DESIGN PATTERNS FOR DYNAMIC COMPOSITE PROGRAMMING

Zbyněk Šlajchrt
AVAST Software a.s., zslajchrt@gmail.com

ABSTRACT:
In contrast to the classical object oriented programming (OOP), which in contrary to its name focuses more on classes than on objects, dynamic composite programming (DCP) works primarily with objects and their interfaces, while classifying objects into classes is regarded as circumstantial. The main goal of this article is to present a catalog of design patterns that similarly to OOP offers a set of verified guidelines and constructions during applying DCP. The catalog is compiled from experiences gained when developing and experimenting a DCP tool Chaplin. Some patterns have been adopted from OOP and others have been discovered and described during applying DCP in practice.

KEYWORDS:
Dynamic composition of objects, DCP, COP, object oriented programming, OOP, aspect oriented programming, Chaplin ACT.

INTRODUCTION

The main motivation for dynamic composite programming (DCP) is maximizing code reusability. Today programmers have to solve a paradoxical situation whenever deciding to develop either a simplistic form of a component without any additional utility code or a full-blown component with a pile of auxiliary routines. If they decide for the first approach the product is an easily reusable component, however using it will require a lot of boilerplate code. On the other hand, deciding for the latter approach will result in a component that can be very easy to use in a particular environment; however its reusability will be probably very low because of tight couplings in the auxiliary methods.
DCP tries to cope with the above-mentioned problem by introducing the notion of fragment, which is an indivisible entity similar to object, however, in contrast to object it does not have to have identity nor be self-reliant. Often, a fragment is not usable until it forms a composite object together with other fragments. Next, the dynamic nature of this approach allows assembling and disassembling objects on the fly, according to client needs, for example. Practicing DCP has led to identification of recurring design patterns. The following text presents the catalog of a subset of these patterns.

CATALOG

Using composite programming has shown that there are several recurring design approaches that can be generalized and compiled into a catalog of reusable guidelines or best practices. During the development of the Chaplin DCP tool (Šlajchrt, 2009a) the author has identified a set of design patterns. Some of them have been used in practice for many years while others are new and specific to the field of DCP. In the following paragraphs some of them are presented.

INHERITANCE

In order to highlight proximity of DCP and OOP I intentionally selected inheritance as the first pattern. In OOP inheritance is not considered a separate design pattern, since it is treated as the fundamental relationship between classes and which is employed in more complex

design patterns. In DCP inheritance is not seen so fundamentally, instead it is taken merely as a special arrangement of object fragments within a composite that can be a combination of three patterns: method overriding, the template method (Gamma, 1994) and extension (Šlajchrt, 2009b).

THIN/RICH INTERFACE

With the help of this pattern (Odersky, 2010) it is possible to solve the dilemma whether to develop a "thin" or a "rich" interface. A thin interface offers only a small set of canonical methods for using the object. On the other hand, a rich interface not only offers the canonical methods, but also a broad range of additional auxiliary methods for a user of the object. Both approaches have their own pros and cons: a thin interface is easy to implement, however, its use is more complicated since the programmer must often develop a lot of additional code. The pros are for example the ease of replacing it for another interface while making only minor changes in the code. Contrary to the thin interface, the rich interface makes the life of the developer, who implements it, harder since it must implement a lot of methods. On the other hand, the developer using the interface has its task easier since a lot of code is already done and provided by the interface. However, because the using code is tightly coupled with the object through the auxiliary methods, replacing the object for some one else is much more difficult.
In composite programming, it is very easy to solve this dilemma by introducing two types instead of one: the first – a thin interface – will contain only the canonic methods while the other – a rich abstract class – will contain all auxiliary methods using the methods from the thin interface. The abstract class actually declares that it implements the thin interface, however, all the thin interface methods leaves unimplemented. The goal is to compose an instance of the abstract rich class with an adequate instance of the thin interface into one composite object at runtime. An advantage of this dynamic composition in comparison with the static composition (Öberg, 2009) is the possibility to define the arrangement of the composite at runtime, eventually to modify its structure dynamically.

INTERCEPTORS

Interceptors allow capturing method invocations and performing some actions between or after invoking the method itself. This is approach, being the domain of aspect oriented programming (Dessi, 2009), is shown advantageous in situations in which we need to separate business logic from accompanying activities like logging, transaction management, input or output validation, security etc. These activities can be categorized into the following groups:

- Side effects – Activities that have no direct relation to the business logic of the wrapped method. Here belongs for instance logging, authorization or transaction management.
- Concerns – Activities that are directly connected to the business logic of the wrapped method.
- Constraints – Declarative restrictions on method invocation like patterns or schemas for input and output values.

In DCP interceptors can be stacked and added to or removed from the stack dynamically. Each interceptor is an object fragment that "wraps" a method and delegates the control to it by means of a special operator.

CHAMELEON

Chameleon is a design pattern closely connected with the concept of dynamic composition. By means of this pattern we can get an object's behavior to adapt to the context in which it is being invoked. The accessed object becomes during the invocation a part of a context composite. Thus it is possible to change temporarily the object's behavior by presence of attributes and methods in the composite overriding the corresponding attributes and methods in the object being invoked.
The Chameleon pattern can be also used for implementing the Thin/Rich pattern or the template method. The object being invoked implements the thin interface and calls an auxiliary method on the composite. The composite than calls back the object's thin interface method.
This pattern is suitable for situations where various clients call the same service, however, its behavior must be customized for a client. Instead of routing the request to the service directly, it passes through a special object representing the client. This client object becomes temporarily a part of the context service composite. The presence of the client object in the service composite will influence the service's behavior so that it can handle the client's request in an appropriate way.

ZOMBIE

In the Thin/Rich pattern both thin and rich objects can live separately until they are fused to a composite. However, while the thin object is fully functional outside a composite, the rich part is not. It needs its thin object counterpart to become fully functional. Until the two form a functional entity – composite – the rich object is unusable. It does not mean that we cannot access the rich object if it is outside a composite. It is possible, for instance, to initialize its attributes, i.e. its state. I call these objects, which are alive (they have some state), however, non-functional, *zombies* or *pseudo-instances*. A zombie cannot be invoked before it becomes a part of a composite.

COMPOSITE FACTORY

Composite factory is a pattern, by means of which it is possible to compose more factory objects into one. The product of such a composite factory is a composite of products of individual factories forming the composite factory. This pattern can be used for producing composites having a pre-defined structure like a composite interceptor that is responsible for both logging and transaction processing.

AGGREGATOR

DCP allows that a composite object contain more fragment objects implementing the same interface. Since the composite is able to expose only one interface of one type to a client it is necessary to specify the behavior of the composite in case there are more instances of the same interface within.
If the client invokes a method on the composite that belongs to an interface implemented by more fragments, the method is simply invoked on all such fragments. As long as the method returns a value there must be some mechanism for aggregating multiple values returned by the fragments. In this case DCP allows insert an *aggregator* fragment into the composite that is responsible for both aggregating the returned values and determining the order of invocations. DCP defines one implicit aggregator that merges all returned values into one composite.

RECIPE

This pattern represents guidelines for assembling a composite. The key role in this pattern plays the *assembler*. The basic idea is that the assembler gets a recipe by means of which it assembles the required composite. The recipe can be specified in various programming languages suitable for the given domain. (Using domains specific languages (DSL) can be advantageous here as they are tailored for solving tasks in a certain domain.) A recipe may contain instructions like: *"Take the first parameter from the request and put it as the sole argument to the given SQL SELECT query. Use the only result row from the query execution for initialization of an instance of class Use. Next, create zombie instances of UserPresentation and UserValidator. Then compose all three fragments into one composite and return it."*

BACKBONE

This design pattern allows enacting order in the structure of a composite. It is possible to see it as both a schema for the composite and a protocol that governs the interactions among fragments. The backbone defines which methods can be called by fragment within the composite (*context services*) and to which events can a fragment respond (*context events*). The backbone is defined by an interface. This interface is intended for internal use within the composite only and clients of the composite are unaware of it. The backbone is supposed to be introduced as long as the structure and interactions are known in advance. It is often the case when building elements of a graphical user interface, for instance according to the Model-View-Controller pattern (Reenskaug).

OTHER PATTERNS

There are other DCP patterns, however, the limited scope of this article does not allow me to deal with them individually in detail. Thus, I am presenting a selection of patterns that I have identified during developing and experimenting with DCP:

- **Nesting** – A pattern for creating hierarchies of composites
- **Component proxy** – Every fragment can be a part of at most one composite. If it is necessary that one fragment belong to more composites it is possible to wrap it into a *proxy* fragment and insert this proxy to a composite instead of the original fragment itself.
- **Default fragment (singleton)** – In DCP it is possible to define an implicit (default) fragment that is used when some fragment is missing in a composite.
- **Enricher** – It serves for dynamic enrichment of persistent entities, for instance, of validation or authorization functionality.
- **State Machine** – It allows defining the lifecycle of a composite's structure. This lifecycle is defined with the help of a finite automaton (state machine).

CONCLUSION

Dynamic composite programming brings another dimension into object-oriented programming. The main motivation for DCP is maximizing code-reuse and overcoming the limits of OOP. An inseparable part of it is the catalog of design patterns that makes the development easier thanks to the guidelines and best practices. Some of the pattern have been taken from OOP practice and adapted to DCP, while others have been identified during

developing and experimenting with DCP and the tool for implementing DCP – Chaplin ACT. The catalog is still under construction and should be finished at the end of this year when I plan to submit my doctoral theses of which the catalog is part.

REFERENCES

1. DESSI, Massimiliano, 2009. *Spring 2.5 Aspect Oriented Programming.* Birmingham: Packt Publishing. ISBN 978-1-847194-02-2.
2. GAMMA, Erich, HELM, Richard, JOHNSON, Ralph a VLISSIDES, John, 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1994, ISBN 0-201-63361-2.
3. ÖBERG, Rickard. *Introduction to Qi4j, 2009.* http://www.qi4j.org/160.html.
4. ODERSKY, Martin, SPOON, Lex, VENNERS, Bill, 2007. *Programming in Scala.* Artima Press, ISBN 978-0981531601
5. REENSKAUG, Trygve. MVC Xerox PARC 1978-79*, 2012.* http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.
6. ŠLAJCHRT Zbyněk, 2009a. *Chaplin - The Agile Java Class Transformer.* http://www.iquality.org/chaplin/generated/doc/book.html.
7. ŠLAJCHRT Zbyněk, 2009b. *Kompozitově orientované programování, sborník Objekty 2009.* Hradec Králové: Gaudeamus. ISBN 978-80-7435-009-2.