Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 1725–1737
The Journal of Systems and Software
www.elsevier.com/locate/jss

Facilitating software extension with design patterns and Aspect-Oriented Programming

Konstantinos G. Kouskouras *, Alexander Chatzigeorgiou, George Stephanides

Department of Applied Informatics, University of Macedonia, 156 Egnatia Street, 54006 Thessaloniki, Greece

Received 20 February 2007; received in revised form 15 December 2007; accepted 27 December 2007
Available online 17 January 2008

Abstract

Software products, especially large applications, need to continuously evolve, in order to adapt to the changing environment and updated requirements. With both the producer and the customer unwilling to replace the existing application with a completely new one, adoption of design constructs and techniques which facilitate the application extension is a major design issue. In the current work we investigate the behavior of an object-oriented software application at a specific extension scenario, following three implementation alternatives with regards to a certain design problem relevant to the extension. The first alternative follows a simplistic solution, the second makes use of a design pattern and the third applies Aspect-Oriented Programming techniques to implement the same pattern. An assessment of the three alternatives is attempted, both on a qualitative and a quantitative level, by identifying the additional design implications needed to perform the extension and evaluating the effect of the extension on several quality attributes of the application.

Keywords: Object-oriented design; Aspect-Oriented Programming; Design patterns; Maintainability; Software metrics

1. Introduction

One of the most compelling properties of software products is their need to continuously evolve. In the case of large software products/applications, both the customer and the producer endorse this characteristic, since they both strive to obtain the most out of their investment, to purchase or develop the products, respectively. Maintenance is becoming a significant part of the software products' life cycle, as the organizations try to keep them operating for as long as possible (SWEBOK, 2004). Even more, most of the maintenance effort concerns adaptive rather than corrective adjustment of the software products (due to missed/revised requirements, customization) (Pressman, 2004). It is then evident that it is extremely important to build software in such a way that it can easily evolve (Bengtsson et al., 2004), to the extent that evolution paths can be foreseen. Understanding the factors that influence maintainability (i.e. ease with which software can be enhanced or adapted) of a system can help contain maintenance cost.

Researchers and practitioners have contributed several methods for building software that meets such expectations. Incorporation of design patterns when building software has been proposed as a way to improve software reusability and maintainability (Gamma et al., 1995). Refactoring techniques, aiming at improving code structure without altering its external behavior, have also been devised (Fowler, 1999). Lately, the Aspect-Oriented Programming paradigm (Kiczales et al., 1997) has been also been presented as a possible way to enhance an object-oriented system, by concentrating within a single entity (the aspect) code that would otherwise be scattered among several classes, thus adversely affecting maintainability.

Several works exist in the literature, related to the abovementioned methods. Some of them present aspect-oriented implementations of design patterns. For example, Nordberg (2001) suggests that aspect-oriented implementa-

* Corresponding author. Tel.: +30 6944 790944/2310 497347.
E-mail addresses: kkous@otenet.gr, kous@intracom.gr (K.G. Kouskouras).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.12.807

tion of certain design patterns may lead to better designs with greater effectiveness in anticipating future changes. Hannemann and Kiczales (2002) present aspect-oriented implementations of all 23 GoF patterns and comment on their modularity based on whether these implementations manifest properties like reusability and (un)plugability. Another category includes works attempting to compare object-oriented and aspect-oriented implementations of the same functionality. Papapetrou and Papadopoulos (2004) identify functionalities of a web-crawler that could be modeled as aspects and implement them following both approaches. Then, they compare the two systems focusing on the amount of changes and coding needed to add those functionalities. Tsang et al. (2004) report on another comparison, on the basis of class metrics adapted to aspects. Kulesza et al. (2006) use their metric suite (based on C & K class metrics and detailed in Sant'Anna et al. (2003)) in order to assess the maintainability of an object-oriented and an aspect-oriented implementation of a specific system.

Our purpose is to elaborate on the extensibility/expandability of software products, in relation with the two design practices mentioned above, i.e. design patterns and Aspect-Oriented Programming. We want to evaluate the different designs in terms of the ease with which a certain extension can be achieved and the influence this extension has on the system's quality attributes. To help us reach reliable conclusions we built a software application in Java, suitable for our investigation purposes, on which we could experiment. More specifically, we built an emulator of a telecommunications exchange, allowing the user to configure it with commands and to perform simple traffic cases.

A specific extension scenario was investigated in the current work. We assumed that new commands and associated parameters are added in the application. We employed three different implementation alternatives, with regards to the application part related to the specific extension. The first one follows obvious (at least to our minds!) and simplistic design decisions. The second makes use of a design pattern (the *Registry* Pattern (Sommerlad and Rüedi, 1998)), which is proposed as a means to overcome some limitations identified in the first alternative. In the third alternative we implemented the *Registry* pattern using Aspect-Oriented Programming techniques.

After implementing the same extension on all three alternatives we attempted an assessment of theirs by exploring the implications of the extension in each one. For each alternative this exploration was twofold. On one hand we qualitatively evaluated the ease with which the extension was achieved, i.e. whether the design facilitates/promotes/encourages such an extension or makes it cumbersome (from a design point of view). On the other hand, we quantitatively checked the effects of the extension on several quality attributes of the application, as reflected in metrics, in order to see whether the extension improves, preserves or has an adverse impact on these metrics.

The rest of the paper is arranged as follows. First, there is a short description of the application and the extension scenario chosen. In the following chapter the assessment framework is analyzed. Then, the first alternative is presented, commenting on metric results and other qualitative observations. In the next chapter the Registry pattern is introduced, along with the alternative implementation resulting from its application. A short description of the Aspect-Oriented Programming paradigm follows and next, the third alternative implementation is presented, with the Registry pattern being applied using the Aspect-Oriented Paradigm. Finally, some conclusive remarks are summarized.

2. The software application and the extension scenario

We chose the field of application to be the telecommunication industry because of the size and continuous evolution (due to standards' revisions and market adaptations) that systems of this area demonstrate. Furthermore, since telecommunication products are developed following specific software engineering processes, maintenance activities are always foreseen.

So, the application is an emulator of an imaginary telephone exchange. In its current form, it enables the user to

- insert definition commands (e.g. define a connection and assign a subscriber number to it) and
- emulate simple calls between subscribers.

From an implementation point of view the application was built in Java. It consists of modules (*packages* in Java) that correspond to specific logical (functional) areas. Regarding its size, it consists of about 50 classes and 1700 non-commented lines of code (before the extension).

As will be seen in the rest of the paper, we view the notion of module of high significance, since it is quite often considered as the basic element of object-oriented software reuse (Booch, 1994). It is seldom the case that a single class can be reused alone, when certain functionality is needed; most often there are closely cooperating classes that together fulfill the functional requirement. If the module[1] contents and structure are well chosen and properly designed, then it can be easily reused, whenever its functionality is needed. Furthermore, in large software systems the module could be seen as a functional area with well-defined boundaries and interfaces, such that it could be delegated as a whole to a specific group of people or organization (within one project or for all its lifetime), thus facilitating the splitting of responsibilities.

In the simple object-oriented implementation of the application, the main packages, which are to their largest extent similar among the alternative implementations, are as follows:

[1] Since we are referring to a specific application built in Java, we will use the equivalent term 'package' throughout the rest of the document.