Meta-architecture separates business logic from design issues

Michel Coriat
LCAT[1]
Thomson-CSF LCR
Domaine de Corbeville - 91404 Orsay Cedex - France
michel.coriat@lcr.thomson-csf.com

Abstract

Reflection is often used to capture variability in the language and handles it at runtime, as reported by the reflection pattern. In this paper, we explore the use of reflection as a prime variability model for evolutionary architectures. This investigation is carried out in the context of product families development where both architecting and designing for change are crucial. We show how reflection enables separation of concern between specification of business logic and decision on *design issues*. we propose to materialize the meta-level part of the architecture with a model called meta-architecture.

Keywords

Specification, Software architecture, software and system families, meta-architecture, meta-model composition, meta-object protocol (MOP), UML.

1 Architecture

A software system (Figure 1) can be compared to the picture of an assembled puzzle. The picture (the application) is defined by a puzzle (the architecture) which is a combination of pieces (the components) respecting a certain layout (style, connector, configuration, structure, ...) so that the puzzle cutting up is independent of the picture nature.

We may suppose that the puzzle layout is defined only by pieces outline taken separately. But, suppose we decide to change the outline of a piece, the puzzle layout remains the same and the piece does not longer fit for the puzzle. Consequently, the puzzle layout is an emergent property of the puzzle due to pieces arrangement. It is not explicitly described by pieces outline but is a consequence of the combination.

To characterize the puzzle layout we need then some materialization of its properties. Moreover, these properties do not have to depend on the picture

[1] LCAT is the acronym of "Laboratoire Commun Alcatel Thomson-CSF"

nature. In fact, they are defined by the cutting die used on the picture to make the puzzle.

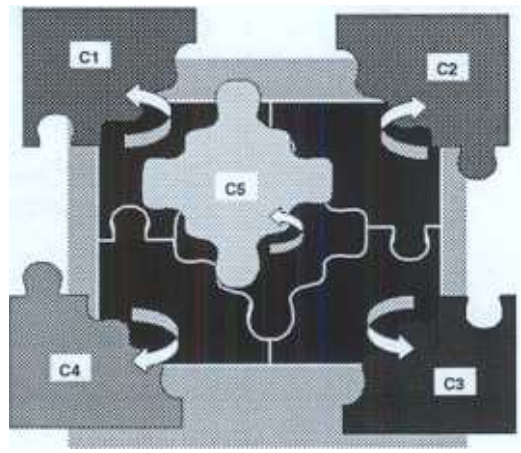

Figure 1 - Components and Architecture (C1 to C5 are pieces)

Back to the software architecture [BCK97]. Pieces of the puzzle are related to components. Components are assigned responsibilities and provide "contracted" interfaces to client components. Puzzle is the architecture where the layout defines the big picture, style, rules, communication and control mechanism guiding application structuring. What we need is the materialization of a cutting die necessary to define the architecture layout such as:

- it specifies the design issues related to application policies (puzzle layout),
- it can be applied independently of the application logic (nature of the picture).

Application logic is about computational/functional and data components expressing the business process imbedded into the application

Design issues are related to technical/technological features such as components interactions (data flow, control flow, ...), constraints (communication protocols, threading, timing, synchronization, ...) and topology (configuration of components linked by connectors).

In the object community these properties are associated to business objects and technical objects [S94] respectively.

We have materialized our cutting die with what we called meta-architecture. Meta-architecture is based on meta level techniques [FD98] which are a

promising approach for the separation of concerns between mechanisms issued at design level and application logic.

2 Meta-Architecture

[BMRSS96] uses meta-level techniques in the architectural pattern "Reflection" to design for change. Two main drawbacks of this pattern are: the necessity of a programming language to support reflection, and the fact that the change of structure and behavior is made at run-time which requires extra-processing and thus reduces performances.

Our approach emphasizes at the architectural level. The main idea is to model design issues (which are addressed by non-functional requirements) by meta-component and business logic (which come from functional requirements) at the base level component.

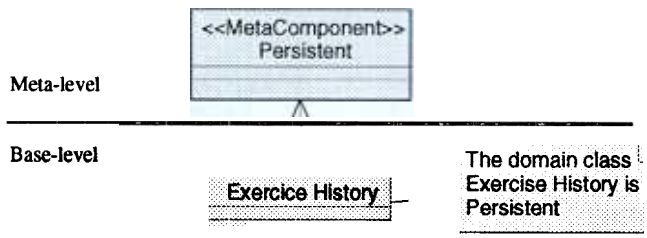

Figure 2 - Persistent Exercise History

A meta-component is a reusable entity that embodies a property without identification of the components that have the property.

The proposed representation of architecture and meta-architecture relies on UML [UML97] diagrams. Components and meta-components are represented by classes (boxes). Meta-components are distinguished by the stereotype <<MetaComponent>>. We can then make a difference between the base level of the architecture (components without the stereotype <<MetaComponent>>) and meta-architecture level (components with the stereotype <<MetaComponent>>).

For example (inspired from [FD96]) suppose that we want to create a component for a flight simulator whose aim is to store exercise history of the student pilot during simulation. The component is called *Exercise History* and must be persistent.

Figure 2 describes the component *Exercise History* which is an instance of the meta-component *Persistent*. This relation assign persistency properties to the

component *Exercise History*. This construct enable the behavior of a component to be customized by binding it to the meta-component that provides the appropriate property. Advantages are transparency, reduction in complexity, and especially explicit separation of concern between business logic and design issues.

Separation of concerns promotes reusability of components because explicit distinction is made between design (*Persistent*) and business component (*Exercise History*). Flexibility of the overall architecture is improved because evolution of these two types of components have to be undertaken separately (switching from a file system to a database system solution for *Persistent* meta-component does no change the logic of login *Exercise History*).

2.1 Composing design solutions at the meta-architecture level

Simple use of meta-component is not enough. It is necessary to compose design solutions at the meta-architecture level. The inheritance mechanism of meta-component enables composition of properties.

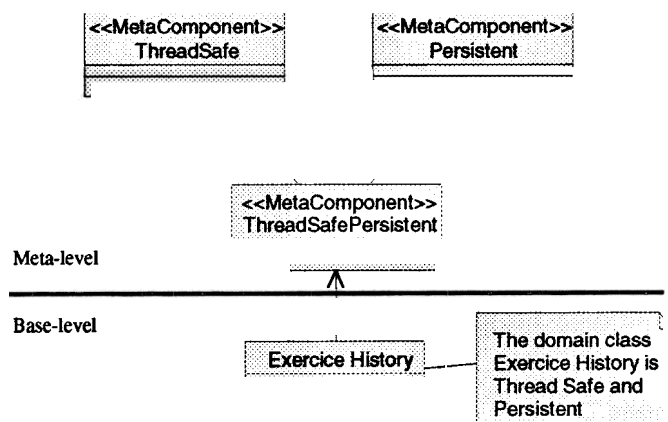

Figure 3 – Thread Safe Persistent Exercise History

Suppose now (inspired from [FD96]) that the exercise history processing is a very critical task that must be always able to work. We then need to treat the Exercise History component as a Thread Safe component.

Figure 3 specifies the *Exercise History* component as Thread Safe Persistent because it is an instance of *ThreadSafePersistent*. Composition of meta-component properties must be made at meta-level (meta-component named *ThreadSafePersistent*) and not at the base class level in order to implement the composite property (Thread Safe and Persistent).

The reader can refers to [FD98] for the definition of inheritance semantics of meta-components.

To customize the behavior of base level components (business components) with properties of meta-components (technical components) we need to describe interface of the meta-components in order to manipulate base level component. This is the meta-object protocol (MOP) [KRB91].

2.2 Designing a MOP

All access to a component are mediated by its associated meta-component which defines a meta-object protocol (MOP) realizing interaction between them.

The sequence diagram of Figure 4 illustrates a MOP related to *Persistent* from the first example (Figure 2). Suppose *aComponent* signals to Exercise History that an exercise step have to be inserted into *Exercise History*.

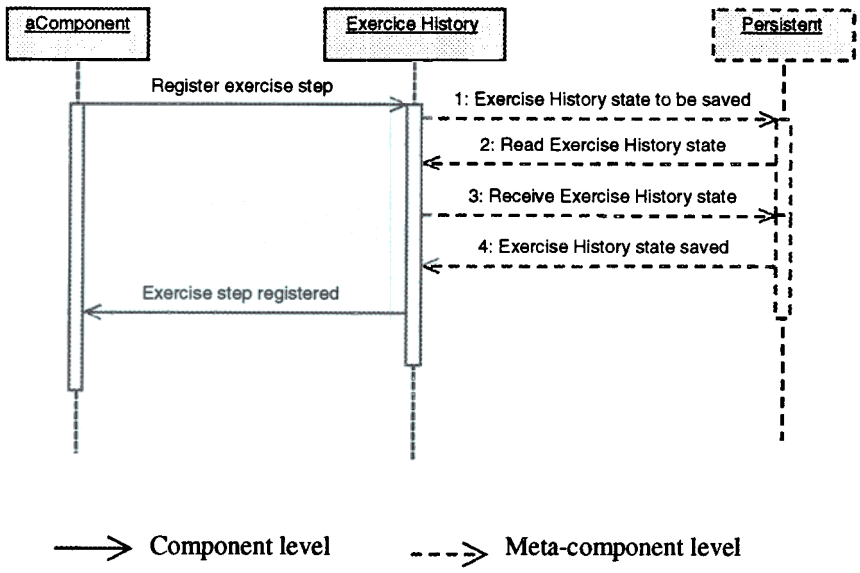

Figure 4 – Sequence diagram of a MOP mechanism

When *Exercise History* is invoked by *aComponent*, *Persistent* catch the call (1), ask Exercise *History* its state (2) which after some process (in particular local history change) is passed to *Persistent* (3) which execute again some process to enable *Exercise History* (4) to signal to *aComponent* that the exercise step has been registered.

Generally, the meta-component interface comprises at least instance creation, and deletion, attribute read or write access, and method call.

The design of a MOP is similar to architecture design. For aspects related to design issues (persistence, thread safety, fault tolerance, secure communications, group based distributed applications, ...) the meta-architect have to describe static and dynamic part of the meta-component.

This approach assumes the possibility that meta-architecting must take into account libraries or frameworks of meta-components objects. These meta-components can be build themselves on infrastructures like CORBA, DCOM, or specific hardware or software libraries or frameworks.

Customization of software architecture with the MOP is achieved by binding it with several possibilities:

- Coding between the lines: pre-processing source code or intermediate code (bytecode),
- Trap/Hooking with the middleware event model (Windows events, JavaBean event model),
- Making use of binding support in middleware: CORBA interceptors,
- Proxying: provide abstraction of the object reference in order to redirect invocations.

The main difference between these possibilities is the MOP implementation which can be at compile-time [C95] or at run-time [KRB91].

3 How meta-architecture promotes practices for software system families

A product family gather a set of applications belonging to the same domain and characterized by common software assets (requirements, architectures models, components, code, ..). Design of software systems families suggests identifying issues in design that are common to all family members, and organizing the design to support changes/variations.

Separation of concerns is encouraged by meta-architecture which facilitates exploitation of commonalties in product families. We have shown that the representation and manipulation of meta-architecture express design constraints on business components. This enables to get control over design decisions depending on the design mechanisms chosen for application of the product family.

Because our approach is based on modeling techniques with the UML notation we do not require a specialized language. Thus we are able to use languages that do not support reflective capabilities.

We do not claim that meta-architecture is the silver bullet for architecting. However, it is a strong variability mechanism which handles commonality and variability of architecture-centric product families.

The overhead caused by the supplementary level induced by meta-architecture generates additional specification and implementation. Nevertheless, it is compensated with the gain in anticipating potential changes of the considered product family.

4 Workshop Interests

The LCAT is a common research laboratory of Thomson-CSF and Alcatel providing to business units methods and techniques for the development of product families for both companies' product lines. In this context, advanced technologies (UML, CORBA, DCOM, frameworks, design patterns, ...) and associated tools are used to support the product line approach.

LCAT research investigates the use of meta-level architectures in the design and implementation of software systems families. Thus, we are interested in related approaches and solutions, in particular those connected to the following problems:

- Composition semantics at the meta-component level,
- Design of MOPs sufficiently robust whose changes on base level are sufficiently localized to enable potential errors,
- Design of meta-component frameworks and libraries based on middleware infrastructure (CORBA, DCOM,...),
- Transformation of architecture at the source code level from meta-architecture descriptions. This is the aim of Aspect-Oriented Programming [AOP97],
- Find the trade-off between compilation-time MOP and run-time MOP,
- Scripting into commercial UML tools to support MOPs and meta-component composition.

5 References

[AOP97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, Aspect-Oriented Programming, Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[BCK97] Bass, Clements, and Kazman, Software Architecture in Practice, Addison-Wesley 1997

[BMRSS96] Buschmann, F. and Meunier, R. and Rohnert, H. and Sommerlad, P. and Stal M., Pattern-Oriented Software Architecture: A System Of Patterns, Wiley & Sons, 1996

Chiba S., A meta-object protocol for C++ - In proceedings of the 10th conference on object oriented programming systems, Languages and programming, pages 285-299, 1995.

[FD96] Ira R. Forman, Scott H. Danforth, Inheritence of Metaclass Constraints in SOM, Proceedings of Reflection'96, page 185-202, San Francisco, April 96

[FD98] Ira R. Forman, Scott H. Danforth, Putting Metaclasses to Work : A New Dimension in Object-Oriented Programming, Addison-Wesley, 1998

[KRB91] Kiczales, G., des Rivieres, J. Bobrow, D.G. – The Art of the Metaobject Protocol - The MIT Press, Cambridge, Massachussets (1991).

Sims O., Business Objects : Delivering Cooperative Objects for Client-Server. McGraw-Hill, 1994.

[UML97] UML Notation Guide, version 1.1, 1 September 1997, OMG.