PART 2

The Zypher Design

CHAPTER *3*

The Zypher Software Artefact

In the domain of software engineering, it is common practice to validate techniques experimentally [NationalAcademy'94]. One builds a prototype for a particular application domain to investigate benefits and drawbacks of certain techniques and generalises the findings to a broader set of application domains afterwards. Such a prototype system is called a software artefact and in our dissertation this is role is played by the Zypher system.

Such an experimental validation methodology is summarised in the proverb "The proof of the pudding is in the eating". In the first part, we have been collecting a number of ingredients and instruments for a delicious pudding — the hypermedia design space, the three levels of tailorability, object-oriented frameworks and meta-object protocols. Also, we have been describing how such a delicious pudding might taste — the framework browser scenario. Finally, we have sketched a recipe to cook the pudding — the framework design methodology.

This second part, describes the cooking process we have carried out in order to cook a real pudding — the Zypher open hypermedia framework. This software artefact serves as a 'proof of concepts' that our framework design methodology is applicable for the construction of a concrete open system that incorporates the three levels of tailorability. Also, we describe the actual proof of the pudding, that is the eating process — the refinement of the Zypher framework into a prototype of a framework browser. This show how the three levels of tailorability helped us to tailor the open system for a prototype of a framework browser.

The Zypher Design Pattern Form

One of the main difficulties in describing a software artefact is that one must list a large number of issues and problems, relate this to the description of a software architecture and somehow convince the reader that this architecture provides a solution for the initial list of problems. This is a very difficult job that requires skills far beyond that of a typical software engineer.

The problem is even more difficult when describing object-oriented software architectures, because such an architecture must be reusable, thus address a number of similar, recurring problems. The object-oriented community acknowledges this problem and forwards design patterns as a way to describe reusable aspects of an object-oriented design. In a design pattern approach, instead of arguing how a particular architecture covers a complete list of problems, one argues how a small aspect of that particular architecture solves a subset of the initial list of problems. This way, certain aspects of that particular architecture can be reused in other architectures. Also, it is more easy to adapt the design to a changing list of problems. Using the design pattern approach, those reusable aspects are described in a systematic way to help authors and readers identifying the important facets; this is called the design pattern form.

There are several attempts to assemble design pattern classified in two main categories. Generic catalogues collect domain independent patterns; examples are [GammaEtAl'95] and [Pree'94]. Domain specific catalogues collect patterns about a specific problem domain; HotDraw [Johnson'92], [Beck,Johnson'94] is a well-known example for the domain of graphical editors.

In what follows, we describe the Zypher open hypermedia framework using the design pattern approach. So the second part of this dissertation is presented as a domain specific design catalogue for open hypermedia systems. We also include a description of the design pattern form as we have applied it in writing the Zypher design pattern catalogue.

The real power of design patterns lies in the idea of overlapping many of them on a small number of system elements. All the following patterns include numerous mutual references, to stress the way they work together. Nevertheless we want to stress that for a first tour, the sequential path is probably the best. Only for an in depth study, the referencing scheme is essential to understand how all patterns work together to achieve the overall design.

The Zypher design pattern catalogue is available in hypertext format as well; among others it has been used as part of the construction of a framework browser. Also, it is available on the world-wide web from the Zypher home page at *http://progwww.vub.ac.be/pools/Zypher/*.

Outlining the Zypher Design Pattern Form

All sections are mandatory, unless stated otherwise.

Title

The title names the pattern and provides a glimpse on the essence of the design. The title consists of a few keywords followed by a colon and a short phrase. The keywords constitute the name used in other design patterns as a reference and alludes to the

problem or solution discussed in the design pattern. The short phrase condenses the core of the pattern and motivates its existence in the global problem.

Intent

The intent section summarises (2 to 4 lines) the design pattern. As such, the intent section elaborates on the short phrase in the title without mentioning the name. An icon reflects the level of tailorability (see the [puppet master metaphor]).

Analysis

The analysis section introduces the problem addressed by the design pattern and lists relevant issues. As such, the analysis section forms a bridge between the intent section and the reference in the title. The analysis section uses examples to illustrate how the problem is manifested in some problem domain without hinting at solutions. The keywords in the title are not necessarily mentioned in the analysis section; this depends whether the keywords refer to the problem or to the solution.

Problem

The problem section recapitulates the problem the design pattern tackles based on the issues addressed in the analysis section. The problem section is always stated as a question; usually a "How ?" question to emphasise that a solution is desired. The problem section is short and does not contain examples.

Solution

The solution section proposes a design that answers the question stated in the problem section. The solution section is written in a clear and concise style that emphasises on the proposed technique and/or structure. Normally, the examples taken from the analysis section are reproduced in the proposed design to demonstrate how it solves the problem.

Contract

The contract section includes charts (applying [class diagrams] or [object interaction diagrams]; see below) specifying the internal structure of the proposed solution. The rest of the contract section enumerates all elements of the charts (including relations between the elements) and charges them with precise responsibilities. Wherever possible, the messages that constitute the contracts are listed and the refinements subclasses are allowed to make are specified. The contract section does not contain any examples.

Motivation

The motivation section answers why the proposed solution solves the issues raised in the analysis domain. Among others, the motivation section includes alternative solutions and why they are rejected. The examples taken from the analysis and solution sections are reconstructed to illustrate certain advantages or drawbacks.

Issues (optional)

The issues section, if present, discusses issues subordinate to the ones listed in the analysis section. Typical issues handled in this section include naming,implementation remarks, relations with similar approaches, references to the literature, … .

Consequences (optional)

The consequences section, if present, discusses design trade-offs and includes references to other design patterns that handle the drawbacks raised by the introduction of the design pattern.

Relations

The relations section links the pattern with other design patterns in the text. This section also includes references to well-known design pattern catalogues. The section includes at least one 'default' reference, pointing to the most obvious next pattern to read. The reference is marked with a triangle symbol in the margin ( ).

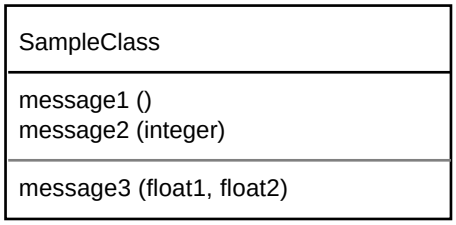
References

All the Zypher design patterns are organised in a mutual referencing structure. References are surrounded by brackets "[]" and use the name in the title (i.e. the keywords preceding the colon) to refer to another design pattern. To refer to a section within the pattern, the reference is followed by the ~ character and the title of the subsection.

Diagrams

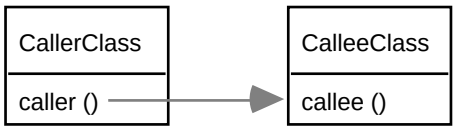
Throughout this document, class diagrams are represented using a variant of the "Object Modelling Technique" (OMT) notation. We refer to [Blaha,Premerlandi,Rumbaugh'88] and [RumbaugEtAl'91] for a full description of OMT. Below is a brief survey of the main elements in adapted notation.

Class & Messages



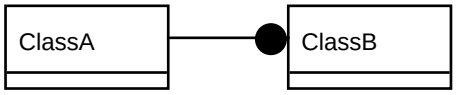
A class is denoted by a box with the class name at the top (i.e. SampleClass). A class name always starts with a capital. The set of messages understood by instances of that class appear below the class name; some messages may be separated by a dotted line to group related messages. Names of messages start with a small letter; names of parameters correspond with the name of the class but start with a small character (i.e. integer, float1, float2). In the example we see a parameterless message (i.e. `message1`), a one parameter message (`message2`) and a two parameter message belonging to another category (i.e. `message3`). Note that none of our classes have instance variables.

Calling Order



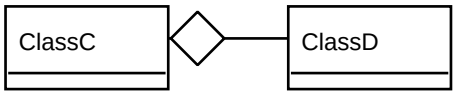
A grey arrow between to classes indicates a calling order from the CallerClass to the CalleeClass, meaning that an instance of CallerClass having received the message `caller` must send the message `callee` to an instance of CalleeClass. Note that, one of the participating messages may be left unspecified, which implies that this side of the calling order is not important in the design. The calling order is only shown when it is essential to the design being explained, as it quickly leads to cluttered schema's; Object Interaction Diagrams (see below) are preferred for larger calling structures.

One-to-Many Association



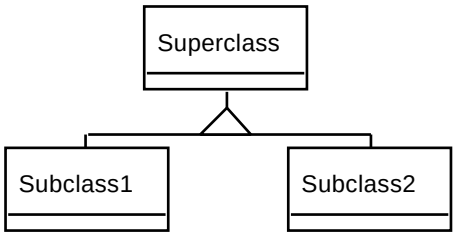
A line with a black circle indicates a one-to-many association from ClassA to ClassB, meaning that —at a certain moment— an instance of ClassA may be associated with multiple instances of ClassB. The line between the classes means that there is an association, the black circle means that this side of the association is the 'multiple' side. Usually, both classes understand messages to modify and query the association relation.

Aggregation



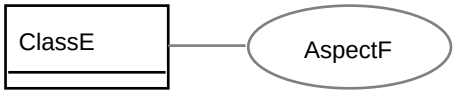
A line with a white diamond indicates an aggregation association from ClassC to ClassD, meaning that —at a certain moment— an instance of ClassC may contain multiple instances of ClassD. The line between the classes means that there is an association, the diamond means that this side of the associations is the 'container' side. Usually, only one class (the container class) understands messages to modify and query the aggregation relation.

Inheritance



A line with a white triangle indicates an inheritance relation with one superclass and two subclasses, meaning that all instances of Subclass1 and Subclass2 inherit the structure and behaviour of Superclass. Nevertheless, the structure and behaviour of the three classes may be different, since Subclass1 and Subclass2 may extend and override the structure and behaviour of the Superclass. The line between the classes means that there is an association; the top of the triangle points towards the superclass; the base points towards the subclasses.

Unspecified Aspect

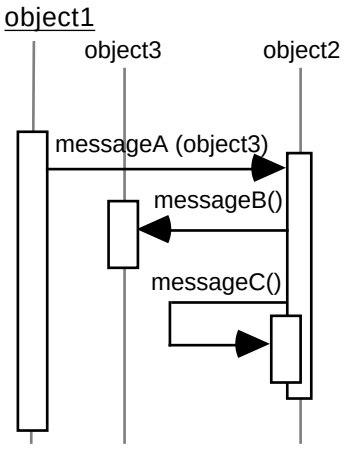


In some occasions, an aspect of the design is left unspecified. This means that its existence plays a crucial role in the overall design, but the actual structure is unimportant. The example shows that ClassE is related with an aspect AspectF, meaning that AspectF participates in the object configurations of ClassE, but that the actual structure of AspectF is yet unknown.

Interaction Diagrams

Throughout this document, object interaction diagrams are represented using an adapted version of the Objectory notation. We refer to [JacobsonEtAl'92] for a full description of the notation. Below is a brief survey of the main elements in the notation.

Message send



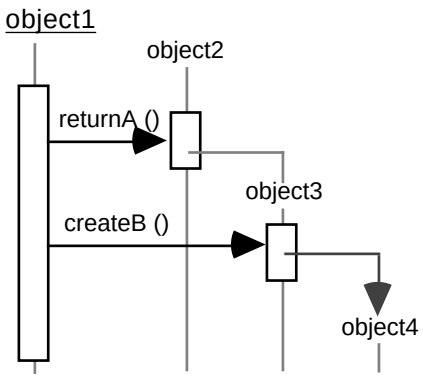
This is an example of an object interaction diagram illustrating message exchange between objects. The message involves three objects (object1, object2, object3) indicated by the vertical lines; the names of the objects are placed on top of the lines. One object (i.e. object1) is the originator of the interaction: its name is placed a little higher than the others and is displayed in a slightly larger and underlined font.

The time flows from the top to the bottom of the diagram. Vertical rectangles denote objects that are handle messages during a certain period: the bottom and top of the vertical rectangles denote the moment the object starts or ends handling the message. The messages themselves are depicted by horizontal arrows starting in the object sending the message and ending in the receiver of the message. The name of the message is written above the horizontal arrow; the names of the argument objects stand between braces.

The example shows an object1 sending a message `messageA` to object2 with object3 as parameter. While object2 is handling the message, it sends a parameterless `messageB` to object3 and thereafter it sends the `messageC` to itself. Since the bottom of the object1 rectangle is below the bottom of the object2 rectangle, control returns to object1 afterwards.

Return Object / Create Object

Existing objects returned as the result of a message send, are depicted with a line emerging from the message handling rectangle; the name of the object appears on top of the line (i.e. object 3). Objects created during a message send are depicted with an arrow emerging from the message handling rectangle; the name of the object appears just below the arrowhead (i.e. object4). To emphasise the difference, the object creation arrow is drawn in a dashed line while usual object lines are dotted grey lines.



The example shows an object1 sending a message `returnA` to object2 that returns an existing object3. Afterwards, object1 sends a `createB` message to object3 that creates an object4.