Using Aspects to Make Adaptive Object-Models Adaptable

Ayla Dantas[1⋆], Joseph Yoder[2], Paulo Borba[⋆⋆], and Ralph Johnson

[1] Software Productivity Group Informatics Center
Federal University of Pernambuco
Recife, PE - Brazil - PO Box 7851
add,phmb@cin.ufpe.br
[2] Software Architecture Group Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
yoder@refactory.com, johnson@cs.uiuc.edu

Abstract. The unrelenting pace of change that confronts contemporary software developers compels them to make their applications more configurable, flexible, and adaptive. In order to achieve this, software designers must provide flexible architectures that can more quickly adapt to changing requirements. Adaptive-Object Model (AOM) is an architectural style intended to provide this flexibility by providing a meta-architecture that allows requirements changes to be performed and immediately reflected at runtime. However, AOMs internal structures are sometimes difficult to extend and maintain. In this case, we can say AOM systems are not adaptable, although they are adaptive [1]. This paper proposes the use of Aspect-Oriented Programming in order to make AOM systems simpler to evolve, specially regarding the inclusion of new adaptive requirements.

1 Introduction

Adaptability is an increasingly important requirement of software systems. To achieve this requirement, software designers must provide flexible architectures that can quickly adapt to changing requirements. Sometimes, user requirements are such that the system will even need to adapt at runtime. In those cases, architectures are designed to adapt to new user requirements by retrieving descriptive information that can be interpreted at runtime. Those are sometimes called "reflective architectures" or "meta-architectures".

This paper focuses on a way to enhance a particular kind of reflective architecture, called Adaptive Object-Model (AOM) architecture [2, 3], through the use of Aspect-Oriented Programming (AOP) [4]. We can make AOMs more flexible by modularizing the adaptation part of the architecture. This modularization through the use of AOP can make AOMs more maintainable and adaptable,

⋆ Supported by CNPq.
⋆⋆ Partially supported by CNPq, process number 521994/96–9.

especially in relation to adaptability requirements. Adaptability here means the ability to change or be changed to fit varying circumstances, such as requirements changes. By using AOM, we organize the code in a way that makes it easier to adapt to such changes. After this organization, a requirement change can be performed, for example, by simply replacing the interpreted metadata, which can be stored on the database or in an XML file. However, AOM systems' code is usually difficult to understand and maintain [3]. In this case, we can say AOM systems are not adaptable because it is not easy to include unanticipated adaptive requirements on them. This happens because the code, not the metadata, should change in this case.

The maintainability problems with AOM's code happen because the adaptive behavior is often mixed with the business logic and GUI code of the application. Business classes that provide dynamic properties or behavior usually contain the code to obtain and interpret the data from a file or database that specifies the new properties or behavior. This is called code tangling. Besides that, such code, sometimes related to the same adaptability requirement, may be scattered throughout many classes, a phenomenon known as code scattering. When this happens, it is hard to understand and change in the code the points where new dynamic data or metadata should be obtained.

Aspect-Oriented Programming is a better way to structure Adaptive-Object Models and implement adaptive applications. For better observing the adaptability implementation problem using only AOM, and how it can be solved combining AOM and AOP, we have gradually implemented some possible adaptive behaviors on a dictionary application.

The remainder of this paper is organized as follows. The two following sections briefly present AOM and AOP respectively. In Section 4, we present the benefits of extending the AOM application by using AOP. Then, Section 5 summarizes this paper, giving some conclusions about using AOP to improve AOM.

2 AOM Overview

Many systems are developed to solve a specific problem and flexibility is not included as one of the requirements. Extending or maintaining these types of systems can be a difficult task. Simple changes can be made by parameterizing system's properties that can be read at runtime from initialization files or databases.

However, parameterizing properties will not work for complex adaptations such as adding new types of entities or properties. Adaptations can be even more complex if the system needs to add new behavior in response to a given property change, or dynamically decide which algorithm to use depending on the property type. For such adaptations, Adaptive-Object Models has been shown to be a good solution.

AOM architectures are usually made up of several smaller design patterns, such as the Composite, Interpreter, Builder, and Strategy [5], along with other dynamic patterns such as TypeObject [6], Property [7], and RuleObjects [8, 9].

Most AOM architectures apply the TypeObject and Property patterns together, which results in an architecture called TypeSquare [2].

By organizing an application using these patterns, we can represent application features, attributes and rules (or strategies) as metadata that can be interpreted in a running system. Since classes, attributes and relationships (which can be a kind of property) are represented as metadata in AOM systems, they have an underlying model based on instances rather than classes. Then, adaptations to the object-model are made by changing metadata, which can then be reflected into a running system by instantiating new EntityTypes, PropertyTypes, and Rules.

The main advantage of AOM systems is ease of change. They can even evolve to where they allow users to configure and extend their systems "without programming" and make the process of changing them quickly. However, there may be a higher initial cost in developing this kind of system, because it must be as general as possible in order to improve extensibility and making possible many sorts of adaptations. Generally, these systems are also difficult to understand because several classes do not represent business abstractions ; the object-model is represented in metadata. However, auxiliary Graphical User Interface (GUI) tools and editors can be used to alleviate this problem. Another problem presented by the AOM architecture is performance, since it is based on the interpretation of metadata.

So, before deciding to use AOM or not, an analysis of the degree of adaptability must be done. This analysis should consider which parts of the system really need to be highly adaptive. For instance, if the developed system has to be a highly configured one, or if the rules or properties of this system might change often, AOM can be a good choice, even considering the problems presented above.

3 AOP

Aspect-Oriented Programming (AOP) is a technology intended to provide clear separation of crosscutting concerns [4]. Its main goal is to make design and code more modular, meaning the concerns are localized rather than scattered and have well-defined interfaces with the rest of the system. In this way, AOP solves the issues raised by some design decisions that are difficult to cleanly capture in code [10]. Those issues are called aspects, and AOP is intended to provide appropriate isolation, composition and reuse of the code used to implement those aspects.

This programming paradigm proposes that computer systems are better programmed by separately specifying the various *concerns* (properties or areas of interest) of a system and some description of their relationships and then, by using AOP environment, these concerns are composed or weaved together into a coherent program [11]. This is especially useful when the *concerns* considered are *crosscutting*. Crosscutting concerns are those that correspond to design decisions that involve several objects or operations, and that, without AOP, would

lead to different places in the code that do the same thing, or do a coordinated simple thing. Some examples of *crosscutting concerns* are: logging, distribution, persistence, security, authentication, performance, transactions integrity, etc.

AspectJ is one of the most widely used AOP languages. It is a general-purpose aspect-oriented extension to Java [12]. This language supports the concept of join points, which are well-defined points in the execution flow of the program [13]. It also has a way of identifying particular join points (pointcuts) and change the application behavior at join points (advice).

4 Using AOP to improve AOMs

In this section, we describe how AOP can solve some of the problems noticed in the AOM implementation of some adaptive requirements in a dictionary application. In order to do that, we have implemented some adaptabilities purely using AOM and then, implemented them again combining AOM with AOP, through the *Adaptability Aspects* [14] pattern. By doing that, we could verify when the AOP use was appropriate and when it was not. The *Adaptability Aspects* is an architectural pattern for structuring adaptive applications using aspects. It was used here in order to make a better use of AOP.

4.1 Using AOM in the Dictionary

The dictionary application is a cellular phone application that is capable of translating words from English into Portuguese. It that presents five screens: presentation, main menu, instructions, info and search screens. The main menu presents three options: Query, Instructions and More info...; info screen displays the source and destination translation languages of the dictionary; and in search screen the search is requested and the translation results are shown.

Considering this simple dictionary application, we have implemented some adaptability requirements. One of them intends to provide dynamic source and target translation languages. Another one is able of providing dynamic search engines for the dictionary, making it able of changing the way it performs a search (e.g. if on memory, on a server, etc). Another adaptability requirement is responsible for providing dynamic properties for the dictionary, which can be shown in info screen or in another application screen where they can be edited. The property types may change frequently, changing the way they are shown on the application or even their ability to be edited or not.

To illustrate AOM's use, we show the implementation of one of the dynamic requirements presented above, which we call "Dynamic Dictionary Properties". In order to implement an adaptation with AOM, we must evaluate which patterns should be used to reorganize the application.

As we have previously seen, the info screen presents two properties of the application: the source and the destination translation languages. These properties are implemented as fields of a class called `InputSearchData`, which is part of the model of the application considering the MVC pattern. If the user

requests a new property, the developer may add a new field to this class. However, the application may need dynamic properties, that may exist or not and even those the developer cannot anticipate. To provide these properties and to avoid codification rework when a new property is requested, we can apply the Property [7] pattern on the dictionary application as shown in Figure 1. Then, instead of many fields representing different properties, the `InputSearchData` class presents a collection of properties that may be stored in a hashtable, for example.

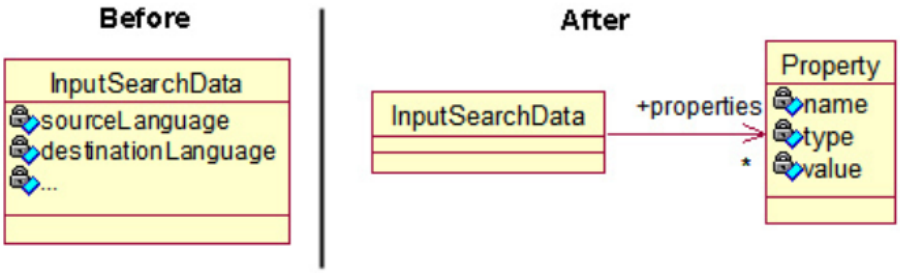

Fig. 1. Property pattern used in the dictionary application

The TypeObject [6] and Strategy [5] patterns can also be used in addition to the Property pattern in order to validate the values of the dictionary properties. Therefore, for each `Property`, there is a `PropertyType` instance associated with it and there are also Strategies for validating dictionary properties. This validation can be used for determining whether a given Property is editable or not and whether it should be presented in a given screen, such as a screen for getting user preferences. Part of the new organization of the application in order to deal with dynamic properties and strategies related to those properties is illustrated by Figure 2.

In the case of the dictionary application, we have implemented an interface, `PropertyValidator`, and classes implementing the `isValid` method, which evaluates if a property is valid according to its type. With such hierarchy, we can use the Strategy pattern to easily change the property validator, by using the `StrategyObject` class. Then, for dynamically changing the validation of the properties shown on a giving screen (such as info screen), we may simply change the `specificInstanceName` attribute of the `StrategyObject` responsible for that.

Following the organization presented by Figure 2, dynamic dictionary properties are obtained from the `InputSearchData` class. The strategies, properties, and their types are represented in XML files that change from time to time and must be interpreted.

There are at least two classes we have implemented that would use the dynamic properties managed by the `InputSearchData` class: the `InfoScreen` class,

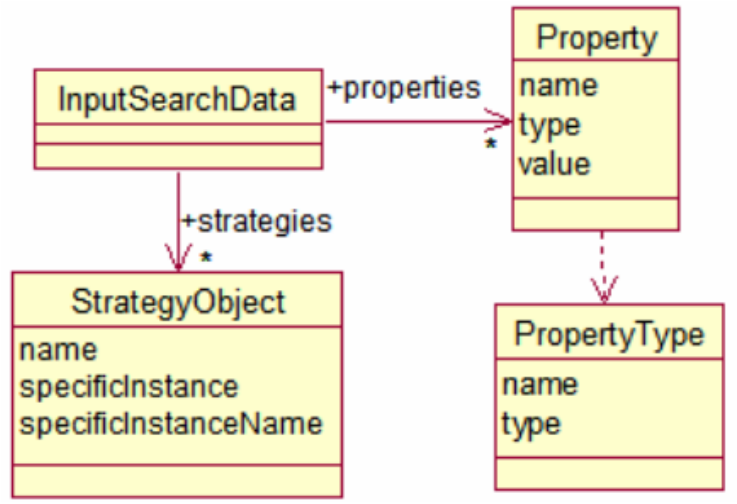

Fig. 2. Using the Properties, TypeObject and Strategy Pattern on the dictionary

which displays all dictionary properties; and the `UserDataScreen`, a new screen responsible for showing dictionary properties that can be edited, such as user name and password.

Reading in the dynamic data from the XML file is done by using an Adaptation Data Provider module implemented using AOM. This module is part of the *Adaptability Aspects* pattern, but can also be used by adaptive systems that do not use aspects (see [14]). It is responsible for obtaining the dynamic properties, types and strategies that represent part of the application. The main class in this module is the `AppAOMManager` class. Any adaptation data is obtained through this class.

In order to build the `InfoScreen` or `UserDataScreen`, the dictionary dynamic properties must be obtained from the `AppAOMManager` and validated, according to their types, which can be dynamically defined. According to this implementation, by simply changing the type of a property, or by including a new property in the metadata that is interpreted, we can dynamically change the application.

This metadata might change frequently and the application might behave differently according to these changes. Therefore, the Adaptation Data Provider objects should be reloaded from time to time. Besides that, the dynamic parts of the application should access this module at certain execution points. For example, before showing `InfoScreen` or `UserDataScreen`, we must rebuild them. To do so, we may request an update of dictionary properties shown by these screens.

The adaptability data is requested when some dynamic information is necessary for the application. This can be done during the dictionary application startup or can be a frequent action performed in several parts of the code. This

is especially true when we remember that AOM systems are also known for their ability to immediately adapt to metadata changes [3]. The Observer pattern can also be used to notify an application that updates to the object-model have been made.

In our pure AOM implementation of the adaptive dictionary application, several points of the code call methods to update the application objects due to changes in the adaptability data. This is a problem known as code scattering. For example, in order to update dictionary properties according to dynamic data, we may have to include reload invocations and application object updates before the `InfoScreen` or the `UserDataScreen` are displayed. We might also have to make calls to the reload invocations after the application startup, at the moment some classes are instantiated, etc.

As we can see, the `DictionaryController` class calls methods intended to obtain new dynamic data and reorganize the application objects and the screens to be shown. However, this behavior is not directly related to this class business logic, and can lead to poor maintainability.

For reload operations at certain execution points, we invoke methods from the `InputSearchData` class that request data from the Adaptation Data Provider module. This request also synchronizes some properties and associates validation strategies to these properties. After the `InputSearchData` class initialization, we must also perform some kinds of reloads.

In fact, several problems arise while implementing adaptability requirements using AOMs or similar techniques and they are not specific of dictionaries. One of them is code tangling. This happens in the dictionary because the `InputSearchData`, or any other class that is supposed to obtain dynamic data, has to know about synchronization mechanisms that may vary according to the Adaptability Data Provider module implementation. Besides that, what must change or not change when dynamic data is obtained may also vary according to new requirements. Adaptive applications, especially very dynamic ones, may change a lot the execution points where they must adapt (obtain new data and change). Consequently, if we want to change those points, we need to modify adaptation code scattered throughout many classes and for many versions. Therefore, code tangling and scattering make the adaptability implementation hard to change and thus less adaptable. As the adaptability concern generally crosses many parts of the code, we can say it is a crosscutting concern.

In order to solve those problems and provide a higher degree of reuse and the ability to easily plug in/out adaptation features, we propose the use of aspects. Aspects help isolate the configuration of dynamic adaptations and thus make AOMs more adaptable. To illustrate this, we show in the following how we have extended the dynamic dictionary properties concern implementation using the AspectJ [12] language.

4.2 AOP Use

In this section we describe how AOP can solve some of the problems noticed in the AOM implementation pointed out in the previous section. In order to do

that, we use some ideas of the *Adaptability Aspects* pattern. We primarily deal with the Dynamic Dictionary Properties adaptability requirement, pointing out the utility of AOP in improving its AOM implementation.

By using aspects we can modularize adaptations and the application execution points that should trigger them (for example, a given method call or execution, a field get or set, etc.). An aspect in AspectJ is a modular unit of crosscutting implementation. As we have previously seen, the configuration of the adaptations is a crosscutting concern, because it generally crosses many parts of the code and can make that code difficult to understand. For implementing this concern in the case of the dynamic properties, we have used an adaptability aspect called `DynamicProperties`. It is part of the Adaptability Aspects module of the *Adaptability Aspects* pattern. It is responsible for verifying if an adaptation should be performed and then performing the necessary changes, using for both tasks elements of other pattern modules.

By defining pointcuts, aspects identify collections of points in the execution flow of a program where behavior changes must happen. In the case of the `DynamicProperties` aspect, some of the pointcuts we have defined are called `showingInfoScreen` and `inputSearchDataCreation`.

In the first one, we identify the execution of the `showScreen` method, from the `DictionaryController` class, when the screen to be shown is `InfoScreen`. A similar definition is also done for the `UserDataScreen`. In order to perform some actions when these execution points are achieved, we must define some advice. In the advice, as in the pointcut definition, there must be some parameters, which are used to expose the application context at those execution points. With a `before` advice that corresponds to the `showingInfoScren` pointcut we may define what must be done before this execution point. The second pointcut definition illustrated above corresponds to the execution of the `InputSearchData` class constructor. By using an `after` advice, we may define the actions to be performed after this execution. The auxiliary methods called inside these advice reload the source of adaptability data and synchronize the current application objects, or part of them, according to the new data. If we want to change the adaptation points, or what must be done at those points, we simply change the pointcuts or the advice declarations respectively. Both are defined in a modular unit, and the access to the source of dynamic data is confined in the aspect code (or auxiliary classes used by it).

In AspectJ, as in other AOP languages, there is a process for composing the base source code or even compiled code with the aspects code. This process is known as weaving. If the user does not want dynamic data for the dictionary properties, the aspect responsible for that is simply not provided as input for the weaving process. If this aspect is provided, the configuration of the adaptability is localized. So, with aspects, it becomes easier to configure the adaptation. Therefore, the adaptation itself is adaptable.

There is also an additional benefit: the AOM code and the ways to integrate it with the normal code are separated. Therefore, we can more easily change the

way we want a system to adapt. We can also more easily change or evolve the non-AOM part of the system.

5 Conclusions

As we could see, adaptability is becoming a common requirement, and implementing it can be hard. Adaptive-Object Models have been, to some extent, successfully used to implement dynamic systems. Understanding AOMs can help developers more quickly build systems that are highly flexible, because part of these systems is represented in metadata that can be easily changed. However, AOMs sometimes lead to solutions that can be hard to maintain in order to include new adaptive capabilities or change the code of the existing ones. This happens because besides reorganizing the application by using some patterns, it also suggests that the system behavior and dynamic elements must be represented using metadata, which is interpreted at runtime. This interpretation of metadata and associated actions are scattered throughout many classes. This makes the business logic code and GUI code become mixed with the code for providing the adaptability.

There may also be some business rules which do not need to be adaptive. By mixing adaptability code with fixed code, code tangling arises. This can lead to problems while maintaining the system; specifically if an extension to the AOM is needed.

In order to minimize those problems, and make AOMs more adaptable, we use Aspect-Oriented Programming. From the previous sections, we could see that AOP can be useful for introducing adaptability with AOMs. It brings two main advantages:

- Makes it easier to change the execution points where dynamic data must be obtained;
- Isolates the adaptability actions from the application business logic and GUI code.

This is a result of the modularization property provided by AOP through the use of aspects. In AspectJ, we change the "adaptability points" by giving new pointcut definitions that generally expose application objects (the pointcut parameters). Then, we define the adaptability actions by using advice (`before`, `after` or `around`), which explore the exposed instances in order to change the application behavior.

By using AOM, we make our applications able to adapt at runtime to users' or developers' new requirements. This happens because we represent the parts of the systems intended to be dynamic in metadata that is interpreted and we organize the system using some patterns. However, retrieving dynamic data and updating application objects is a crosscutting concern related to many adaptability requirements. Implementing it using pure OO programming may lead to code that is difficult to understand and evolve. Therefore, we propose the use of aspects for modularizing this concern in each adaptability requirement.

Besides improving AOM implementations, AOP can also be used to implement some of AOM patterns avoiding direct changes in the application code. However, in some cases, this may bring maintainability problems, because the code may become more difficult to understand, as we could see in other adaptability requirement implementation.

By using the *Adaptability Aspects* pattern, lightweight aspects are used in order to avoid problems that may result from a bad use of aspects. The aspects should only be used to avoid code tangling and scattering while implementing adaptability, and where they allow a better comprehension of the code.

After this work, we conclude that AOM and AOP are a good combination in order to provide flexible applications that are easy to evolve both by interpreting metadata or by changing source code. In the latter case, this will happen because the adaptability configuration will be better isolated.

References

1. Lieberherr, K.: Workshop on Adaptable and Adaptive Software. In: Addendum to the Proceedings of the 10th annual Conference on Object-Oriented Programming Systems, Languages, and Applications, ACM Press (1995) 149–154
2. Yoder, J.W., Balaguer, F., Johnson, R.: Architecture and Design of Adaptive Object-Models. ACM SIGPLAN Notices **36** (2001) 50–60
3. Yoder, J.W., Johnson, R.: The Adaptive Object-Model Architectural Style. In: Working IEEE/IFIP Conference on Software Architecture 2002(WICSA), Montral, Qubec, Canada (2002)
4. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. Communications of the ACM **44** (2001) 33–38
5. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1994)
6. Johnson, R., Wolf, B.: "Type Object". Pattern Languages of Program Design 3. Addison-Wesley (1998)
7. Foote, B., Yoder, J.: Metadata and Active Object-Models. Collected papers from the PLoP '98 and EuroPLoP '98 Conference Technical Report wucs-98-25, Dept. of Computer Science, Washington University (1998)
8. Arsanjani, A.: Rule Object Pattern Language. In: Proceedings of PLoP2000. Technical Report wucs-00-29, Dept. of Computer Science, Washington University Department of Computer Science. (2000)
9. Arsanjani, A.: Using Grammar-oriented Object Design to Seamlessly Map Business Models to Component-based Software Architectures. In: Proceedings of The International Association of Science and Technology for Development, Pittsburgh, PA (2001)
10. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect–Oriented Programming. In: European Conference on Object–Oriented Programming, ECOOP'97. LNCS 1241, Finland, Springer–Verlag (1997) 220–242
11. Elrad, T., Filman, R.E., Bader, A.: Aspect-Oriented Programming. Communications of the ACM **44** (2001) 29–32
12. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: Getting Started with AspectJ. Communications of the ACM **44** (2001) 59–65

13. Team, A.: The AspectJ Programming Guide. At http://www.eclipse.org/aspectj (2003)
14. Dantas, A., Borba, P.: Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications. In: Third Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2003, Porto de Galinhas, Brazil (2003) Temporary version at http://www.cin.ufpe.br/~sugarloafplop/-acceptedPapers.htm.