Application and Extension of Design Patterns in the Technology Enabled Active Learning Project

by

Ying Cao

M.S. Civil
Tsinghua University,1998

SUBMITTED TO THE DEPARTMENT OF CIVIL AND ENVIRONMENTAL ENGINEERING IN PARTIAL FULFILLMENT OF THE REQURIEMENTS FOR THE DEGREE OF

MASTER OF SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUNE 2003

Author _____
Department of Civil and Environmental Engineering
May 9, 2003

Certified by _____
Steven R. Lerman
Professor
Department of Civil and Environmental Engineering
Thesis Supervisor

Accepted by _____
Oral Buyukozturk
Chairman, Department Committee on Graduate Studies

Application and Extension of Design Patterns in Technology Enabled Active Learning Project

by

Ying Cao

Submitted to the Department of Civil and Environmental Engineering
on May 9, 2003, in Partial fulfillment of the
Requirements for the degree of
Master of Science

Abstract:

This thesis describes the application and extension of design patterns in Technology Enabled Active Learning (TEAL) project. The aim of TEAL is to help students developing better intuition about conceptual models of physical phenomena and understanding the physical concepts of these phenomena better. Computer simulation is one of the most important parts of TEAL.

Design patterns are reusable solutions to recurring problems that occur during software development. The purpose of design patterns is to systematically name, explain and evaluate an important and recurring design experience in Object-Oriented systems so that developers can use it effectively. Design patterns play an important role in the design and refine process of TEAL simulations. We used several design patterns, such as Factory Method, Adapter, Iterator and Template in TEAL. Some experiences derived from TEAL are also summarized as design patterns in this thesis.

Thesis Supervisor: Steven R. Lerman
Title: Professor of Civil and Environmental Engineering

Acknowledgements

I would like to thank Professor Steven R. Lerman for his great help and valuable advice, which made this thesis possible. I'd also like to thank Professor John W. Belcher for giving me a chance to study and work in TEAL project group. I'd like to thank them also for the tremendous support they gave me when I was in the difficult time with visa problem.

I'd like to thank Andrew Micknney, Philip Bailey and Pierre for the tremendous help and patience they gave me in the last two years.

Thanks to all the RAs and staffs working in CECI for the great help, support and friendship they gave me.

Thanks to my husband, his endless love help me walked through the most difficult time in my life. Thanks for my parents' love and encouragement.

And I'd also like to express my thanks to all the friends at MIT. Their friendship is the most valuable experience I got here.

Context

Chapter 1: Introduction 6
1.1 Objective 6
1.2 Motivation 6
1.3 TEAL (http://caes.mit.edu/research/teal) 9
1.3.2 Simulations 10
1.3.3 Design Patterns 11
1.4 Area Of Research 11
1.5 Thesis Roadmap 12

Chapter 2: Design Patterns Overview 13
2.1 Introduction 13
2.1.1 What are design patterns? 13
2.1.2 Why design patterns? 14
2.2 Brief History of Design Patterns 16
2.3 The Catalog of design patterns 17
2.3.1 Creational Patterns: 18
2.3.2 Structural Patterns 20
2.3.3 Behavioral Patterns 21
2.4 Application of design patterns in Java 23

Chapter 3: Application of Existing Design Patterns 26
3.1 Introduction 26
3.1.1 Overview 26
3.1.2 Brief description of UML 26
3.2 Application 1: Factory Method Pattern 29
3.3 Application 2: Adapter Pattern 34
3.4 Application 3: Iterator Pattern 37
3.5 Application 4: Template Pattern 41
3.6 Summary: 45

Chapter 4: Extensions of Existing Design Patterns 47
4.1 Introduction: 47
4.2 Property Route Pattern 47
4.3 GRASP Patterns 56
4.3.1 Concept of GRASP patterns 56
4.3.2 Low Coupling 56
4.3.3 Expert 59
4.3.4 Creator 61
4.3.5 Polymorphism 62
4.3.6 Pure Fabrication 64
4.3.7 Controller 66
4.3.8 A refined TEAL framework 67
4.4 Summary 71

Chapter 5: Summary 72
5.1 Conclusion 72
5.2 Future Work 72

References 74

Chapter 1: Introduction

1.1 Objective

The work presented in this thesis is part of the Technology Enabled Active Learning (TEAL) project at CECI center of MIT. The objective of TEAL project is to reform the way physics is being taught in undergraduate classes. As an important part of a studio-style class, a large number of computer simulations are required to simulate the physical phenomena, which are not easily understood by student without the aid of tools that make it easier for students to visualize abstract ideas such as electromagnetic field. The research of TEAL project focuses on developing a fairly extensible and comprehensive toolset, which make the generation of new simulations quite easy. Design pattern are widely used in TEAL to make the framework of TEAL more understandable, portable and reusable. The focus of this thesis will be on the design patterns that are used in the design process of TEAL as well as summarizing the experience derived from TEAL in form of design patterns.

1.2 Motivation

Physics is a fundamental element of any technical education. It's a required course for most undergraduate students of science and engineering schools. However, physics is a difficult subject to master. It's a subject in which mathematical complexity can quickly overwhelm physical intuition. It's also a subject each student learns at a different pace. Often some students fall back in the class and gradually lose interest in the class after a

period of study. For this reason, it is very important to use innovative teaching methods to increase students' interest and help students developing better physical intuition. However, recent research shows that the traditional methods of teaching physics are inadequate. For example, students usually learn physics by associating what they learn in class with real life experiences. This works very well for Newtonian physics where the concepts taught in class can be directly correlated with everyday experiences. However, it gets complicated for advanced topics like electromagnetism where the students find it difficult to comprehend and visualize what is being taught in the class.

Realizing the drawbacks and limitations in the traditional methods of teaching physics, many universities and organizations are actively involved in physics education research. Through many research and experiments, researchers found physics is best learnt when the students are active rather than passive in the class. [Michael, 1998], [Jeffery, 1998] Any tool that can help the student interactively visualize the physical principles and actively engaged in the class is of great value. Based on this idea, many new methods of physics education, such as Desktop Experiment, Conceptual Questions, Peer Instruction, Studio Physics, have been invented and tested. They have had got varying degrees of success[PERG].

Among those methods, the Studio Physics approach is the most important and comprehensive shift from the traditional methods in the way of physical education. The main differences between studio class and the traditional lecture format are the desktop experiments and computer simulation of physical phenomenon, active group discussions,

and question-answer sessions. Desktop experiments are an ideal way to provide the students practical experience, which leads to thorough understanding of physical theories taught in class. However, since there are many practical limitations, other methods are required as to supplement of desktop experiments. With the development of technology in computer hardware and software it becomes possible to create high quality simulations. These computer simulations can simulate a wide variety of physical phenomena ranging from the very simple to the very complex. They can be easily integrated with other course materials. They also offer the advantages of conveniently changing, adding and reconfiguring the content of simulations without adding any cost.

The Studio Physics approach has been deployed in some universities with very impressive results. Examples are given as follows:

- The Teal Project at MIT (http://www-caes.mit.edu/research/teal/)
- Sophomore Studio Physics at Ohio State University (http://www.physics.ohio-state.edu/~physedu/projects/index.html)
- Studio Physics with a Notebook Computer at Acadia University (http://ace.acadiau.ca/science/phys/homepage_Jun07_2002/Information/studio.html)
- Sherman Visual Lab (http://www.shermanlab.com/science/physics/index.php)

These prototypes of Studio Physics may be different from each other in the content of their studio classes. However they all have a common feature that engaging the students to be active participants in the class. The TEAL project is one of the most important implementations of Studio Physics approach.

1.3 TEAL (http://caes.mit.edu/research/teal)

1.3.1 Introduction

The TEAL project at MIT is based on the Studio Physics approach to physics education. The educational component of the project is funded by the d'Arbeloff Initivative and the MIT School of Science. The software development for this effort is funded by the MIT/Microsoft I-Campus Alliance. The project is sponsored by the Department of Physics and administered through the MIT Center for Educational Computing Initiatives.

The aim of the TEAL project is to help students develop better intuition about conceptual models of physical phenomena and understand the physical concepts of these phenomena better. The basic idea of TEAL is "active learning"- that is making the student an active rather than passive participant in a class. This approach uses a highly collaborative, hands-on environment, with extensive use of educational technology to help the student interactively visualize the physical principles and actively engage in the class.

The basic plan of the TEAL project is to merge lecture, recitations, and hands-on laboratory experience into a technologically and collaboratively rich experience for incoming freshmen. Students gather in groups of nine, with twelve or so such groups in a common area, for five hours per week. The students are exposed to a mixture of instruction, laboratory work with desktop experiments, and collaborative work in smaller groups of three, in a computer rich environment. The desktop experiment data give the students direct experience with the basic phenomena. Formal and informal instruction,

aided by media-rich interactive software for simulation and visualization, then aids students in their conceptualization of this experience.

1.3.2 Simulations

As mentioned before, desktop experiments are ideal way to provide the students practical experience. However, since there are many practical limitations to desktop experiments, computer simulations play an important role in the way of helping students thoroughly understand the physical concepts. In the following parts of this thesis, except when specifically stated, the terms TEAL or TEAL project refer to the simulation part of TEAL project.

The design of these simulations should not only simulate the physical phenomena relevant to the curriculum, but should also make it to conveniently add, modify and create new simulations. To achieve this aim, a software framework was developed, on which all the simulations could be implemented. In the framework, the whole simulation process is decentralized and abstracted into a set of abstract classes and interfaces. According to their different functionalities, these classes and interfaces are further grouped into modules. Each module can perform some required task. Different modules communicate with each other through well-defined interfaces.

To make it more portable, extensible and interactive, the framework was revised several times. The first prototype implementation was in Fall 2001. Based on the results of the first use in Fall 2001, the framework was revised in Spring and Summer of 2002 and used

again in teaching during Fall 2002. Based on this second trial, the simulations were revised again and used in Spring 2003, with 700 students, about seven full time Physics faculty, seven teaching assistants, and seven undergraduate aids. The current work is the fourth version of TEAL simulations.

1.3.3 Design Patterns

Design patterns play an important role in the design and refinement process of the TEAL framework. Design patterns are reusable solutions to recurring problems that occur during software development. The purpose of design patterns is to systematically name, explain and evaluate an important and recurring design experience in Object-Oriented systems so that developers can use these experiences effectively. The detail of design patterns will be described in Chapter 2. Many design patterns such as Factory Method, Adapter, Iterator and Template are widely used not only in the GUI development, but also in the model and viewer parts of TEAL. Since experience is so important for software developer, using design patterns to accumulate and codify experience derived from TEAL framework design and refinement will also be beneficial for future work.

1.4 Area Of Research

The aim of the research is to use design patterns in the development of refinement of the TEAL framework, which makes the software more portable, extensible and fairly comprehensive. The experience derived from TEAL will also be described in the form of

design patterns. The focus of this research will be on how design patterns summarize experience and make the software work in better way.

1.5 Thesis Roadmap

The introduction of design patterns will be given in Chapter 2. Chapter 3 will describe the application of design patterns in TEAL. Samples of Factory Method, Adapter, Iterator and Template will be given in this chapter. In Chapter 4, the extension of existing design patterns in TEAL will be introduced. A new design pattern, named Property Route pattern will be described. The GRASP Patterns, which are used to restructure software and lead to design patterns will also be explained in Chapter 4. Chapter 5 summarizes the work done and the outline for the future work.

Chapter 2: Design Patterns Overview

2.1 Introduction

2.1.1 What are design patterns?

The first recognized definition of design patterns appeared in *Design Patterns, Abstraction and Reuse of Object Oriented-Design* [Gamma, 1993]. In that paper, E. Gamma stated, "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." After that, in *Design Pattern, Elements of Reusable Object-Oriented Software* [Gamma, 1995] E. Gamma gave a further definition, "The design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." With the development of research in this area, some other useful definitions of design patterns appeared.

- "Design patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development" [Pree, 1994];
- "Design patterns are recurring solutions to design problems we see over and over" [Alpert, 1998];
- "Design patterns are reusable solutions to recurring problems that occur during software development" [Grand, 2002].

In a short word, design patterns describe the reusable aspects of an architecture during the process of software development. In general, a pattern has four essential elements [Gamma, 1995]:

- The pattern name: We can use a pattern's name to describe a design problem, its solutions, and consequences. Naming a pattern also makes it convenient for us to communicate with other software developers. Finding a good name is difficult, but worthwhile when we developing a new catalog.
- The problem: It describes the circumstances where the pattern is applied. It explains the problem's context or conditions that must be met before the application of the pattern.
- The solution: The solution provides an abstract description of design problem and general elements arrangement instead of detail description of particular design. The solution describes the make up of design elements, their relationships, responsibilities, and collaborations.
- The consequences: The consequences describe the costs and benefits of applying the pattern. It often includes the space and time trade-offs, language and implementation issues, and its impact on a system's flexibility, extensibility, or portability.

2.1.2 Why design patterns?

What makes the productivity of experienced object-oriented designers differ from new programmers is experience, reusing solutions that have worked for them in the past. We all know the value of design experience. However, most of us don't do a good job of recording experience in software design. As stated earlier, the purpose of design patterns is to systematically name, explain and evaluate an important and recurring design experience in Object-Oriented systems so that people can use it effectively.

In general, design patterns can help software designers in the following aspects:

- Design patterns provide solutions to common problems, which can make a software developer a better designer.
- Design patterns help us in choosing alternative designs to make a system more reusable.
- Design patterns make proven techniques more accessible to other developers.
- Design patterns can increase the speed of generating new frameworks, improving the structure of developing software.
- Design patterns can simplify documentation and maintenance of existing systems.
- Design patterns create a common design language, which improves communication among software developers.
- Design patterns empower less experienced personnel to produce high-quality designs.
- Design patterns explaining existing frameworks better. Design patterns make a system seem less complex by letting us talk about it at a higher level of abstraction. Describing a system in terms of the design patterns will make the system easier to understand.

Put simply, design patterns help a designer get a "right" design faster. The design patterns show how to use primitive techniques such as objects, inheritance, and polymorphism. They not only provide a way to record the results of designers' decisions but also describe the reasons for a specific design.

2.2 Brief History of Design Patterns

The idea of software design patterns originally came from the field of architecture. An architect named Christopher Alexander wrote two revolutionary books that describe patterns in building architecture and urban planning: *A Pattern Language: Town, Buildings, Construction* [AIS+, 1977] and *The Timeless Way of Building*. The ideas presented in these books are applicable to a number of fields outside of architecture, including software. The definition of software design patterns is largely based on C. Alexander's definition, "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that we can use this solution a million times over, without ever doing it the same way twice." [AIS+, 1977]

In 1987, Ward Cunningham and Kent Beck used some of Alexander's ideas to develop five patterns for user-interface design. They published a paper on the UI patterns at OOPSLA-87: *Using Pattern Languages for Object-Oriented Programs.* [Beck-Cunningham, 1987]

Model-View-Controller (MVC) for Smalltalk [Krasner and Pope, 1988] is the most common pattern cited in early literature on programming frameworks, in which user interface problem is divided into three parts:

- Data Model: includes the computational components of the program
- View: user interface "view" the data model
- Controller: interacts between the user and the view

More formal recognition of design patterns began in the early 1990s when Erich Gamma [Gamma, 1993] described patterns incorporated in the GUI application framework. Programmers began to meet and discuss these ideas. These discussions and meetings evolved into the essential part of *Design Patterns-Elements of Reusable Software* [Gamma, 1995]. This now classic book has had a powerful impact on those seeking to understand how to use design patterns. It describes 23 common, generally useful patterns and comments on how and when they can be applied.

Since the publication of *Design Patterns*, several other useful books have been published. One closely related book is *The Design Patterns Smalltalk Companion* [Alpert, 1998], which covers the same 23 patterns but from the view of Smalltalk. The other similar publication is *Java Design Patterns, A Tutorial* [Cooper, 2000]. It also describes the same patterns but from the Java point of view.

2.3 The Catalog of design patterns

In *Design Patterns* [Gamma, 1995], software design patterns are classified in two dimensions. The first dimension is purpose. In this dimension patterns are separated into three groups: creational patterns, structural patterns and behavior patterns. The second dimension is scope. It distinguishes whether a pattern applies primarily to classes or to objects. Since whether a pattern has one or the other property is difficult to determine when the pattern itself is unknown, the distinction based on dimension and scope may meet some problems when they are used to search for patterns.

In *A System of Patterns* [Buschmann 96], patterns are separated into architectural patterns, design patterns, and idioms. Architectural patterns provide a top-level structural division of software, while design patterns refine components at a medium level, and idioms are low-level. The problem with this division is that it is difficult to place patterns consistently. A pattern may be a top-level in one system while middle-level in the other system.

In *A Catalogue of General-Purpose Software Design Patterns* [Tichy, 1998], the classification scheme concentrates on the problems solved by patterns. The top-level categories based on this scheme are: Decoupling, Variant Management, State Handling, Control, Virtual Machines, Convenience Patterns, Concurrency, and Distribution. This is a new attempt to organize design patterns. It still needs feedback of usage before it can evolve into a catalog of patterns.

In the thesis, we'll discuss patterns mainly following Gamma's separation, from the view of the internal purpose of a pattern.

2.3.1 Creational Patterns:

When we design software, we often find that the creation of objects requires making decisions dynamically. These decisions will typically involve dynamically deciding which class to instantiate or which objects some function will be delegated to. Creational patterns help us in following aspects on how to create objects in such circumstances:

- How to make decisions dynamically;
- How to structure and encapsulate these decisions;
- How to abstract the instantiation process;
- How to make a system independent of how its objects are created, composed and represented.

All of the Creational patterns deal with ways to create instances of objects. Since the system only knows about the objects' interfaces defined by abstract classes, the creational patterns give us a lot of flexibility in what gets created, who creates it, how it gets created, and when to create it. This is important because our program should not depend on how objects are created and arranged. The creation and arrangement of objects should be able to vary with the needs of the program. Abstracting the creation process into a special "creator" class can make our program more flexible and general. To achieve this aim, the Creational design patterns should be able to encapsulate knowledge about which concrete classes the system uses as well as hide how instances of these classes are created and put together. A class creational pattern uses inheritance to vary the class that's instantiated, whereas an object creational pattern will use delegation to instantiate another object.

Some typical Creational patterns include [Gamma, 1995]:

- Factory Method pattern: provides a common interface for creating families of objects without specifying their concrete classes. Through a simple decision-making class, we can select one of several possible subclasses of an abstract base class.

- Abstract Factory pattern: provides an interface to create and return one of several families of related objects.
- Builder pattern: separates the construction of a complex object from its representation. The client object can construct a complex object by only specifying its type and content.
- Prototype pattern: allows an object to create customized objects without knowing the details of how to create them. It starts with an instantiated class, which is copied or cloned to make new instances.
- Singleton pattern: ensues that there may be no more than one instance of a class is created. It provides a single global point of access to that instance.

2.3.2 Structural Patterns

Structural patterns describe common ways that different types of objects can be organized to work with each other to form larger structures. They are particularly useful for making independently developed class libraries work together.

The structural class patterns describe how inheritance can be used to provide more useful program interfaces. The derived class can combine the properties of its parent classes, which is very useful to make independently developed classes work together. The structural object patterns help on how to compose objects to realize new functionality. Their ability to change the composition at run-time adds the flexibility of object composition.

There are some typical Structural patterns [Gamma, 1995]:

- Adapter pattern: implements an interface known to its clients and converts it into another interface unknown to its client for easier programming.
- Bridge pattern: separates an object's interface from its implementation so the two parts can be varied separately.
- Composite pattern: lets the simple and composite objects be treated uniformly.
- Decorator pattern: makes it easy to add responsibilities to objects dynamically.
- Flyweight pattern: allows each object share the storage of their states externally.
- Façade pattern: used to make a single class represent an entire subsystem.
- Proxy pattern: creates a simple object that takes the place of a more complex object which may be invoked later.

2.3.3 Behavioral Patterns

Behavioral patterns are most specifically concerned with the interconnection and communication between objects. Behavioral patterns describe not only patterns of objects or classes but also the organization, management and combination of behavior among them. Behavioral class patterns use inheritance to distribute behavior among classes. Behavior object patterns use composition to cooperate a group of objects to perform a task that no single objects can carry out by itself.

There are some typical Behavior patterns [Gamma, 1995]:

- Chain of Responsibility pattern: passes a request from one object to the next in a chain until the request is recognized. It can avoid the need for a direct coupling between sender and receiver.
- Command pattern: encapsulates the execution of commands in simple objects, which makes control and manipulation of those commands easier.
- Interpreter pattern: deals with how to define and interpret language elements in a program.
- Iterator pattern: provides a standard interface to move through a list of data without knowing that data's internal details.
- Mediator pattern: encapsulates objects and communication between objects in a separate object to keep objects from tightly coupling with each other.
- Memento: provides a way to save and restore the state of objects without exposing much of their internal state.
- Observer pattern: defines a one-to-many dependency among objects so that when one object changes state, all its dependents are notified and updated automatically.
- State pattern: allows an object to modify its behavior when its internal state changes.
- Strategy pattern: encapsulates an algorithm inside of a class, allows the algorithm vary independently from the client using it.
- Template Method pattern: provides an abstract definition of an algorithm, while leaving some of the implementation in subclasses.

- Visitor pattern: creates an external class to act on data in other classes. It can add polymorphic function to that class.[1]

Often, there is more than one pattern that can be applied to a situation. Sometimes multiple patterns can be combined together to achieve a better result. In other cases, the patterns may be competing as well. The design patterns may also generate similar results although they have different intents. It's important to be familiar with the existing patterns if we want to have an appropriate application of these patterns in software design.

2.4 Application of design patterns in Java

Since Java is one of the most important Object-Oriented languages, how to write Java programs using the most common design patterns is an important issue in design patterns. Actually, all the examples in *Design Patterns* [Gamma, 1995] are implemented with Smalltalk or C++. After the publishing of this book, more and more programmers and researchers began to pay attention to Java design patterns.

In *Java Design Patterns, A Tutorial* [Cooper, 2000], James W. Cooper summarized the usage of 23 design patterns in Java. In this book, where to use design patterns, how to use design patterns, and the consequence of using design patterns in Java are well described. A large number of Java examples are given as well. GUI is the most common usage of

[1] Polymorphic function here means a function can be implemented differently in different classes based on the needs of that specific class

design patterns in Java. The Swing classes utilize a number of the basic design patterns and make extremely good examples. For instance, the relationship between menu and action is a good example of the Command pattern; the separation of the data from the viewer in the JTable class is a typical Observer pattern.

In a broad sense, a number of adapters are already built into the Java language. In this case, the Java adapters serve to simplify an unnecessarily complicated event interface. One of the most commonly used of these Java adapters is the WindowAdapter class. One inconvenience of Java is that windows do not close automatically when you click on the Close button or window Exit menu item. The general solution to this problem is to have your main Frame window implement the WindowListener interface and leave all of the Window events empty except for windowClosing. But it looks awkward. We can use Window Adapter class instead. It has the default implementations of all seven of the previous WindowEvents. You then need only an override the handler for a windowClosing event and insert the appropriate exit code.

In recent years, besides programming GUI elements, design patterns are used in more and more area, such as: transaction design, distributed computing, concurrency, and databases. In *Patterns In Java* [Grand, 1998], the design pattern catalog is greatly enlarged. Besides the Creational, Structural and Behavioral patterns, Grand also summarized Partitioning Patterns, Concurrency Patterns, GRASP Patterns, Organizational Coding Patterns, Code Optimization Patterns, Robustness Coding Patterns, Testing Patterns and so on.

With the development of the Internet, the information propagating mechanism becomes more and more important. In *Propagator--A Family of Patterns* [Feiler, 1998], Feiler describes a new catalog of design patterns, Propagator Patterns. They are a family of patterns for consistently updating objects in a dependency network. The propagator patterns can be found in such applications as Make, spreadsheets, GUIs, reactive control systems, simulation systems, distributed file systems, distributed databases, workflow systems, and multilevel caches. Propagator Patterns can support smart propagation for avoiding redundant work as well as concurrent updates.

In short, the design patterns have many applications and have their use evolved in Java programs. We also use Java design patterns to improve the design and programming in the Technology Enabled Active Learning (TEAL) project, and refine some design patterns as well.

Chapter 3: Application of Existing Design Patterns

3.1 Introduction

3.1.1 Overview

TEAL requires a large number of computer simulations. We designed and refined the framework of TEAL several times to achieve the following requirements:

- The creation of new simulations becomes simpler.
- The maintenance of existing simulations becomes easier.
- Any improvements made to the framework are automatically reflected in all the simulations.
- The components should be reusable and portable.
- The physical concepts should be easily integrated into simulations.
- The simulations should be easy to use and understand.

Since design patterns describe reusable methods to solve general software development problems, they are very useful during the design and refinement process of TEAL.
Some typical applications of design patterns will be given in this chapter. The extension of existing patterns will be described in chapter 4.

3.1.2 Brief description of UML

We use Unified Modeling Language (UML) for software analysis and design. UML defines a number of different kinds of diagrams. The kind of diagram used in TEAL is

the class diagram. A class diagram is a diagram that shows classes, interfaces, and their relationships.

The most basic element of a class diagram is a class. Classes are drawn as rectangles. The rectangles can be divided into three parts, which list the name of the class, the class's variables and the class's methods respectively. Interfaces are drawn in a similar way. The only difference is that the name in the top compartment is preceded by an <<interface>> stereotype. The name of an abstract class is italicized. A basic class is shown in Figure 3.1:

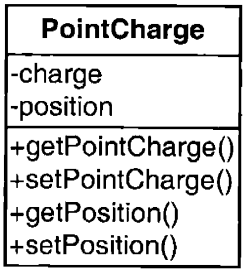

Figure 3.1 Basic Class

A solid line with a closed arrowhead like the one in Figure 3.2 indicates the relationship with a subclass that inherits from a superclass. A similar sort of line shown in Figure 3.3 is used to indicate that a class implements an interface. It is represented with a dotted or dashed line with a closed head.

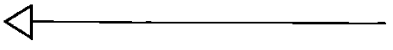

Figure 3.2 Subclass inherits from superclass

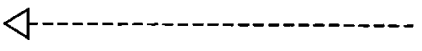

Figure 3.3 Class implements an interfce

The type of one-to-many relationship in which one object contains a collection of other objects is called an aggregation. A hollow diamond at the end of an association indicates aggregation. UML has another notation that indicates a stronger relationship than aggregation. This relationship is called composite aggregation. For an aggregation to be composite, the aggregated instance must belong to only one composite at a time. Some operations must propagate from the composite to its aggregated instances. The aggregation and composite aggregation are shown in Figure 3.4 and Figure 3.5 respectively:

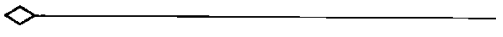

Figure 3.4: Aggregation

Figure 3.5: Composite aggregation

The association is a type of relationships between the classes and the interface. The association exists if a class requires an attribute with a data type of another class definition. There are a number of things that can appear with an association that provide information about the nature of an association. The following items are optional:

- Association Name: Somewhere around the middle of an association there may be an association name. The name of an association is always capitalized.
- Navigation Arrows: Arrowheads that appear at the ends of an association are called navigation arrows. Navigation arrows indicate the direction in which we can navigate an association. Looking at the association named creates shown in figure 3.6, we can see that it has a navigation arrow pointing from class A to class B. that

means class A will create a reference that allows it to access the instance of class B, but not the reverse way.

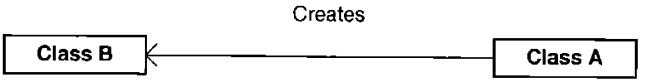

Figure 3.6: Association

The dependency relationship shows that a change to one element's definition may cause changes to the other. A dependency is shown in Figure 3.7:

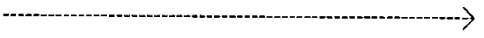

Figure 3.7: Dependency

The classes in a class diagram can be organized into packages. A package is drawn as a large rectangle with a small rectangle above it. A comment in UML is drawn as a rectangle with its upper right corner turned down.

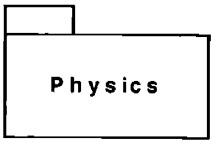

Figure 3.8: Package

3.2 Application 1: Factory Method Pattern

Intent:

The Factory Method defines an abstract class or interface that creates objects, but lets subclasses decide which object to create. In the Factory Method pattern there is no single class that makes the decision which subclass to instantiate. Instead, the superclass defers the decision to its subclasses.

Motivation:

During the software developing process, we may meet some sophisticated circumstances:

- A class wants to leave some classes to be instantiated in its subclass
- A class doesn't know what class of objects it must create
- The knowledge of which objects gets created should be localized

In order to achieve these aims, we want to create some reusable classes in our system. In these classes, the instantiation of some classes is not dependent on any instantiation of other classes. The superclass is able to defer the choice of which class to instantiate to its subclass and refer to the instantiated class through a common interface.

In TEAL, there are several applets and applications. Each applet or application may require different looks. To meet this requirement, several user interfaces should be created. Each interface includes a set of GUI components, the arrangement of these components, and different responses to user input. To achieve this aim, the creation of user interfaces must be separated from the creation of applications. All the user interface classes should be derived from the same abstract class or have the same interface, which makes the use of this class independent of the details of this class. Making it easy to use different GUIs in the same system is the reason why we use the Factory Method pattern in TEAL.

Solution:

The Factory Method pattern provides an application-independent object with an application-specific object to which it can delegate the creation of other application-specific objects. There are four essential parts in the Factory Method pattern: Product, ConcreteProduct, Creator and ConcreteCreator. The role of Product is to define the abstract superclass or interface of objects the factory method creates. The ConcreteProduct implements the Product interface and is the concrete class that is actually instantiated. The Creator calls the factory method to create a Product object. The ConcreteCreator overrides the factory method and returns an instance of a ConcreteProduct. Figure 3.9 shows a class diagram with the interfaces and classes that typically make up the Factory Method pattern. It also shows the roles that classes and interfaces play in the Factory method pattern.

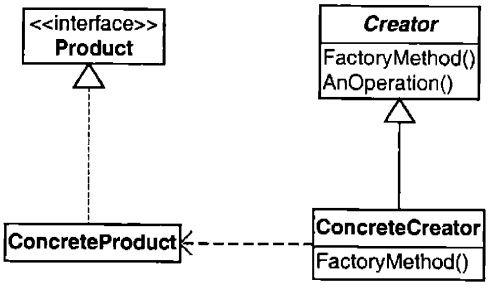

Figure 3.9 Class Diagram of Factory Method

Implementation:

In TEAL, the Factory Method pattern is used to create different styles of user interfaces. The roles of the interface/classes are introduced as following:

- Product (TealGUI): defines the common interface of different GUIs. The key methods in TealGUI are addProObject and remove. They are used to add and remove

GUI components to and from the user interface. The definition of TealGUI is as follows:

```
public interface TealGUI{
    ...
    public void addProObject(ProObject te);
    public void remove(ProObject te);
}
```

- ConcreteProduct (SLabGUI, PChargeGUI, etc): defines the specified GUI classes required by different applets or applications. All ConcreteProducts implement TealGUI interface. The examples given here are SLabGUI and PChargeGUI. The implementations of addProObject and remove methods are quite different in these two classes. The SLabGUI uses Java default feel and look while the PChargeGUI has a new feel and look named CompiereTheme. The arrangement of GUI components is also different in SLabGUI and PChargeGUI.
- Creator (TEALApp): defines the abstract class that requires an object of TealGUI. The methods creatTealGUI, getTealGUI and addGUIElement are all abstract. They need to be overridden in TEALApp's concrete subclasses.

```
public abstract class TEALapp extends JApplet implements TealFramework, ProObject {
        public abstract void createTealGUI(){}
        public abstract TealGUI getTealGUI() {}
        public abstract void addGUIElement(ProObject){}
}
```

- ConcreteCreator (SLabTest2Charge, PCharge): overrides the abstract methods in TEALApp and returns the instance of concrete GUI classes. The samples given here are SlabTest2Charge and PCharge. They are all concrete subclasses of TEALApp. The SlabTest2Charge class returns a concrete GUI instance of a class named SLabGUI while the PCharge class returns a concrete GUI instance of a class named PChargeGUI.

```
//SLabTest2Charge is a concrete creator  of SLabGUI
public class SLabTest2Charge extends TEALapp{
    protected TealGUI simGUI = null;
    // create a SLabGUI for this applet
    public void createTealGUI(){
            simGUI = new SLabGUI();
    }
    //return the SLabGUI created in this applet
    public TealGUI getTealGUI(){
            return simGUI;
    }
    //add GUI components into SLabGUI
    public void addGUIElement(ProObject ele){
            TealGUI gui = getTealGUI();
            gui.addProObject(ele);
}

//PCharge class is a concrete cretor of PChargeGUI
public class PCharge extentds TEALapp{
    protected TealGUI simGUI = null;
    //create a PChargeGUI for this applet
    public void createTealGUI(){
            simGUI = new PChargeGUI();
    }
    //return the PChargeGUI created in this applet
    public TealGUI getTealGUI(){
            return simGUI;
    }
    //add GUI components into PChargeGUI
    public void addGUIElement(ProObject ele){
            TealGUI gui = getTealGUI();
            gui.addProObject(ele);
}
```

The relationships among these classes are shown in Figure 3.10:

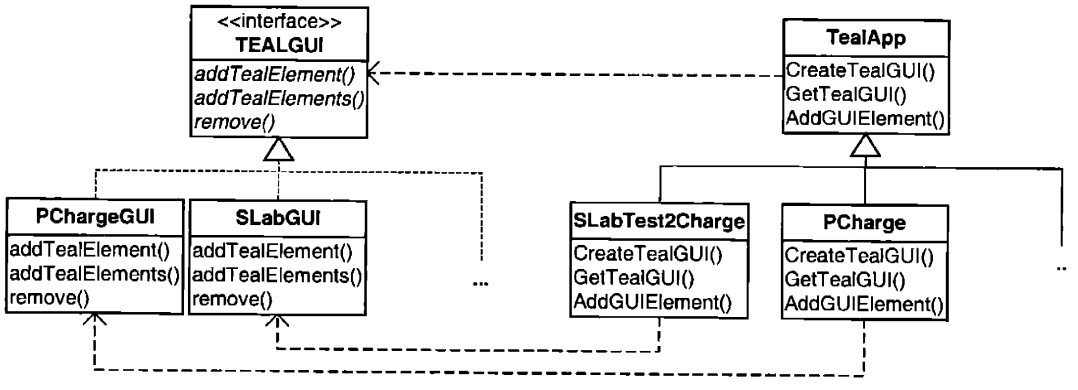

Figure 3.10 The Relationship of GUIs and Applications

Consequences:

The Factory Method pattern reduced the combination of application-specific code with the application-independent code. It makes the localizing object creation easier. It increases the reusability and portability of system. There is a disadvantage that users might have to subclass the Creator class only for creating a particular ConcreteProduct object.

3.3 Application 2: Adapter Pattern

Intent:

The Adapter pattern converts the programming interface of one class into that of another which is expected by a client. The adapters let unrelated classes work together without worrying about the incompatible interfaces.

Motivation:

The Adapter pattern is a good way to let unrelated classes work together in a program. It converts the programming interface of one class into that of another. The implementation of an adapter is relatively easy: create a class that has the desired interface and then use this class to communicate with the original class. There are two kinds of adapters: class adapters and object adapters. They achieve their aim in different ways: inheritance and object composition. By inheritance, we can derive a new class from the original one and add the desired methods. By composition, we include the original class inside the new one and create the desired methods to communicate with the original one.

In TEAL, we need to deal with a lot of mouse, window, and key events. Most of the time, only one or two of them are of interest to us. Adapters are used to simplify an unnecessarily complicated event interface.

Solution:

The Adapter pattern converts the programming interface of one class into that of another. There are four essential parts in the Adapter pattern: Client, TargetIF, Adapter and Adaptee. The Client is a class that calls a method which does not belong to a specific class. The TargetIF is an interface that declares the method that a Client class calls. The Adapter implements the TargetIF interface by calling a method of the Adaptee class. The Adaptee class does not implement the TargetIF method but has a method that we want the Client to call. The relationship of these classes are shown in Figure 3.11:

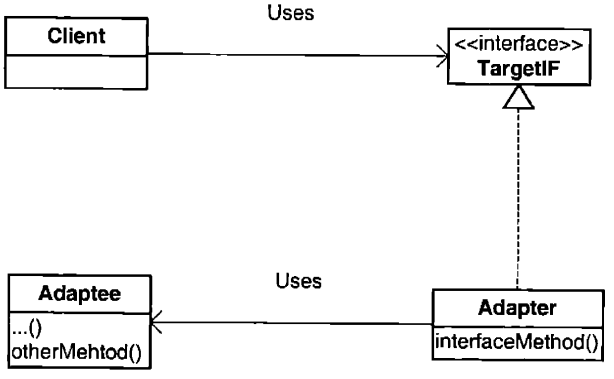

Figure 3.11: Class Diagram of Adapter pattern

Implementation:

Implementation of the adapter class is rather straightforward. One of the most commonly used adapters in TEAL is the MouseAdapter class, which is defined in Java Swing package. In our system, we often deal with some of the mouse events, such as mouse

pressed, released, clicked, entered or existed. The general solution to this problem is to have the class which deals with mouse events implement the MouseListener interface and leave all of the mouse events handlers empty except for the desired one. However, this approach is awkward because the class will be filled with a lot of empty methods. We can use MouseAdapter class instead. It implements MouseListener and defines all the five methods in MouseListener, although the methods in this class are all empty. We then need only override the events we care about. PickListener is a subclass of MouseAdapter, which is used to pick the object in a 3D viewer panel. Since we only care about mousePressed and mouseReleased events, these two events' handlers are overridden in our code.

```
class PickListener extends MouseAdapter{
        //define the action after mouse pressed
        public void mousePressed(MouseEvent me){
                ...
        }
        //define the action after mouse released
        public void mouseReleased(MouseEvent me){
                ...
        }
}
```

Following is the sample code to use PickListener class in a 3D viewer:

```
public class SimViewer3D extends SimViewer implements Viewer3D{
        public SimViewer3D(){
                myCanvas.addMouseListener(new PickListener());
                ...
        }
}
```

Another adapter used in TEAL is the WindowAdapter class, which is also part of Java Swing Package. One inconvenience of the Java Swing GUI package is that windows do not close automatically when we click on the Close button or window Exit menu item. The general solution to this problem is to have main Frame window implement the

WindowListener interface and leave all of the Window event handlers empty except for the handler for the windowClosing event. However, the Window Adapter class is provided to simplify this procedure. This class contains empty implementations of all seven of the windowEvents. We then need only to override the windowClosing event and insert the appropriate exit code.

```
public class TEALapp extends JApplet implements ActionListener, TealFramework, TealElement{
        ...
        public static void main(String args[]) {
                ...
                appFrame.addWindowListener(new WindowAdapter() {
                        public void windowClosing(WindowEvent e) {
                                System.exit(0);
                        }
                });
        }
}
```

Consequence:

Using Adapter pattern, the Client and Adaptee classes remain independent of each other. . The adapters let unrelated classes work together without worrying about their incompatible interfaces. The disadvantage of Adapter pattern is that it introduces an additional indirection into a program, which may increase the difficulty of understanding the program.

3.4 Application 3: Iterator Pattern

Intent:

The Iterator pattern allows user to move through a collection of data in a standard way without exposing the details of that data's internal representations.

Motivation:

The motivation for the Iterator pattern is quite simple. We want to move through a set of data elements without exposing what happens inside those data. We also expect to perform some special processing and return only specified elements of the data collection. For example, all the physical objects in TEAL, such as pointCharge, dipole, magnet, etc have a 3D shape. We want to display the 3D shape on the 3D viewer panel. Then we need a collection of these 3D objects which implements an interface that allows us sequentially access this collection, add and remove objects from this collection easily. The Iterator pattern can meet this requirement.

Solution:

Figure 3.12 shows the organization of the classes and interfaces that participate in the Iterator pattern. Those classes and interfaces are Collection, IteratorIF, Iterator and CollectionIF separately. The Collection is a class that encapsulates a collection of objects or value. The IteratorIF is an interface defines methods to sequentially access the Collection object. The Iterator is a class implements an IteratorIF interface. The CollectionIF declares a method for creating an Iterator object.

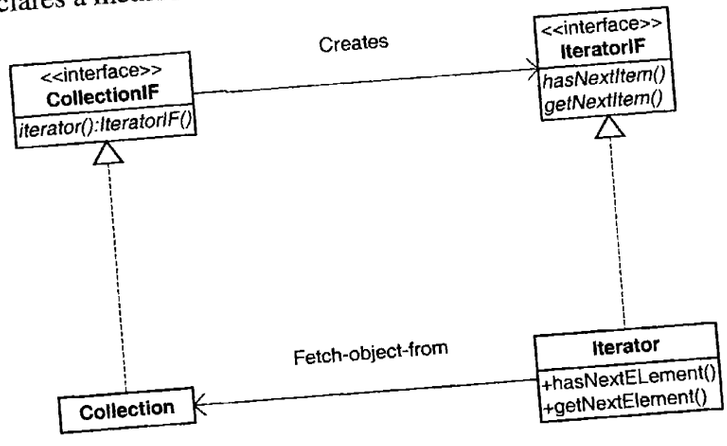

Figure 3.12: Class Diagram of Iterator pattern

Implementation:

Java defined Iterator interface as IteratorIF. The definition of Iterator is as following:

```
Public interface Iterator{
        public Boolean hasNext();
        public Object next();
        public void remove();
}
```

Java defined Collection interface as CollectionIF. The Collection interface has an iterator method that returns an instance of the class implementing Iterator interface. The definition of Collection is as following:

```
public interface Collection{
        public Iterator iterator();
        ...
}
```

Vector is a typical class that implements Collection interface. It's used to store a collection of objects in TEAL. The use of Vector is as following:

```
public class SLabTest2Charge extends TEALapp{

        Vector  controllers = new Vector(); //controllers is used to store a collection of GUI components
        ...
        TealGUI chargeGui = new PChargeGUI();
        chargeGui.AddTealElments( controllers); //add GUI components into graphical interface
}

public class PChargeGUI extends SimPanel implements TealGUI {
        //add a colletion of GUI compoments into graphical interface panel
        public void addTealElements(Collection list) {
                Iterator it = list.iterator(); //get Iterator for vector
                while (it.hasNext()) {      //get individual GUI component from vector
                        addTealElement((TealElement) it.next());
                }
        }
        ...
}
```

We also developed other iterator interfaces in TEAL. For example, interface VectorIterator is used to iterate over a set of points in a field. It defines nestVec, hasNext and reset methods.

```
public interface VectorIterator {
        public Vector3d nextVec();
        public boolean hasNext();
        public void reset();
}
```

RandomGridIterator is a class implementing VectorIterator interface. It produces pseudo-random points that cover a rectangular region. The Java code for the RandomGridIterator class is as follows:

```
public class RandomGridIterator implements VectorIterator {

        private Vector3d v = new Vector3d();
        private int width, height;
        private Random random;

        public RandomGridIterator (int width, int height, Random random){
                this.width = width;
                this.height = height;
                this.random = ( random == null ) ? new Random() : random;
        }

        public boolean hasNext(){
                return true;
        }

        public void reset()
        {
        }

        public Vector3d nextVec(){
                v.x = random.nextInt( width );
                v.y = random.nextInt( height );
                v.z=0.0;
                return v;
        }
}
```

The RandomGridIterator is used in TealGUI to create random points. The code follow is one example.

```
public class PChargeGUI extends SimPanel implements TealGUI {
        public void getRandom(int width, int height, int num) {
                RandomGridIterator it = new RandomGridIterator(width, height, new Random());
                Int i=1;
                while (it.hasNext() && i<num) {
                        Store((TealElement) it.nexVec());
                        i++;
                }
        }
        ...
}
```

Consequences:

By using Iterator, it is possible to access a collection of objects without knowing the source of the objects. By using multiple iterator objects, it is simple to manage multiple transversals at the same time. A problem may be caused by modifications to a collection object while an iterator is traversing its contents. If no provisions are made for dealing with such modifications, an iterator may return an inconsistent set of results. Such potential inconsistencies include skipping objects or returning the same object twice. The simplest way of handling modifications to an underlying collection during a traversal by an iterator is to consider the iterator to be invalid after the modification.

3.5 Application 4: Template Pattern

Intent:

The Template pattern formalizes the idea of defining an algorithm in a class, while leaving some of the details to be implemented in subclasses. It also lets the subclasses redefine some steps of the algorithm without changing the algorithm's structure.

Motivation:

We want to design a class that could be used in several programs. Although the overall algorithm of the class will be the same, part of the details will be different in each program in which it is used. To insure the programmer uses the correct logic, an abstract class may be used to define the logic and leave the detailed implementation to its subclasses. We may also meet another situation in which an existing design contains a number of classes that do similar things. To minimize code duplication, we factor those classes, identifying what they have in common and what is different. Then the classes are reorganized so that their differences are completely contained in discrete methods with common names. The remainder of the code in the classes should be the common code that is defined in a superclass.

Actually, we met the latter situation while refining TEAL's framework. For example, all the physical objects, such as point charge, dipole, magnet, etc have some similarities. So we find out the common part and define a superclass named physicalObject, in which some common methods are defined. All the physical objects should be derived from this class directly or indirectly, using or overriding some of the methods defined in physicalObject. Another example is that since the Swing components can't meet our requirements, a new set of GUI components needs to be developed. So a basic class is defined as ControlPanel. All the other components are derived from it. Template pattern provides a simple way to organize these classes.

Solutions:

There are two essential parts in Template pattern: AbstractTemplate and ConcreteTemplate. The AbstractTemplate is an abstract class which defines both concrete methods and abstract methods. The abstract methods vary in each of the subclasses of abstractTemplate class. The ConcreteTemplate is a concrete class derived from AbstractTemplate. It overrides the abstract methods defined by its superclasses. The roles of these classes are shown in Figure 3.13:

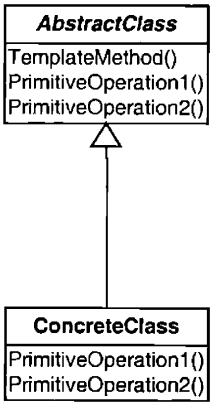

Figure 3.13: Class Diagram of Template pattern

Implementation:

The Template pattern is very simple and widely used in TEAL. The sample given here is a physical object named Dipole.

- AbstractClass: defines both the abstract and concrete TemplateMethods. The name of the AbstractClass used here is Dipole. The updateNode method is an abstract method to update status of Dipole. While the setLength and setRadius are concrete methods which will not be changed in the subclasses of Dipole.

```
public abstract class Dipole extends PhysicalBody implements GeneratesB,GeneratesE,GeneratesP{
        //an abstract method used to update node in model
        protected abstract void updateNode();
        //a concrete method
        public void setLength(double h){
```

```
            length = h;
            if (mNode != null){
                    updateNode();
            }
            clearShapes();
        }
        //a concrete method
        public void setRadius(double w){
            radius = w;
            if (mNode != null){
                    updateNode();
            }
        }
    ...
}
```

- Concrete Class: There are two concrete subclasses of dipoles in TEAL: electronic dipole and magnet dipole. They have some similarities as well some differences. The common part and general logic is extracted to be in the abstract class Dipole. The MagneticDipole class stands for the dipole having magnetic characteristic while ElectricDipole stands for the one having current. The implementation of the updateNode method is quite different in MagneticDipole and ElectronicDipole:

```
public class MagneticDipole extends Dipole{
    //the implementation of updatNode
    protected void updateNode(){
            ((MagDipoleNode3D)mNode).update();
            ((HasRotation)mNode).setRotation(directionControl.getQuaternion());
    }
    //the inner class MagDipoleNode3D stands for a magnatic node having 3D shape
    private class MagDipoleNode3D extends SimNode{
            ...
            void update()
            {
                ....
            }

    }
}

public class ElectricDipole extends Dipole implements HasCharge,GeneratesEPotentiel{
    //the implementation of updateNode
    protected void updateNode(){
            ((ElectricDipoleNode3D)mNode).update();
    }
    //the inner class ElectricDipoleNode3D stands for a electronic node having 3D shape
    private class ElectricDipoleNode3D extends SimNode{
```

```
            ...
            void update(){
                    ...
            }

    }
}
```

The relationship of these classes are shown in Figure 3.12:

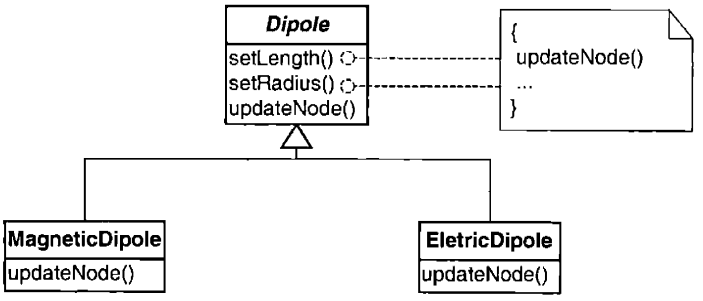

Figure 3.12: Class Diagram of Template pattern in TEAL

Consequences:

Template pattern is one of the most useful techniques for code reuse. It's important for template methods to specify which methods may be overridden and which must be overridden. To reuse an abstract class effectively, users must be familiar with which methods are designed for overriding. It's also very useful to extend a parent class's behavior while writing subclasses. The users should remember to call the inherited method.

3.6 Summary:

As mentioned in Chapter 2, design patterns provide simple and reusable solutions to a great number of common problems that could be met during software development. Design patterns provide a convenient way of reusing object-oriented code between

projects and programmers. Actually, during the design and refinement process of TEAL, we used many design patterns either consciously or unconsciously. The design patterns we mentioned above are only some of the typical applications used in TEAL. Most of the time, design patterns are not used separately. Several of them are often combined together to achieve an aim better.

Chapter 4: Extensions of Existing Design Patterns

4.1 Introduction:

Since experience is so important for a software developer, accumulating experience with design patterns will be very helpful in our future work. Some of the experiences, which are derived from the implementation of TEAL, will be summarized in this chapter. The first part of this chapter will introduce the Property Route pattern, which is evolved from Observer pattern. It will describe how to dynamically link and automatically change properties between objects. It's widely used in TEAL and is useful in a Model-View-Controller model. The second part of this chapter will talk about how to assign responsibilities among classes to make a well-structured design. General Responsibility Assignment Software Pattern (GRASP) is beyond the range of traditional design patterns, but will lead to the using of traditional design patterns.

4.2 Property Route Pattern

Intent

Designing the one-to-many dependency among objects' properties so that when the property of one object is changed, all its dependents are notified and updated automatically. There is no need for an observable object and the observer to know of each other's existence or properties.

Motivation

In some sophisticated models, we often want to display data in more than one form, and sometimes want one controller to control data from several objects. We don't want to achieve these aims by making the objects and their properties tightly coupled since it will reduce their reusability. The idea of the Property Route pattern evolved from the Observer pattern. In the Observer Pattern, objects dynamically register dependencies between objects, that is we can add observers to observable objects dynamically. When the state of an observable object changes, the observers will be notified and respond accordingly. There are several limitations of the Observer pattern:

- All the observers need special code for the observable object. In other words, the observers must "know" what object it will observe and contain a specific code for it.
- All the changes of an observable object will be sent to observers instead of the only "useful" information to related observers. Delivering notifications can take a long time if an object has a large number of objects to deliver notifications to. Most of the time, observers are only interested in a subset of the properties of observable objects.
- It's hard to coordinate the changes of observers. Sometimes the observer may be observable object and observer at the same time.

Based on the above, we extended the Observer pattern into the Property Route pattern.

Solution:

There are several key elements in Property Route pattern:

- ProObject: defines the property change methods required by both property change sender and property change listener. The Property Route pattern assumes that each

object, whether sender or listener, implements the proObject interface. ProObject interface often includes the following methods:

```
public interface proObject extends propertyChangeListener{
        //add a listener to a property
        public void addPropertyChangeListener (propertyName, propertyChangeListener listener);

        //remove a listener to a property
        public void removePropertyChangeListener(propertyName, propertyChangeListener);

        //revoke property change event
        public void firePropertyChange(PropertyChangeEvent evt);

        //setProperty and getProperty are a pair of methods which are required by a property in a
        class
        public Object getProperty(String attName);
        public boolean setProperty(String attName, Object value);
}
```

The addPropertyChangeListener and removePropertyChangeListener are two methods which add or remove link among property and listeners. The firePropertyChange is a method to revoke the propertyChange event. The getProperty and setProperty are a pair of methods which are required to get or modify properties in a class.

- PropertySender: implements the proObject interface. It's the object whose property will be observed by other objects. The instance of PropertyRoute inside propertySender class is used to store a collection of objects listening to the property change notifications sent by propertySender. The addRoute method adds a listener to a sender's property. The removeRoute method removes a listener from a sender's property.

```
public class propertySender implements proObject{
    protected PropertyRoute mRoutes;

    // add a listener to a property
    public void addRoute(proObject obj, String att, proObject listener, String target){
```

```
            mRoutes.addRoute(obj, att, listener,target);
    }

    //remove a listener to a property
    public void removeRoute(proObject obj,String attName, proObject listener,  String target{
            mRoutes.removeRoute(obj, attName, listener, target);
    }
    ...
}
```

- PropertyListener: implements proObject interface. The property of propertyListener will be changed automatically when it receives notification from the propertySender.

```
public class propertyListener implements proObject{
    ...
}
```

- PropertyRoute: acts as the bridge between propertySender and propertyListener. The instance of Vector named routes is used to store the links between two object's properties. The addRoute method will add new links between the sender's and receiver's properties. Whenever a link is created, it will be added to routes. While the removeRoute method is used to remove the link. When a link is removed, it will be deleted from routes. The propertyChange method can send notification and automatically modify the listener's property stored in routes.

```
public class propertyRoute implements propertyChangeListener{
    Vector routes;
    //add link between sender and listener
    public void addRoute(proObject obj, String att, proObject listener, String target){
            ...
    }
    //remove link between sender and listener
    public void removeRoute(ProObject obj,String attName, ProObject listener,  String target){
            ...
    }
    //send notification to listeners
    public void propertyChange(PropertyChangeEvent ev)
    {
            ...
    }
}
```

The relationship between these interface and classes is shown in Figure 4.1.

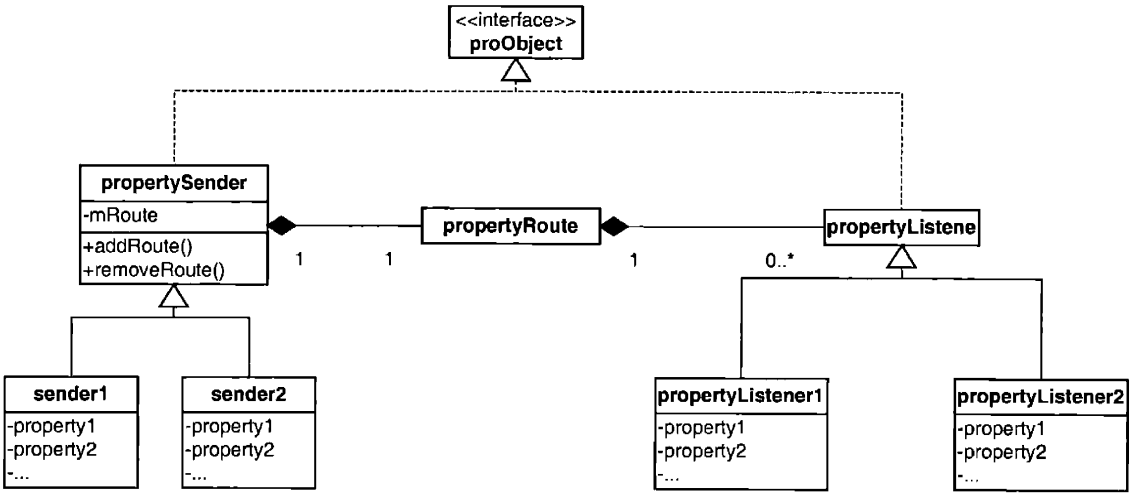

Figure 4.1 Property Route class diagram

Implementation:

In the TEAL project, we use Property Route pattern to control data/properties stored in objects. It's quite easy to add and remove views, controllers from model objects, which increase the flexibility of system. An example using Property Route Pattern in TEAL is shown in Figure 3.2:

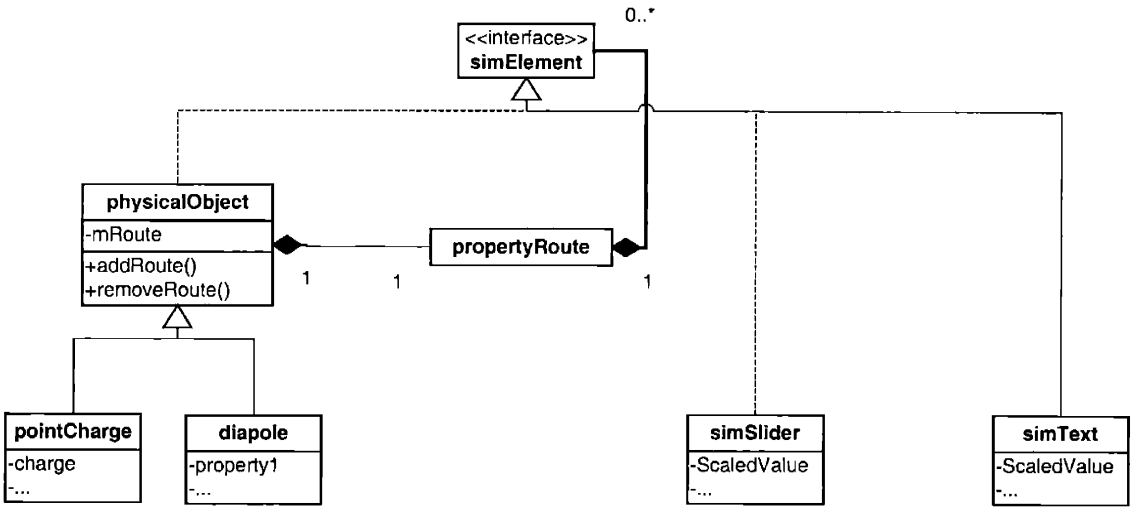

Figure 4.2 Application of Property Route in TEAL

- SimElement (proObject): defines the common interface for simulation elements.
- PhysicalObject (propertySender): defines a class which implements the SimElement interface. A collection of listeners to the property of PhysicalObject can be added into the instance of PropertyRoute.
- PointCharge (propertySender and propertyListener): a physical object whose charge property will be linked with other property listeners. PointCharge class can also be used as a propertyListener. The getCharge and setCharge methods are combined as shown in the code below to control the charge property in PointCharge class.

```
public class PointCharge extends PhysicalObject implements HasCharge, simElement{
    protected double charge;
    public double getCharge() {
            return this.charge;
    }
    public void setCharge(double ch) {
            if (ch != charge) {
                    PropertyChangeEvent pce= PCUtil.makePCEvent(this, "charge", charge, ch);
                    charge_d= ch;
                    charge= ch;
                    firePropertyChange(pce);
            }
    }
...
}
```

- SimSlider (propertySender and propertyListener): used to control an object's property in the form of slider. It also has a property named value. When the value property is linked with the charge property in PointCharge class, they can be changed automatically when either one of them has any changes. The Java code below illustrates how this is done:

```
public class SimSlider extends JSlider implements ChangeListener, TealElement, Control{
    double lastValue;
    public void setValue(double val){
    ...
    }
    public double getValue(){
    ...
    }
    public void stateChanged(ChangeEvent ev){
            int val = getValue();
            PropertyChangeEvent pc = PCUtil.makePCEvent(this,"scaledValue",
                                                        lastValue * scale,val * scale);
            mRoutes.propertyChange(pc);
            firePropertyChange(pc);
            lastValue = val;

    }

...
}
```

- SimText (propertySender and propertyListener): used to control an object's property in form of text. It's similar to SimSLider.

```
public class SimText extends JText implements ChangeListener, TealElement, Control{
    //almost the same as SimSlider
}
```

The use of property route pattern is as follows:

```
public class SLabTest2Charge extends TEALapp{
        public SLabTest2Charge()
        {

                super();
                PointCharge pc1 = new PointCharge();
                pc1.setCharge(0.1);

                SimSlider slider1 = new SimSlider();
                slider1.setMinimum(-10);
                slider1.setMaximum(10);
```

```
            slider1.setScale(0.5);
            slider1.addRoute("scaledValue",pc1,"charge");

            SimText text1 = new SimText();
            text1.setMinimun(-10);
            text1.setMaximum(10);
            text1.setScale(0.5);
            text1.addRoute("scaledValue",pc1,"charge");
            text1.addRoute(("scaledValue",slider1,"scaledValue");
            slider1.addRoute(("scaledValue",text1,"scaledValue");
        ...
        }
}
```

In this application, the pc1 (an instance of PointCharge) is a propertyListener; the slider1 and text1 are both propertyListeners and propertySenders. When the user changes the value in slider1 or the value in text1, this will cause the propertyRoute inside them to change the related property of their listeners. In this way, the pc1, slider1 and text1 can work synchronously.

Consequences:

There are several advantages of using the Property Route pattern:

- It allows the property sender to deliver only useful or registered property changing notifications to property listeners, which improve the efficiency of object delivery and partly solves the long time delay problem caused by sending notifications to too many listeners.
- There is no need of the sender and listeners being aware of each other, which improves the portability of system.
- The listeners can respond to the notifications automatically. Since property route class's main function is to link the senders' and listeners' properties together, the receiving of notification will change the listeners' corresponding property automatically. There is no need of additional code to respond to this change.

On the other hand, there are also some situations in which the Property Route pattern may have unforeseen or undesirable results:

- Although the delivery efficiency is largely improved by only delivering the useful information to registered listeners, it may also take a long time for the sender to deliver notifications to a large number of listeners if a sender has many indirect listeners and its notifications will be cascaded by other objects.
- Another problem will be caused by cyclic dependencies. When this happens, objects call each other's setProperty methods until the stack fills up and a StackOverflowError is thrown. Though serious, this problem can be easily solved by adding appropriate code to a class which is involved in the cycle and can detect a recursive notification. For example, in a pointCharge class, we can add an *if* statement in the setCharge method. Only when the *charge* field stored in the object of pointCharge gets a new value, propertyChange method will be fired and the notification will be sent to the registered listeners. Otherwise, no action will be performed at all. Through this approach, recursive call can be avoided.
- Some more consequences should be considered if a notification could be delivered asynchronously of other threads. We need to ensure that the asynchronous delivery of notifications is done in a way that ensures the consistency of the listeners that receive the notifications. For example, we should consider the situation that a notification is sent to a "sleeping" thread, how the thread can receive this notification when it's "wakes up". It may also be important to ensure that notification does not block waiting for another thread for any extended length of time.

4.3 GRASP Patterns

4.3.1 Concept of GRASP patterns

General Responsibility Assignment Software Patterns (GRASPs) are not traditional design patterns. Traditional design patterns help users solve specific design problems in general situations. However GRASP patterns provide guidance in assigning responsibilities to classes. GRASP patterns present fundamental object-oriented design principles in the form of patterns. They are used to lead the designers to a situation where design patterns are applicable. GRASP patterns were first documented by Craig Larman [Larman, 1998]. His motivation for formulating this set of patterns was to help people learn object-oriented design. Typical GRASP patterns include: Low Coupling/High Cohesion, Expert, Creator, Polymorphism, Pure Fabrication, Law of Demeter and Controller. All these patterns provide skillful assignment of responsibilities among classes. They are often combined to make the design well structured and easy to understand or maintain. Samples of how to use multiple GRASP patterns to decide the structure of design in TEAL and how these GRASP patterns lead to traditional design patterns will be given in the following part of this chapter.

4.3.2 Low Coupling

One of the main tasks of refining the TEAL framework is decoupling highly coupled classes in order to make the design easier to modify and maintain. Highly coupled classes are difficult to understand because they tend to introduce internal dependencies between

unrelated functions. Highly coupled classes are also difficult to reuse because they must be used with the classes on which they depend. Highly coupled classes require additional maintenance effort because the more classes that a class depends on, the more changes are required to reuse the coupled classes. For example, at the very beginning, the user interface of TEAL was highly coupled with the model of application as follows:

```
public class TEALapp extends JApplet implements ActionListener, TealFramework, TealElement{
        protected JPanel displayPane;
        protected JPanel treePane;
        protected JPanel viewPane;
        protected JPanel controlPane;
        public TEALapp() {
                super();
                getContentPane().setLayout(new BorderLayout());
                createFrame();
                createModel();
        }

        //create the GUI frame
        private void createFrame(){
                displayPane = new JPanel();
                treePane = new JPanel();
                ...
                getContentPane().add("Center",displayPane);
                ...
        }

        //add GUI components into frame
        public void addElement (TealElement e) {
                if (e instance of controller)
                        addInTreePane(e);
                else if (e instance of viewer )
                        addInviewPane(e)
                ...
        }

        //add  GUI components into tree view
        private void addInTreePane(){
        ...
        }

        //create model
        private createModel(){
                ....
                addElement(e);
        }

        //other essential methods
        public object getProperty(){
        ...
```

```
        }

        public  void setProperty(object o){
        ...
        }
        ...
}
```

Looking at the above code, we can notice that the application is responsible for many tasks, such as model control, property control, frame control and the user interface control. It's not good to mix a lot of indirectly associated functions in one class. For example, whenever we want to change user interface, we have to change the whole application class, which may influences model, frame or property. To make the modification of user interface easier, the coupled code should be separated into application and GUI classes. Actually, it leads into the Factory Method pattern which is mentioned in chapter 3. The changed code shown below is much easier to modify:

```
public class TEALapp extends JApplet implements ActionListener, TealFramework, TealElement{
        //set GUI as a property of application
        protected TealGUI simGUI = null;

        public TEALapp() {
                super();
                getContentPane().setLayout(new BorderLayout());
                ...
        }

        //get GUI property
        public TealGUI getTealGUI() {
                return simGUI;
        }

        //set GUI property
        public void setTealGUI(TealGUI gui) {
                simGUI = gui;
                if (simGUI != null)
                        getContentPane().add(BorderLayout.CENTER, simGUI.getPanel());
        }

        //add GUI components into user interface
        public void addElement(TealElement e){
                if (e instance of GUIElement)
                        simGUI.addElement(e);
                ...
```

```
        }
        ...
}
```

In this new structure, various user interface classes, which implement TealGUI interface, can be created and included in TEAL project. We can load the preferable user interface into the same application just through the setTealGUI method. It's much easier to maintain and modify.

4.3.3 Expert

During the design process of TEAL, we often met the problem of how to assign responsibility to classes. Most of the time, there is more than one way to achieve the desired result. The basic rule to make this decision is to assign a responsibility to the class or classes that have the information required to carry out the responsibility. For example, suppose we want to display the 2D shape of PointCharge in a 2D panel named SimViewer2D panel. There are two options on how to draw the 2D shape. One is implementing a "draw" method in a SimViewer2D. The other is implementing the "draw" method in PointCharge. If the 2D shape is drawn in SimViewer2D, the code should be as follows:

```
public class SimViewer2D extends SimViewer implements Viewer2D{
        //render objects in  physical world
        public void render(boolean doRepaint) {
                Iterator i = drawObjects.iterator();
                while (i.hasNext()) {
                        drawable = (Has2DShape) i.next();
                        if(checkDraw(drawable))
                                draw((drawable)i); //call for draw method in Viewer2D
                }
        }

        //draw all the objects having 2D shapes
        private void draw(drawable i){
```

```
                if (i instant of pointCharge){          //draw pointCharge
                        Graphics2D g2= getGraphics2D();
                        Vector3d pos= i.getV().project(i.getPosition()); //get property inside object
                        double radius = i.getRadius();
                        g2.setPaint(
                                new GradientPaint(
                                        (float) (pos.x - 1.5 * radius),
                                        (float) (pos.y - 1.5 * radius),
                                        Color.white,
                                        (float) (pos.x),
                                        (float) (pos.y),
                                        i.getColor(),
                                        false));
                        g2.fill(new Ellipse2D.Double(pos.x - radius, posy - radius, 2 * radius, 2 *
radius));
                }

        }
        ...
}
```

If the 2D shape is drawn by PointCharge, the code looks like the following:

```
public class SimViewer2D extends SimViewer implements Viewer2D{
        //render objects in physical world
        public void render(boolean doRepaint) {
                Iterator i = drawObjects.iterator();
                while (i.hasNext()) {
                        drawable = (Has2DShape) i.next();
                        if(checkDraw(drawable))
                                drawable.draw(this);  //draw object by itself
                }
        }
        ...
}

//point charge, a kind of the physical object
public class PointCharge extends PhysicalObject implements drawable {
                //draw object by itself
                public void draw(Viewer2D v) {
                        Graphics2D g2= v.getGraphics2D();
                        Vector3d pos= v.project(position);
                        g2.setPaint(
                                new GradientPaint(
                                        (float) (pos.x - 1.5 * radius),
                                        (float) (pos.y - 1.5 * radius),
                                        Color.white,
                                        (float) (pos.x),
                                        (float) (pos.y),
                                        mColor,
                                        false));
                        g2.fill(new Ellipse2D.Double(pos.x - radius, pos.y - radius, 2 * radius, 2 *
        radius));
                }
}
```

Comparing these two versions, we can find that if the "draw" responsibility is assigned to the SimViewer2D class, a lot of inside state of PointCharge needs to be revealed to SimViewer2D. This makes the SimViewer2D highly reliant on the PointCharge classes. On the other hand, if the "draw" method is implemented in PointCharge class, the coupling will be highly decreased.

Actually, the Expert pattern is highly related to the Low Coupling pattern. It promotes low coupling by putting methods in the classes that have the information required by the methods. Classes whose methods require only the class's information have less need to rely on other classes.

4.3.4 Creator

Since some objects will be used in several places in an application, it's important to decide where and when to initiate them. This decision should be made based on the relationship between the potential creator classes and the class to be instantiated. For example, in an application of TEAL, WorldModel is the model of physical world, where all the software counterparts of physical objects exist. Almost all the physical objects need it as a reference. Since WorldModel is created in an application named SLabTest2Charge, according to the Creator's rule, it's appropriate to create all other physical objects which need it as reference in SlabTest2Charge also. This is shown in the code below.

```
public class SLabTest2Charge extends TEALapp{
```

```
WorldModel theModel;
public SLabTest2Charge()
{

        super();
        title ="sLab Test";
        TealGUI gui = new SLabGUI();     //create GUI
        theModel = new SimWorld();       //create new physical world model

        SimViewer3D v3 = new SimViewer3D(); //create 3D viewer
        gui.addTealElement(v3);                  //add 3D viewer into GUI
        theModel.addSimElement(v3);              //add 3D viewer into model

        SimViewer2D v2 = new SimViewer2D(); //create 2D viewer
        v2.setBackground(Color.darkGray);
        gui.addTealElement(v2);                  //add 2D viewer into GUI
        theModel.addSimElement(v2);              //add 2D viewer into model

        SimModelControl smc = new SimModelControl(); //create model controller
        smc.setModel(theModel);
        smc.setVisible(true);
        gui.addTealElement(smc);

        PointCharge pc1 = new PointCharge();   //create point charge
        pc1.setID("PontCharge 1");
        pc1.setPosition(new Vector3d(1,-1,0));
        pc1.setModel(theModel);
        theModel.addSimElement(pc1);

        PointCharge pc2 = new PointCharge();  //create point charge
        pc2.setID("PontCharge 2");
        pc2.setPosition(new Vector3d(-2,-0.8,0));
        pc2.setCharge(-.1);
        pc2.setModel(theModel);
        theModel.addSimElement(pc2);
        ...
}
```

The creator pattern promotes low coupling by making instances of a class responsible for creating objects they need to reference. By creating the objects themselves, they avoid being dependent on another class to create the object for them.

4.3.5 Polymorphism

Polymorphism is the most widely used GRASP pattern in TEAL. If alternate behaviors are selected based on the type of an object, two approaches are commonly used to solve

this problem. One is using a chain of *if* statements to test the type. This is not a good choice since programmers have the opportunity to introduce bugs into a program when writing and maintaining the code. On the other hand, using polymorphism makes it easy to overcome this shortcoming. The polymorphic method call also increases the ease with which new cases can be added to code.

An example of using a polymorphic method call is given here. PropertyControl, ControlCheck, ControlCombo, ControlInteger, and ControlDouble are all derived from PropertyControl class. They all have setValue and getValue methods. But the implementation of these methods is quite different in each subclass. To use polymorphism, PropertyControl is declared as abstract class, which has two abstract methods setValue and getValue. The concrete methods are given in concrete subclasses of PropertyControl. The implementation of the polymorphic methods is as follows:

```
public abstract class PropertyControl extends ControlPanel{
        public abstract void setValue(Object obj);
        public abstract Object getValue();
        ...
}

//a GUI component to control property in form of CheckBox
public class ControlCheck extends PropertyControl implements ActionListener,ChangeListener{
        JCheckBox mCheck;
        public void setValue(boolean newValue){
                PropertyChangeEvent pc= PCUtil.makePCEvent(this, "value", lastValue, newValue);
                mRoutes.propertyChange(pc);
                mCheck.setSelected(newValue);
                 lastValue = newValue;
        }

        public Object getValue(){
                return new Boolean(lastValue);
        }
}

// a GUI component to control property in form of ComboBox
public class ControlCombo extends PropertyControl
        implements TextListener, ItemListenerpublic void setValue(Object obj){
        public void setValue(Object obj){
```

```
                Object value = mValues.elementAt(mCombo.getSelectedIndex());
                mCombo.setSelectedItem(target);
                lastValue = value;
        }

        public Object getValue(){
            return mValues.elementAt(mCombo.getSelectedIndex());
        }
}
...
```

Actually, polymorphism can lead to other design patterns, such as Factory Method. Consider a situation that an application can have many user interfaces. From the above analysis, it's clear that polymorphism is a better choice than use of an *if* statement. An abstract class named application can be created, from which several concrete applications are derived. The method of creating user interface in application class will be redefined in its subclasses. Actually the application and concrete application classes are the Creator and ConcreteCreator in the Factory Method pattern. Polymorphism can also lead to the Strategy pattern. If there are many algorithms existing in the same method, use of an *if* statement is not a good choice to make decision on which algorithm should be used. We can use polymorphism to deal with such kind of problems. An interface or abstract class Strategy can be defined as a superclass. In concreteStrategy classes, which are derived from the Strategy class or implement a Strategy interface, the same method will be redefined or overridden. Now Strategy pattern can be used to optimize the design. Polymorphism can also lead to other design patterns. It's one of the most important ideas in Object-Oriented software design.

4.3.6 Pure Fabrication

A great number of images or icons are used in TEAL user interface. Since the path of folder where TEAL package is installed is not fixed, the user should find out the path of the icon before importing images. For example, if we want to use an icon button in the application, the code should be like the following:

```
Icon theIcon = null;
String strPos = Swing.pathToButton + "help";
ClassLoader cl = Class.forName("teal.gui.IconCreator").getClassLoader();
URL url = cl.getResource(strPos);
if (url == null)
        url = cl.getSystemResource(strPos);
if (url != null){
        theIcon = new ImageIcon(url);
}
else {
        theIcon = new ImageIcon(strPos);
}

JButton help = new Jbutton(theIcon);
```

Since there are a lot of icons used in the application, it is awkward to use code like the above each time icon is used. The responsibility of importing an icon is required, but not essential in the application. When this happens, a pure fabrication class named IconCreator is created:

```
public class IconCreator {
        //create Icon according to the icon's name
        public static Icon getIcon(String name) {
                Icon theIcon = null;
                 String strPos = Swing.pathToButton + name;
                try{
                        ClassLoader cl = Class.forName("teal.gui.IconCreator").getClassLoader();
                        URL url = cl.getResource(strPos);
                        if (url == null){
                                url = cl.getSystemResource(strPos);
                        }
                        if (url != null){
                                theIcon = new ImageIcon(url);
                        }
                        else {
                                theIcon = new ImageIcon(strPos);
                        }
                }
                catch(ClassNotFoundException e){
```

```
                }
                return theIcon;
    }
}
```

We can use it as follows:

```
JButton help = new JButton(IconCreator.getIcon("help"));
```

There are some other pure fabrication classes in TEAL. Examples:

- ClassComparator: tests a list of classes or instances which may or may not be used under some condition.
- TDebug: helps developer debug and test existing code.
- ResourceFinder: creates an URL according to the class name passed to it by users.

Using pure fabrication we can add a responsibility to a program and still maintain the class's low coupling and high cohesion.

4.3.7 Controller

If a program receives events not only from the graphical user interface but also from other external sources, we should add an event class to receive external events and forward them to the appropriate internal event-handling object. Such an object that coordinates external events is called a controller object. The Property Route pattern is developed based on this idea. Actually, a property route, which is introduced in 4.2, is a kind of controller.

4.3.8 A refined TEAL framework

Based on the idea of GRASP Patterns, the TEAL framework was modified several times to be well structured and easier to modify. The class diagram of reorganized framework is shown in Figure 4.3, Figure 4.4 and Figure 4.5 respectively.

To make the GUI components meet application's requirements better, a new GUI package was developed in TEAL project. Figure 4.3 shows the class diagram of GUI package. All the GUI components are derived from the PropertyControl class either directly or indirectly. The PropertyControl class implements PropertyChangeListener, HasPropertyChange, Serializable and Control interfaces. It's a typical Java Bean and can be used to control the properties of physical elements much easier than the GUI components in Java Swing.

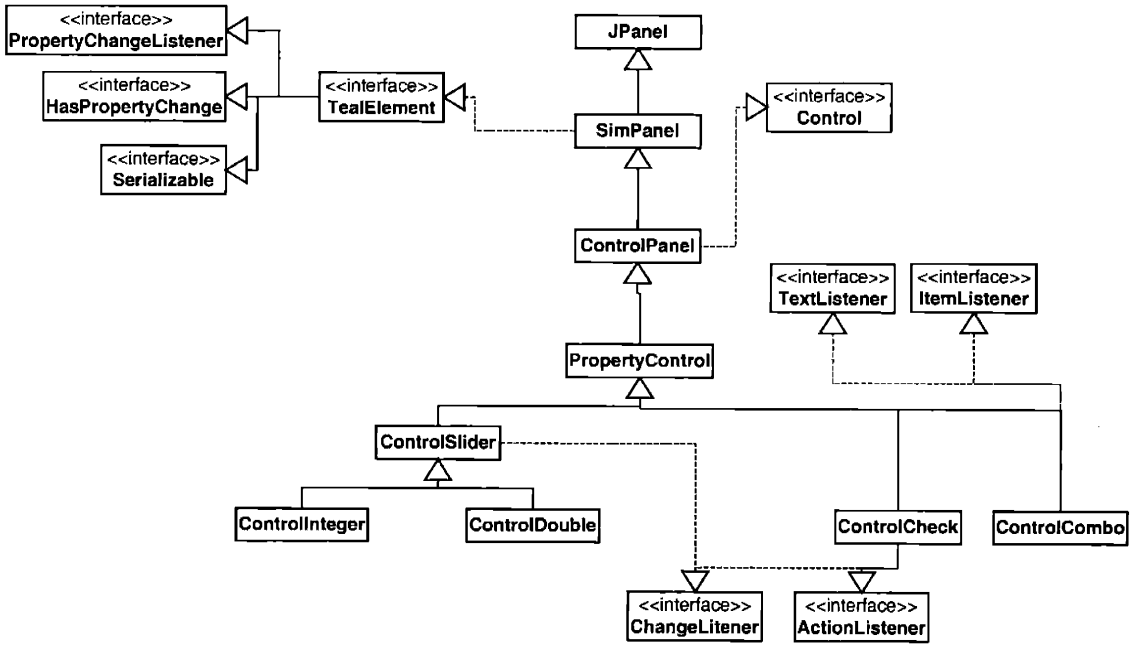

Figure 4.3 The Class Diagram of Graphical User Interface

There are five kind of physical objects in TEAL project currently. They are PointCharge, ElectricDipole, MagneticDipole, RingofCurrent and InfiniteWire. They have a lot of similarities, such as: they all have positions, 2D and 3D shapes, can be rendered and moved in a physical world, etc. There are also some differences between them, such as: they have different shapes and physical characteristics. Through decoupling and polymorphism, a new class diagram of physical objects is as shown in Figure 4.4. The common properties and methods of physical objects are summarized in the PhysicalObject class. All the physical objects are derived either directly or indirectly from it.

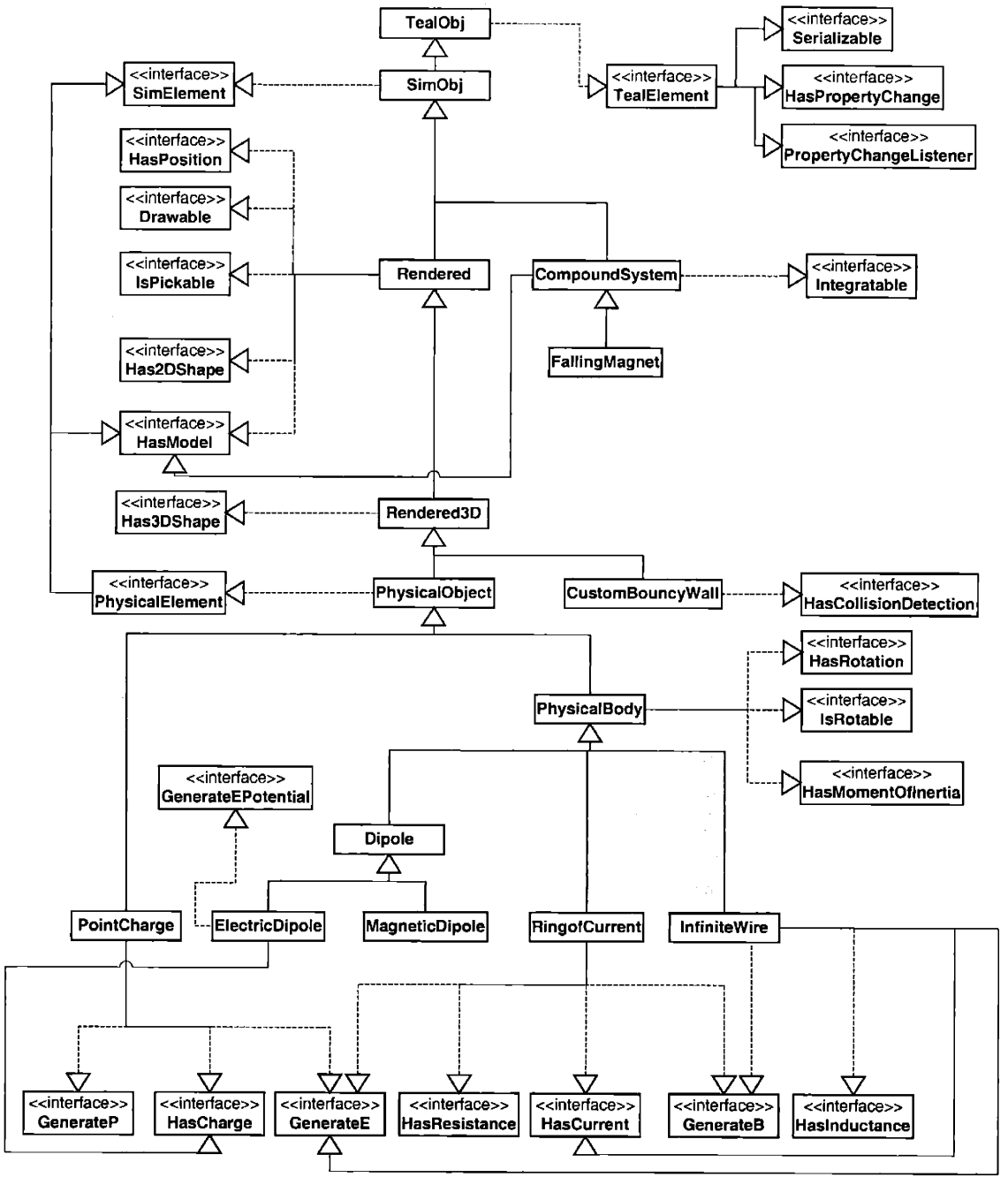

Figure 4.4: The Class Diagram of Physical Objects

An example of a typical application of TEAL is given in Figure 4.5. There are several main elements in this application:

- SimWorld: all objects are running in this world. It's unique in an application.
- SLabGUI: one of the user interfaces of TEAL.
- Sim3Dviewer: displays the 3D view of simulation model.
- Sim2Dviewer: displays the 2D viewer of simulation model.
- SimModelControl: a controller of simulation model.
- SimSlider: GUI component to control the property of physical objects.
- FieldVisualization and FieldValue: creates the physical field in simulation model.

The whole structure is better organized in new framework. The creation and modification of physical simulation is quite simple now.

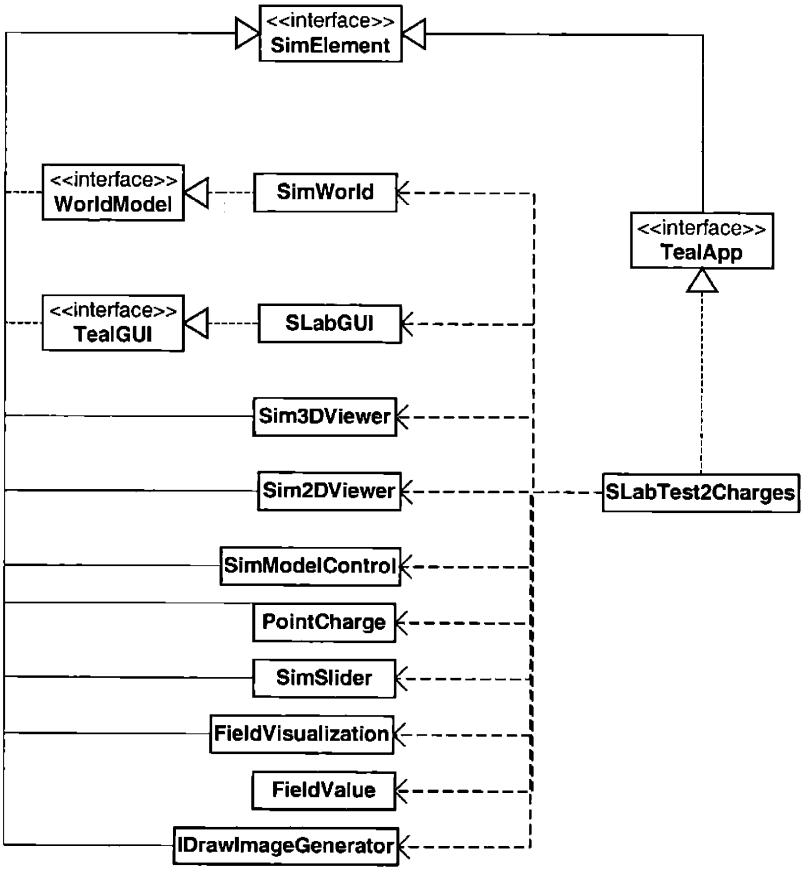

Figure 4.5: The Class Diagram of SlabTest2Charges Application

4.4 Summary

The extension of existing design patterns in TEAL is discussed in this chapter. In the first part of this chapter, a new design pattern named Property Control Pattern is introduced. Since it's a new pattern, the intent, motivation, solution, implementation and consequence are described. In the second part of this chapter, the GRASP Patterns are introduced. These are beyond the traditional concept of design patterns, and are used to summarize the experience of how to assign responsibilities among classes. The GRASP patterns are helpful to reorganize design structure and often lead to traditional design patterns. The refined model of TEAL framework is shown at the end of this chapter.

Using design patterns, the TEAL framework is well designed and reorganized. Through the designing and refining process, we accumulated a lot of experience and summarize this experience in the form of design pattern, which will be very helpful in our future work.

Chapter 5: Summary

5.1 Conclusion

After several years' development and refinement, the fourth version of TEAL simulations has been finished. Besides the object-oriented idea inherited from the former versions, there are still some new features in this version:

a. Using design patterns, the TEAL framework is developed and modified to make the creation and modification of the simulations much easier.

b. Using design patterns, the GUI, model and viewer parts are well separated and structured, which makes the change of each part much easier.

c. Using design patterns, the GUI part is well developed, which make the simulations are more interactive and configurable.

d. The simulations are currently rendered not only in 2D but also in 3D. Since the model and viewers are separated, the simulations can be viewed in even more viewers.

e. Some experiences are summarized in the form of design patterns. The Property Route pattern was invented and described in this thesis.

5.2 Future Work

There are still some work should be done in the future:

a. The TEAL framework can be better optimized with design patterns.

b. More physical objects should be added into TEAL library, so that more simulations could be created.

c. The GUI should be more interactive. For example, some physical concepts should be added into TEAL. So that only meaningful operations are permitted.

d. More exception handlers should be added into simulations

e. "Redo" and "Undo" operations can be realized using the Command Pattern.

f. More experience could be summarized from TEAL simulations in the form of design pattern. It will be useful for future work or for other developers as reference.

References

[AIS+, 1977] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. A Pattern Language. Oxford University Press, New York, 1977.

[Alpert, 1998] Alpert, Sherman, K. Brown, and B. Woolf. The Design Patterns Smalltalk Companion. Reading, Mass.: Addison-Wesley, 1998.

[Beck-Cunningham, 1987] Kent Beck, Ward Cunningham. Window per Task. http://c2.com/cgi/wiki?WindowPerTask. 1987.

[Buschmann, 1996] F.Buschmann et al.. A System of Patterns. John Wiley & Sons, 1996.

[Cooper, 2000] James W. Cooper, Java Design Patterns, A Tutorial. Addison-Wesley, 2000.

[Feiler, 1998] Peter Feiler and Walter F. Tichy, Propagator--A Family of Patterns, in Proc. Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society, 1998.

[Gamma, 1993] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Abstraction and Reuse of Object Oriented-Design. Proceedings of ECOOP, Kaiserslautern, Germany, 1993: pp. 405-431.

[Gamma, 1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley Publishing Company, Boston, Mass, 1995.

[Grand, 1998] Mark Grand. Patterns in Java, Volume1. Wiley Computer Publishing, John Wiley and Sons, Inc, 1998.

[Grand, 1998] Mark Grand. Patterns in Java, Volume2. Wiley Computer Publishing, John Wiley and Sons, Inc, 1998.

[Grand, 2002] Mark Grand. Java Enterprise Design Patterns. John Wiley and Sons, Inc., 2002.

[Jeffery, 1998] Jeffery M. Saul. Beyond problem solving: Evaluating introductory physics courses through the hidden curriculum, Ph.D dissertation, 1998.

[Krasner, 1988] Krasner, G.E., and S.T. Pope. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. Journal of Object-Oriented Programming I(3), 1988.

[Larman, 1998] Craig Larman. Applying UML and Patterns. Upper Saddle River, New Jersey: Prentice Hall PTR, 1998.

[Michael, 1998] Michael C. Wittmann. Making sense of how students come to an understanding of physics: An example from mechanical waves. Ph.D dissertation, 1998

[PERG] http://www.physics.umd.edu/perg/ecs/phe.html

[Pree, 1994] Pree, Wolfgang. Design Patterns for Object-Oriented Software Development. Reading, Mass.: Addison-Wesley, 1994.

[TEAL] http://caes.mit.edu/research/teal

[Tichy, 1998] Walter F. Tichy, A Catalogue of General-Purpose Design Patterns, in Proc. Technology of Object-Oriented Languages and Systems (TOOLS 23), IEEE Computer Society, 1998