Developing Adaptive J2ME Applications Using AspectJ

Ayla Dantas
add@cin.ufpe.br

Paulo Borba
phmb@cin.ufpe.br

Informatics Center
Federal University of Pernambuco
Recife, Pernambuco, Brazil

ABSTRACT

This paper evaluates the use of AspectJ, a general-purpose aspect-oriented extension to Java to provide adaptive behavior for Java 2 Micro Edition (J2ME) applications in a modularized way. Our evaluation is based on the development of a simple but non-trivial dictionary application where new functionalities were incrementally implemented using AspectJ.

Our main contribution is to show that AspectJ can be used to implement several adaptive concerns. Concerns of this type would provide the application the ability to have a different behavior according to changes in its environment. We also analyze advantages and disadvantages of using AspectJ for this intent. During the experiment, we observed some possible patterns and guidelines that could be useful for developers interested in making a J2ME system adaptive. Our implementation was also compared with corresponding purely Object Oriented alternatives, which have significant disadvantages.

Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques – Aspect-Oriented Programming D.3.2 [**Programming Languages**]: Language Classifications - AspectJ.

General Terms

Experimentation, Standardization, Languages.

Keywords

Aspect-oriented programming, separation of concerns, AspectJ, AOP applications, software architecture, J2ME, adaptability

1. INTRODUCTION

In this paper we evaluate AspectJ [6], a general-purpose aspect-oriented extension to Java [4], as a tool to develop adaptive Java 2 Micro Edition [12] applications. Adaptive applications require that changes on the environment lead to new system behaviors. However, it is important to provide adaptive behavior following quality and productivity requirements. We have made an experiment on which new functionalities were incrementally added using AspectJ and where those constraints were satisfied. During our experiment, we analyzed the advantages and drawbacks of using AspectJ to implement several adaptive concerns, which was our main contribution. We also observed some good practices and possible patterns during the development and compared our implementation with purely Object Oriented (OO) solutions.

In order to evaluate the applicability of AspectJ for the development of modular adaptive applications, we have developed a J2ME dictionary, which was at first designed with its basic functionalities only. Then, we implemented some new concerns using AspectJ, making the application self-adaptive, which means that it is capable of modifying its own behavior in response to changes in its operating environment [11]. A self-adaptive behavior example could be the inclusion of more options in the application main menu according to a server response. However, our dictionary would be classified as a closed-adaptive one, because it is self-contained and not able to support the addition of new behaviors during runtime. This means that all the adaptive behavior should be programmed before, but only activated or deactivated in response to environment changes. This happens because installation of new functionalities at runtime is not currently supported by J2ME. Some more details about this technology are described through out this paper.

Some of the adaptive concerns implemented on the dictionary were internationalization, changing and addition of screens, customization, and cashing. We used AspectJ because it is supposed to provide adequate support for separating concerns and minimizing the efforts necessary when reconfiguring a system to add, modify, or delete features. However, to be sure about the benefits of it, we have also compared our solution to purely OO implementations of these concerns, which had significant disadvantages.

The remainder of this paper is organized as follows. Section 2 describes the dictionary application and Section 3 gives an overview about the AspectJ language. Section 4 explains the implemented concerns, describing some techniques and possible adaptive patterns used. Finally, Section 5 shows some related works and concludes the paper, discussing some results and problems found during the development.

2. THE APLICATION

In order to evaluate AspectJ in J2ME, we decided to develop a dictionary application. It is a very simple MIDP-based application (also known as MIDlet [8, 12]) capable of translating an entered word from English to Portuguese. Although it is simple, our dictionary has the complexity of a typical J2ME application of this nature. It contains four different screens: presentation screen, main menu (with two options: Query and Instructions), dictionary screen (where the search will be requested and the results shown),

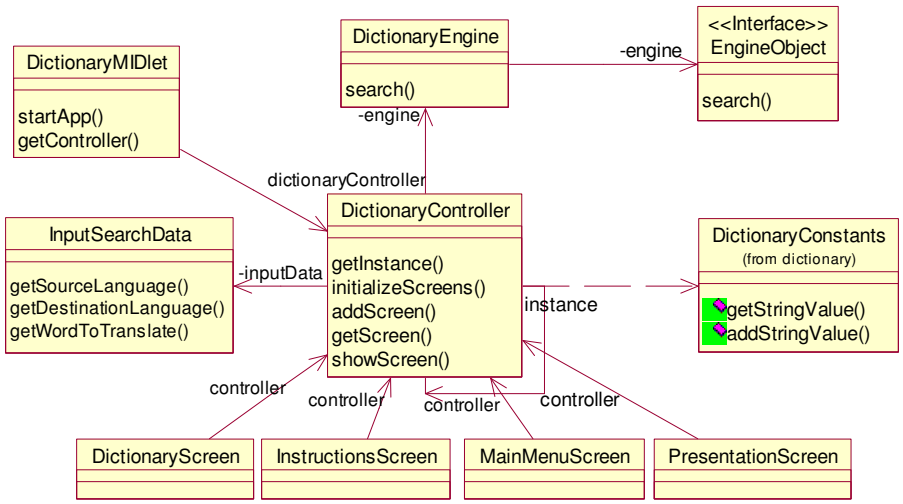

Figure 1 – Dictionary application UML model

and instructions screen. None of them are colorized. The dictionary searches the requested English word on memory and displays its translation to Portuguese on the screen.

The application structure follows the Model View Controller architectural design pattern [2], on which the main classes are the `DictionaryMIDlet` and the `DictionaryController`, which contains a `DictionaryEngine` object that is responsible for the search (see Figure 1). The `DictionaryMIDlet` inherits from `javax.microedition.midlet.MIDlet` class and must be specified on the application descriptor file (JAD). The `DictionaryController` is the controller, and manages the application screens, its main operations and properties. The application initial screens are represented by four classes, which implement `javax.microedition.Displayable` interface: `DictionaryScreen`, `InstructionsScreen`, `MainMenuScreen` and `PresentationScreen`. All of them store the single controller instance in order to have the ability to execute some operations when their commands are selected by a key press.

A dictionary is a good example of an application that should be adaptive and where many aspects could be introduced and analyzed. Besides that, we were looking for an application where we could observe a ubiquitous behavior. From the simple application described earlier, we wanted to build an adaptive one. This new dictionary could, for example, be able to change its source and destination translation languages, according to the user's country. It could also be able to decide its search mechanism (server, local storage, or memory) according to some information collected (as the time spent for each kind of search, the number of successful answers, and so on). This application could also be colorized or internationalized according to the user logged into the system.

3. ASPECTJ OVERVIEW

Aspect-oriented programming is a programming technique that provides explicit language support for modularizing design decisions that cross-cut a functionality-decomposed program. Instead of spreading the code related to a design decision throughout a program's source, a developer is able to express the decision within a separate, coherent piece of code [15]. For these characteristics, AOP has been considered one of the methods to support separation of concerns.

AspectJ is a general-purpose aspect-oriented extension to Java [6]. Aspect-oriented languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points. Join points are well-defined points in the program flow [14].

In AspectJ, we have the concept of an aspect, which is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, constructors, initializers, named pointcuts, and advice. For instance, we could have the following aspect definition:

```
public aspect Screens {…}
```

AspectJ join points are well-defined points in the execution flow of the program. Pointcut designators identify particular join points by filtering out a subset of all the join points in the program flow [6]. A pointcut example is shown in the following:

```
pointcut showingScreen(): execution (public
  void showScreen(byte)) ;
```

This pointcut captures the execution of any public method called `showScreen`, that has a byte parameter, and has `void` as its return type. This is just one example of the several kinds of join points AspectJ provides.

Advice declarations are used to define additional code that runs when a pointcut is reached. For example, we can define code to be run before a pointcut, as the following example shows:

```
before(): showingScreen(){
  System.out.println("A screen will be shown");
}
```

With this code, before the method `showScreen` is executed, a message is displayed on the standard output. Besides the `before` advice, AspectJ also provides `after` and `around` advice. `after` advice runs after the computation under the join point finishes. `around` advice runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all [14].

AspectJ also has a way of affecting a program statically that is called Introduction. With it, a program static structure can be changed, namely, the members of its classes and the relationship between classes. For example, we could insert a new constant on `DictionaryController` class, as the following code ilustrates:

```
public static final byte
  DictionaryController.REGISTRATION_SCREEN = -3;
```

Aspects are AspectJ's unit of modularity for crosscutting concerns and are defined in terms of pointcuts, advice and introduction. The aspect code is combined with the primary program code by an aspect weaver [15]. In AspectJ current version (1.0.6) this can only be done at compile time, and the source code is required for this process.

4. INTRODUCED CONCERNS

Using AspectJ, we have introduced several adaptive concerns on the simple dictionary application: Customization, Screens, Caching, Internationalization, and Exception Handling concerns. Customization Concern is responsible for customizing the application, changing some of its parameters, what makes it present new functionalities. Screens concern modifies the application current screens and also adds new ones to it. Caching concern provides caching of adaptive data obtained remotely, avoiding network accesses every time this data is needed. Internationalization concern internationalizes the strings used on the application, giving its value according to a defined application language. Exception Handling Concern changes the application behavior when any exception is handled.

4.1 Customization Concern

This concern is useful to support application functionalities changes, such as the source and destination languages used for translation and the mechanism used to search a word. This is done by changing some of the application attributes, as the `InputSearchData` instance used by the controller, and the `EngineObject` used by `DictionaryEngine` respectively (see Figure 1). Customization Concern was implemented using three aspects (`Customization`, `DictionaryCustomization`, and `UpdateObjectConfig`) and some auxiliary classes. They are illustrated on Figure 2, which is an adapted UML class diagram, in which a class abstraction is used to represent an aspect. This figure also shows their relation with some of the auxiliary classes and the dictionary application affected code (`DictionaryController` and `DictionaryMIDlet`).

The Customization aspect was designed to affect the `MIDlet` startup, capturing at that point the `MIDlet` instance, allowing easier application functionality changes. `Customization` is an abstract `aspect` composed of one `pointcut`, three `advice`, and three abstract methods. Its `pointcut` is shown below:

```
pointcut MIDletStart(MIDlet midlet):
  execution(void startApp()) && target(midlet);
```

Note that this `pointcut` has a parameter, which is a `MIDlet` object. This parameter must be present in the `advice` definition too, and can be used in their bodies, allowing them to change the application functionalities, since the `MIDlet` controls the application lifecycle and has access to its components.

The `Customization` aspect also defines three general `advice`:

```
before(MIDlet midlet): MIDletStart(midlet){
  doBeforeAdaptations(midlet);
}

void around (MIDlet midlet) :
  MIDletStart(midlet){
  boolean shouldProceed =
  doAroundAdaptationAndPossiblyProceed(midlet);
  if (shouldProceed)
    proceed(midlet);
}

after(MIDlet midlet): MIDletStart(midlet){
  doAfterAdaptations(midlet);
}
```

These `advice` change the application behavior by invoking three methods that are called before (`doBeforeAdaptations`), after (`doAfterAdaptations`), and instead of (`doAroundAdaptationAndPossiblyProceed`) the `MIDletStart` pointcut. These methods are abstract and should be implemented by another ap-

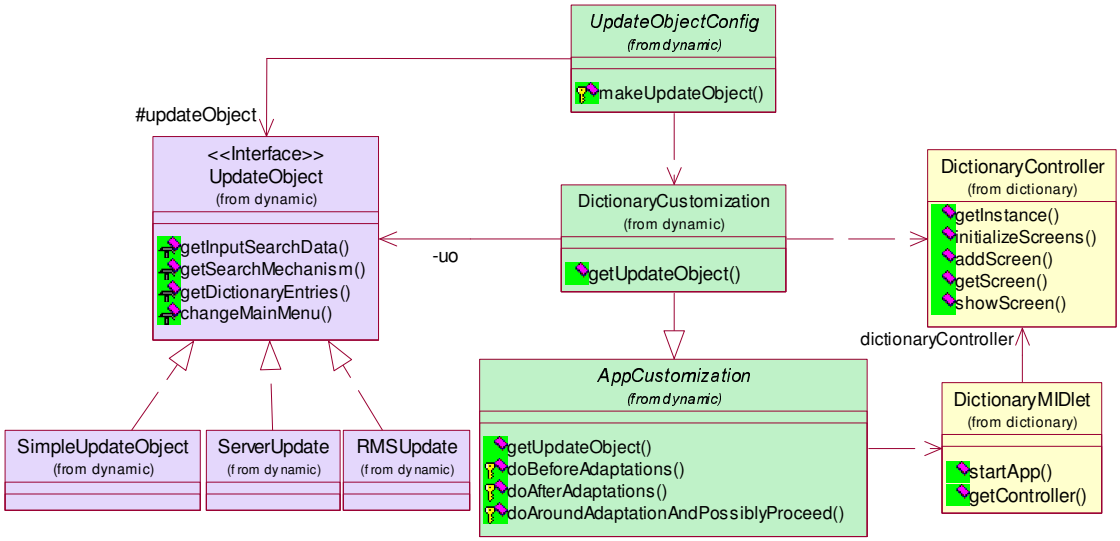

Figure 2 – Customization concern implementation in AspectJ

plication specific aspect, specific for the application. In fact, the `Customization` aspect is very general and can be used by any J2ME application because all of them have a `MIDlet`, and therefore, a `startApp` method is always executed. The specificities should be given by its implementing `aspect`, which does not need any AspectJ special construction, making the developer work easier.

In the dictionary application, `DictionaryCustomization` aspect implements `Customization` aspect, actually changing the dictionary application using `UpdateObject` interface operations (see Figure 2). By these operations, we can obtain the data necessary for performing these changes such as a new `InputSearchData` object or a new `EngineObject` (see Figure 1). An `UpdateObject` instance can be obtained by invoking `getUpdateObject` method, which is a `DictionaryCustomization` aspect member. The following piece of code extracted from this aspect exemplifies the use of `UpdateObjects` and some application functionality changes:

```
1: protected void doBeforeAdaptations(MIDlet mid){
2:   try {
3:     DictionaryMIDlet midlet =
4:       (DictionaryMIDlet) mid;
5:     InputSearchData data =
6:       getUpdateObject().getInputSearchData();
7:     InputSearchData current =
8:       midlet.getController().getInputData();
9:     if (data!=null){
10:      current.setDestinationLanguage(
11:        data.getDestinationLanguage());
12:      current.setSourceLanguage(
13:        data.getSourceLanguage());
14:    }
15:  } catch (ClassCastException e){}
16:}
```

This method casts the `MIDlet` object passed as a parameter to the application specific `MIDlet` class (line 4). This object is used to change the source and destination languages of the dictionary according to an `InputSearchData` object returned by the invocation of `getInputSearchData` method on an `UpdateObject` (lines 5 and 6). From the MIDlet object obtained at line 4, we access the controller and its current `InputSearchData` object (lines 7 and 8) and change its source and destination languages attributes (lines 10, 11, 12 and 13).

The `UpdateObjectConfig` aspect resolves the type of `UpdateObject` to be used by `DictionaryCustomization` aspect to provide the adaptive behavior. It defines the `gettingAdaptationObject` pointcut:

```
pointcut gettingAdaptationObject(
  DictionaryCustomization adap):
    target (adap) && execution(public
    UpdateObject getUpdateObject());
```

This pointcut captures the execution of `getUpdateObject` method executed from `DictionaryCustomization` aspect. There is also an `around` advice, which substitutes the normal execution of this method by returning an `UpdateObject` instance, which was created by invoking `makeUpdateObject` method.

```
UpdateObject around(Customization adap):
  gettingAdaptationObject(adap){
    return this.makeUpdateObject();
}
```

We have noticed that this aspect works as an AbstractFactory design pattern [2] which helps the Bridge pattern composed by `DictionaryCustomization` aspect, `UpdateObject`, and its possible implementing classes in the definition of the specific object to be instantiated (see Figure 2). Besides that, we can also visualize the `DictionaryCustomization` aspect implementing the State Pattern, as it changes the dictionary application state, partially represented by the `InputSearchData` object.

Another design pattern also observed, but not explained before was the Strategy [2]. `DictionaryCustomization` aspect also changes the `DictionaryEngine` class substituting its current `EngineObject` instance, by another one by using the

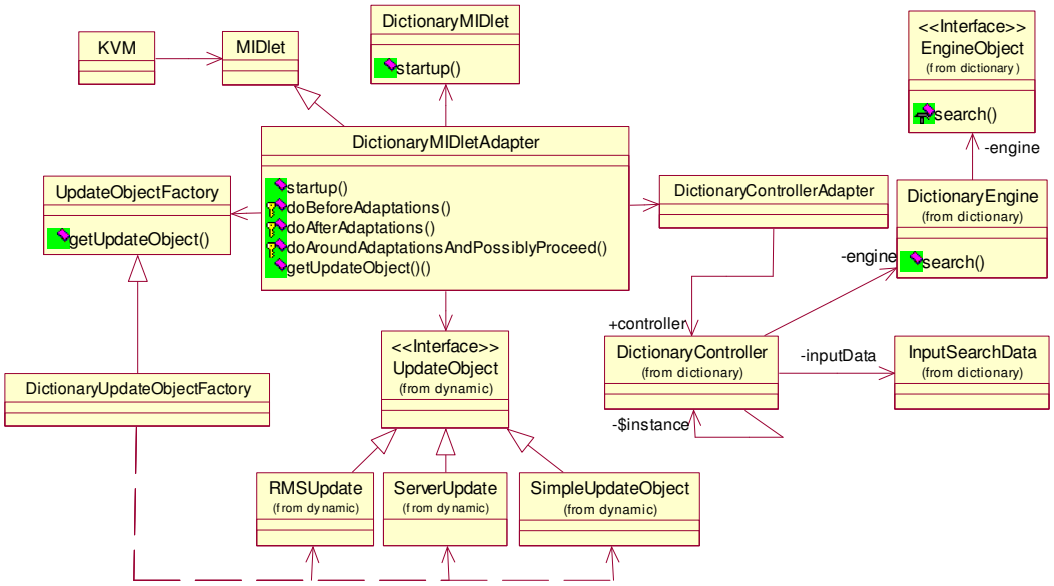

Figure 3 – Customization concern purely OO implementation

doAfterAdaptation method, which is called by the after advice from Customization aspect. Three different EngineObjects were implemented: LocalPersistenceEngineObject, which searches the translation locally, on the MIDP Record Management System (RMS); ServerSearcherEngineObject, which connects to a URL and requests the translation; and VolatileEngineObject, which searches on the memory. Therefore, changing the EngineObject instance, we have a different search algorithm being used.

As it could be observed, Customization aspect was designed to be reused by other J2ME applications, because all of them have to define a class that is a MIDlet with a startApp method. This aspect would be more useful if a controller class were also provided, because with it, from the aspect, we could easily customize the application.

The DictionaryCustomization and UpdateObjectConfig aspects cannot be reused because they are application specific, but their pattern can be easily followed. The former defines three methods, which specify what should be done before, after, or instead of the MIDlet startup. It also has a method called getUpdateObject which can be called by the other three methods and returns an object that provides adaptability data specific for the application. The second aspect configures the UpdateObject type that should be returned by this method.

In a purely OO implementation, this concern could be implemented using some design patterns, as Figure 3 illustrates.

We would need two Adapters [2], one for the DictionaryMIDlet and another for the DictionaryController. DictionaryMIDletAdapter would have to extend MIDlet class defining its methods and the methods shown on the Figure 3 to make adaptation actions before, after and around the MIDlet startup. The DictionaryControllerAdapter would use a DictionaryController instance and change the application parameters, represented in the diagram by InputSearchData and EngineObject classes. UpdateObjectFactory would factor the UpdateObject used by DictionaryMIDletAdapter to change the application.

As we can see, the AspectJ implementation is simpler. Besides that, it does not require to change the application descriptor file (JAD) because the MIDlet class will be the same. This does not happen on the purely OO implementation, where an adapter class is necessary and it should be defined on the JAD file. On the other hand, with AspectJ, we need to introduce another step before the application deployment: we must preprocess the code, generating a weaved source code, and then compile and package the .class files using J2ME tools.

4.2 Screens Concern

This concern is related to some changes on the application screens. It was implemented by a single aspect, called Screens, and additional auxiliary classes. This aspect was responsible for the colorization of presentation screen, the addition of a new screen after presentation, called Registration screen, the inclusion of new options on the application main menu, and the creation of the new screens related to the new menu options. These changes are good examples of desired adaptive behaviors on J2ME applications. They can be introduced according to an UpdateObject also stored by this aspect. The Screens aspect, its auxiliary classes, and the application affected classes are shown on Figure 4.

As the introduction or changing of screens usually requires new strings to be used on the application and all dictionary application strings are described on DictionaryConstants class, we have introduced the new strings there using AspectJ introduction as shown below:

```
public static final String
  DictionaryConstants.SOURCE_LANGUAGE =
  "Source Language";
```

The DictionaryConstants class also resolves these constants values on the application by its getStringValue method. So, we

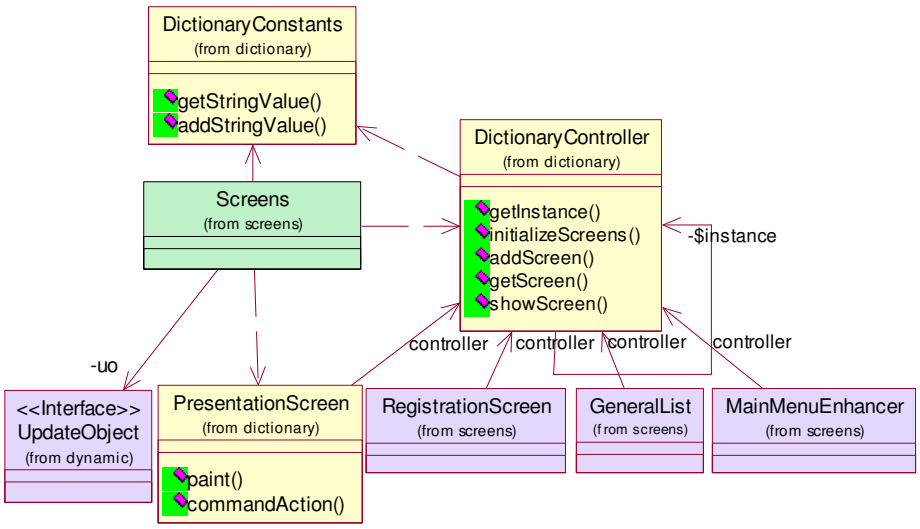

Figure 4 – Screens concern implementation using AspectJ

must also include the values of these new strings, what was done in a static block:

```
static {
  ...
  DictionaryConstants.addStringValue(
    DictionaryConstants.SOURCE_LANGUAGE,
    "Source Language");
}
```

If new screens are introduced, they would also require a constant to identify them in `DictionaryController` class. Using introduction, we included new screens constants as the following example shows:

```
public static final byte
  DictionaryControler.
  SOURCE_LANGUAGE_SELECTION_SCREEN = -4;
```

On the remainder of the aspect definition and on the auxiliary classes invoked later, we can then consider the existence of these screens and strings constants on the corresponding classes.

Now, we explain how the screens of the dictionary application could be changed. To colorize presentation screen we had to define the following pointcut:

```
pointcut changingPresentation(Graphics g,
  PresentationScreen p):
    args(g) && execution(public void
    paint(Graphics)) && target(p);
```

This pointcut affects the execution of the `paint` method on a `PresentationScreen` object with a `Graphics` parameter. These pointcut parameters were used, because on the advice definition, we needed both the `Graphics` and the `PresentationScreen` instances to paint the screen in a new way. This was done using an `around` advice, which substitutes the application `paint` method previously defined.

To introduce some more screens on the application, we defined the following pointcut:

```
pointcut initializingScreens(
  DictionaryController con): this(con)
  && execution(public void initializeScreens());
```

This pointcut affects the execution of `initializeScreens` method from `DictionaryController`. We would like to recommend the creation of such a method that should be called at `startApp` execution from the `MIDlet`. This method is responsible for creating each application screen and including them on the controller managed screens group.

The advice that affects the `initializingScreens` pointcut is an `after` advice, which consequently executes after the initialization of the current application screens. This advice creates a new screen that requests the user login and password, called `RegistrationScreen`. We have also invoked the `changeMainMenu` method from the `UpdateObject` instance. This method has a `boolean` return type and in case of a true answer, the current application main menu would be enhanced with more options, and consequently, new screens should be added to the controller. This was done by `MainMenuEnhancer` auxiliary class, which requires the controller instance that is passed as a parameter for the advice. This controller instance will also be stored by the aspect.

On the advice previously described, we introduced a new screen to the controller, Registration screen. However, in order to show this screen, we need another application change, described on the following pointcut and advice:

```
pointcut presentationCommandAction(Displayable
  p, Command c):
  args(c, p) && if (p instanceof
    PresentationScreen)&&
    execution(public void commandAction(Command,
    Displayable));

void around(Displayable p, Command c):
  presentationCommandAction(p,c){
  if (c==DictionaryConstants.START_CMD){
    this.controller.showScreen(
      DictionaryController.REGISTRATION_SCREEN);
  } else {
    proceed(p,c);
  }
}
```

The pointcut defined above affects the execution of `commandAction` method on which the `Displayable` taken as argument is a `PresentationScreen` object. On the advice definition, we require that every time the START command is pressed from presentation screen, Registration screen should be shown. In case of another command, the `commandAction` method should proceed with its normal execution.

As it could be seen, this aspect cannot be reused because it is specific for the dictionary application. However, if every J2ME application followed a model view controller similar to ours, they could have the same pointcuts definition, and only the advice bodies should be changed. To make this aspect reusable we could follow the same idea described on the Customization concern, that is, we should have an abstract `Screens` aspect and then a `DictionaryScreens` aspect, which defines methods called by each application specific join point and a `getUpdateObject` method which will be called by the others. We could also have a `DictionaryScreensConfig` aspect which would define the kind of `UpdateObject` to be returned by `getUpdateObject` method. This kind of structure was named “Adaptability Pattern”. However, we need more experiments involving other developers to classify it as a real pattern.

In a purely OO implementation, we would need adapters to the `DictionaryController` and also the `PresentationScreen` classes. The additional code invoked by the advice would be there with the calls to the already defined methods from the adaptees, which would mean a bigger source code.

4.3 Caching Concern

The data used for adaptation might constantly change. Therefore, an option would be to have a network point as its source. However, wireless applications that must contact a remote server to access their data present special problems for the developers. Protocol support in J2ME is much more limited than in J2SE (Java 2 Platform, Standard Edition), and some devices might require multiple access methods. In addition, radio communications are much less reliable than the landline connections most Internet standards assume [3].

Observing those problems, we propose a structure that could be used by those who think about getting from servers the data that will be used for adaptation. According to it, the data should be obtained, especially in XML format, and stored on the RMS (Record Management System) in the same format. The connection to the server should rarely be done (for example: once a month) and the data obtained by the server and by the RMS should be handled in the same way, as explained in the following.

The Caching Concern is related to the ability of locally storing any information used for adaptation obtained by a server. This is done by a `Caching` aspect, which affects the methods execution of `ServerUpdate` that is a specific `UpdateObject` as illustrated on Figure 2.

The `Caching` aspect presents four pointcuts. Each of them affects a different `ServerUpdate` method execution. Four after advice are also provided and are responsible for writing on the RMS the object returned as a server response by each method. They do that by invoking `writeServerContentsOnXML` method, which is implemented in the `Caching` aspect. This method uses auxiliary classes specially designed to deal with XML and the RMS in a reusable way.

We can observe that the Caching aspect can affect other aspects execution, as they use `UpdateObjects`. In fact, this happens only when the `UpdateObject` instance being used is a ServerUpdate one. However, this is an advantage, because avoids the duplication of work and modularizes the concern of storing server data. This relation between some dictionary application aspects can be illustrated on Figure 5.

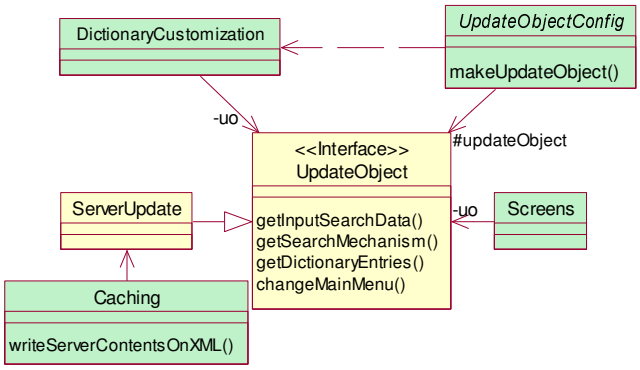

Figure 5 – Caching aspect relation with other aspects

With the implementation of caching concern, the adaptation data can be stored and an `UpdateObject` of type `RMSUpdate` can be used by other classes or aspects to retrieve this data avoiding a server communication.

A purely OO modularized implementation would be the creation of a `ServerUpdateAdapter` class which would use the previously defined `ServerUpdate` class and would place an invocation of a `writeServerContentsOnXML` method after each of its methods. However, this would require application changes in many points, where a `ServerUpdate` object is used.

4.4 Internationalization Concern

This concern's intent is to provide a specific value for a string used on the application according to the application language. It was easy to implement because the application was designed to obtain any of its string values from a single point: by invoking `getStringValue` static method from `DictionaryConstants` class. However, this was a result of a restructure process on the base code.

Internationalization concern was implemented by a single aspect, named `InternationalizationAspect`, and the `Internationalization` auxiliary class. `InternationalizationAspect` stores an instance of this auxiliary class and delegates the function of getting string values to it, replacing the execution of the method `getStringValue` from `DictionaryConstants`. This behavior is shown on the following pointcut and advice declaration:

```
pointcut internationalizing(Object key):
  execution(public static String
  DictionaryConstants.getStringValue(Object))&&
  args(key);

String around(Object key):
  internationalizing(key){
    return internationalization.
    getStringValue(key);
}
```

The `Internationalization` class stores a hash table with the string values related to a specific language. However, the strings inserted by other concerns are not being considered, and therefore, are not internationalized. For example, the new strings used on `Screens` aspect won't be found. We solved this problem by preserving modularization, with the creation of another aspect that was called `InternationalizedScreens` aspect. This aspect affects `Internationalization` class initialization and introduces the values for the new strings used by the Screens concern. The pointcut used is shown below:

```
pointcut internationalizingNewScreensStrings(
  Internationalization in):
  initialization(public
    Internationalization.new()) && target(in);
```

An OO implementation of Internationalization concern could use again the Adapter pattern, but would produce many changes around the application with the substitution of every `DictionaryConstants` access by its adapter. We would have a great impact because strings are used all around the application.

4.5 Exception Handling Concern

This concern is implemented by a single aspect that changes the application behavior when any exception is handled. This concern should be useful when we do not want the developer to worry about the action to be taken when an exception is caught. It was implemented using `ExceptionHandling` aspect, which defines the following pointcut and advice:

```
pointcut handlingException(Exception e):
  handler(Exception) &&  args(e);

before(Exception e): handlingException(e){
  //… Action to be taken
}
```

Note that this aspect is generic and can be reused by any application. In the advice body we can, for example, write the exception message on a log file, or build a screen and show an error message for the user when an exception is handled. Even though the Exception caught is generic, we can use artifacts like the `instanceof` operator to handle in a special way a specific exception.

Without AspectJ, it would not be so simple to perform a change in the way an exception is handled in a modular way.

5. CONCLUSIONS

We have concluded that AspectJ is useful to implement several adaptive concerns in a modular and simple way. Corresponding

purely object-oriented solutions provide modularization, but usually require more changes to the application source code and to its general structure, when this adaptive behavior is incrementally inserted, as in our experiment. We have considered in our experiment a J2ME system and showed that, even in a platform characterized by so many restrictions that lead to less modularization, less use of design patterns, and a greater focus on performance, it was possible to provide modularization without significant performance impacts.

Using J2ME, we have identified some special points of execution (join points) that simplify changes to the application behavior in order to make it adaptive. Some of these points are present in every J2ME application, and others appear when we use the model view controller architecture and some guidelines we propose.

We have observed, as a good practice, the use of auxiliary classes by the aspects. Those classes encapsulate complex services which are invoked by the advice. In this way, those classes could be reused and its maintainability is improved, since the aspects code becomes much simpler. We actually consider that our implementation consists of three clear parts: the base code, the aspects, and the auxiliary classes. The aspects part acts as a "glue" between the other two, what has been already observed elsewhere [10]. Also, every change in the aspects should be done with care since AspectJ is a powerful language: a simple change to an aspect can affect several parts of the application.

We have noticed that better quality and productivity can be obtained by restructuring the base code [10], so that it exposes necessary join points and we can reuse auxiliary code. We have also defined special auxiliary classes to handle server connections, XML data and RMS manipulations that should improve the reuse. We have also concluded that being able to supply new behaviors after the implementation activity using aspect-oriented programming does not diminish the importance of the design activities to obtain applications prepared for adaptation.

Some of the problems identified in other experiments [5, 13] refer to the interferences between aspects. We have also faced this problem but could solve it by creating aspects that should be included when conflicting aspects have to be used together. This was explained when we discussed about Internationalization concern and the creation of `InternationalizedScreens` aspect (see Section 4.2) and justifies again the importance of design activities.

In relation to the aspects reuse, we could observe that every concern discussed here could be implemented using reusable aspects, following the Customization concern idea (see Section 4.1). Each reusable aspect would contain the definition of pointcuts, advice and abstract methods to be implemented by other aspects specific for the application being developed. With such structure, the developer of the adaptive behaviors would not have to know AspectJ, but simply to implement some methods which will be executed at special points. However, with such implementation, we would have a negative performance impact due to the number of classes that must be loaded.

AspectJ source code showed to be around 20% smaller than other corresponding object-oriented alternatives where modularization was considered. A similar result has also being obtained in another work [13]. However, with the current AspectJ version (1.0.6), the generated bytecodes were bigger, up to 15%, than similar pure OO solutions. This difference reduces to 10% using obfuscators.

Another problem detected during our experiment was the lack of J2SE (Java 2 Standard Edition) classes in J2ME. Some classes that can provide dynamic adaptability, such as `ClassLoader` and those from the Reflection package, are not required by the J2ME specification. In the future, this is likely not be a problem and the proposals presented here will be able to be applied for dynamically adaptable software systems too. The lack of some J2SE classes was also a problem in the definition of the aspects, because some constructions from AspectJ, as `cflow`, required other non available classes, and were consequently avoided. However, such constructions, if possible, would increase a lot the application size, since their required classes would have to be included on the JAR file too, which contains all the application classes.

Despite the pointed drawbacks, AspectJ seemed to be useful for providing adaptive behavior, represented by several concerns, for J2ME applications. This fact could also be true for other Java platforms. However, we would like to make more experiments involving several different developers and application domains to actually observe the accuracy of our evaluation and the usefulness of our guidelines and possible patterns.

6. REFERENCES

[1] Chamberlain, J. Master J2ME for live data delivery: Support multiple applications on multiple devices with this J2ME framework. May 31, 2002. http://www.javaworld.com/javaworld/jw-05-2002/jw-0531-j2me.html

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Desing Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[3] Glosh, S. J2ME record management store: Add data storage capacities to your MIDlet apps. May 2002. http://www-106.ibm.com/developerworks/java/library/wi-rms/?dwzone=java

[4] Gosling, J., Joy, B., Steele, and Bracha, G.. The Java Language Specification. Addison-Wesley, second edition, 2000.

[5] Kenzle, J., and Guerraoui, R. AOP: Does It Make Sense? The Case of Concurrency and Failures. In ECOOP 2002, pp. 37-61.

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting Started With AspectJ. Communications of the ACM 44(1), PP. 59-65, October 2001.

[7] Knudsen, J.. Parsing XML in J2ME[tm] XML in a MIDP Environment. March 7, 2002. http://wireless.java.sun.com/midp/articles/parsingxml.

[8] Mahmoud, Q.H. J2ME MIDP and WAP Complementary Technologies. February 2002. http://wireless.java.sun.com/midp/articles/midpwap.

[9] Murphy, G. C., Lai, A., Walker, R. J., and Robillard, M. P. Separating features in source code: An exploratory study. In Proceedings of the 23rd International Conference on Software Engineering (May 2001).

[10] Murphy, G. C., Walker, R. J., Baniassad, E. L. A., Robillard, M. P., Lai, A., and Kersten, M. A. Does Aspect-Oriented Programming Work? Communications of the ACM 44(1), PP. 75-77, October 2001.

[11] Oreizy, P., Gorlick, M. M., Taylor, R. N., Heimbigner, D., Jonhson, G., Medvidovic, N., Quilici, A., Rosenblum, D. S., and Wolf, A. L. An Architecture-Based Approach to Self-Adaptive Software. In IEEE Intelligent Systems, 1999, pp. 54-62.

[12] Piroumian, Vartan. Wireless J2ME Platform Programming. Sun Microsystems Press. 2002.

[13] Soares, S., Laureano, E., and Borba, P. Implementing Distribution and Persistence Aspects with AspectJ. In OOPSLA'02. November 4-8, 2002.

[14] The AspectJ Team. The AspectJ Programming Guide. (1.0.6) At http:// aspectj.org, 2002.

[15] Walker, R. J., Baniassad, E. L. A., and Murphy, G. C. An initial assessment of aspect-oriented programming. In Proceedings of the 21st International Conference on Software Engineering , May 1999.