs
	learnability	degree to which a product or system can be used by specified users to achieve specified goals of learning to use the product or system
	operability	degree to which a product or system has attributes that make it easy to operate and control
	user error protection	degree to which a system protects users against making errors
	user interface aesthetics	degree to which a user interface enables pleasing and satisfying interaction for the user
	accessibility	degree to which a product or system can be used by people with the widest range of characteristics and capabilities
Reliability		degree to which a system, product or component performs specified functions user specified conditions for a specified period of time
	maturity	degree to which a system, product or component meets needs for reliability under normal operation
	availability	degree to which a system, product or component is operational and accessible when required for use

	fault tolerance	degree to which a system, product or component operates as intended despite the presence of hardware or software faults
	recoverability	degree to which, in the event of an interruption or a failure, a product or system can recover the data directly affected and re-establish the desired state of the system
Security		degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization
	confidentiality	degree to which a product or system ensures that data are accessible only to those authorized to have access
	integrity	degree to which a system, product or component prevents unauthorized access to, or modification of, computer programs or data
	non-repudiation	degree to which actions or events ca be proven to have taken place, so that the events or actions cannot be repudiated later
	accountability	degree to which the actions of an entity can be traced uniquely to the entity
	authenticity	degree to which the identity of a subject or resource can be proved to be the one claimed
Maintainability		degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers
	modularity	degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on the other components
	reusability	degree to which an asset can be used in more than one system, or in building other assets
	analysability	degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identity parts to be modified
	modifiability	degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality
	testability	degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met
Portability		degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another
	adaptability	degree to which a product or system can effectively and efficiently be adapted to different or evolving hardware, software or other operational or usage environments
	installability	degree of effectiveness and efficiency with which a product or system can be successfully installed and/or uninstalled in a specified environment
	replaceability	degree to which a product can replace another specified software product for the same purpose in the same environment

References

Aksit, M., Bergmans, L., & Vural, S. (1992). An Object-Oriented Language-Database Integration Model: The Composition Filters Approach. *In Proc. of the 7th European Conference on Object-Oriented Programmin*.

Berg, K., Conejero, J., & Chitchyan, R. (2005). *AOSD Ontology 1.0 – Public Ontology of Aspect-Orientation*, AOSD-Europe.

Bloch, J. (2008). *Effective Java* (2nd ed.). Addison-Wesley.

Booch, G. (1991). *Object-Oriented Analysis and Design with Applications*. Benjamin Pub.

Buttner, F., Radfelder, O., Lindow, A., & Gogolla, M. (2004). Digging into the Visitor Pattern. *In Proc. of 16th International Conference on Software Engineering & Knowledge Engineering*, Alberta, Canada.

Cameron, P., Fedotov, A., and Malitsky, N. (2002). Tune Measurement in the SNS Ring, *In Proc. of 8th European Particle Accelerator Conference*, Paris, France.

Carey, D.C. & Iselin, F.C. (1984). Standard Input Language for Particle Beam and Accelerator Computer Programs, *In Proc. of Snowmass*, Snowmass, USA.

Dean, J., & Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *In Proc. of 6th Symposium on Operating Systems Design and Implementation*, San Francisco, CA.

D'Imperio, N., Boine-Frankenheim, O., Luccio, A., and Malitsky, N. (2006). Parallel 3-D Space Charge Calculations in the Unified Accelerator Library. *In Proc. of European Particle Accelerator Conference*, Edinburg, UK.

Dirksen, J. (2013). *Learning Three.js: The JavaScript 3D Library for WebGL*. Packt Publishing.

DOE ASCAC (2013). Synergistic Challenges in Data-Intensive Science and Exascale Computing. Data Subcommittee Report

DOE ASCR (2013). Accelerating Scientific Knowledge Discovery. Working Group Report

Ekman, T. and Hedin, G. (2007). The JastAdd Extensible Java Compiler, In Proc. of International Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)

Fedotov, A., et al. (2000). Effect of Non-linearities on Beam Dynamics in the SNS Accumulator Ring, *In Proc. of 7th European Particle Accelerator Conference*, Vienna, Austria

Fedotov, A., et al. (2001). Excitation of Resonances due to the Space Charge and Magnet Errors in the SNS Ring, *In Proc. of Particle Accelerator Conference*, Chicago

Fedotov, A., Wei, J., and Gluckstern, R. (2001). Effect of Space Charge on Stability of Beam Distribution in the SNS Ring, *In Proc. of Particle Accelerator Conference*, Chicago

Fedotov, A., et al. (2002), Exploring Transverse Beam Stability in the SNS in the presence of Space Charge, *In Proc. of 8th European Particle Accelerator Conference,* Paris, France

Fisher, W., Pilat, F. and Ptitsin, V (1999). The Application of the SXF Lattice Description and the UAL Software Environment to the Analysis of the LHC, *In Proc. of Particle Accelerator Conference,* New York

Fine, V., Malitsky, N., and Talman, R. (2006). Interactive Analysis Environment of Unified Accelerator Libraries. *Nuclear Instruments and Methods in Physics Research*, A 559.

Forax, R., Duris, E., & Roussel, G. (2005). Reflection-based implementation of Java extensions: the double-dispatch use-case. *In ACM Symposium on Applied Computing*.

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*, Addison-Wesley.

Galambos, J., et al. (1999). ORBIT – A Ring Injection Code with Space Charge, *In Proc. of Particle Accelerator Conference*, New York

Gamma, E., Helm, R., Johnson, R., & Vlissides, J.(1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

Gagnon, E.M., & Hendren, L.J. (1998). SableCC, an object-oriented compiler framework. *In TOOLS USA 98 (Technology of Object-Oriented Languages and Systems)*, IEEE.

Ghemawat, S., Gobioff, H., & Leung, S.-T. (2003). The Google Filesystem. *In Proc. 19th ACM Symposium on Operating Systems Principles*. Lake George, NY.

Gross, L. & Yellen, J. (2005). *Graph, Theory and Its Applications* (2nd ed.), Chapman and Hall.

Grothoff, C. (2003). Walkabout revisited: The Runabout. *In Proc. of European Conference on Object-Oriented Programming*.

Gulliford, C., *et al*. (2010). The NTMAT EPICS-DDS Virtual Accelerator for the Cornell ERL Injector. *In Proc. of International Particle Accelerator Conference*, Kyoto, Japan.

Harrison, W., & Ossher, H.(1993). Subject-Oriented Programming (A Critique of Pure Objects). *In Proc. of the 8th Conf. Object-Oriented Programming Systems, Languages, and Application*.

The HDF Group. Hierarchical Data Format, version 5, 1997-2015: http://ww.hdfgroup.org

Hedin, G. & Magnusson, E. (2003). JastAdd: and aspect-oriented compiler construction system. *Science of Computer Programming*, *47(1): 37-58*.

Hedin, G. (2010). An Introductory Tutorial on JastAdd Attribute Grammars. *In Proc*. *of GTTSE*.

Heck, M. (2010). Open Inventor by VSG ~ http://oivdoc93.vsg3d.com/content/getting-started-guide

Hey, T., Tansley, S., & Tolle, K. (2009). The Forth Paradigm: Data-Intensive Scientific Discovery. *Microsoft Research*. Redmond, WA.

Iselin, F.C. (1996). The Classic Project. *In Proc*. *of 4th International Conference on Computational Accelerator Physics*. Williamsburg, USA.

ISO/IEC (2011). System and Software Quality Requirements and Evaluation. ISO/IEC 25010.

Jacobson, I. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley Professional.

Jones, F. (1997). Development of the ACCSIM Tracking and Simulation Code, *In Proc*. *of Particle Accelerator Conference,* Vancouver, Canada

Kanellopoulos, Y., et al. (2010). Code Quality Evaluation Methodology using the ISO/IEC 9126 Standard, *International Journal of Software Engineering & Applications (IJSEA)*, 1(3).

Kiczales, G., *et al*. (1997). Aspect-Oriented Programming. *In Proc*. *of ECOOP*, Springer-Verlag.

Koyama, T., Malitsky, N., and Talman, R. (1997), Beam-Beam Simulation Using the Unified Accelerator Libraries, *In Proc*. *of Particle Accelerator Conference, Vancouver, Canada*

Laddad R. (2003). *AspectJ in Action*, Munning Publications Co.

Lieberherr, K. (1996). *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing Company.

Lin, F., *et al*. (2009). Overview of (Some) Computational Approaches in Spin Studies. *In Proc*. *of 10th International Computational Accelerator Physics Conference*, San Francisco, CA.

Los Alamos Accelerator Control Group (1987). A Compendium of Computer Codes Used in Particle Accelerator Design and Analysis. *In Proc*. *of Summer School on High Energy Particle Accelerators*. AIP 184.

Low, Y., *et al*. (2010). GraphLab: A New Framework for Parallel Machine Learning. *In Proc*. *26th Conference on Uncertainty in Artificial Intelligence*. Catalina Island, CA.

Luccio, A. (1995). Spin Tracking in RHIC (Code Spink). Proceedings of the Adriatico Research Conference on Trends in Collider Spin Physics, Trieste, Italy

Luccio, A., D'Imperio, N., and Malitsky, N. (2006), Numerical Methods for Simulation of High-Intensity Hadron Synchrotrons, Nuclear Instruments and Methods in Physics Research, A 561: 216 - 222

Mai, Y., & Champlain, M. (2001). Reflective Visitor Pattern. *In Proc. of 6th Annual European Conference of Pattern Languages of Programs*.

Malewicz, G. et al. (2010). Pregel: A System for Large-Scale Graph Processing. *SIGMOD'10*, Indianapolis, IN

Malitsky, N. (1994). Application of a Differential Algebra Approach to a RHIC Helical Dipole, RHIC Project Note 006.

Malitsky, N., et al. (1995). A Proposed Flat Yet Hierarchical Accelerator Lattice Object Model, Particle Accelerators, 55 (2): 313-328

Malitsky, N. & Talman, R. (1996). Unified Accelerator Libraries. *In Proc. of 4th International Conference on Computational Accelerator Physics*. Williamsburg, USA.

Malitsky, N. and Pelaia, T. (1998). Integration of Unified Accelerator Libraries with CESR, CBN 98-8

Malitsky, N. and Talman, R. (1998). Study of the LHC Aperture Dependence on Tune Separation Using Thin Lenses, Phase Trombones, and Unified Accelerator Libraries, LHC Project Note 130

Malitsky, N. and Talman, R. (1998). Accelerator Description Exchange Format, *In Proc. of 5th International Conference on Computational Accelerator Physics*, Monterey, USA

Malitsky, N. & Talman, R. (1998). The Framework of Unified Accelerator Libraries. *In Proc. of 5th International Conference on Computational Accelerator Physics*. Monterey, USA

Malitsky, N. *et al*. (1999). UAL-Based Simulation Environment for Spallation Neutron Source Ring. *In Proc. of Particle Accelerator Conference*, New York, NY.

Malitsky, N. (2000). A Prototype of the SNS Optics Database, BNL Technical Note 085

Malitsky, N., et al. (2002), Development and Applications of the UAL-based SNS Ring Simulation Environment, *In Proc, of 20th ICFA Advanced Beam Dynamics Workshop on High Intensity and High Brightness Hadron Beams*, Batavia, USA

Malitsky, N., Fedotov, A.V., & Wei, J. (2002). Application of UAL to High-Intensity Beam Dynamics Studies in the SNS Accumulator Ring. *In Proc. of 8th European Particle Accelerator Conference*. Paris, France.

Malitsky, N. *et al*. (2003). Configurable UAL-Based Modeling Engine for Online Accelerator Studies. *In Proc. of Particle Accelerator Conference*, Portland, OR.

Malitsky, N., et al. (2004), MADX-UAL Suite for Off-Line Accelerator Design and Simulation, *In Proc. of 9th European Particle Accelerator Conference*, Lucerne, Switzerland

Malitsky, N. & Talman, R. (2005). Accelerator Simulation Using the Unified Accelerator Libraries. *U.S. Particle Accelerator School*, Cornell University, NY.

Malitsky, N. et al. (2005). Joining the RHIC Online and Off-Line Models. *In Proc. of Particle Accelerator Conference*, Knoxville, TN.

Malitsky, N. & Talman, R. (2006). Accelerator Description Formats, *In Proc. of 9th International Conference on Computational Accelerator Physics*, Chamonix Mont-Blanc, France.

Malitsky, N. (2008). Processing Heterogeneous Abstract Syntax Trees with the Mutable Class Pattern. *In Proc. of 23rd International Conference on Object-Oriented Programming, System, Languages, and Applications*. Nashville, US.

Malitsky, N., Shah, J., and Hasabnis, N. (2009). EPICS-DDS, In Proc. of Particle Accelerator Conference, Vancouver, Canada

Malitsky, N., et al. (2009). Prototype of a DDS-Based High-Level Accelerator Application Environment, In Proc. of International Conference on Accelerator & Large Experimental Control Systems, Kobe, Japan

Malitsky, N. et al. (2010). EPICS-DDS: Rationale, Status and Applications. *In Proc. of Control, Diagnostics, and Automation*, Novosibirsk, Russia.

Malitsky, N. et al. (2010). Application of Model Independent Analysis with EPICS-DDS. *In Proc. of International Particle Accelerator Conference*, Kyoto, Japan.

Malitsky, N. (2012). Bringing Large-Scale Analytics to Accelerators. *In Proc. of 11th International Conference on Computational Accelerator Physics*, Warnemunde, Germany.

Malitsky, N. et al. (2015). Precise High-Performance Simulator for EDM Experiments, DOE ASCR-HEP Proposal

Mariscal, G., Marban, O., and Fernandez, C. (2010). A Survey of Data Mining and Knowledge Discovery Process Models and Methodologies, The Knowledge Enfinerring Review, Vol 25:2

Martin, R. (1996). The Dependency Inversion Principle, *C++ Report*.

Martin, R., Riehle, D., & Buschmann F.(1997). *Pattern Language of Program Design 3*. Addison-Wesley.

Martin, R. (1998). The Interface Segregation Principle, *C++ Report*.

Martz, P. (2007). *OpenSceneGraph Quick Start Guide*, Computer Graphics System Development Corporation, CA.

Melnik, S. et al. (2010). Dremel: Interactive Analysis of Web-Scale Datasets. *In Proc. 36th International Conference on Very Large Data Bases*, Singapore

Oliviera, B. (2007). *Generosity, Extensibility and Type-safety in the Visitor Pattern*, PhD Thesis, University of Oxford.

OMG (2005), Data Acquisition from Industrial Systems Specification, *formal/05-06-01*

OMG (2011). *Unified Modeling Language, Infrastructure*, V2.4.1, formal 2011-08-05, OMG

OMG (2015), Data Distribution Services, V 1.4, *formal/15-04-10*

Ossher, H., & Tarr, P.(1999). Multi-Dimensional Separation of Concerns using Hyperspaces. *Technical Report 21452*, IBM Research Report.

Page, L., Brin, S., Motwanl, R., and Winograd, T.(1998). The PageRank Citation Ranking: Bringing Order to the Web. *Technical report*. Stanford University, Stanford, CA

Papaphilippou, I. (2001), SNS Ring Optics Tuning, *In Proc. of Particle Accelerator Conference*, Chicago

Parr, T. & Quong, R.W. (1995). ANTLR: A Predicated-LL(k) Parser Generator. *Software – Practice and Experience, Vol. 25 (7), 789-810.*

Parr, T. (2013). *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

Palsberg, J., & Jay, C. B. (1998). The Essence of the Visitor Pattern. *In Proc. of 22nd Annual International Computer Software and Applications Conference*. Washington, DC, USA.

Pati, T., & Hill, J. (2010). *A Survey Report of Enhancements to the Visitor Software Design Pattern, Software – Practice and Experience*. Wiley & Sons.

Pazandak, P. (2015). Integrated Platform for Data-Intensive Science. DOE SBIR proposal, Real-Time Innovations (submitted)

Pilat, F., et al. (1997), Modeling RHIC Using the Standard Machine Format Accelerator Description, *In Proc. of Particle Accelerator Conference*, Vancouver, Canada

Pilat, F., et al. (1998), The Standard eXchange Format (SXF) for Accelerator Description, *In Proc. of 5th International Conference on Computational Accelerator Physics*, Monterey, USA

Pilat, F., et al. (1999), Processing and Analysis of the Measured Alignment Errors for RHIC, *In Proc. of Particle Accelerator Conference*, New York

Rumbaugh, J., Blaha, M., Lorensen, W., & Eddy, F. (1990). *Object-Oriented Modeling and Design*. Prentice-Hall

Sakr, S. (2013). Processing Large-Scale Graph Data: A Guide to Current Technology. *developerWorks*, IBM.

Schachinger, L. and Talman, R. (1987). TEAPOT: A Thin Element Program for Optics and Tracking, Particle Accelerators, 22 (35)

Schauerhuber, A., et al. (2006). Towards a Common Reference Architecture for Aspect-Oriented Modeling. *In Proc. of 8th International Workshop on Aspect-Oriented Modeling*.Germany.

Schmidt, D., Stal, M., Rohnert, H., & Buschmann (2000), *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2*. Wiley and Sons.

Shreiner, D., et al. (2013). *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3* (8th ed.). Pearson Education, Inc.

Sun, Y. & Han, J.(2012). *Mining Heterogeneous Information Networks*, Morgan & Claypool Publishres.

Valiant, L. (1990). A Bridging Model for Parallel Computation. *Comm. ACM 33(8),* 103-111.

Visser, J. (2001). Visitor Combination and Traversal Control, *In Proc. of 16th ACM SIGPLAN Conference on OOPSLA*. NY, USA.

Walls, C. (2013). *Spring in Action* (4th ed.). Manning Publications Co.

Wang, R., & Qian, X. (2012). *OpenSceneGraph 3 Cookbook*. Packt Publishing.

Wernecke, J., et al. (1994). *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor* (2nd ed.). Addison-Wesley.

Wernecke, J., et al. (1994). *The Inventor Toolmaker: Extending Open Inventor* (2nd ed.). Addison-Wesley.

Wimmer, M. et al. (2011). A Survey on UML-Based Aspect-Oriented Design Modeling. *ACM Computing Surveys*, Vol. 43, No. 4, Article 28.

Wu, X. et al. (2005). A Two-Dimensional Separation of Concerns for Compiler Construction. *In Proc. of the 2005 ACM Symposium of Applied Computing*.

Wu, X. et al. (2006), Separation of Concerns in Compiler Development Using Aspect-Orientation. *In Proc. of the 2006 ACM Symposium of Applied Computing*.

Xin, R. et al (2013). GraphX: A Resilient Distributed Graph System on Spark, Proc of the First International Workshop on Graph Data Management Experience and Systems, New York

Zaharia, M. (2013), An Architecture for Fast and General Data Processing on Large Clusters, PhD thesis, Berkley