The 7th International Scientific Conference
eLearning and Software for Education
Bucharest, April 28-29, 2011

USE OF STRUCTURAL PATTERNS IN e-LEARNING

Crina-Grabriela TIRON, Teodor-Ciprian SAVA
Faculty of Textiles, Leather and Industrial Management, "Gheorghe Asachi" Technical University, Bd-ul D. Mangeron, Iasi, Romania
ctiron@tex.tuiasi.ro, sava.ciprian@gmail.com

Costea BUDULAN
Department of Knitting and clothing, "Gheorghe Asachi" Technical University, MySecondStreet, Bd-ul D. Mangeron, Iasi, Romania
cbudulan@tex.tuiasi.ro

***Abstract**: Design Patterns are the problems that experienced designers have often met in practice and they do the same. These design solutions can be applied in different areas where that can highlight items that should be created, structured or interact according to certain rules. Structural Patterns, in number 7 are a subset of patterns whose purpose is to address the processes of composition of classes and objects (types and specific products). The paper explains how to use them in e-learning, both in the context of training for teaching and testing, and application-level software based on distance learning.*

***Keywords:** design, structural pattern, class, interface, object.*

I. INTRODUCTION TO DESIGN PATTERNS

A pattern records the design decisions taken by many builders, in many places, over many years, in order to resolve a particular problem [1]. The idea has been adapted for various other disciplines, including computing science (*Design Pattern* and *Architectural Pattern*) and education (*Pedagogical Patterns* = documenting good practices in teaching).

The range of situations in which the problems and solutions addressed in a pattern apply is called its context. The idea expressed in a pattern should be general enough to be applied in very different systems (a house, an e-learning software, a commercial product) within its context, but still specific enough to give constructive guidance. However, even a very abstract pattern will usually contain examples that are, by nature, absolutely concrete and specific.

There are 23 proposed solutions [2] that are independent of the programming language, and the program's application domain. Related to scope, design patterns are divided into three categories: creational, structural and behavioral patterns. These patterns are graphically described using UML (*Unified Modeling Language*), a standardized general-purpose modeling language in the field of *Object-Oriented* software engineering. Because of abstraction of UML and OOP, *structures* can be considered as a set of properties, *interfaces* – as a set of actions, and a *class* can be considered as a concrete type which has both structural (*data fields*) and behavioral (*methods*) characteristics.

II. USE OF STRUCTURAL PATTERNS

Structural design patterns concerned with how classes and objects are composed to form larger structures. According to OOP concept, structural *class* patterns use *inheritance* to compose interfaces

or implementations. Structural *object* patterns, on the other hand, describe ways to compose objects (*aggregation*) to realize new functionality [2].

A **Student** "IS A" **Person**, and a **Teacher** "IS A" **Person**, too (*inheritance*). A **Syllabus** "HAS A" **MethodologySection**, it "HAS A" **ReferenceSection** and so on (*aggregation*).

There is some overlap among these patterns and even some overlap with the behavioral patterns (not discussed here) [3]. Structural patterns occur in an intermediate stage between establishing the types of objects implementation functionality. It happens often in various fields, so that dynamic processes will be hampered (or might not even be able to perform tasks!) because of inefficient structuring. For example, the reliability of a software of educational management is dictated by the implementation of user specifications and ensure the data persistence. The first stage of static chart (the creation) is not difficult because of the educational process necessary items can be identified relatively easily. On the other side, the dynamic chart of the educational process is complex and simplify/optimizing its operation through repeated analysis of structural dependencies.

2.1 Facade

Frequency of use in programming: high (5 /1÷5). It is a single class that represents an entire subsystem [4]. Usually only one **Façade** object is required, so it is a **Singleton**. Subsystem classes handle work assigned by the **Façade** object, but have no knowledge about it.

This pattern should be regarded as a modular control panel to manage a large application module, which consists in several classes and interfaces having many dependencies between them. Thus, a software application for distance education to achieve: curriculum management (a façade object), student management (another façade object) etc. (Figure 1):

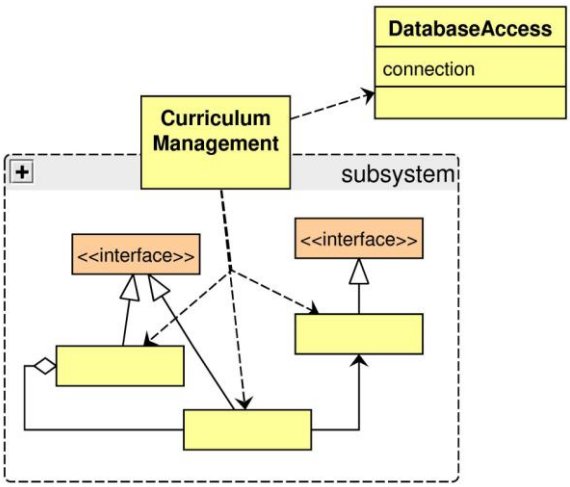

Figure 1. Façade pattern

A desktop application consist in a single object for the executable, that controls the main frame, database access (so an Façade); main frame is also unique and controls the pairs document/view for persistent/view data and other types of objects necessary (another Façade). Thus, desktop applications can be simplified as a tree of objects Façade.

Also the information activities and top-down control from a university is subject to this pattern pattern (a rector → deans; a dean →a department chief; a department chief → members); bottom-up communication breaks a restriction/constraint of a model **Façade**: elements of a subsystem should Not have knowledge of their representative.

If we ignore the *Object-Oriented Model*, the main page of a Web site is subject of this pattern: respect the uniqueness and is the access panel for different features offered by web site.

2.2 Composite

Frequency of use in programming: medium high (4 /1÷5). It is a tree structure of simple and composite objects; some objects may be nodes with additional branches and some may be leaves. A client manipulates primitive and composite elements through their common interface.

The most frequent use of this pattern occurs in graphics, whitch draw of a complex model is achieved by drawing primitive components. This solution may be applied to graphic representations (Views) for the documents with composite structures (e.g. DOM hierarchy of an HTML document and its visual representation in the browser). Generally, a typed material course includes paragraphs of text, images, equations –primitive objects – than can be grouped into sections etc. (Figure 2):

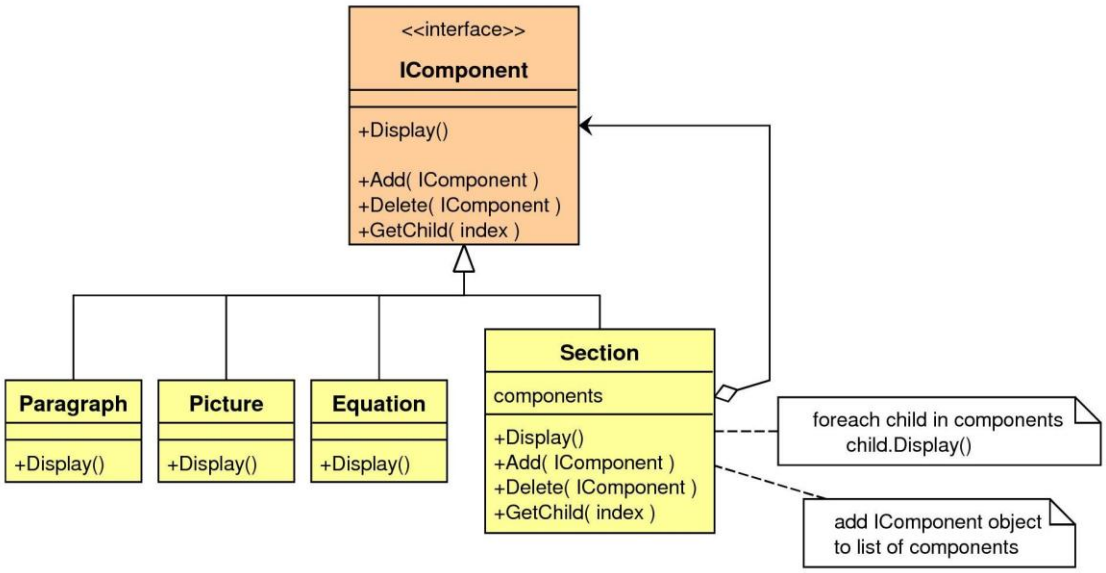

Figure 2. Composite pattern: treat primitive and composite elements uniformly

Another use is to represent the organizational plan of an institution, and scroll through the tree method can analyze a particular characteristic (Min/Max) or group (Sum). If salary is a common field for all employees (leaves: **Teacher**, **Lecturer**, **Engineer** etc.), each composite object (**IT Department**… **Faculty**) can use the special method to sum the salaries of its children (Figure 3).

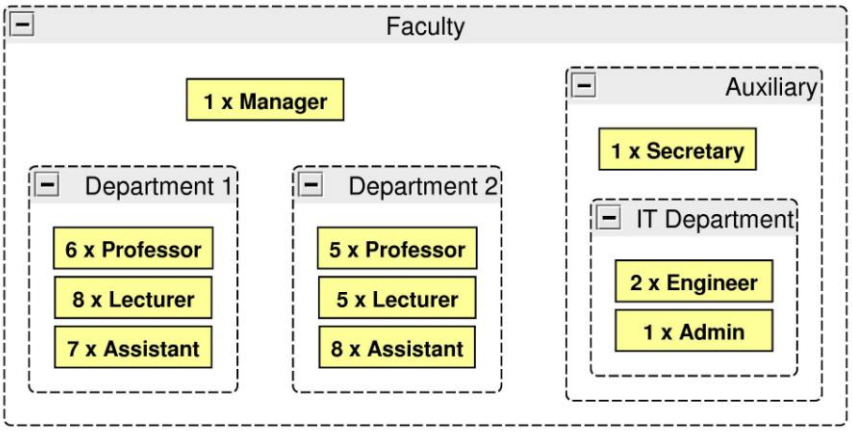

Figure 3. Composite pattern: organizational plan of a faculty

2.3 Proxy

Frequency of use in programming: medium high (4 /1÷5). Provide a placeholder for another object to control access to it, or (optionally) to add new functions [4]. There are 3 kinds of proxy:

- **Remote Proxy** is responsible for communicating with real object in a different address space. This kind of proxy meets technology for distributed applications (CORBA, JavaRMI); call is made by remote communication between the *client stub* (*proxy*) and *server stub* (*skeleton*).
- **Virtual Proxy** may cache additional information about the real object. For example a proxy for an MP3 audio resource can store additional information about the duration, artist, album etc.

- **Protection Proxy** checks that the caller has the access permissions required to perform a specific request. Connection to a database can be encapsulated by a proxy that restricts permissions to query data for a login session on a student account.

In any case, a proxy is an envelope for other object. When a **Researcher** sends a paper to be published (**RequestToPublish()** action from **Journal** interface) in a **ScientificJournal**, the editorial team is a **Proxy**, as an intermediary between the client and the real journal (Figure 4):

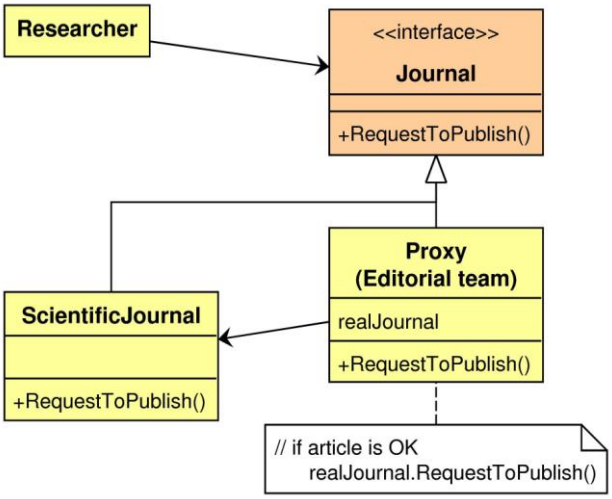

Figure 4. By proxy, the editorial team handles the requests for publication in a scientific journal

The editorial team receives the researcher's request, and then analyzes the article. If it is consistent with the scientific and editorial requirements, it will be considered for publication. In this example, the editorial team is a **Protection Proxy**, because of restriction for publication ("if article is OK"); furthermore, it can also be **Virtual** if additional information about the real object is extracted and stored (Figure 5):

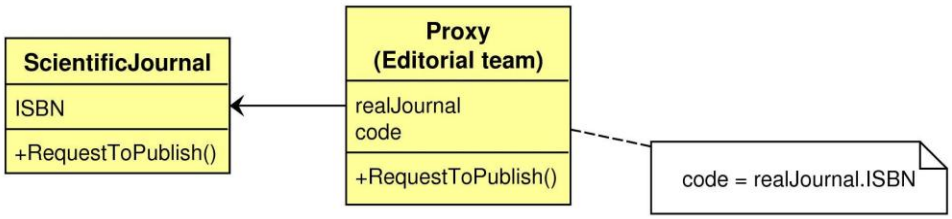

Figure 5. Virtual Proxy that caches additional data

2.4 Adapter

Frequency of use in programming: medium high (4 /1÷5). Also known as Wrapper, this structural pattern match interfaces of different classes. **Adapter** lets classes work together that couldn't otherwise because of incompatible interfaces [4].

This pattern is difficult to evidence outside the software domain, and should not occur in the design of an application, but later re-use of its libraries. It looks like **Proxy**, but hide functionality, not the object, as are similarities **Decorator** and **Bridge** patterns.

A simple example for **Adapter** is that complex calculation function can be implemented based on others already mentioned, but from another set of methods (Figure 6):

As we will show in next picture, **PersonnelManager** class implements **CountTeachers** function – inherited from **IPersonnelManager** interface – using existing functions from **ITeacherManager** interface.

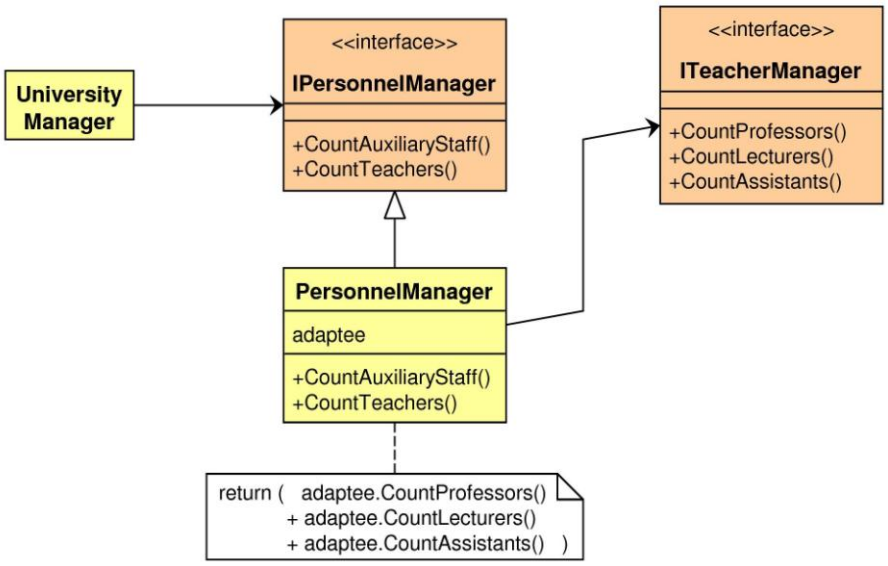

Figure 6. Functionality described by an existing interface is adapted to other

2.5 Bridge

Frequency of use in programming: medium (3 /1÷5). It separates an object's interface from its implementation (a kind of class **Adapter** pattern). The **Bridge** pattern is intended to keep the interface to your client program constant while both the class, as well as what it does, vary often [3].

It is used to create portable User Interface Toolkit (on different operating systems, or different browsers, and so on) and can be easily extended with new functionality, because all operations of the object's interface are high-level operations, which are implemented using low-level functions from the concrete classes (Figure 7):

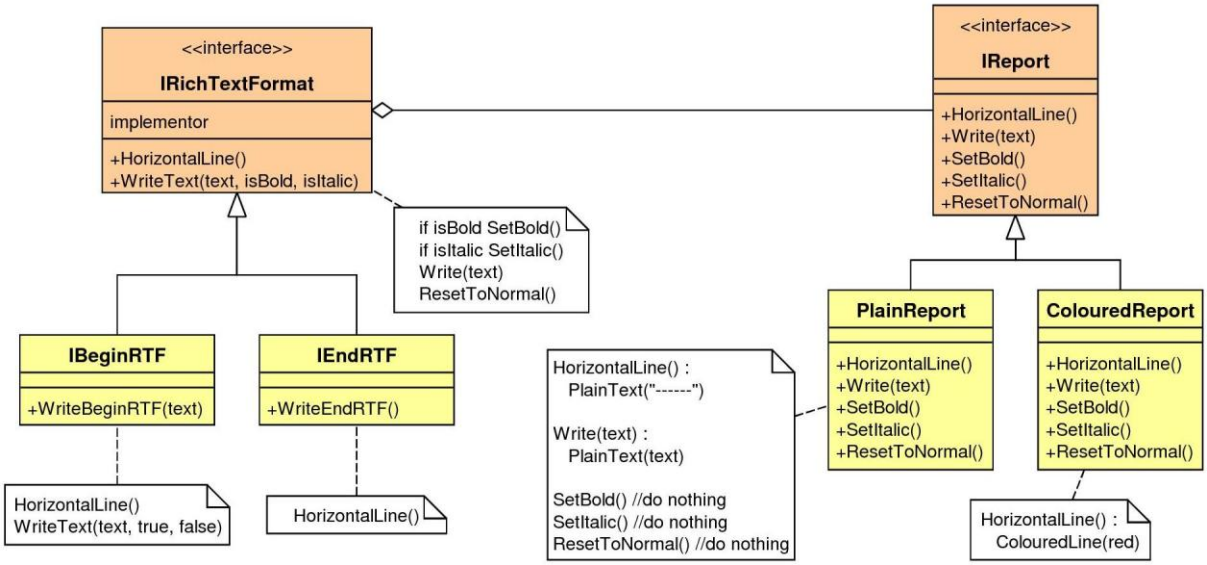

Figure 7. A detailed Bridge between RTF abstraction and report implementor

Adding new interfaces (left) or implementations (right) don't require any changes in diagram.

2.6 Decorator

Frequency of use in programming: medium (3 /1÷5). This pattern adds responsibilities to objects dynamically and provides us with a way to modify the behavior of individual objects without having to create a new derived class [3].

A **Decorator** is a wrapper for a visual or non-visual component. A component that implements a calculation for a test score online can be encapsulated in a **Decorator**; after calculation of the score, it sends a message to the user (Figure 8):

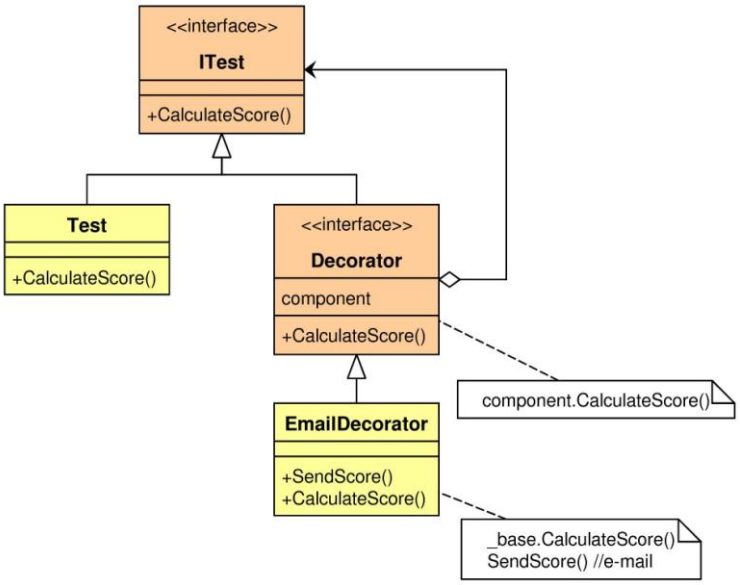

Figure 8. Decorator: after the score calculation, it will be sent via e-mail

2.7 Flyweight

Frequency of use in programming: low (1 /1÷5). It uses sharing to support large numbers of fine-grained objects efficiently. Flyweight is rarely used (e.g.: an application with facilities for editing and formatting text).

III. CONCLUSIONS

Design patterns influence the way design is done through the following ways: a common design vocabulary for all designers (to communicate, document, and explore design alternatives); documentation and learning aid (help designers to understand complex systems); an adjunct to existing method; a target for refactoring [2]. We use **Façade** pattern when we want to stratify our subsystems and **Proxy** when a plus of security or authentication is important. **Composite** helps us to construct complex structures and to manage them using a set of simple (primitive) objects having the same functionality. **Adapter** pattern is a trick to develop a new functionality based on an existing set of actions, while **Decorator** only changes object's responsibilities, not its interface. **Bridge** and **Flyweight** patterns are too complicated to be applied in non-programming activities.

References

[1] Alexander, C., 1977. A Pattern Language: Towns, Buildings, Construction, Oxford University Press, ISBN: 0195019199.

[2] Gamma, E., Helm, T., Johnson, R. and Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Professional Computing Series, ISBN: 0-201-63361-2.

[3] Cooper, J., 1998. The Design Patterns Java Companion, Addison-Wesley Design Patterns Series, ISBN: N\A.

[4] http://www.dofactory.com/Patterns/Patterns.aspx , accessed: 30 December 2010.