Dynamic Adaptation of Non-Functional Concerns*

Pierre-Charles David & Thomas Ledoux
{pcdavid,ledoux}@emn.fr

École des Mines de Nantes
4, rue Alfred Kastler. B.P. 20722
F-44307 NANTES Cedex 3

Abstract

In order to build programs able to adapt themselves to changing execution conditions, we propose a *Separation of Concerns* approach distinguishing functional and non-functional concerns. These two kinds of concerns are composed together, at run-time, by a *weaver* which is aware of the execution conditions so that it can adapt its weaving to their evolution.

1 Introduction

Today's world is characterized by change, and information technologies is probably one of the domains where this is the most dramatically true. At the same time, we rely more and more on computer technologies, and we need them to work, and work well, in this ever-changing world. This situation has a very concrete impact on the software development field: today's software systems must deal with an increasing diversity and complexity. First, concerning execution platforms and environments, new types of computing platforms appear regularly like Personal Digital Assistants and mobile phones, each with its own particularities. Users needs also evolve, as these new technologies allow them to be more mobile, and make them rely more and more on electronic devices.

From a software development perspective, this makes the developers work harder than ever: they have to deal with a wide spectrum of hardware platforms, each with dynamically varying resources. This unpredictable variability means that it is not feasible to develop applications in an ad-hoc way, to work only in very specific conditions. For an application to last, it must be built to be *adaptable*.

In this context, the goal of our work is to make it easy to developers to build such adaptable applications, and to provide the required infrastructure to actu-

*This research is supported by the RNTL project ARCAD (http://arcad.essi.fr) ally adapt them, more or less automatically[1]. Our idea is to use a *Separation of Concerns* approach for application development, to make explicit in applications the separation between what depends on the changing environment, and what does not. This separation is only the first step: to obtain a working application, it is necessary to compose the different concerns together. By making this composition process aware of the evolutions of the environment, the application resulting from the composition is always *adapted* to its environment.

This paper is organized as follows: in Section 2, we present more specifically the kind of adaptations we address, and the general idea of our approach. Section 3 constitutes the main part of the paper: it describes in detail our proposed solution and the prototype we developed to demonstrate it. Then, we finally conclude in Section 4.

2 General Approach

In an ideal world, programmers would only deal with the intrinsic features of the application and their program would work perfectly in every circumstances. In practice, they also deal with all kinds of non-functional concerns like limited resources (sometimes very limited, as in embedded systems) and potential failures (both hardware and software). Applications must be designed very carefully in order to work correctly and efficiently. As a consequence, applications generally include both very specific code intended to solve a well defined problem, *functional code*, and code whose sole purpose is to make sure the functional code performs correctly and efficiently in its environment, *non-functional code*.

As stated in the introduction, the execution environment of an application varies a lot during its lifetime. The changes range from new hardware platforms every few months (PDAs, cell-phones...) to fine-grained variations (temporary drop in the available bandwidth). Our goal being to make applications adaptable to these changes, our target is the non-functional code, which by definition is the part of the applications dealing with the environment. However, to do this it is essential to be able to manipulate non-functional code independently from the rest of the application.

Using traditional object and component based approaches, non-functional code is generally interwined in the functional code, making it impossible to manipulate it independently. However, Aspect-Oriented Programming [5] techniques make it possible to cleanly encapsulate non-functional concerns and then to *weave* these concerns with the functional ones. The benefit is that it now becomes possible, without modifying the functional part of an application, to modify which non-functional concerns affect the functional code, and how they affect it, simply using different *weaving instructions*.

In our particular case, we are interested in the run-time execution conditions. Our goal is thus to make the weaver aware of the program's execution conditions so that it can adapt, at run-time, which non-functional concerns to apply, and

[1]We believe that complete automation of software adaptation is not achievable, because adaptation is by nature linked to the semantics of an application.

how to apply them, to the different parts of the functional code. To allow this, we need to reify the weaving instructions, and make them dependant on the execution conditions.

3 Proposed solution

This section represents the main part of the paper. It describes a prototype we developed in Java to allow dynamic adaptation of non-functional concerns using *adaptive weaving*.

Reusing the terminology introduced by Kiczales [4] in a broader context than AspectJ, we consider that an AOP system consists in a *base program* containing *join-points* (principled points in the execution of this program), and *aspect programs*. These two kinds of code are composed together by a *weaver* at specific places called *pointcuts* (sets of join-points), following instructions expressed in a specific weaving language. In some systems like AspectJ, these *weaving instructions* are (at least syntaxically) part of the aspect programs. We believe that a reification and externalization of these weaving instructions can lead to more flexible and powerful AOP systems [6]. In this section, we show how we used this idea to define a context-aware weaving language which allows to build dynamically adaptable applications.

3.1 Base Program and Join-Points

In our system, the base program is identified with the functional code, which role is to implement a solution independently of the specific characteristics of the execution environment. We assume that all the interactions of this code with the environment are dealt with somewhere else in the system (see Section 3.2).

The functional code itself is composed of standard Java code, and deals only with business logic. We don't impose any constraints on this code, which is developed by application programmers who are only concerned with solving the business problem. However, we need some control on the execution of this code if we want it to be affected by non-functional code. To get this control, we associate a `Container` object to each functional object. This container is responsible to manage all the interactions of its object with others, essentially by intercepting messages (it plays essentially the same role as EJB containers [3]). This is the point at which it is possible to weave aspects at run-time, and as such, the set of all method receptions of a particular functional object can be though of as a *join-point*.

To insert the required indirection to the `Container` in every public method call in the standard Java code written by programmers, the code must be compiled using a tool described in [2] which transparently inserts *hooks* to redirect method calls to the `Container`. In the original implementation the `Container` is actually a Meta-Object from the RAM Meta-Object Protocol [1]. The important point is that at this stage, the application programmer does not have to think about non-functional concerns at all.

3.2 Non-Functional Aspects

As we said earlier, the aspects we want to weave with the base program correspond to non-functional code; they encapsulate all the interactions of the base program with the execution environment.

The `Container` attached to each functional objet manages a list of meta-level objects (programmed in standard Java) implementing non-functional aspects. When it intercepts a message, it passes it to all the non-functional aspects currently attached to it before executing the original method. Each aspect then have the opportunity to execute its code (what AspectJ would call an *advice*), thus composing[2], or *weaving*, the non-functional behaviors with the base behavior.

Because this weaving is done entirely at run-time, it can evolve during the program execution to match the evolution of the execution environment, thus realizing *dynamic adaptation* of the program. To allow this, the `Container` supports a simple interface to dynamically attach, detach or reconfigure the non-functional aspects it manages.

3.3 Adaptive Weaving Specification: Adaptation Policies

The main contributions of our system are the reification and externalization of the weaving instructions, and the fact that these instructions are made aware of the execution context. We call the resulting weaving programs *adaptation policies* because, as we show in this section, they allow to build dynamically adaptable applications. These policies encapsulate all the weaving instructions, from the identification of pointcuts in the base program to their associations with non-functional aspects. Our system proposes two kinds of policies which are to be used together to form a complete weaving specification. This weaving specification can take into account evolutions of the execution context so that the non-functional code weaved with the base program is always well adapted.

3.3.1 Dynamic Pointcuts Definition using Application Policies

The first kind of adaptation policies is the *application policy.* It is meant to be written mainly by application programmers, and thus remains relatively abstract and free from low-level details. The role of these policies is:

- to define pointcuts (sets of join-points) in the base program;
- to identify which non-functional aspects are required at these pointcuts, without detailing how these aspects are implemented.

As we saw in Section 3.1, a join-point in our system is the set of methods receptions on a single functional object (`around(): execution(public *.*(..))` using AspectJ's notation for pointcuts). Because of this one-to-one

[2]Our current implementation is very limited in this regard, in the sense that the composition of multiple aspects is very simplistic: they are simply called one after the other.

relationship between base objects and join-points, we identify the two from now on. Thus, a pointcut being a set of join-points, our pointcuts are simply *groups* of functional objects.

The application policies allow application programmers to define these groups of objects using a Scheme-like syntax. Concretely, the pointcut language allows to define each group implicitly by a *predicate* over run-time attributes attached to functional objects. These attributes are managed by the functional objects' `Container` and include:

- some generic, predefined attributes like `className` (respresenting the name of the class the object is instance of);
- object-specific attributes representing reifications of the fields belonging to the object (for example, a `com.mybank.Account` object would have a `balance` numerical attribute).

Using run-time attributes means that the join-points which are part of a given pointcut may evolve during the program execution. This makes it possible to model the fact that a given object can be treated differently during its lifetime, depending on its state.

Given these informations, some predefined predicates and composition operators (comparisons and standard boolean operators), it is possible to create complex pointcut definitions. The following example shows such a predicate identifying all the instances of class `com.mybank.Account` whose `balance` field is below $300:

```
(and (= (attr "className") "com.mybank.Account")
     (< (attr "balance") 300))
```

These predicates are applied to already existing groups, and "filter" their content: the content of a group is the set of all the elements in its parent group for which the predicate holds. As shown in the example, the attributes of an object can represent dynamic properties (the `balance` field). The content of the groups is thus dynamic and the system is responsible to track the revelant changes in attributes values and to update groups content accordingly.

To bootstrap this recursive definition mechanism, the system provides a special group, named `all`, and containing all the functional objects existing in the application. Here is the syntax used to define a group:

```
(def-group "low-accounts" :parent "all" :select predicate)
```

As a result of this recursive definition, the groups are organized in a hierarchy which can be thought of as a "class hierarchy" orthogonal to the language-level one, and which is used to model the non-functional dependencies of the base program. This hierarchy of pointcuts have two characteristics which make it particularly powerful: it is expressed at the object-level (fine-grained), and it is completely dynamic.

Once the application programmer has defined the appropriate groups (pointcuts) for his applications, he must indicate which *system policies* should be weaved to them. System policies are described in the next section, and provide adaptive implementation of non-functional aspects. However, at this point, the only thing the application programmer needs to know is which of these non-functional aspects he is interested in, not the way they will be implemented.

This constitutes the last part of a group definition. The following example shows the complete syntax used to define a group, including the specification of such an attachment. A group inherits all the attachments of its parent, but can override the configuration parameters of the policies.

```
(def-group "low-accounts" :parent "all"
           :select (and (= (attr "className") "com.mybank.Account")
                        (< (attr "balance") 300))
           :bind   '((policy "logging")))
```

To conclude on this first kind of policies, we can say that they are used to model dynamic pointcuts which depend on the state of the functional objects, and bindings of these pointcuts to system policies.

3.3.2 System Policies for Context-aware Weaving

The second kind of adaptation policies required for an application are *system policies*, which introduce execution context-awareness in the weaving. In application policies, the programmer only identifies relatively abstract non-functional aspects, like persistence or distribution. The role of system policies is to ensure that the aspects with base objects are always well adapted to the evolutions of the execution context. The end result is a context-aware weaving of non-functional aspects upon functional objects.

An obvious prerequisite for a context-aware system is a monitoring subsystem. We have developed a simple monitoring framework consisting in *probes*, which observe the environment and provide values for several characteristics of this environment (CPU usage, free memory, used bandwidth...), and *environmental conditions* expressed using the values reified by probes (CPU usage above 90% *and* free memory below 5Mb).

A system policy consists in a set of *adaptation rules*, each rule associating an environmental condition with a constraint on non-functional aspects. Its semantics is that whenever the system is in a state which matches an environmental condition, the constraint associated to this condition must hold. Concretely, constraints are simply requirements for non-functional aspects, including configuration parameters.

The example in Figure 1 shows the syntax of system policies. This example policy will adapt the logging aspect implementation (and parameters) to the amount of available free disk space and network bandwidth: as long as there is enough free space on the local disk, logging informations are saved on it in a verbose format (XML), but when the space becomes rare, the information is sent over the network using a more concise format (to save bandwidth), and only if this leaves enough bandwidth for the application.

```
(def-policy "logging"
  (when (> (attr "/system/storage.free_space_kb") 5000)
    (ensure (attached "logging.file" ((format . 'xml)))))
  (when (and (<= (attr "/system/storage.free_space_kb") 5000)
             (>  (attr "/system/network.free_bandwidth_kbs") 10))
    (ensure
      (attached "logging.network" ((target-host . "appserver.mybank.com")
                                   (target-port . 99)
                                   (format . 'text))))))
```

Figure 1: An example system policy

The runtime system we provide is responsible for setting up the monitoring framework according to the conditions expressed in the system-policies. Then, when this monitoring framework detects that a condition's boolean value has changed, it notifies the system which attaches or detaches the corresponding aspect(s). Because system policies are bound to groups of objects, this results in adapting which aspects are weaved to the objects (and hence their behaviour) to the evolutions of the environment.

3.4 Putting it all together

To use our system, an application must provide:

- the functional code (pure Java source compiled using a special tool provided);
- the non-functional code (implemented as reusable libraries of meta-objects);
- the adaptation policies (both application and system policies).

When the system starts, it loads the policies, sets up the monitoring framework, and launches the application. From this point on, it becomes completely passive, and wakes up only in the following circumstances:

1. A new functional object is created. The new object is introspected, put in the appropriate groups, and weaved with the non-functional aspects currently active for these groups.
2. A significant property of a functional object changes (significant meaning "appearing in a predicate defining object groups"). The system determines the new set of groups the object is member of, adds and removes it from the appropriate groups and updates the non-functional aspects weaved to it accordingly.
3. The monitoring framework detects a change in the environment which activates or deactivates a rule in a system-policy. The system updates the non-functional aspects weaved to every member of the groups bound to this system-policy.

4 Conclusion and Future Works

In this paper we presented a prototype using Aspect-Oriented Programming concepts to enable dynamic adaptation of applications to changing execution contexts. The main contributions of our system are the reification and externalization of the weaving instructions, and the fact that these instructions are made context-aware. The resulting system is able to dynamically adapt the weaving of non-functional aspects over functional objects according to the changes in the execution context.

In the current prototype, the weaving instructions are loaded by the system at startup time, and can not evolve during runtime. The next step would be to enable free modifications of the adaptation policies at run-time, enabling real unanticipated software adaptation.

References

[1] N. M. N. Bouraqadi-Saädani, T. Ledoux, and M. Südholt. A reflective infrastructure for coarse-grained strong mobility and its tool-based implementation. In *Invited presentation at the* International Workshop on "Experiences with reflective systems" *(held in conjunction with Reflection 2001)*, Sept. 2001.

[2] P.-C. David, T. Ledoux, and M. N. Bouraqadi-Saâdani. Two-step weaving with reflection using AspectJ. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, USA, Oct. 2001.

[3] L. DeMichiel, L. Ümit Yalçinalp, and S. Krishnan. *Enterprise JavaBeans^{TM} Specification.* SUN Microsystems, Aug. 2001. Version 2.0, Final Release.

[4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001*, volume 2072 of *LNCS*, pages 327–353. Springer-Verlag, 2001.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*. Springer-Verlag, June 1997.

[6] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. Dynamic wrappers: Handling the composition issue with JAC. In *TOOLS USA 2001.* IEEE Computer Society, July 2001.