

OOPSLA 2001  
Workshop on The Three Tier Architecture Pattern Language  
POSITION PAPER

# Lessons learnt from developing and changing an educational three tier system

Letizia Jaccheri, Tor Stålhane, Marco Torchiano  
{letizia, stalhane, marco}@idi.ntnu.no

Department of Computer and Information Science, Norwegian  
University of Science and Technology, Trondheim Norway

Submitted to OOPSLA 2001

Workshop on The Three Tier Architecture Pattern Language

## Abstract

In the context of a software architecture course, we have collected experience and findings related to the architectural assessment and change of a small three tier system. In this paper, we present those findings, which may be of interest to persons working in the software pattern field.

## 1 The eCourse system

The educational project was centered on a three tier based system, called eCourse. eCourse enables its users to share documents and other information on the three tier. eCourse has been designed and implemented using the Rational Unified Process, Rational Rose, Enterprise Java Beans, and XML. eCourse provides support for document exchange, and was developed before the start of the course. Documents describing requirements, architecture, design, source code, and tests were made available to the students. eCourse is a relative small system (100 classes and 6000 lines of Java code). The students view the system eCourse from both user and developer perspective.

Students use the system to access documentation distributed by the teacher, to share documentations, and to deliver project results.

As developers, students assess, design, and implement changes to improve the architecture with respect to quality attributes (maintenance, performance, usability). Different groups are assigned to different quality attributes.

The goals of the project were [4]: make the students acquainted with methods concerning architecture evaluation and change, software design styles and patterns, and domain specific architecture. At the same time, we want students to experiment with architecture issues, e.g., evaluate and choose alternative designs for the same system, apply patterns, and above all experience the difficulty of changing software rather than build it from scratch.

The original system uses the following design patterns: Model-View-Controller (MVC), Front Component, Façade, Data Access Object, and Value Object.

## 2 Lessons learnt

Patterns were one of the driving concepts when the system was built. The system was build mainly following the guideline provided by Sun [7].

### 2.1 Model-View-Controller

The MVC pattern is a way of breaking an application, or a piece of an application's interface, down into three parts: model, view, and controller. The model represents an abstraction of the data, the view takes care of the presentation, while the controller handles the input and performs updates when required. In eCourse the MVC pattern has been slightly changed compared to its original version (see Figure 2). The model does not notify views when it changes the state. This is due to the asymmetric structure of the HTTP protocol.

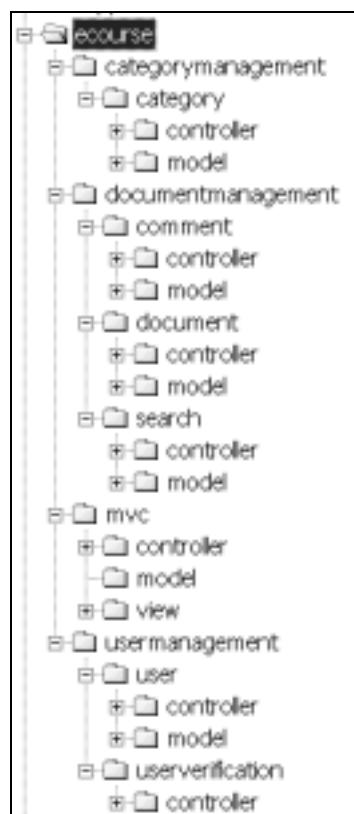


Figure 1: Java packages.

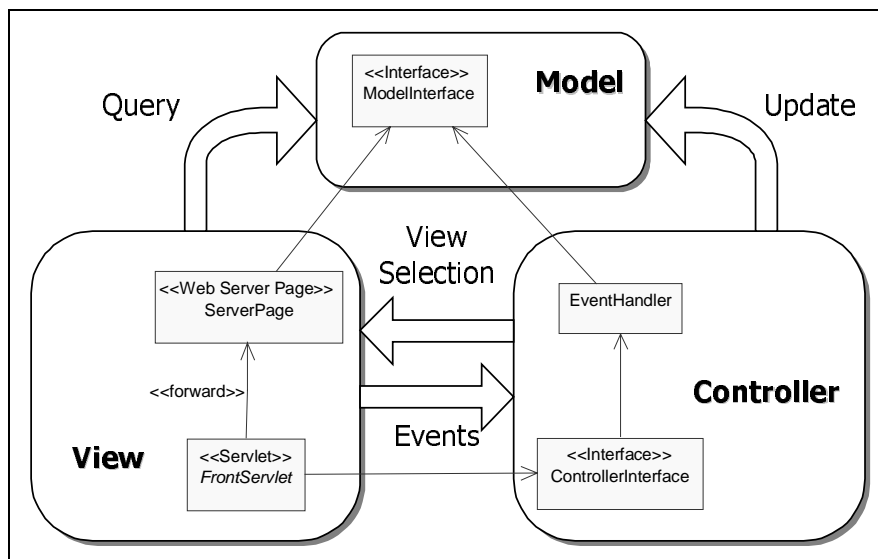


Figure 2: The MVC Pattern and its application in eCourse

In the eCourse system MVC is used in a fractal manner; it is adopted at both the architectural level, the implementation level and for package organization. (micro, macro and intermediate levels). The structure of the packages is shown in Figure 1.

**Positive findings:**

- The architecture, built according to MVC and 3-tier, is not affected by usability changes. This indicates that usability issues are well isolated by the MVC pattern.

- Once the MVC pattern is well known, it is easy to identify where changes should be made.

**Negative findings:**

- The MVC pattern introduces dependencies between the three components and a tight coupling between the viewer and the controller. Therefore, most changes must be applied in three different places. E.g. a change in the structure of the database can require changes in components in the other layers.
- The documentation is spread among many components instead of being packed in one single place making it difficult to understand the system.
- The MVC pattern tends to give a large amount of update messages between model and view components, resulting in poor performance. On the other hand, the MVC pattern is so deeply rooted into the system that avoiding its use (even in some case) appears to be impractical.

**2.2 Front Component**

A front component [7], [2] is a single point of entry for an Internet application. This means that all URL requests from clients (browsers) to the application are automatically directed to the URL of the front component. This component is then responsible for handling the requests. All information presented to the user is dispatched from the front component. This pattern decreases the dependencies in the system since all the client pages requests are transmitted to the front component. In addition, it represents a single point of entry for the system, and therefore makes it easy to enforce security policies.

**Negative findings:**

The front component does not have a clean design. Thus, the complexity which it is intended to control is concentrated in it the component itself. For this reason it becomes difficult to maintain.

Our negative findings may stem from a bad design of this system. On the other hand, it is always the risk that a component that has several relationships to several other components will become too complex.

### **2.3 Façade**

The Façade [3] pattern is intended to provide a unified and high-level interface to a set of interfaces in the model subsystem. The `ModelInterface` class represents the façade to the model sub-system.

#### **Findings:**

This use of this pattern may introduce a performance problem in the system. Students who should improve the system performance removed it from the system. This resulted in better performance. We will later make a new assessment to see if the maintainability decreased as we expect.

### **2.4 Data Access Object**

The Data Access Object (DAO) [7] or component [6] pattern introduces an extra layer of abstraction by encapsulating data access specific tasks. This makes the session Java bean independent of how data is accessed from the data source, and what data source is used. Different vendors of databases offer special versions of SQL. Having a DAO makes the session bean independent of the database used.

#### **Negative findings:**

Because of this pattern in conjunction with the Façade pattern, any change in the data layer were propagated up the layers to the model interface.

### **2.5 Value Object**

A value object is an internal representation or cache for a database object. The extensive use of value objects represents a threat to data consistency. The data cached in a value object can become inconsistent with the database.

#### **Positive findings:**

Forty students used the system intensively during a three months period. We never experienced consistency problems.

## **3 Short conclusions**

We have reported about some pattern related results of a bigger case study in the field of software architecture. Students applied the architecture transformation method proposed in [1] to improve an existing software system. We extracted some findings by carefully studying the documents they produced. We are now in the process of planning more formal experiments around the same system.

## **4 References**

- [1] Bosch, J., Design & Use of Software Architecture, 2000, Addison Wesley.
- [2] Buschmann, F. et. Al., Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons Ltd., 1996.
- [3] Gamma, E., Helm, R., Johnson, R., Vlissides, J., Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [4] Jaccheri, L., Software architecture project course, Forum for Advancing Software engineering Education (FASE), Volume 11 Number 06 (137th Issue) – June 15, 2001.
- [5] Krasner, G.E., Pope, S., A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1, 3, 1988, 26-49.
- [6] Matena, V., Stearns, B., Applying Enterprise JavaBeans: Component-Based Development for the J2EE Platform. Addison-Wesley, 2001.
- [7] Sun Microsystems. “Java™2 Platform, Enterprise Edition BluePrints”, 2001. URL: <http://java.sun.com/j2ee/blueprints/>.