Extending UML To Visualize Design Patterns In Class Diagrams

Jing Dong and Sheng Yang
School of Engineering and Computer Science
University of Texas at Dallas, Richardson, TX 75083, USA
{jdong,syang}@utdallas.edu

Abstract

A design pattern describes a general solution to a design problem that recurs repeatedly in many projects. Software designers adapt the pattern solution to their specific project. Design patterns are usually modeled using UML. However, UML does not keep track of pattern-related information when a design pattern is applied or composed with other patterns. Thus, it is hard for a designer to identify design patterns in software system designs. The benefits of design patterns are compromised because the designers cannot communicate with each other in terms of the design patterns they use and their design decisions and tradeoffs. In this paper, we present the essential features of a new member of the UML language family that supports working with object-oriented design patterns. This UML extension allows the explicit representation of design patterns in software designs. We also discuss some of the relevant aspects of the UML profile which is based on standard UML extension mechanisms. A case study shows how it can be used to assist pattern-based software development.

1 Introduction

Design patterns [8] are commonly used in designing large-scale software systems. They have become increasingly popular among software developers since the early 1990s. A pattern is a recurring solution to a standard problem. It has a context in which it applies. Design patterns help developers communicate architectural knowledge, help people learn a new design paradigm, and help new developers ignore traps and pitfalls that have traditionally been learned only by costly experience. Design patterns are usually modeled and documented in natural languages and visual languages, such as the Unified Modeling Language (UML) [3, 13, 9]. UML is a general-purpose language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. It provides a collection of visual notations to capture different aspects of the system under development.

Graphical notations include diagrammatic, iconic, and chart-based notations. A graphical notation can be beneficial in many ways. First, it can be used for conveying complex concepts and models, such as object-oriented design. Notations like UML are very good at communicating software designs. Second, it can help people grasp large amount of information more quickly than straight text. Third, as well as being easy to understand, it is normally easier to learn drawing diagrams than writing text because diagrams are more concrete and intuitive than text written in formal or informal languages. Fourth, graphical notations cross language boundary and can be used to communicate with people with different nationalities.

The standardization efforts of the UML offer a chance to harness UML as notational basis for visualizing design patterns. However, the constructs provided by the standard UML are not enough to visualize design patterns in their applications and compositions. The model elements, such as classes, operations, and attributes, in each design pattern usually play certain roles that are manifested by their names. The application of a design pattern may change the names of its classes, operations, and attributes to the terms in the application domain. Thus, the role information of the pattern is lost. It is not obvious which model elements participate in this pattern. UML does not track pattern-related information when a pattern is applied in a software system. Using an example, we have shown that it is difficult to identify design patterns in a system containing five patterns in [6]. There are several problems when design patterns are implicit in software system designs: first, software developers can only communicate at the class level instead of the pattern level since they do not have pattern-related information in system designs. Second, each pattern often documents some ways for future evolutions, which are buried in system designs. Third, it may require considerable efforts on reverse-engineering design patterns from software system designs [11].

UML provides extension mechanisms that allow us to define appropriate labels and markings for the UML model

elements. In order to retain the pattern-related information even after the pattern is applied or composed, we define an extension to UML. In this extension, pattern-related information is explicit so that a design pattern can be easily identified when it is applied and composed. The extensions have been defined mainly by applying the UML built-in extensibility mechanisms, such as stereotypes, tagged values, and constraints. A case study shows how it can be used to assist pattern-based software development.

The remainder of this paper is organized as follows. In the next section, we provide a brief introduction to the UML built-in extensibility mechanisms. In Section 3, we present our extensions to UML in terms of a profile and provide an example. In Section 4, we use a case study to illustrate our approach. In the last two sections, we discuss related work and conclude this paper.

2 UML Extension Mechanisms

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. It is a multi-purpose language with many notational constructs. UML provides extension mechanisms to allow the user to model software systems if the current UML technique is not semantically sufficient to express the systems. These extension mechanisms are stereotypes, tagged values, and constraints.

Stereotypes allow the definition of extensions to the UML vocabulary, denoted by <<stereotype-name>>. The base class of a stereotype can be different model elements, such as Class, Attribute, and Operation. A stereotype groups tagged values and constraints under a meaningful name. When a stereotype is branded to a model element, the semantic meaning of the tagged values and the constraints associated with the stereotype are attached to that model element implicitly. A number of possible uses of stereotypes have been classified in [2].

Tagged values extend model elements with new kinds of properties. Tagged values may be attached to a stereotype, and this association will navigate to the model element to which the stereotype is branded. Basically, the format of a tagged value is a pair of name and an associated value, i.e., {name=value}. The tagged values attached to a stereotype must be compatible with the constraints of the stereotype's base class.

Constraints add new semantic restrictions to a model element. Typically constrains are written in the Object Constraint Language (OCL) [16]. Constraints attached to a stereotype imply that all model elements branded by that stereotype must obey the semantic restrictions which constraints state. Note that the constraints attached to a stereotyped model element must be compatible with the constraints of the stereotype and the base class of the model element.

A profile is a stereotyped package that contains model elements that have been customized for a specific domain or purpose by extending the metamodel using stereotypes, tagged values, and constraints. A profile may specify model libraries on which it depends and the metamodel subset that it extends.

3 The Proposed Extensions

This section introduces the UML extensions through an example. It summarizes the new stereotypes, tagged values and constraints, and presents a general description of their semantics. It also presents a description of how the UML extensibility mechanisms have been applied in the definition of a UML profile for design patterns.

3.1 Motivating Example

Figure 1 shows a system design that manages the connections to different types of databases, such as Oracle and MySQL (www.mysql.com). This system provides a connection pool for accessing each type of database. The connection pool restricts a limit number of accesses to a database and reuses connections to the database. The system has the capability to handle different types of database connections. The ConnectionPool class defines an interface for the creation of a connection pool for the appropriate type of database. The concrete classes, OracleConnectionPool and MySQLConnectionPool, use the createConnection operation to create the corresponding connections, OracleConnection and MySQLConnection, respectively. All connection instances have the same interface which is defined in the Connection class.

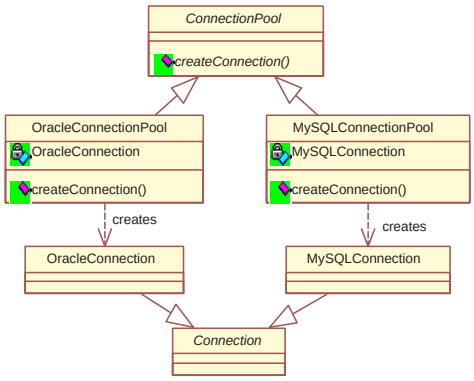

Figure 1. Connection Pool for Database

There are two design patterns, Abstract Factory and Sin-

gleton, applied in the system design[1]. The ConnectionPool, OracleConnectionPool and MySQLConnectionPool classes play the roles of abstract and concrete factories, whereas the Connection, OracleConnection and MySQLConnection classes play the roles of abstract and concrete products in the Abstract Factory pattern, respectively. OracleConnectionPool and MySQLConnectionPool are the Singleton classes, which restrict only a limited number of connections for each database.

While pattern-related information is not explicit in the class diagrams, e.g., Figure 1, without further textual explanation, several graphical notations have been proposed. Vlissides [14] described a Venn diagram-style annotation for identifying design pattern. Different patterns are differentiated with different levels of shading. All classes that participate in the same pattern are shaded with the same color. Although this approach may allow visualizing a small number of patterns, it is hard to scale up. Moreover, the overlapping regions may become hard to distinguish when some classes play several roles in different patterns. In addition, this notation cannot represent roles a model element, e.g., a class, an operation, and an attribute, plays in a design pattern.

UML Collaboration notation [9] has been proposed, which uses dashed ellipses with pattern names inside to denote patterns and dashed lines with participant names to associate patterns with their participating classes. The shortcoming of this notation is pattern information is mixed with class structure, making hard to distinguish both. The dashed lines may also twist together when the number of patterns increases. Furthermore, this notation can only represent roles a class, not an operation or an attribute, plays in a design pattern.

Vlissides [14] described a "pattern:role" annotation which uses a shaded box containing the pattern and/or participant name(s) to associate with the corresponding class. The drawback of this notation is that it cannot represent roles an operation (or attribute) plays in a design pattern. Explicitly representing this kind of information is very important because many patterns are based on polymorphism, delegation and aggregation, which are often presented based on the relationships among operations and attributes. Explicit representation of the key operations and attributes can not only help on the application (instantiation) of a pattern because the pattern impose some restrictions through the relationship among operations and attributes, but also assist on the traceability of a pattern since it allows us to trace back to the design pattern from a complex design diagram.

In [4], we have provided detailed descriptions of those previous notations and proposed the tagged pattern notation by extending UML. This notation can visualize different model elements. We have described the notation informally in [4]. In this paper, we will provide a formal description of the tagged pattern notation in terms of a UML profile.

3.2 UML Profile for Patterns

To explicitly visualize design patterns in class diagrams, we define a UML profile which includes three stereotypes (see Table 1): PatternClass, PatternAttribute and PatternOperation, whose base classes are Class, Attribute and Operation, respectively. Each stereotype also defines one tagged value as shown in Table 2. These tagged values define exactly what role a class, an attribute or an operation plays in a design pattern. The name of the tagged value is "pattern" and the value of the tagged value is a tuple in the format of <name:string [instance:integer], role:string>[2]. The "name" in the tuple is the pattern name in which a model element, such as Class, Attribute or Operation, participates. The name fields of PatternAttribute and PatternOperation can be omitted if the class plays a role only in one pattern, and this omission will not create any ambiguity. But the name fields of PatternClass is mandatory. Sometimes there is more than one instance of a pattern in the system, and it is useful to distinguish the instances. The "instance" in the tuple indicates the instance of the pattern the model element participates. The "role" in the tuple shows the role that a model element plays in the pattern. For instance, the OracleConnectionPool class plays the role of ConcreteFactory in the Abstract Factory pattern in the example shown in Section 3.1. Then, the stereotype <<PatternClass{<Abstract Factory[1], ConcreteFactory>}>> is branded to the OracleConnectionPool class, where the branded class participates in the first instance of the Abstract Factory pattern and plays the role of ConcreteFactory in the pattern. As another example, the OracleConnection attribute plays the role of UniqueInstance in the Singleton pattern in the example shown in Section 3.1. Then, the stereotype <<PatternAttribute{<Singleton[1], UniqueInstance>}>> is branded to the OracleConnection attribute, which participates in the first instance of the Singleton pattern and plays the role of UniqueInstance in the pattern.

A model element may simultaneously play different roles in different patterns. In this case, a new tagged value with the same format as "name[instance],role" is branded to the model element for each additional pattern it participates. For instance, the OracleConnectionPool class also plays the role of Singleton in the Singleton pattern in

[1] To illustrate our approach, we use this small example. It may be not hard, if not obvious, to discover the two design patterns used in this example. We have shown that it is not easy to identify design patterns in a larger system design with five patterns in [6].

[2] For simplicity, we omit the name of the tagged value such that each tagged value is represented by {<name[instance], role>} instead of {pattern = <name[instance], role>}.

Stereotype	Applies To	Definition
<<PatternClass>>	Class	Indicate that this class is a part of a design pattern
<<PatternAttribute>>	Attribute	Indicate that this attribute is a part of a design pattern
<<PatternOperation>>	Operation	Indicate that this operation is a part of a design pattern

Table 1. Stereotypes

Tagged Value		Applies To	Definition
Name	Value		
pattern	<name[instance],role>	<<PatternClass>>	Indicate that the attached class plays the role of $role$ in the $instance$ of a design pattern named $name$
pattern	<name[instance],role>	<<PatternAttribute>>	Indicate that the attached attribute plays the role of $role$ in the $instance$ of a design pattern named $name$
pattern	<name[instance],role>	<<PatternOperation>>	Indicate that the attached operation plays the role of $role$ in the $instance$ of a design pattern named $name$

Table 2. Tagged Values

the example shown in Section 3.1. Then, the stereotype <<PatternClass{<Abstract Factory[1], ConcreteFactory> <Singleton[1], Singleton>}>> is branded to the OracleConnectionPool class, where the branded class also participates in the first instance of the Singleton pattern and plays the role of Singleton. As another example, the createConnection operation also plays the role of Instance in the Singleton pattern in the example shown in Section 3.1. Then, the stereotype <<PatternOperation{<Abstract Factory[1], CreateProduct> < Singleton[1], Instance>}>> is branded to the createConnection operation, where the branded operation also participates in the first instance of the Singleton pattern.

The PatternClass, PatternAttribute and PatternOperation stereotypes and their tagged values are defined in Table 1 and Table 2, respectively. The constraints of these stereotypes are defined formally in OCL as follows:

```
<<PatternClass>>:
self.baseClass = Class and self.taggedValue -> exists
(tv:taggedValue | tv.name = "pattern" and tv.value =
"tuple<name:string[instance:integer],role:string>")
<<PatternAttribute>>:
self.baseClass = Attribute and self.taggedValue ->
exists (tv:taggedValue | tv.name = "pattern" and
tv.value = "tuple <name:string[instance:integer],
role:string>")
<<PatternOperation>>:
self.baseClass = Operation and self.taggedValue ->
exists (tv:taggedValue | tv.name = "pattern" and
tv.value = "tuple <name:string[instance:integer],
role:string>")
```

where the base classes of the PatternClass, PatternAttribute and PatternOperation stereotypes are Class, Attribute and Operation, respectively. Each stereotype has a tagged value with the name of "pattern" and the value of "name[instance],role". The types of "name" and "role" are string and the type of "instance" is integer. These stereotypes, together with their tagged values and constraints, form a new UML profile for design patterns.

3.3 Constraints

There are several constraints for the stereotypes and tagged values we defined in the previous section. For brevity, we only provide two constraints in this section. We refer to [5] for other constraints.

1. All pattern names found in the << PatternOperation >> and << PatternAttribute >> stereotypes should also be in the << PatternClass >>.

```
<<PatternOperation>>:
self.taggedValue->exists(tv:taggedValue,pc:PatternClass
| tv.value.name = pc.taggedValue.value.name)
<<PatternAttribute>>:
self.taggedValue->exists(tv:taggedValue,pc:PatternClass
| tv.value.name = pc.taggedValue.value.name)
```

2. In the tagged value with format as < name[instance], role >, the "name" field of the tagged value in << PatternClass >> is mandatory.

```
<<PatternClass>>:
self.taggedValue.value.name -> notEmpty
```

After applying the new UML extensions to the previous example, the new class diagram of the system is shown in Figure 2. Note that the instance fields of the tagged values are omitted for all stereotypes except in the Singleton pattern because there is only one instance of the Abstract Factory pattern and two instances of the Singleton pattern. From this diagram, we can identify the two patterns and their participants. For example, from the stereotype branded to the class OracleConnectionPool, i.e., <<PatternClass{<Abstract Factory, ConcreteFactory> <Singleton[1], Singleton>}>>, we know that OracleConnectionPool participates in two design patterns, Abstract Factory and Singleton. It plays the

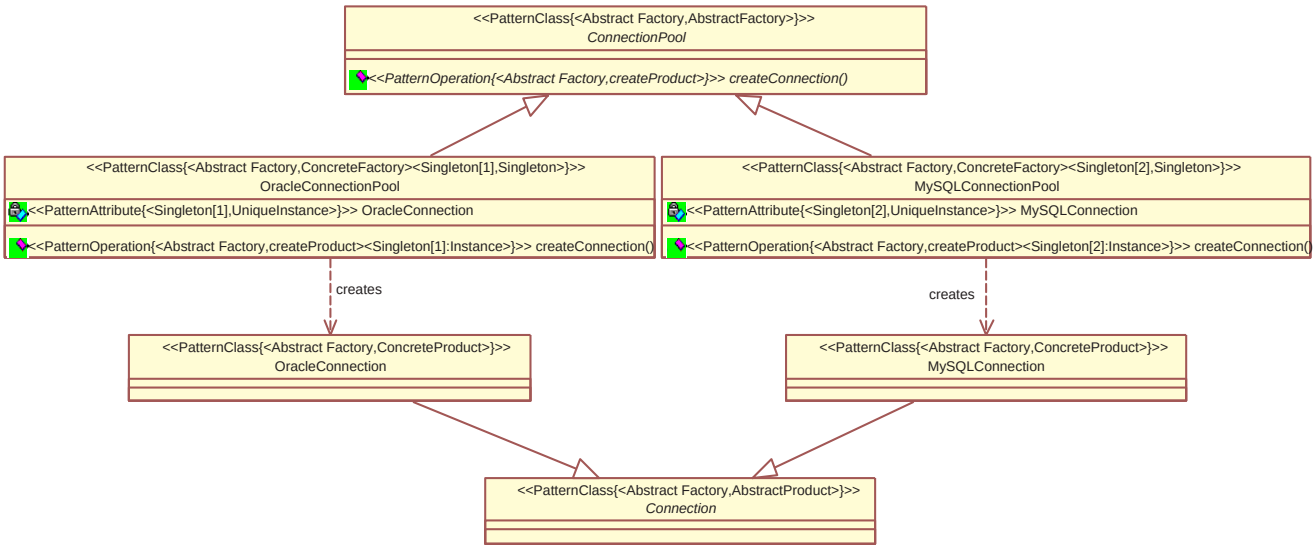

Figure 2. Connection Pool Modeled with the New Stereotypes

role of ConcreteFactory in the Abstract Factory pattern and the role of Singleton in the Singleton pattern. There is only one instance of the Abstract Factory pattern since the instance fields are omitted.

3.4 Virtual Meta Model

A virtual meta model (VMM) is the UML expression of a formal model with a set of UML extensions. A VMM can graphically represent the relationship among the newly defined elements, i.e., PatternClass, PatternAttribute and PatternOperation, and those defined by UML specification. The VMM for the newly defined extensions is represented as a set of class diagrams in Figure 3. The VMM represents a Stereotype as a Class stereotyped <<stereotype>> and a TaggedValue associated with a Stereotype as an Attribute of the Class that represents the Stereotype. The Attribute is stereotyped <<TaggedValue>>. The Attribute name is the name of the tagged value. The value of a tagged value is enclosed between "<" and ">" signs which means the format of the value is <name[instrance],role>. The multiplicity following the Attribute name ([1..∗]) indicate that the tagged value may have one or more values.

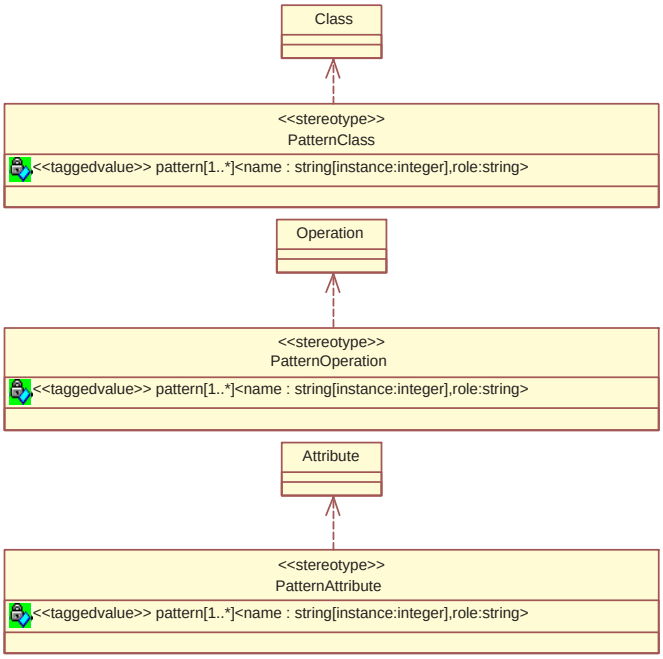

Figure 3. Virtual Meta Model

4 Case Study

In this section, we describe a case study to illustrate how to use the UML profile to visualize design patterns. This case study considers a software system for the management of student information in a school. The system allows the

instructor to search, review and update student information, such as courses enrollment and grades anywhere in and out of the campus. At the initial stage, the system needs to handle limited number of students due to budget limitation. If the student enrollment increases, the system is able to scale up easily with increasing budget. There are three design patterns used in this system: two instances of the Abstract Factory pattern [8], two instance of the Singleton pattern [8] and one instance of the Data Access Object pattern [1].

The Data Access Object (DAO) pattern encapsulates the connection to a database so that the user is able to access the database only through the DAO classes. The DAO pattern provides the flexible and transparent accesses to different database engines. Figure 4 shows the class diagram of the DAO pattern. The BusinessObject defines the business logic and uses the DataAccessObject to access any persistent data in a database or a flat file system. The DataSource represents a data source implementation, which can be a database or a flat file system. The ValueObject is a data carrier. The DataAccessObject encapsulates the access to the DataSource and the underlying data access implementation for the BusinessObject, which enables transparent access to the data source. The BusinessObject, as a client, requires data from the DataSource by using the DataAccessObject and does not know what the DataSource is. The DataAccessObject then returns the required data to the client using the data carrier, a value object. When the BusinessObject requires updates in the DataSource, the DataAccessObject will get the updated data from the client using the value object and update data in the data source.

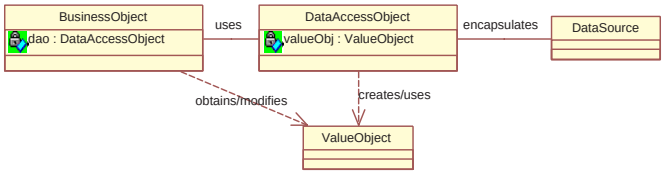

Figure 4. The Data Access Object Pattern

The DAO pattern is applied in this system for flexible and transparent change of database engines. Business data can be moved from an open-source database engine (e.g. MySQL) with low capability and price to a commercial one (e.g. Oracle) with high capability and price when the data processing demand increases, and vice versa. In addition, the application of this pattern prevents vender lock-in in that business data is free to move from one database to another transparently due to the merger or change of business and organizations.

Figure 5 shows the detailed system design with two instances of the Abstract Factory pattern, two instance of the Singleton pattern and one instance of the Data Access Object pattern. The QueryServlet and UpdateServlet classes play the role of the BusinessObject in the DAO pattern and define the business logic of searching and updating student information. The StudentInfo plays role of the ValueObject in the DAO pattern. It defines the functions, such as getStudInfo() and updateStudInfo(), and carries the student information either as search results returned to the QueryServlet or as the data prepared by the UpdateServlet and to be updated by the DAO class. The DAO class plays the role of DataAccessObject in the DAO pattern. It defines all the functions to manipulate the database, such as to get required data from the database and to update the required data in the database. To be able to move data between databases transparently, the DataAccessObject needs to decide what type of database to be accessed, and the Abstract Factory pattern is used to generate the appropriate type of DAO objects according to the kind of database engine used, e.g. MySQLDAO, OracleDAO and/or other database DAO objects, which provides the flexibility to add and delete a type of database transparently. At the initial stage, the MySQL database is used. The MySQLStudInfoDAO classes realizes the StudInfoDAO class to allow the access of student information to the MySQL database. Later on, when the database is changed to Oracle, OracleStudInfoDAO will be implemented to allow the BusinessObject to access and manipulate data in the Oracle database.

The DataSource in the DAO pattern provides the connections to the corresponding database engine, e.g., initially the MySQL database and later the Oracle database. To allow dynamic creation of the connections to the corresponding database engine, the Abstract Factory pattern is applied the same way as shown in Figure 2. Similarly, the Singleton pattern is applied the same way as shown in the example in Section 3.1.

5 Related Work

UML extension mechanisms have been used to expand the expressive power of UML to model and represent object-oriented framework [7], software architecture [12, 10, 17], and agent-oriented systems [15] when the original UML is not sufficient to represent the semantic meaning of the design.

Medvidovic et al. [12] applied the UML extension mechanism for modeling software architectures. They extended the UML to model software architecture in UML. Kande and Strohmeier [10] extended the UML by incorporating key abstractions in ADLs, such as connectors, components and configurations. They focus on how UML can be used for modeling architectural viewpoints. Zarras et al. [17] applied the UML extension mechanism for architecture description and provided a base UML profile for existing Architecture Description Languages (ADLs).

Fontoura et al. [7] proposed a UML extension, called

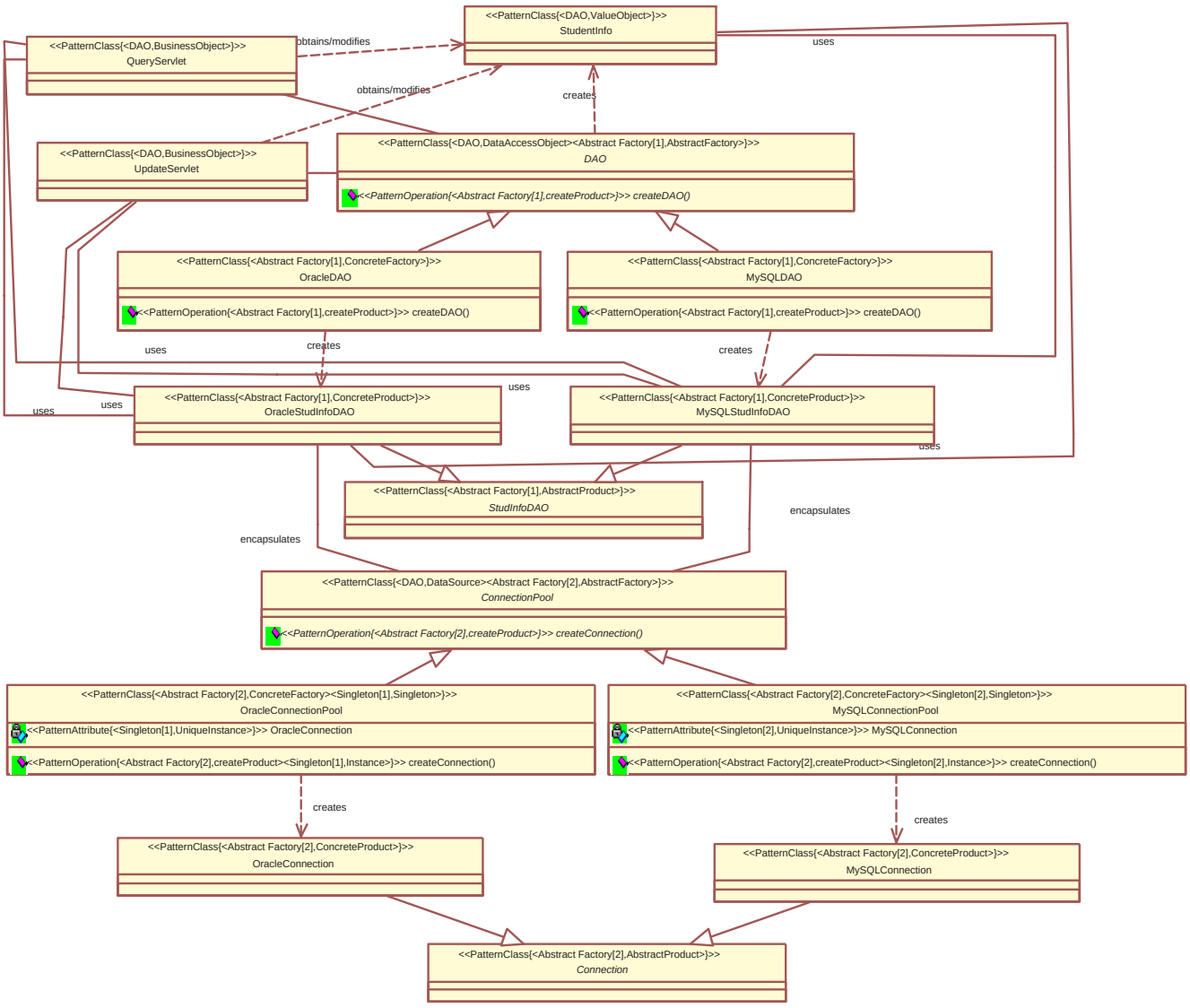

Figure 5. The Student Information Management System

UML-F, to represent object-oriented frameworks. The authors defined a set of new tagged values which can help to represent an object-oriented framework more meaningfully by UML. But the authors failed to give the complete UML profiles for the newly defined stereotypes and tagged values.

Wagner [15] applied the UML extension mechanisms for agent-oriented modeling. A set of new stereotypes are defined to model agent-oriented systems.

Our work uses the UML profiles and extension mechanisms to visualize the pattern-related information hidden in a class diagram. We defined new stereotypes, tagged values and provided the constraints applied to these stereotypes and tagged values. By using these new stereotypes and tagged values, the user can easily identify the patterns in a class diagram.

6 Conclusion

Standard UML is normally used to describe a design pattern. However, UML does not keep track of pattern-related information in system applications. In this paper, we introduced a UML profile for the explicit visualization of design patterns in system designs. It is important for designers to describe explicitly patterns in a design diagram because the goals of design patterns are to reuse design experience, to improve communication within and across software development teams, to capture explicitly the design decisions made by designers, and to record design tradeoffs and design alternatives in different applications. The application of a design pattern may change the names of classes, operations, and attributes participating in this pattern to the terms of the application domain. Thus, the roles that the classes, operations, and attributes play in this pattern have lost. This pattern-related information is important to accomplish the goals of design pattern. Without explicitly representing this information, the designers are force to communicate at the class and object level, instead of the pattern level. The design decisions and tradeoffs captured in the pattern are lost too. Therefore, the notations provided in this paper help on the explicit representation of design patterns and accomplishing the goals of design patterns.

Our approach uses the UML extension mechanisms to define a UML profile for visualizing design patterns. Three new stereotypes, PatternClass, PatternAttribute and PatternOperation, are defined. Each stereotype has a tagged value with the name of "pattern" and the value of the format as "name[instance],role". Several constraints are also defined. Using this new UML profile to model software system design in class diagrams, one can identify pattern-related information, such as how many design patterns are used in the system, what is the role of each model element, how many instances of a design pattern are applied, and where is each instance of design pattern in the class diagram.

References

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Sun Microsystems, 2001.

[2] S. Berner, M. Glinz, and S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages. *Proceedings of the Second International Conference on the Unified Modeling Language (UML), LNCS1723, Springer-Verlag*, pages 249–264, October 1999.

[3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[4] J. Dong. Representing the Applications and Compositions of Design Pattern in UML. *Proceedings of the ACM Symposium on Applied Computing (SAC), Melbourne, Florida, USA*, pages 1092–1098, March 2003.

[5] J. Dong and S. Yang. Visualizing Design Patterns With A UML Profile. *Technical Report UTDCS-11-03, Computer Science Department, University of Texas at Dallas*, 2003.

[6] J. Dong and K. Zhang. Design Pattern Compositions in UML. *Software Visualization - From Theory to Practice, Kluwer Academic Publishers*, pages 287–308, 2003.

[7] M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP)*, pages 63–82, July 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, 1995.

[9] Object Management Group. *Unified Modeling Language Specification, Version 1.4*. http://www.omg.org, 2001.

[10] M. M. Kande and A. Strohmeier. Towards a UML Profile for Software Architecture Descriptions. *Proceedings of the Third International Conference on the Unified Modeling Language (UML), LNCS1939, Springer-Verlag*, pages 513–527, October 2000.

[11] R. K. Keller, R. Schauer, S. Robitalille, and P. Pagé. Pattern-Based Reverse-Engineering of Design Components. *Proceedings of the 21st International Conference on Software Engineering, Los Angeles, USA*, pages 226–235, May 1999.

[12] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling Software Architectures in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology*, 11(1):2–57, January 2002.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[14] J. Vlissides. Notation, Notation, Notation. *C++ Report*, April 1998.

[15] G. Wagner. A UML Profile for Agent-Oriented Modeling. *Proceedings of the Third International Workshop on Agent-Oriented Software Engineering, Bologna, Italy*, July 2002.

[16] J. B. Warmer and A. G. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.

[17] A. Zarras, V. Issarny, C. Kloukinas, and V. K. Nguyen. Towards a Base UML Profile for Architecture Description. *Proceedings of the ICSE Workshop on Architecture and UML*, 2001.