Using Aspects to Structure Small Devices Adaptive Applications

Position Paper

Ayla Dantas
add@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Vander Alves
vra@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil

ABSTRACT

This paper briefly presents an architectural pattern, *Adaptability Aspects*, for structuring adaptive applications using Aspect-Oriented programming in a specific way. We also evaluate the applicability of this pattern for resource constrained devices. The *Adaptability Aspects* pattern is intended to improve modularity and reuse when adaptability is required. By providing three versions of the same J2ME application, we also compare our pattern implementation with pure Object–Oriented solutions using GoF patterns and with a less flexible implementation where the adaptability concerns are not so well isolated, evaluating aspects such as code size, memory use and performance.

1. INTRODUCTION

Adaptability is a common requirement nowadays [8], specially for ubiquitous applications, but its implementation is usually scattered throughout the code, what leads to less reuse. In order to deal with that, we have used [2, 1] Aspect-Oriented Programming (AOP), a technology intended to provide clear separation of concerns [3] and to make design and code more modular. In this position paper we discuss the implications of using such a technique in environments with restricted resources for implementing adaptability. We argue that it is possible to use AOP in this context, by following a pattern, and analyze its impacts on memory use, application size and performance.

The discussion and analysis presented here are drawn from an experience on implementing adaptive applications, which are able to change their behavior in response to context changes [9], using *Adaptability Aspects* architectural pattern [1]. This pattern proposes the use of aspects to provide adaptability in a flexible, modular and reusable way. We have applied it to some Java 2 Micro Edition (J2ME) [10] applications. For one of them, we have also implemented other versions not using AOP, measuring the impact of AOP in resources use.

In the remaining of the paper we give a brief overview of the *Adaptability Aspects* pattern and then we discuss the applicability of AOP for providing adaptive behavior with reuse, modularity and flexibility in constrained resources environments. We outline some results of a comparative study between our pattern implementation and pure Object-Oriented solutions. In the last section, we conclude the paper.

2. ADAPTABILITY ASPECTS

Adaptability Aspects [1] is an architectural pattern for structuring adaptive applications using AOP in a specific way. With AOP we can cleanly capture some implementation aspects that affect many parts of a system, providing appropriate isolation, composition, and reuse of the code used to implement those aspects [7]. We can, for example, define in an isolated module of the system the execution points (join points) that should be affected, and the code to be executed at those points, changing the normal execution of an application. This is the approach used by AspectJ, which is a general-purpose aspect-oriented extension to Java [6] and one of the most widely used AOP languages. For composing the AOP constructs with the code to be affected, we use code weavers.

The *Adaptability Aspects* pattern structure consists of five modules or elements, which are illustrated by Figure 1. It describes a solution for the problem of modularizing and improving reuse in the development of adaptive applications. The main idea behind our solution is to provide lightweight aspects (*Adaptability Aspects*), which are in an isolated module and are able to crosscut the other parts of the application and change their behavior, but just working as a glue between the other pattern elements. In fact, most code for these behavior changes are defined in another module (*Auxiliary Classes*), whose classes can be reused by several adaptive applications. The core application functionalities (*Base Application* element), including its business rules and GUI code, is also isolated, and it is the target module of the aspects actions.

The *Adaptability Aspects* element interacts with a module for monitoring the context (*Context Manager*) that can, for example, collect some information about the device environment such as its location. According to changes in the context, some new application behaviors can be triggered. This module can have reusable components that can be easily and cleanly plugged in other applications by aspects. There is also a module (*Adaptation Data Provider*) for providing

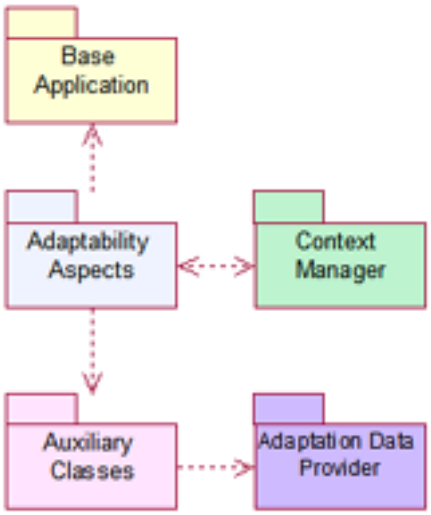

Figure 1: *Adaptability Aspects* elements

dynamic data for the adaptations and whose core infrastructure can also be reused.

As we can see, the pattern is composed by elements that are fully reusable, and by using aspects as suggested, the adaptability functionality is better isolated and can be easily plugged in/out. Besides that, some internal concerns related to adaptability are also isolated, such as context monitoring (*Context Manager*), dynamic adaptability data providers (*Adaptation Data Provider*), application changers (*Adaptability Aspects* and *Auxiliary Classes*) and the base application itself with business and GUI code (*Base Application*).

The dynamic behavior of the *Adaptability Aspects* pattern was depicted by two scenarios [1]. In the first one the aspects request some data from the context manager in order to know if a certain adaptation should be performed or not whereas in the other one, when a context change is detected, the aspects are immediately notified. The first scenario is illustrated by Figure 2.

The example implementation of this pattern uses Java 2 Micro Edition, targeted to pervasive devices that have adaptability as a common requirement.

3. COMPARATIVE STUDY

In order to evaluate the impact of aspects in constrained environment devices, we have developed the same application, a J2ME dictionary, in three different ways: using our pattern with the AspectJ language (*AspSol* version); using only pure OO patterns [4] (*PatSol*); and a tangled one, with less flexibility and separation of concerns, not properly using *Adaptability Aspects*(*TangSol*) or GoF patterns. We compared the three approaches by analyzing several metrics.

All versions evolved from a simple dictionary, capable of translating a given word from English into Portuguese. It presents four screens: presentation, main menu, instructions and dictionary screen (where the search is requested and the results shown).This initial application is our *Base Application* module, considering the *Adaptability Aspects* pattern. Some of the adaptability concerns included were: changes on application screens, dynamic addition of new screens, invocation of the garbage collector, changes of the search mechanism, internationalization, and changes in its target, and source translation languages. All of these adaptations should be triggered by changes in the application context, which can be represented by metadata or obtained through sensors, responsible for context management.

The*AspSol* version, using *Adaptability Aspects* pattern, presents the pattern elements illustrated by Figure 1. The *Adaptability Aspects* module is composed by six aspects which isolate each of the adaptations and use auxiliary classes, in order to avoid heavy aspects and to improve reuse. The context manager is also implemented using AspectJ, following an AOP implementation of the observer pattern [5]. The adaptation data provider is a small framework implemented following Adaptive Object-Models [11] architectural style that was also used for *TangSol* and *PatSol*.

For *TangSol* and *PatSol*, the context manager is implemented using the observer pattern [4] and some auxiliary classes from *AspSol* version could be reused. The difference between them is that in *TangSol* the adaptability concern is scattered throughout the code, leading to code tangling between each class normal responsibilities and the adaptability concern itself. In order to avoid that in *PatSol* we have used the Decorator, Abstract Factory and Bridge patterns so that it is easier to obtain an application version either with or without certain adaptations. In *AspSol* this is even easier, because we should only select the aspects we want to include as inputs for the weaving process whereas in *PatSot* it becomes more difficult to isolate each adaptability concern and to understand the code when the number of such concerns increases.

Considering the three versions of the adaptive application, we have analysed code size (in lines of code), bytecode size, memory use, initialization and execution times. The data used for this analysis are shown in Table 1. We have observed that *AspSol* has 2.6% more lines of code than *TangSol*, but 11.6% less than *PatSol*. However, *AspSol* requires less effort, because most of its additional code corresponds to class or aspects declarations, import lines and constructors. Considering bytecode size, *AspSol* showed to be 35.7% bigger than *TangSol* and 16% bigger than *PatSol*. This happens because the AspectJ compiler instruments the Java code in order to provide some reflection capabilities (which is an inherent characteristic of the language current version), generating new and bigger classes, which are a result of the composition process and present additional members.

With relation to initialization time, it takes 25% and 10.4% more time in *AspSol* than for *TangSol* and *PatSol* respectively. This is a result of the bytecode size increase caused by the AspectJ instrumentation, that also leads to the performance and memory use results shown in the following. In order to evaluate application execution time, we have chosen the most frequently used method of the dictionary, which corresponds to the search itself. For it, we had 30.9% and 12.5% more time than for *TangSol* and *PatSol* respectively. Regarding memory use, *AspSol* uses 23% and 14.1% more memory than *TangSol* and *PatSol* respectively.

It is important to notice that while using AspectJ language with J2ME, we have some limitations that are a result of the lack of J2SE classes in J2ME. Because of that, we cannot use some of the language constructions, such as the *cflow* pointcut, or we have to modify some of the AspectJ classes

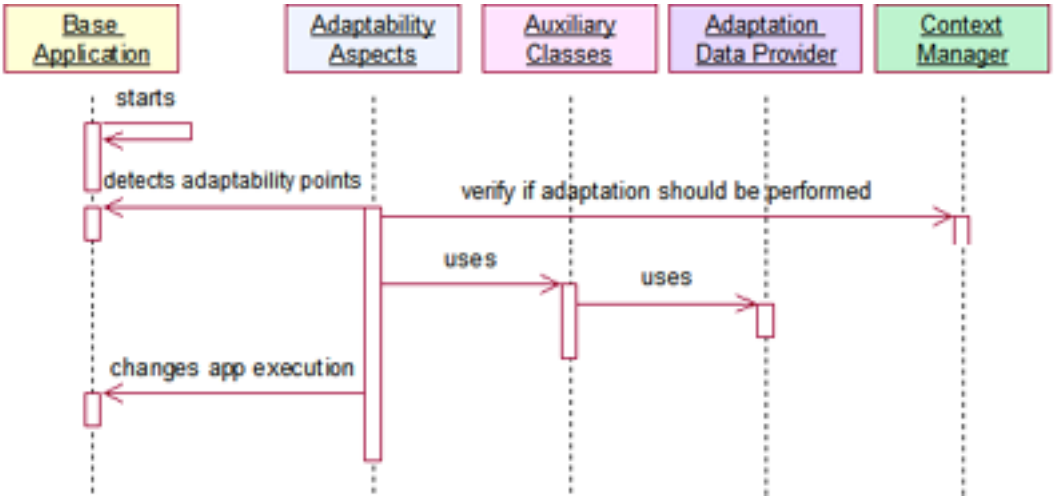

Figure 2: *Adaptability Aspects* dynamics

Table 1: Comparative study results

	Initialization(%)	Execution(%)	Used memory(%)	JAR size (%)	Source(%)
AspSol x TangSol	25,0	30,9	23,0	35,7	2,6
AspSol x PatSol	10,4	12,5	14,1	16,0	-11,6

and include them on the resulting application, what may increase its size.

4. CONCLUSIONS

From the obtained data, we have concluded that the burden on memory use, performance and code size while using aspects is not so significative when compared to a solution with patterns, but can achieve about a third in relation to a tangled version, what might not be desirable depending on the system resources and on the wanted degree of flexibility. The advantages brought by the *Adaptability Aspects* pattern use, such as better reuse, the separation of the adaptability concerns from business code, the pluggability of these concerns (which makes it easy to generate different versions of the same application depending on the device), better maintainability and the ability to divide the team for the implementation of the pattern modules are worthwhile, but the target system characteristics should be analysed. As future work, we want to perform formal experiments in order to obtain quantitative data relative to each of the *Adaptability Aspects* pattern advantages.

5. REFERENCES

[1] A. Dantas and P. Borba. AdapPE: An Architectural Pattern for Structuring Adaptive Applications. In *Third Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2003*, Porto de Galinhas, Brazil, 12th–15th August 2003. To appear. Temporary version at http://www.cin.ufpe.br/~sugarloafplop/-acceptedPapers.htm.

[2] A. Dantas and P. Borba. Developing adaptive J2ME Applications Using AspectJ. In *Proceedings of the 7th Brazilian Symposium on Programming Languages*, pages 226–242, May 2003.

[3] T. Elrad, M. Aksit, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of aop. *Communications of the ACM*, 44(10):33–38, October 2001.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[7] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer–Verlag.

[8] K. Lyytinen and Y. Yoo. Issues and challenges in ubiquitous computing: Introduction. *Communications of the ACM*, 45(12):62–65, 2002.

[9] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Jonhson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self- adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[10] V. Piroumian. *Wireless J2ME Platform Programming*. Sun Microsystems Press, 2002.

[11] J. W. Yoder, F. Balaguer, and R. Johnson. Architecture and design of adaptive object-models. *ACM SIGPLAN Notices*, 36(12):50–60, 2001.