Patterns Instruction in Software Engineering

Cornelia Novac- Ududec[1], Diana Stefanescu[1]

„Dunarea de Jos" University of Galati, Romania

Keywords: Design patterns, software engineering, OOP design, multi-agent system, tutoring system.

Abstract:

The paper deals with a training system with a client-server architecture created for students who are learning to design and implement object-oriented software systems with reusable components, and also targeting those users who wish become accustomed with (or improve their skills related to) software design. The paper briefly presents the design patterns which are to be found in the software library, as well as the manner of applying them. The user of such a system has five types of patterns at his/her disposal (Factory Method, Builder, Command, Mediator, Observer), out of which will be able to chose one/more than one, depending on the requirements of the system proposed for design.

1. Introduction

Motto: "The software design patterns help you learn more from other people's successes than from your own failures" [Mark Johnson] .

A software system is a dynamic entity that undergoes multiple changes during its life-cycle, under the influence of factors such as: new user requirements, adding new functionalities, environment adjustment, etc. Therefore, apart from fulfilling those functions which it was designed to fulfill, a software application must be easily changeable, with some of its modules eventually re-used. Having this goal, the application must fulfill certain requirements regarding robustness, mobility, flexibility, scalability, etc. The software design principles are regarded as ways of obtaining the right and re-usable design. By applying abstractization and OOP languages polymorphic techniques, the open-shut principle and the substitution principle (Liskov) upon classes of problems, common solutions were reached – solutions which make up the so-called design patterns. The software patterns facilitate re-using design and architecture. If the design and/or architecture of a system were built with design patterns (techniques which have proven efficient), then they will be more accessible and easier to understand by those software developers who build new systems based on old design. The patterns help the software specialist chose that design that makes the system re-usable. In software engineering, "design pattern" (DP) [2] means a solution applicable to a certain type of problems. A design pattern is a description / model of how a problem should be solved. A pattern cannot be directly transformed in source code.

The purpose of design patterns is to enable a "good" object-oriented design, and more than that – to make it reusable (which is more important than reusing the code, regardless of the fact that reusing the design often implies reusing the code). The quality of software design is proportional to the experience and previous knowledge of the one performing it. An expert applies solutions previously used and proven efficient for similar problems. When the expert finds a solution considered acceptable, he/she uses it again. Design patterns solve specific design problems, making the OO more flexible, elegant and reusable. DP help beginners with software design, proposing patterns based on previous experience. A user who is familiar with the DP-s can apply them instantly in order to solve specific problems, without having to

rediscover them. By using the software design patterns while designing an object-oriented architecture, one is guaranteed to obtain system compliance to software engineering principles, while also preserving the qualities of "good design": reusable, flexible, robust [3].

2. Design Patterns

Depending on the DP purpose, the authors of GoF [4] proposed three main categories of patterns: creational patterns, structural patterns and behavioral patterns. These are the most well-known DP, but not the only ones. Other types of patterns emerged – more or less popular – as well as variations of the GoF patterns.
A pattern has four essential components:

1. **Name**: the name of the DP must describe in a few words a design pattern problem, its solutions and the solution's effects.
2. **Problem**: describes the situations when the pattern is applicable, explaining the problem as well as its context.
3. **Solution**: describes the components of the design, the relationships between them, the responsibilities of each component and the way they interact. The solution does not describe a specific design for a certain situation or a correct implementation, because a pattern has a general character, referring to a plethora of contexts. The DP presents an abstract description of a problem and a general arrangement of the components (classes and objects) that solve the described problem.
4. **Effects**: refer to the results of applying the pattern, which are very important when we evaluate the alternatives and we estimate the costs and benefits of applying the pattern. The effects of a pattern are also to be perceived in the flexibility, extensibility and portability of a system.

DP identifies the classes and instances which participate, their role, collaboration and responsibility sharing between them.

The design patterns that the user of the proposed system has at hand are: Factory Method and Builder (from the Creational Patterns category), Command, Mediator and Observer, (from the Behavioral Patterns category).

2.1. Factory Method Pattern

Factory Method, also known as Virtual Constructor, defines an interface for creating an object, but allows the sub-classes decide which one of them will be instanced. Factory Method defers to the sub-classes the instancing operation.
Factory Pattern is quite common to object-oriented programming. It returns an instance of a class out of many more classes possible, according to the data received. Ordinarily, all classes that can be returned have a common super-class and common methods, but accomplish tasks differently and accept different types of data. **Figure 1** presents the general structure of the Factory Method pattern.
In this diagram:

- **Product** is the interface of the objects created by `factoryMethod()`;
- **ConcreteProduct** implements the interface `Product`;
- **Creator** declares the method `factoryMethod()` which returns an object of the type `Product`. In some cases `factoryMethod()` can be a specific method which returns directly an object `ConcreteProduct`.
- **ConcreteCreator** re-writes `factoryMethod()` so that it returns an object `ConcreteProduct`.

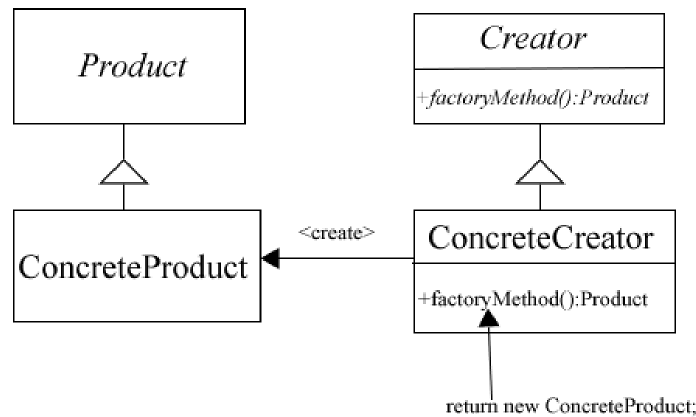

Figure 1. Factory Method Structure

The class `Creator` counts on its sub-classes, `ConcreteCreator`, for them to define the `factoryMethod()` so that it returns an instance `ConcreteProduct`.
The Factory Method DP should be used when:

- A class can't anticipate which should be instanced;
- A class uses its own sub-classes to specify which objects are created;
- The decisions of creating objects are localized.

Conclusions:

By using Factory Method:

- There is a separation of the code which represents the logic of the data program. The logic is established by Factory Method. By separating the logic from the program data, the extensibility of the programs improves;

- Creating an object by Factory Method is much more flexible than creating an object directly;

- The code of the `Client` class doesn't need to know the sub-classes, it works with the interface `Product`, so, if the programmer subsequently wants to add subclasses to `Product`, the class `Client` can use them without any alterations.

One disadvantage of the Factory Method is that a class `Client` will have to derive from the class `Creator`, just to create an object `ConcreteProduct`.

2.2. Builder Pattern

The Builder design pattern separates the process of building a complex object from its representation, so that the same building process can create different representations. **Figure 2** presents the general structure of the Builder as established by GoF.

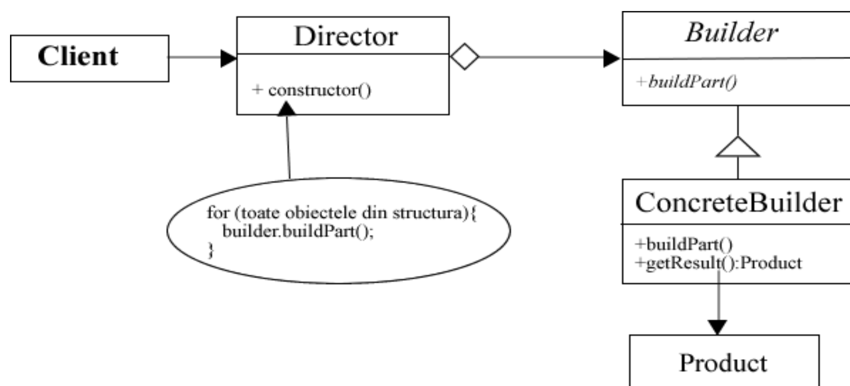

Figure 2. Builder Pattern Structure. Classes Diagram

In this diagram:

- **Builder** is the interface used to create the parts of the object `Product`;
- **ConcreteBuilder** implements the method `builderPart()` of building and assembling the parts of `Product` and offers an interface to obtain the object `Product` built;
- **Director** builds an object using the interface `Builder`, as this class is the one which knows the internal representation of the object to be built;
- **Product** represents the complex structure of the object to be built. `ConcreteBuilder` builds internal representation and defines the process of assembling the parts of the object `Product`. The abstract class `Product` includes classes which define the components, as well as interfaces used for assembling the parts with a view to reach the final goal.

This pattern is advisable when:

- The algorithm of creating a complex object must be independent of its components and of the process of assembling them;
- We need more representations for the same object which is to be built.

The Builder pattern can be compared to the Abstract Factory pattern (another one of the Creational Patterns category), as they both return classes consisting of a certain number of methods and objects. The main difference is that, while Abstract Factory returns a family of kindred classes, Builder builds a complex object, depending step by step on the data received.

2.3. Command Pattern

The Command pattern encapsulates a request in an object which is submitted towards a **specific** module that deals with the request. The object that encapsulates the request implements a public interface. Thus, a class `Client` is given the possibility to make requests without knowing anything about the operation to be executed, because `Client` uses the public interface. Also, action implementation code changes can be made without needing `Client` program to be changed.

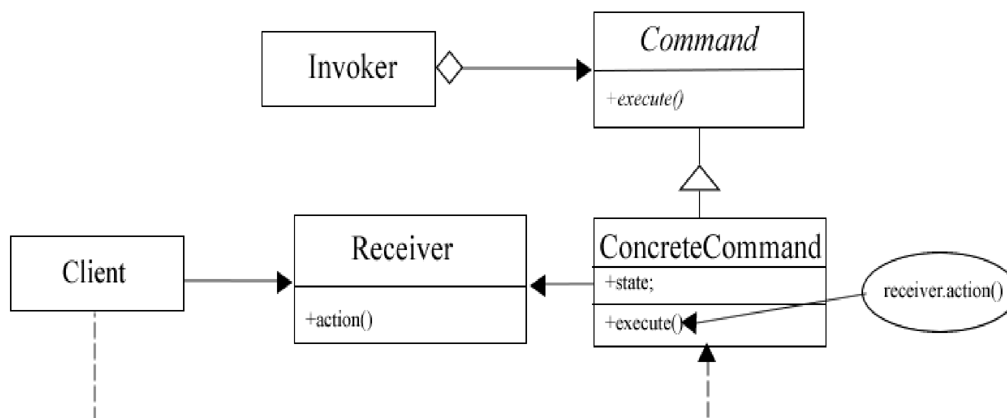

Figure 3. Command Pattern Structure

The classes of **Figure 3** are:

- **Command** an interface used to execute an operation;
- **ConcreteCommand**
  - Sets a connection between `Receiver` and an action;
  - Implements the method `execute()`, invoking the object `Receiver`, the method that performs the operation itself.
- **Client** creates an object `ConcreteCommand` and sets the object `Receiver`;
- **Invoker** asks the object `Command` to complete the request;

- **Receiver** is the class that implements the operations of completing the request. Any class can be a `Receiver`.

This pattern is advisable when:

- An object is called as having as parameter the action it is about to perform. For example, if there is a request to open a document, this request is used as parameter in the call;
- The incoming requests are placed in a pile and completed later, at different moments in time. For example, if one wants to copy a document in another document, one must open the first document, select the content, copy the content then Paste it into the second document, which must be open as well;
- The application offers the user possibilities like Undo-Redo;
- The application offers support for tracking changes in the system ("logging changes"), so that the system's status can be restored in case of failure. It is also possible to keep into a persistent log the actions performed against the system, so that - through Load and Restore operations implemented by means of the Command interface – to restore the status of the system. That restoration means loading the operations from the log and then re-executing them.

2.4. Mediator Pattern

When a system consists of more than one class, logic and algorithm are shared between the classes. The more the classes in a program, the more problems are regarding **communication** between them. If each class needs to know the methods of all the others, the structure of the program becomes very complicated, the program becomes harder to read and understand, and then, harder to maintain. The system will be hard to change, because a change will reflect upon the code of more than one class. The **Mediator** pattern solves this aspect by **reducing the coupling between classes**. Briefly, the classes inform the mediator when changes happen, then class `Mediator` informs all the classes that they must know the respective changes, as it will be the only one knowing the details of the methods of all classes, thus avoiding the class objects to call themselves directly. `Mediator` is a class that **encapsulates the interactions between the objects of the system**.
The structure proposed by the Mediator pattern is shown in **Figure 4**:

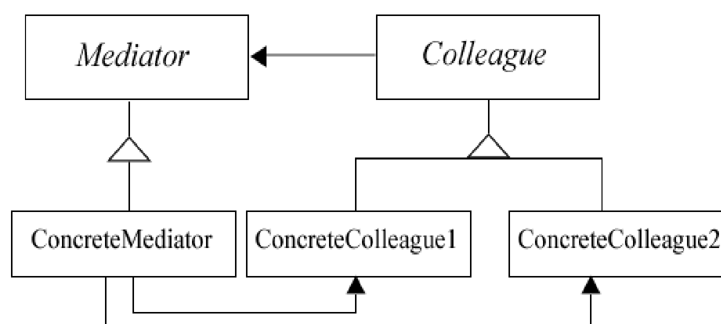

Figure 4. Structure of the Mediator Pattern

In this diagram:

- **Mediator** defines an interface for communication between objects of the type *Colleague*;
- **ConcreteMediator** implements the behavior of the object *Colleague*. A ConcreteMediator knows and coordinates all objects *Colleague*;
- Each class **Colleague** knows its *Mediator*, and each *Colleague* communicates with its mediator when it has something to say to its colleagues.

This pattern is to be used when:

- A group of objects communicates in a well-defined, complex manner;
- Inter-dependencies between objects lack structure and are difficult to understand;
- Re-using an object is hard because it refers/communicates directly with many objects;
- A behavior distributed to more than one class must be customized without using too many sub-classes.

Using the pattern Mediator is not limited to programs with visual interfaces, although this is the most common situation. It can be used in any situation that deals with a complex communication problem in a group of objects.

2.5. Observer Pattern

The Observer pattern defines an object inter-dependence of the type "one to more", so that, when one object changes status, the objects depending on it are automatically notified of the change and update their status.

The **Observer** pattern proposes a solution for obtaining system consistency, simultaneously with component objects being re-used independently. The objects are shared between **observers** and **subjects.** A subject can have more observers (objects depending on the subject status). As soon as the subject status changes, all observers receive automatic announcements. In response, each observer synchronizes its status with the subject's. **Figure 5** presents the general structure of the Observer pattern.

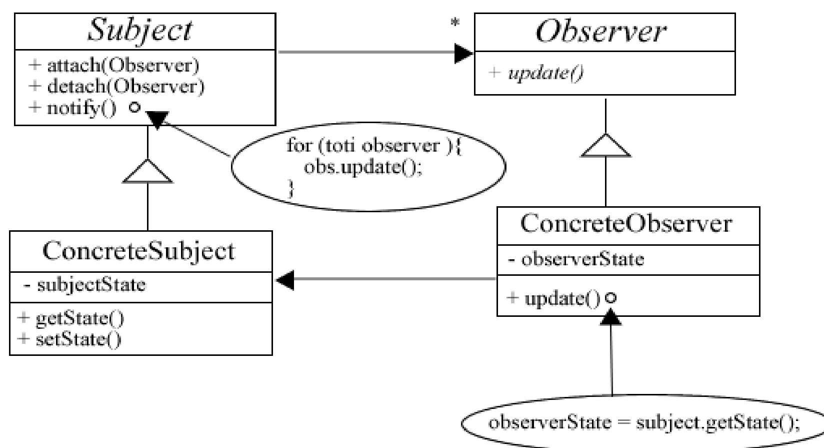

Figure 5. Structure of the Observer pattern

Classes participant in the Observer:

- **Subject** declares an interface for adding or extracting the observer objects.
- **Observer** defines an interface used to notify the observers when there are changes in the subject's status;
- **ConcreteSubject** preserves the status which is of interest for the observers and sends notifications when it changes;
- **ConcreteObserver** implements the interface used to update, keeps a reference to the subject and preserves the status which must be consistent to the subject's status.

Using the Observer pattern is recommendable in the following situations:

- When an abstractization has two aspects depending upon each other. Encapsulating these aspects in separate objects contributes to the possibility of re-using them in new contexts, independent of each other. Also, the system becomes elastic, allowing the modules to modify without a chain reaction of change;
- When an object must notify other objects without knowing who they are. In other

words, they are not characterized by strong, tight coupling;
- When the changing of an object leads to changes in other objects, without knowing how many objects will have to be changed in the end.

3. The Architecture of the System

The platform is designed as a distributed application, providing the option of distance learning/tutoring by means of a computer network or the Internet.
The application can be classified as having “three - tier" architecture, the Client part being represented by the configuration module and the tutoring material presentation module. The user connects to the system by means of a Login module, monitored by an Administrator agent (CerberusAgent).
After authentication the user may have access to a configuring instrument (if the user connected to the system is the tutor) or to the user interface by means of which the learning material will be presented (if the user is a student). This part of the application will be monitored by an interface agent (InterfaceAgent).
The application server contains the agent platform that monitors the student activity, and a web server responsible for delivering the learning material to the user interface (Apache Tomcat).
The third level consists of a xml database (Apache Xindice), which stores information about the structure of the course, student models, etc.
The MTS training system is designed for students which learn to design and implement object-oriented software systems, with re-usable components, but also for those users who wish to become familiar with (or improve) software design. We present the approach of a multi-agent intelligent platform for tutoring student which provides a Student Model. The “modeling student process” used the *overlay* architecture (common for many tutoring systems) combined with the Bayesian Theory of Probability [5].
The system’s tasks are distributed among the intelligent agents (software agents), each having clearly specified individual roles as following: administrator (Cerberus Agent); interface (InterfaceAgent); tutor (Tutor Agent); student (Student Agent). Moreover, the system has “three-tier” architecture and was implemented in Java language [5].
We may well imagine software agents attached to every application or tutoring environment, each having an explicit representation of the users’ objectives, of their plans and resources. These agents communicate and negotiate with each other in order to fulfill their individual or group objectives.
The multi-agent tutoring system (MTS) was designed and implemented within the Computer Science Department, in which research focused upon “modeling” the student with a view to identifying the most suitable methods and tutoring strategies by taking into account individual knowledge and abilities.
The user of this kind of system has all five types of previously described patterns at his/her disposal (Factory Method, Builder, Command, Mediator, Observer), out of which will be able to use one or more, depending on the requirements of the system that needs to be designed.
Depending on the needs of the user that must fulfill the requirements of the system, one selects from Tutor Agent the design patterns best suited for those functions. The patterns help build the menus of the application and the user interface. Then the database is built, as the user has permanent access to an example of application as similar as possible to the one which needs to be carried out. In order to develop an application, the student must know the Java programming language, must be familiar with MySql data servers and must be able to use the Eclipse platform. The Eclipse platform is an open source framework from IBM which aims to provide a variety of basic services for software developers [6].

4. Conclusions

The system wishes to be a useful working instrument for software application development training, for both students and other types of users. A design pattern together with an example of using that pattern may become a way of working in software design. Moreover, in this case applying design patterns – therefore design principles - in carrying out an object-oriented application results into a reusable **design** and reusable **modules / components**.
Also, the resulting code enjoys more **robustness and flexibility**, these two characteristics being essential for developing new modules that would add new functionalities and operations to the application; the design/code of the application is open to extension and closed for changes, according to OCP.
Another important aspect relates to the **maintenance activities** which the designed application will undergo along its life-cycle: they will **more cost effective**, as no change will disseminate through the entire system, the application code being built so that it isolates the volatile classes (according to the principle of reversing dependencies), the dependencies being in the way"details depend upon abstractization" and not the other way around.
The software patterns ensure compliance to design principles, as they contribute to obtaining a steady and efficient system, and they provide design discipline for students. Last but not least, using such a system in software engineering learning makes the process much more attractive and efficient. The design patterns approach in software engineering training fits the pedagogical Schemas Theory (created by Bartlett at the beginning of the 20th century) [1]. A brief representation of the particular knowledge of the training domain can help the trainee to easier absorb the new information, while the organization and structuring of the knowledge as patterns will guide him/her in finding (remembering) the necessary information in a certain context [4]. We believe that within such an approach one can create a training ontology from Software Engineering.

References:

[1] Brewer W.F., Nakamura G. V., The Nature and Functions of Schemas, Wyer & Srull, Eds. Handbook of Social Cognition, vol.1, pp.119-160, 1984.
[2] Design Patterns in C# and VB.NET- Gang of Four (GOF), http://www.dofactory.com/Patterns/Patterns.aspx.
[3] Design Pattern Tutorial- www.csc.calpoly.edu/~dbutler/tutorials/winter96/patterns/
[4] Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software (GOF), Ed. Addison-Wesley, 1995
[5] Novac-Ududec C., Barla R., A Multi-Agent Intelligent Platform for Student Tutoring, Proc. SEFI-IGIP 2007, Miskolc, Hungary, July, 2007, pp.65-69
[6] Eclipse Project http://www.eclipse.org/

Author(s):

Cornelia Novac-Ududec, Professor, PhD
„Dunarea de Jos" University of Galati, Department of Computer Science and Applied Informatics, 47 Domneasca Street, Galati, Romania
Cornelia.Novac@ugal.ro

Diana Stefanescu, lecturer, PhD candidate
„Dunarea de Jos" University of Galati, Department of Computer Science and Applied Informatics, 47 Domneasca Street, Galati, Romania
Diana.Stefanescu@ugal.ro