INTRODUCTION TO COMPOSITE-ORIENTED SOFTWARE DESIGN

Zbyněk Šlajchrt
ICZ a. s., Na Hřebenech II 1718/10, 140 00 Praha 4,
University of Economics, Prague, Faculty of Informatics and Statistics, Department of Information Technologies
zslajchrt@gmail.com

ABSTRACT

This article briefly introduces an emerging architectonical style – composite-oriented design. To demonstrate its potential two other wide-spread architectonical styles (service-oriented and object oriented styles) are shortly presented with focus on their weaknesses. The article claims that these weaknesses are inherent and relate to dividing applications into tiers and layers. Composite-oriented design abandons the concept of tiers and offers a solution based on the concept of fragments that are local analogs to the global tiers. These fragments are reusable building blocks that are put together to form bigger autonomous units called composites. The fragments collaborate within the composite to implement the requested functionality. A part of the author's thesis is development of a Java language extension called Chaplin ACT, whose purpose is to introduce dynamic composition of objects by means of the tools and concepts of the Java language.

KEYWORDS:

Software architecture, object-oriented programming, SOA, multi-tier architecture

1. INTRODUCTION

There are two principal and de facto orthogonal architectonical styles that dominate in the domain of designing enterprise applications – the multi-tier (aka service-oriented) and the object-oriented styles. While the first considers a service as the key concept and promotes dividing the system into physically separable tiers and separation of data from the business logic, the latter favors keeping both the data and logic in a compact unit called object. For either style there are boundaries within which the one style thrives better than the other. However, there exists a certain domain of applications in which these styles suffer from some inherent drawbacks that cannot be solved easily with the framework given by either paradigm. This article briefly acquaints the reader with these issues and proposes a solution, which stems from an emerging architectonical style called composite-oriented design [1].

The main objective of this article is to explain the basic principles of the composite oriented architecture of software applications along with the motivations for it. Before delving into the ideas of the composite design the two above-mentioned and more or less antagonistic architectonical styles are briefly explained stressing theirs pros and cons. All design approaches are presented within the context of a simple web application for managing photos that is helping to illustrate the key traits of the approaches, such as extensibility and reusability. The author has intentionally chosen this application as a representative of the application domain, in which using the composite design can be the better choice comparing to the other presented approaches. This domain gathers the applications, where a smooth interaction with the user is the priority, in contrast to the applications, where the user’s presence is secondary. The key factor for these user-centric applications is that their behavior and responses accommodate to the user’s profile and needs. The user is not only an isolated consumer of services; however, he or she enters actively into the interactions with the

application and is becoming an active element of the system.

2. MULTI-TIER ARCHITECTURE STYLE

The name of this architectonical style prompts that the key characteristics of this architectonical style are tiers. Each tier provides a set of services that are utilized by the upper neighbor tier. A tier represents a certain domain in which a particular type of tasks can be solved. At the same time, this domain determines a specific vocabulary used for formulating problems and goals. Therefore, the communication between the tiers can be seen as a translation between two languages. Figure 1depicts the tiers, which an application can be composed of along with typical vocabularies used within them.

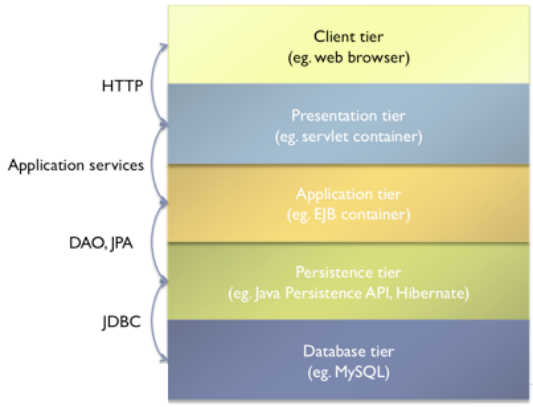

Figure 1: Multi-tier design

Let's remind the key virtues that are often associated with the multi-tier architecture:

- **scalabity**
- **reliability**
- **availabilty**
- **maintainability**
- **security**.

Of course, there are some limitations, among who's there belong to for example:

- **demanding administration**.
- **increasing chance of a failure of a node** as the number of nodes increases
- **worse response**
- **costs**

Let's return to the motivation application mentioned in the introduction. Let's inspect how we could design its architecture within the multi-tier paradigm. The following picture 2 shows the communication between the client and the application.

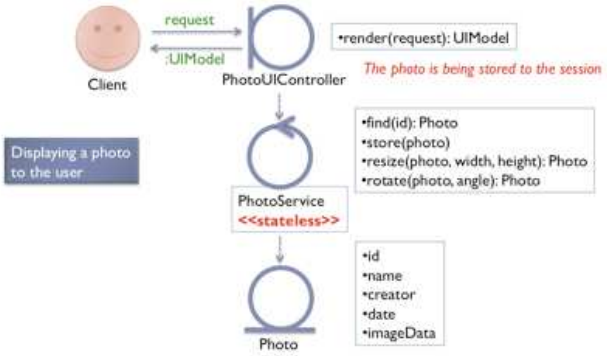

Figure 2: Three-tier design of the application

The scenario, in which the client sends a request for showing the page with a selected photography, proceeds as follows: the request is captured by component

`PhotoUIController`, which is looking for the photography in the database by means of method `find` of service `PhotoService`. If the photography is found the `PhotoUIController` constructs an HTML representation of the photo and returns it to the client in the form of HTTP response.

Though, this approach manifests some weaknesses. The first one relates to polymorphism. As long as the application is to support multiple types of photography, while there are some methods that behave differently with respect to the distinct photography types, the application tier must be aware of this difference in the implementation of each service that is affected by the difference, in contrast to the object-oriented approach, which allows hiding this difference through its natural support of polymorphism.

Let's look now at another scenario, in which we are trying to integrate the application with another one. Let's imagine it is necessary to integrate the functionality of our photo album with a student information system. It should be possible to open a student's photography from the student's page in the information system. The page with the photography should contain also the basic information about the student. Furthermore, the page contains a button for changing the format of the photography. It is natural to require that the integrator reuse the functionality of the photo album at the most. Figure 3 illustrates a possible schema of the integration.

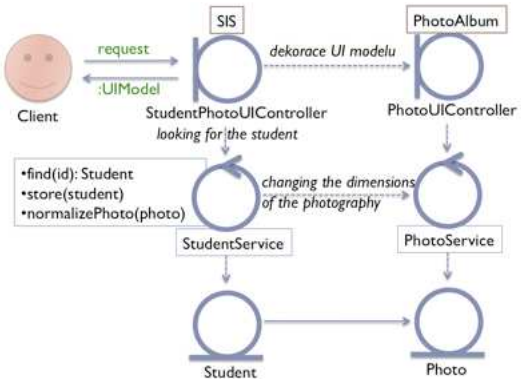

Figure 3: The design of the integration scenario in the three-tier design

The main weakness of this approach is a low reuse of the photo album's data domain.

If the data of the student information system and the photo album is stored in the same database, it would be efficient to fetch both student and photography related data by one database query. Furthermore, this single query would return only the data required for the pending operation. It is being shown that because of the strict separation of the two systems it is not possible at the same time to reuse the functionality of component `PhotoUIController` and to query for the necessary data by a single query. This is a general trait of strictly separated applications that communicate by means of services, which may cause some limitations in case of a need to share and reuse the data model.

3. OBJECT-ORIENTED ARCHITECTURE STYLE

It is interesting that the best practices and patterns used during developing applications within the framework of the multi-tier paradigm often contradict the best practices applied in the object-oriented design. Another interesting aspect is that the service oriented applications (i.e. layered applications) are being developed in object-oriented languages like Java or C #. The object orientation of these languages, i.e. their most important feature, is often used only marginally. In the service-oriented applications the key element is a service, i.e. a procedural element that processes and/or provides data. The data and the business logic are separated. On the other hand, the object oriented approach is exactly the opposite. The key element is an object that encapsulates both the data and the logic.

Let’s go back to the photo album application. The following figure 4 shows the schema of an object-oriented design for the application.

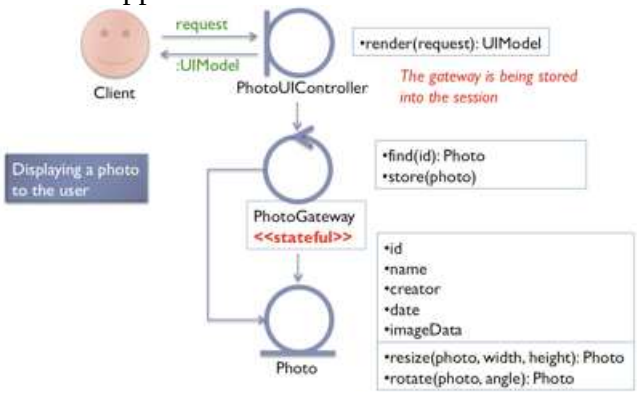

Figure 4: The object oriented design of the application

At first look, the schema is very similar to that of the multi-tier approach. The first significant difference is moving the business logic to the Photo entity. The second important difference is that the application keeps the state on the server in contrast to the multi-tier approach. While in the case of the multi-tier architecture it is not necessary to keep the state of conversation between the application and the client, this design stores the photo in the session on the server and every operation that the client invokes is performed on it. The state of this entity, i.e. the effect of the client’s operations, is kept in the memory until the client decides to store it back to the database.

In contrast to the multi-tier design, this approach does support polymorphism. Anytime it happens that a new kind of photography should be incorporated to the application, a new class is created and derived from the base class representing a general photography.

Let’s look now, how the object-oriented approach makes out the problem of integrating applications. (Figure 5)

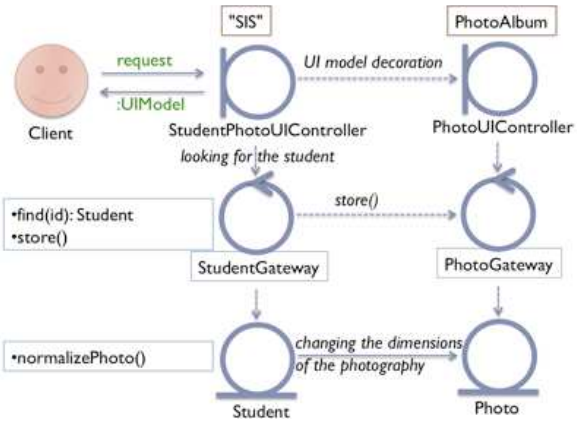

Figure 5: The object oriented design of the integration

The schema is very similar to the one shown in the case of the multi-tier approach. The key difference is that it contains stateful components and that the communication between the applications is carried out at all three tiers and not only at the topmost one. Unfortunately, it turns out that the object-oriented approach does not help either to resolve the problem of the simultaneous reuse of `PhotoUIController` component and the database model. The following chapter deals with an alternative approach called the composite design and that is able to resolve all the above-mentioned problems.

4. COMPOSITE DESIGN[1]

The composite design stems from the idea of special building blocks called *fragments*. A fragment can posses the typical object properties like identity, encapsulation, inheritance and polymorphism, however, it is not always necessary. A fragment does not have to be utilizable until it becomes a part of some other composite entity. It usually represents a certain narrowly defined aspect of composite's existence, for example a piece of data, a set of coherent operations, crosscutting concerns like logging, security, various constraints and so on. A fragment may also require a presence of another fragment in the composite for its correct functionality. A simple illustrative example of the composite design is here [5]. Some of these ideas can be implemented in dynamic languages like Python or Ruby [6], and also the Scala language provides a very useful concept of traits that is very close to the concept of fragments [3]. The Qi4J framework is attempting to provide a platform for designing statically composed applications in Java [4]. As a part of his thesis the author develops a Java language extension called Chaplin ACT, which is aimed at introducing **dynamic** composition of objects by means of the tools and concepts of the Java language [2].

In the case of the photo album we can identify two data fragments. The first represents the picture data itself while the other represents the metadata, like height, width, format etc. Furthermore we can identify two behavioral fragments. The first is for the operations on the photography and the other for creating the HTML presentation of the photography.

The next important concepts are that of *assembler* and *formula*. The assembler is a constructor of composites which builds them according to a given formula. The formula contains guidelines written in a special language for assembling a composite. (In some sense, the formula replaces the concept of class as it is used in Java, for instance).

The following picture depicts the schema of a composite design of the photo album (Figure 6).

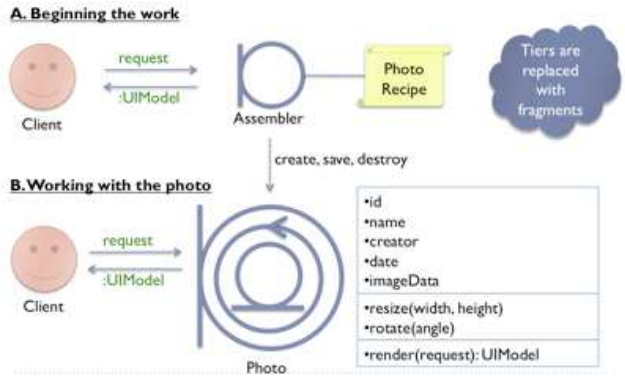

Figure 6: The schema of the composite oriented design of the application

The most distinctive trait of this design embodies in the absence of tiers. They are replaced by fragments. The scenario for beginning to work with the application performs as follows: the client sends a request containing the identifier of the photography. The assembler captures the request, which assembles a new composite according to the given formula. For the sake of simplicity, let's consider only three fragments: one data fragment `PhotoData`, one behavioral fragment `PhotoLogic` and one presentational fragment `PhotoUILogic`. The formula may look as follows:

[1] I intentionally use term *composite design* instead of the similar term *component design* to emphasise the fact that the composite is of primary concern in this approach. The traditional concept of component defines a component as an autonomous, independent and reusable unit providing a defined functionality and depending on other components through interfaces. The building blocks of composites in my approach – fragments – have a similar purpose, however, in contrast to the components, they are often not capable of an independent existence and they must be integrated to a higher unit, i.e. the composite, to provide their functionality.

1. *Formula parameter: photography identifier*
2. *Create a fragment* `PhotoData` *and initialize it from the database by means of the following query: „*`SELECT id, name, creator, date FROM photo WHERE id = %1`*“, where the unique parameter is the photography identifier.*
3. *Create instances of fragments* `PhotoLogic` *and* `PhotoUILogic`
4. *Compose all three fragment instances into one composite object.*

The fragment composition illustrates the following figure 7.

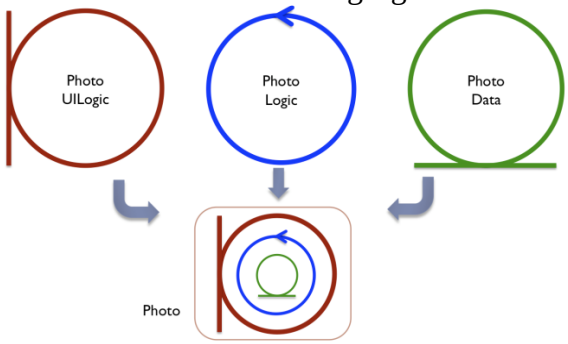

Figure 7: Composing the fragments

The composite is then stored into the session similarly as in the case of the object-oriented approach. Simultaneously, the assembler calls a predefined method for creating the default HTML presentation that returned to the client. The subsequent request will be routed directly to the composite as the assembler's responsibility is to manage the life-cycle of objects.

Let's try to find out, how to integrate the photo album with the student information system. It is the task which neither the multi-tier nor the object-oriented approach does gracefully because of impossibility to share the object model.

The schema of the integration is shown on the following figure 8.

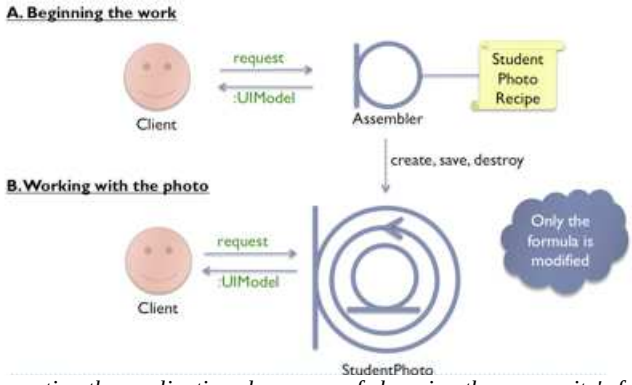

Figure 8: Integrating the applications by means of changing the composite's formula

It is clear, that this schema is practically identical with the previous one, but another formula and extended fragments. The extended fragments are depicted on the following figure 9.

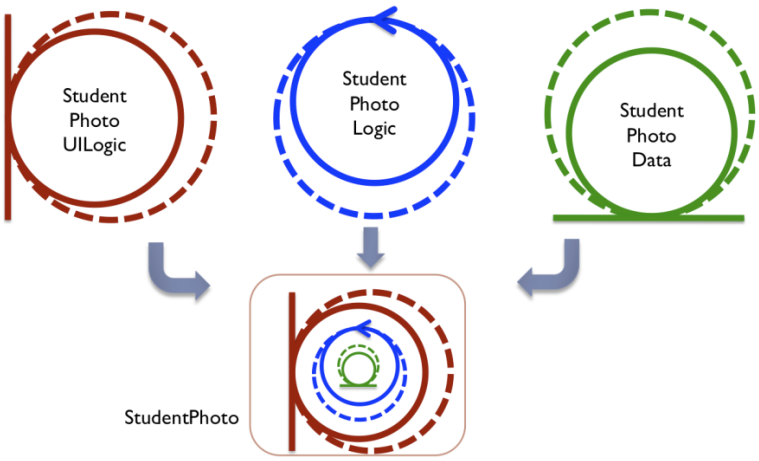

Figure 9: Extending the fragments

All three fragments for this integration scenario are derived from the original fragments by means of inheritance. The `StudentPhotoData` data fragment contains the student's personal data in addition. The `StudentPhotoLogic` behavioral fragment posses in addition the logic for modifying the size of the photography, while the `StudentPhotoUILogic` presentation fragment overrides the original method for generating the HTML representation of the photography and adds additional HTML elements.

The formula for the assembler can look as follows:

1. *Formula parameter: the photography identifier*
2. *Create an instance of fragment* `PhotoData` *and initialize it from the database by means of this query: „*`SELECT * FROM photo INNER JOIN student ON student.photoID=photo.ID WHERE student.id = %1`*“, where the unique parameter corresponds to the photography identifier.*
3. *Create instances of fragments* `StudentPhotoLogic` *a* `StudentPhotoUILogic`
4. *Compose all three instances into one composite object.*

In the query we used the JOIN clause for joining the both models. We can conclude now, that we have utilized at the most all contemporary components from the photo album application by means of inheritance and simultaneously we have achieved optimal sharing data models of both applications.

5. CONCLUSION

The main goal of this article was to explain the basic ideas of the composite design in comparison with two other architectonical styles – SOA and OOA&D – and in the context of a simple web application. In contrast to the two other approaches the composite oriented design is being evolved and is not very established yet.

It has been shown that for a certain domain of applications using the composite paradigm may be the better choice since the traditional approaches are not able to cope with the inherent problems such as uneasy extending the application with other types that inherit from existing entities (the multi-tier approach) and the problem with sharing data models between the integrated applications (the object oriented design).

The composite oriented design, which can be considered a generalization of the object oriented design, solves the illustrated problems and simultaneously offers an alternative view at the modeled system, in which the global tiers are replaced with local fragments as the building blocks for the composite structures in the application.

At present, the composite oriented approach can be applied with the help of the dynamic programming languages. The downside of this way is the lack of the static type system. The Qi4j framework allows the programmer to apply the static composition of fragments. As a

part of author's thesis is developing a Java language extension called Chaplin ACT, which is aimed at introducing dynamic composition of objects by means of the tools and concepts of the Java language.

REFERENCES

[1] Reenskaug, Trygve; Coplien, James O.: *The DCI Architecture*, http://www.artima.com/articles/dci_vision.html
[2] Šlajchrt, Zbyněk: *Chaplin ACT*, http://www.iquality.org/chaplin
[3] *Scala Language*, http://www.scala-lang.org/
[4] *Qi4j*, http://www.qi4j.org/
[5] Composite Oriented Programming, http://iridescence.no/post/Composite-Oriented-Programming.aspx
[6] Metaprogramming in Ruby and Python, http://codeblog.dhananjaynene.com/2010/01/dynamically-adding-methods-with-metaprogramming-ruby-and-python/