Journal of Universal Computer Science, vol. 9, no. 8 (2003), 935-955
submitted: 24/2/03, accepted: 30/5/03, appeared: 28/8/03 © J.UCS

Developing Adaptive J2ME Applications Using AspectJ

Ayla Dantas
(Federal University of Pernambuco, Brazil
add@cin.ufpe.br)

Paulo Borba
(Federal University of Pernambuco, Brazil
phmb@cin.ufpe.br)

Abstract: This paper evaluates the use of AspectJ, a general-purpose aspect-oriented extension to Java, to provide adaptive behavior for J2ME applications in a modularized way. Our evaluation is based on the development of a simple but non-trivial dictionary application where new adaptive behavior was incrementally implemented using AspectJ. Our main contribution is to show that the AspectJ language can be used to implement several adaptive concerns, which allow the application to have different behaviors according to changes in its environment. We also compare our implementation with corresponding pure Java alternatives, identify disadvantages of using AspectJ and propose some possible patterns.
Key Words: Aspect-Oriented Programming, Separation of Concerns, AOP Applications, Software Architecture, Adaptability
Category: D. 3, D. 2, D.2.11

1 Introduction

In this paper we evaluate the AspectJ [Kiczales et al. 2001] language, a general-purpose aspect-oriented extension to Java [Gosling et al. 2000], as a tool to develop adaptive Java applications, especially the Java 2 Micro Edition (J2ME) [Piroumian 2002] ones. Although we focused on this application domain, many of our results can be applied to other Java platforms. We use J2ME because it is targeted to applications domains that have adaptability as a common requirement.

Adaptive applications behave differently, according to changes on the environment. Implementing this kind of application involves complex issues, so it is important to provide adaptive behavior following quality and productivity factors. We have made an experiment on which new adaptive behavior was incrementally added using AspectJ and where those constraints were considered. During our experiment, we analyzed the advantages and drawbacks of using AspectJ to implement several adaptive concerns, which is our main contribution. We also observed some good practices and possible patterns during the development and compared our implementation with pure Java alternatives using some design patterns, considering aspects such as modularization, code size, and maintainability.

In order to evaluate the applicability of AspectJ for the development of modularized adaptive applications, we have developed a simple but non trivial J2ME dictionary, which was at first designed without adaptiveness. Then, we implemented some new concerns using AspectJ, making the application adaptive, that is, capable of modifying its own behavior in response to changes in its operating environment [Oreizy et al. 1999]. For example, an implemented adaptive behavior is the inclusion of more options in the application main menu according to a server response. But our dictionary is 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 the adaptive behavior should be programmed before deployment, but only activated or deactivated in response to environment changes. This happens because loading code at runtime is not currently supported by J2ME. Some more details about this technology are described throughout the paper.

Using AspectJ, we have introduced some adaptive concerns to the initial dictionary application: Customization, Screens, and Internationalization concerns. We have also introduced the Caching concern, which provides support for the others. The Customization concern is responsible for customizing the application by changing some of its parameters. This makes it behave differently when performing its core functionalities. The Screens concern modifies the application current screens and also adds new ones to it. The Internationalization concern internationalizes the strings used on the application, giving their values according to a defined application language. The Caching concern provides caching of data obtained remotely, avoiding network accesses every time this data is needed.

We used AspectJ because its constructs are supposed to provide adequate support for separating concerns and minimizing the efforts necessary when reconfiguring a system to add, modify, or delete features. However, some refactorings were sometimes necessary in order to achieve our objectives.

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. Sections 4, 5, 6 and 7 explain the implemented concerns, describing some techniques, possible adaptive patterns and our experiment evaluation. Finally, Section 8 discusses some related work and concludes the paper, presenting some results and problems found during the development.

2 The Application

In order to evaluate AspectJ for developing J2ME applications, we developed a dictionary application. It is a simple MIDP-based application (also known as MIDlet [Mahmoud 2002] [Piroumian 2002]) capable of translating a given word from English into Portuguese. Although simple, our dictionary has the

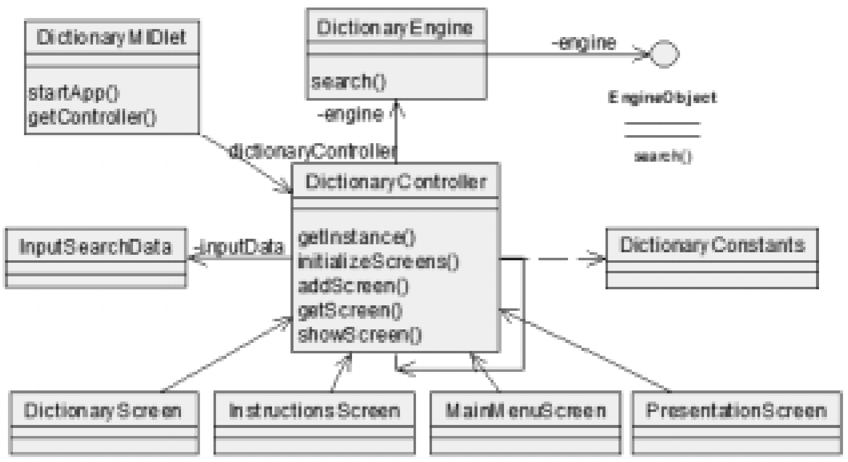

Figure 1: Dictionary application UML model

complexity of a typical J2ME application. It contains four different screens: presentation screen, main menu (with two options: Query and Instructions), dictionary screen (where the search is requested and the results are shown), and instructions screen. None is colorized. The dictionary searches the requested English word on memory and displays its Portuguese translation on the screen.

The application structure follows the Model View Controller architectural pattern [Gamma et al. 1994]. The main classes are `DictionaryMIDlet` and `DictionaryController`, which refers to a `DictionaryEngine` object that is responsible for the search (see Figure 1).

The `DictionaryMIDlet` class inherits from `javax.microedition.midlet.MIDlet` class and must be specified on the application descriptor file (JAD). The `DictionaryController` class is the *controller* and manages the application screens, its main operations, and properties. The screens, or the *view* element of the MVC, are represented by four classes, which implement the `javax.microedition.Displayable` interface: `DictionaryScreen`, `InstructionsScreen`, `MainMenuScreen`, and `PresentationScreen`. All of them refer to the single controller instance in order to notify it when their commands are selected by the user. The *model* element of the MVC is represented by the `DictionaryEngine` class and `InputSearchData`, which is a class representing some application properties, such as the source and destination languages used for the translation, and the word to be translated.

From this simple application, we developed an adaptive one. This new dictionary can, for example, change its source and destination translation languages, its search mechanism (for searching on a server, local storage, or memory), its screens, and its current language. These changes are performed according to context changes. In the current version of our application, the information about the context is requested from a server or from the user. Nevertheless, depending on the device and on the APIs provided by it, we could have other ways of providing the mentioned adaptations. For example, the languages used for translation and the current application language could be automatically changed according to the device location, obtained by a connected GPS. Other possible change would be to select the search mechanism according to some collected information (such as the time spent for each kind of search, the number of successful answers, and so on).

3 AspectJ Overview

Aspect-oriented programming is a programming technique that provides explicit language support for modularizing design decisions that crosscut a decomposed program. Instead of spreading the code related to a design decision throughout source code, a developer is able to express the decision within a separate, coherent piece of code [Walker et al. 1999]. For these characteristics, AOP has been considered to support separation of concerns.

AspectJ is a general-purpose aspect-oriented extension to Java [Kiczales et al. 2001] that supports the concept of join points, which are well-defined points in the execution flow of the program [Team 2002]. It also has a way of identifying particular join points (pointcuts) and change the application behavior at join points (advice).

Pointcut designators identify particular join points by filtering out a subset of all the join points in the program flow [Kiczales et al. 2001]. 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 pointcuts AspectJ provides.

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

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

With this advice, a message is displayed on the standard output before the execution of the `showScreen` method. Besides the `before` advice, AspectJ also provides `after` and `around` advice. The first runs after the computation under the join point finishes. The second 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 [Team 2002].

AspectJ also has a way of affecting a program statically. With Introductions a program static structure can be changed. For example, we can change the members of a class and the relationship between classes. The following code illustrates how we can insert a new constant into the `DictionaryController` class:

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

In AspectJ, we also 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 {  }
```

Aspects are defined by grouping pointcuts, advice, and introductions. The aspect code is combined with the primary program code by an aspect weaver [Walker et al. 1999]. In AspectJ current stable version, this can only be done at compile time, and the source code is required for this process.

4 Customization Concern

This concern is useful to support behavior changes in the application core functionalities, such as changing the source and destination languages used for translation and the mechanism used to search a word. This is achieved by changing some of the application parameters, which yield changes to the `InputSearchData` instance used by the controller, and the `EngineObject` instance used by `DictionaryEngine`, respectively (see Section 2). In order to be reusable, the Customization Concern was implemented using three aspects (`Customization`, `DictionaryCustomization`, and `UpdateObjectConfig`) and some auxiliary classes. They are illustrated by Figure 2, which is an adapted UML class diagram, where the *aspect* stereotype represents an aspect. This figure also shows their relation to auxiliary classes and some of the classes (`DictionaryController` and `DictionaryMIDlet`) they affect.

The `Customization` aspect was designed to change the behavior of the `MIDlet` startup, capturing the `MIDlet` instance, and invoking methods before

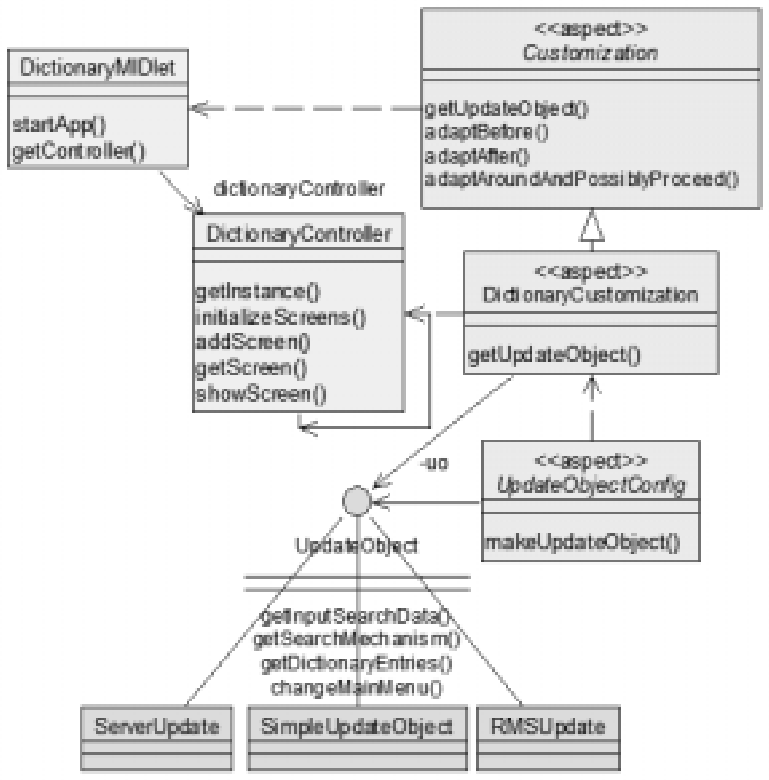

Figure 2: Customization concern implementation in AspectJ

and after the startup. Those methods allow the application to easily change the behavior of the core functionalities at startup. It was designed in a way that it can be reused by any MIDP application, so it is an abstract aspect having the following `pointcut`:

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

Note that this `pointcut` has a parameter, which is a `MIDlet` object. This parameter might be present in the advice definition too, and can be used in their bodies to allow them to change the core application functionalities, since the `MIDlet` instance controls the application lifecycle and has access to its components. The `startApp` execution occurs when the application name is selected on the Java applications list of the device, and is invoked on a `MIDlet` object, which is called the target of the execution.

The `Customization` aspect also defines three general advice:

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

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

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

These `advice` can change the application behavior by invoking three methods that are called before (`adaptBefore`), after (`adaptAfter`), and instead of (`adaptAroundAndPossiblyProceed`) the `MIDletStart` pointcut. These methods are abstract and should be implemented by another aspect, specific for a given application. In fact, the `Customization` aspect is very general and can be used by any MIDP application because all of them have a `MIDlet` and, therefore, a `startApp` method is executed when the application starts up. The specific aspect does not need any AspectJ construction, making the developer work easier, since this aspect will look like a class defining three methods.

In the dictionary application, the `DictionaryCustomization` aspect extends the `Customization` aspect, defining the specific methods, which actually change the dictionary application using the `UpdateObject` interface operations (see Figure 2). With these operations, we can obtain the data that specifies changes such as a new `InputSearchData` object or a new `EngineObject` instance (see Figure 1). An `UpdateObject` instance can be obtained by invoking the `getUpdateObject` method, which is defined in `DictionaryCustomization`. The following piece of code extracted from this aspect illustrates the use of `UpdateObject` for getting data that specifies how the behavior of a core functionality should be changed:

```
1: protected void adaptBefore(MIDlet mid) {
2:   if (mid instanceof DictionaryMIDlet){
3:     DictionaryMIDlet midlet =(DictionaryMIDlet) mid;
4:     InputSearchData data =
5:       getUpdateObject().getInputSearchData();
```

```
 6:     InputSearchData current =
 7:       midlet.getController().getInputData();
 8:     if (data!=null) {
 9:       current.setDestinationLanguage(
10:         data.getDestinationLanguage());
11:       current.setSourceLanguage(
12:         data.getSourceLanguage());
13:     }
14:   }
15: }
```

This method casts the `MIDlet` object passed as a parameter to the application specific `MIDlet` class (line 3). This object is used to change the source and destination languages of the dictionary specified by the `InputSearchData` object returned by `getInputSearchData` (line 5). If this method returns null, no adaptation is performed. With the `MIDlet` object obtained at line 3, we access the controller and its current `InputSearchData` object (line 7) and change its source and destination languages attributes (lines 9, 10, 11 and 12). We could also have separately obtained each application property from `updateObject`. We used the `InputSearchData` object in order to make the application easily evolvable, but not necessarily adaptive. However, grouping the main application properties in a single object makes this task easier. This means that these configuration changes were anticipated, but the fact of making these changes in a dynamic adaptive way not.

4.1 Configuring the Adaptation

The `UpdateObjectConfig` aspect configures the adaptation, resolving the type of `UpdateObject` to be used by `DictionaryCustomization` aspect to provide the adaptive behavior. It can be a `SimpleUpdate`, a `RMSUpdate` or a `ServerUpdate`, among others. It defines the `gettingAdaptationObject` pointcut:

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

This pointcut captures the execution of the `getUpdateObject` method, whereas an `around` advice substitutes the normal execution of this method by returning an `UpdateObject` instance, which is created by invoking `makeUpdateObject` method. The `UpdateObject` type is defined by a code read from a local file and it indicates how the data used for adaptation will be retrieved.

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

4.2 Comparing with Design Patterns

We notice that the UpdateObjectConfig aspect works as the Abstract Factory design pattern [Gamma et al. 1994] in the sense that it helps the Bridge pattern composed by the DictionaryCustomization aspect, UpdateObject, and its possible implementations in the definition of the specific object to be instantiated (see Figure 2). In this pattern implementation, the Abstract Factory participant is represented by an aspect instead of a class. 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 is the Strategy [Gamma et al. 1994]. The DictionaryCustomization aspect also changes DictionaryEngine instances substituting their current EngineObject instance. Three different EngineObjects were implemented: LocalPersistenceEngineObject, which searches for 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 collection structure stored on main memory. Therefore, by changing the EngineObject instance, we have a different search strategy being used.

4.3 Reusability

As observed, the Customization aspect was designed to be reused by other J2ME applications, because all of them have a MIDlet with a startApp method.

The DictionaryCustomization and UpdateObjectConfig aspects cannot be reused because they are application specific, but the aspects of different J2ME applications might follow the same pattern. The former defines three methods that specify what should be done before, after, or instead of the MIDlet startup. It also has a method called getUpdateObject, which is usually called by the other three methods and returns an object that provides application specific adaptability data. The second aspect configures the UpdateObject instance returned by that method.

Although we have focused on a J2ME example, the technique explained for customization can be used on other kinds of applications, replacing the MIDlet

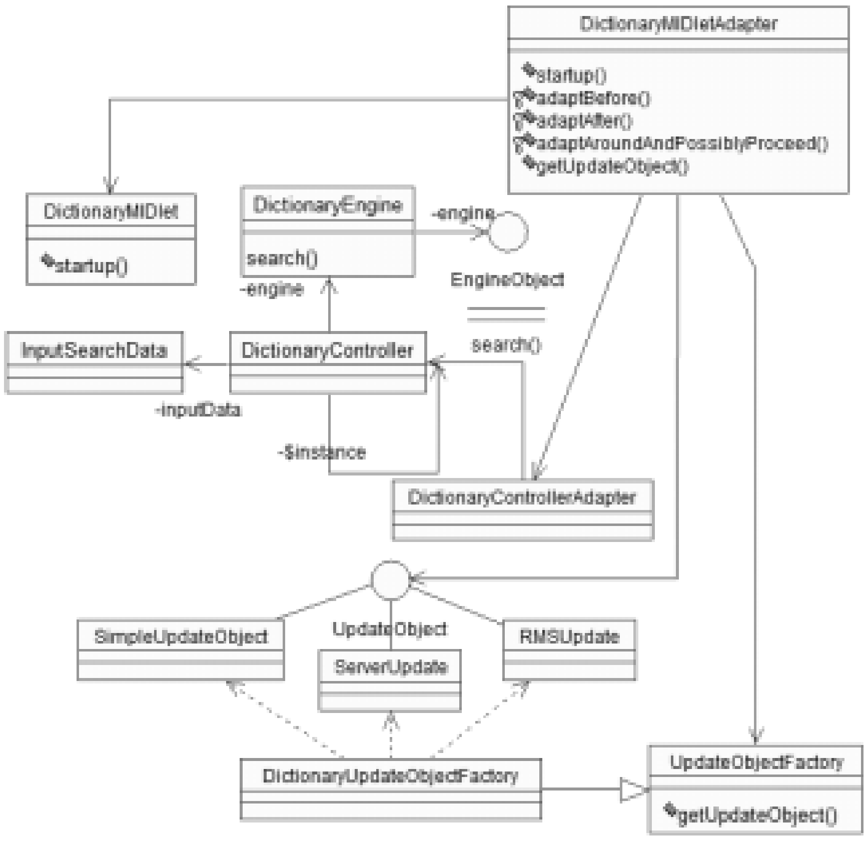

Figure 3: Customization concern purely OO implementation

object by another one that enables application properties (and consequently behavior) changes.

4.4 Comparing with a pure Java implementation

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

One of these patterns is the Adapter [Gamma et al. 1994], which allows us to use delegation in some method executions and to invoke some actions before or after this delegation. In the dictionary application, we would need two Adapters, one for the `DictionaryMIDlet` and another for the `DictionaryController`. If this aspect intercepted more joinpoints, many other adapters would be necessary. The `DictionaryMIDletAdapter` would have to extend the MIDlet class defining its methods and the methods shown on 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 by comparing Figure 2 and Figure 3, the AspectJ implementation is simpler and has a smaller source code. Besides that, it does not require us to change the application descriptor file (JAD) because the `MIDlet` class will be the same for the original and the adaptive dictionary. This does not happen on the purely OO implementation, where an adapter class is necessary and it, instead of the `DictionaryMIDlet`, should be mentioned on the JAD file. On the other hand, with AspectJ, we need to introduce another step before application deployment: we must preprocess the code, generating a weaved source code, and then compile and package the bytecodes files using J2ME tools. However, this is just a limitation of the current stable version of AspectJ that can be easily solved.

It is important to notice that the implementation shown before provides adaptability by customizing the application when it is starting. Nevertheless, in order to provide other adaptability points, we must simply define a pointcut exposing the application specific MIDlet instance and advice that invoke `ApplicationCustomizationAspect` methods (where *Application* is the application name). An example related to the Dictionary application is shown below:

```
pointcut otherAdaptationPoints(DictionaryMIDlet
  midlet): execution (public DictionaryController
    getController()) && this(midlet);

before (DictionaryMIDlet midlet):
  otherAdaptationPoints(midlet){
    DictionaryCustomizationAspect.
      aspectOf().adaptBefore(midlet);
}
```

This way we can reuse the methods defined on the application specific customization aspect in other pointcuts. This is done by using the `aspectOf` method, which returns an aspect instance.

An alternative implementation would be to redefine the pointcut from `Customization` aspect defining every point where the customization adaptation should be performed.

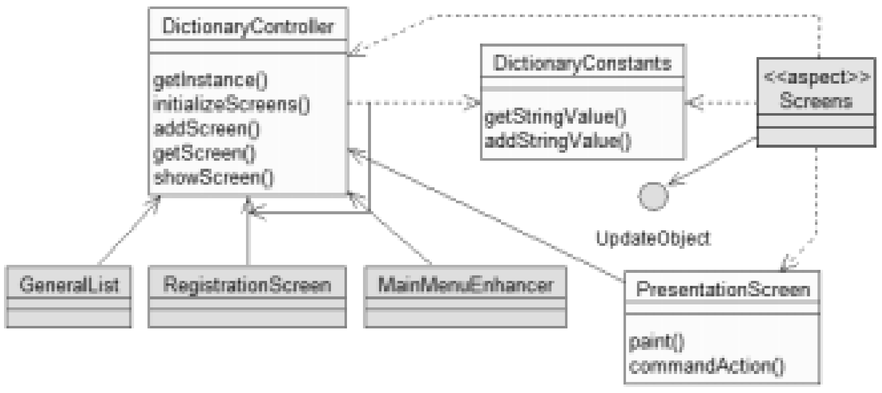

Figure 4: Screens Concern implementation using AspectJ

5 Screens Concern

The Screens concern is related to changes on the application screens. It was implemented by a single aspect, called `Screens`, and additional auxiliary classes. This aspect is responsible for the colorization of the 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 in J2ME applications. They are introduced or removed when data provided by `UpdateObject` instance indicates that screens updates should be made. The `Screens` aspect, its auxiliary classes, and the application affected classes are shown in Figure 4. This use of aspects as a glue between the application core functionality and auxiliary classes has also been considered good practice elsewhere [Murphy et al. 2001].

5.1 Using AspectJ introduction for the Screens Concern

As the introduction or modification of screens usually requires new strings to be used in the application, and all dictionary application strings are described in the `DictionaryConstants` class, we introduce the new strings there using AspectJ introduction as shown below:

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

Once the Screens aspect is used, these introductions are always done, because this is a static change.

The `DictionaryConstants` `getStringValue` method yields these constants values. This method was included after the refactoring made while introducing the Internationalization concern, as will be explained in Section 7, and is invoked every time a `String` value is used in the application. Therefore, we must add the values of the strings from the new screens to the hash of `String` values from `DictionaryConstants` class. This is done in a static block:

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

When new screens are introduced, we need constants to identify them. We thus included new screens constants as the following example shows:

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

The Controller needs these constants because every screen change is made by it, using this constant to identity each `Displayable` instance (screen) to be shown.

5.2 Changing the application screens

We now explain how the screens of the dictionary application could be changed. To colorize the presentation screen we defined the following pointcut:

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

This pointcut refers to the execution of the `paint` method with a `Graphics` parameter on a `PresentationScreen` object. The `paint` method is internally invoked when the presentation screen is shown. The pointcut parameters are necessary because the advice definition needs both the `Graphics` and the `PresentationScreen` instances to paint the screen in a new way. This is specified by an around advice, which substitutes the application `paint` method previously defined when the data provided by the `UpdateObject` indicates that this adaptation should be carried on.

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

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

This pointcut refers to the execution of the `initializeScreens` method from `DictionaryController` class. This method is called by the `startApp` method. This method is responsible for creating the application screens and including them on the controller managed screens group. We have also introduced an after advice that affects this pointcut and uses some auxiliary classes (`RegistrationScreen`, `MainMenuEnhancer` and `GeneralList`) to build and show new screens on the application. Nevertheless, this adaptation is activated or not, according to this aspect `UpdateObject` instance.

In order to show the introduced Registration screen, we need to adapt the application in another point, as described on the following pointcut and advice:

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

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

The pointcut defined above denotes the join points where the execution of the `commandAction` method has as argument a `PresentationScreen` object. The advice shows the Registration Screen every time the START command is pressed. For the other commands, the `commandAction` method proceeds with its normal execution. This method is invoked every time a command is pressed, its arguments represent the specific command pressed, and the `Displayable` (screen) where it happened.

This aspect cannot be reused; it is specific for the dictionary application. However, every J2ME application that follows the same model view controller pattern shown here could have the same pointcuts, and only the advice bodies would be changed. To make this aspect reusable, we could follow the same idea described on the Customization concern: we should have an abstract Screens aspect and then a `DictionaryScreens` aspect, which declares methods called by each application specific advice and a `getUpdateObject` method which is called by the others. We could also have a `DictionaryScreensConfig` aspect that would define the kind of `UpdateObject` to be returned by the `getUpdateObject` method. This actually constitutes an *Adaptability Pattern* which we have applied in some concerns and think that can be applied in several other contexts.

In a purely OO implementation using design patterns, we would need adapters to the `DictionaryController` and also the `PresentationScreen` classes. The additional code invoked by the advice would be tangled with the calls to the methods from the adaptees and we would not achieve a good separation of concerns.

6 Caching Concern

The data used for adaptation might change. Therefore, in order to have access to the corresponding updates, we can obtain this data by server requests. However, wireless applications that contact remote data servers present problems for the developers. For example, 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 [Glosh 2002].

In order to solve those problems, our implementation obtains XML data from the server and stores it on the RMS (Record Management System) in the same format. Therefore, the server does not need to be accessed often, and the data obtained by the server and by the RMS is handled in the same way, as explained in the following. The Caching Concern, which is a support for the adaptability concern, is responsible for locally storing any information used for adaptation obtained by a server. This is done by the `Caching` aspect, which affects the execution of the `ServerUpdate` methods, as illustrated by Figure 5.

The `Caching` aspect has four pointcuts. Each of them corresponds to a different `ServerUpdate` method. Four after advice are responsible for writing on the RMS the server response returned by each method. They do that by invoking the `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 notice that the Caching aspect affects other aspects, as they use `UpdateObjects` as well. In fact, this happens only when the `UpdateObject` instance being used is a `ServerUpdate` one. This relation between the aspects has advantages and disadvantages, as explained in the following. At some moments, it can make more difficult the understandability of the code if the developer does not have much experience with AspectJ or does not use any tools to help in the identification of join points affected by many advice. An advantage of

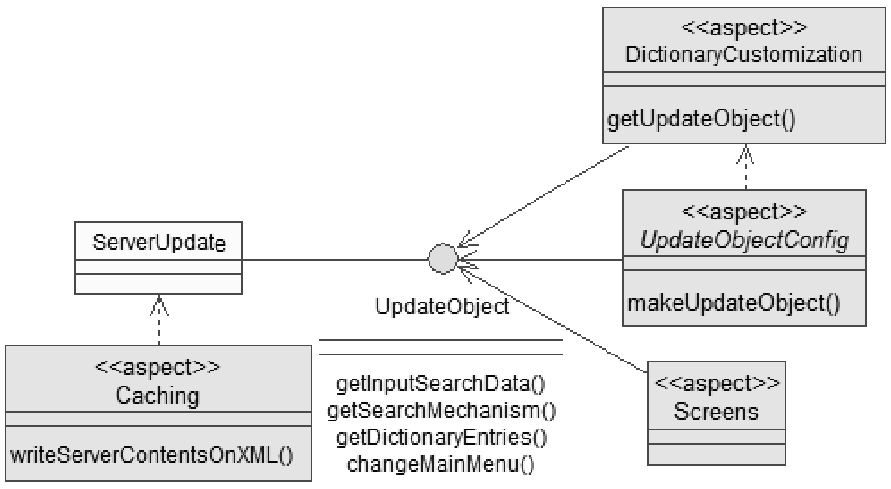

Figure 5: Caching aspect relation with other aspects

this implementation is that it avoids duplication of code, providing reuse, and modularizing the caching concern. Another important thing to observe is that the caching concern does not need to be implemented after the other aspects whose executions are changed by it, because this interference is indirect, via the `UpdateObject` instance. This relation between some dictionary application aspects is illustrated on Figure 5.

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

A purely OO modularized implementation would be similar to the one explained on the end of Section 5, using Adapters.

7 Internationalization Concern

This concerns intent is to internationalize the application by providing a specific value for a given string used on the application according to the application language. It was easy to implement because the application was refactored to obtain its string values in a single way: by invoking the `getStringValue` static method from `DictionaryConstants` class. In fact, when using AspectJ sometimes we need to refactor the code in order to expose some join points or to provide better pointcut definitions, which are easier to understand. Instead of refactoring the application, we have thought about capturing every access to a `String` constant from `DictionaryConstants` class that we wanted to internationalize, but the access to final fields cannot be captured by AspectJ.

After the refactoring, the Internationalization concern was implemented by a single aspect, named `InternationalizationAspect`, and the `Internationalization` auxiliary class. The aspect stores an instance of this auxiliary class and delegates the task of obtaining string values to it, replacing the execution of the method `getStringValue` from `DictionaryConstants`:

```
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 the `Screens` aspect will not be found. We solved this problem by preserving modularization, with the creation of another aspect that was called `InternationalizedScreens` aspect. This aspect affects the `Internationalization` class initialization and introduces the values for the new strings used by the Screens concern. The pointcut used is shown below:

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

An OO implementation of the Internationalization concern could again use the Adapter pattern, redefining the `getStringValue` method and delegating its calls to `DictionaryConstants` class sometimes. Nevertheless, this would require many invasive changes throughout the application, with the substitution in this case of every `DictionaryConstants` instance by its adapter.

On the other hand, we had a similar problem during the refactoring process in order to implement the concern with AspectJ. It was necessary to refactor the code to call the `getStringValue` method. But we have not created additional classes (adapters) as it would be necessary in a pure OO implementation.

We preferred the use of AspectJ to implement this concern because it is part of an adaptability issue. The application language might change occasionally, according to user inputs or by accessing proprietary libraries that could give information about the user localization. We have made alternative AspectJ solutions for the Internationalization concern in the dictionary application, but we have decided to show this one because it presents one of the problems we have while programming in AspectJ, the dependency between aspects, and how we may be able to solve it.

8 Conclusions

We have concluded that AspectJ is useful to implement several adaptive concerns in a modularized and simple way. Corresponding purely object-oriented solutions using design patterns 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. Our J2ME experiments showed that it was possible to provide modularization without significant performance impacts in relation to an OO tangled version. As our focus was on the adaptability concern, this Java platform was chosen. However, many of our results can be also applied to other platforms as well.

Using J2ME, we have identified some special execution points (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 or when some refactoring is done.

A good practice observed during the experiment is 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 their 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 [Murphy et al. 2001]. Also, every change to 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 [Soares et al. 2002].

We have noticed that better quality and productivity can be obtained by restructuring the base code [Murphy et al. 2001]. Sometimes, just with refactoring, some necessary join points are exposed and the code from auxiliary classes can be used in many places. An example is the development of auxiliary classes to handle server connections, XML data and RMS manipulations that should improve the reuse, which are used in many of the concerns related to adaptation. We have also concluded that being able to dynamically 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. But its important to notice that some application changes should be anticipated, that is, the application sometimes must be prepared for change.

Some of the problems identified in other experiments [Kienzle and Guerraoui 2002][Soares et al. 2002] 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 the Internationalization concern and the creation of the `InternationalizedScreens` aspect (see Section 7). This justifies again the importance of design activities when using AOP.

In relation to reuse, we could observe that every concern discussed here could be implemented using reusable aspects, following the Customization concern idea (see Section 4). Each reusable aspect contains the definition of pointcuts, advice and abstract methods to be implemented by other aspects specific for the application being developed. Following such structure, the developer of the adaptive behaviors does not have to know AspectJ, but simply to implement some methods that will be executed at special points. Nevertheless, 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 is considered too. A similar result has also been obtained in another experiment [Soares et al. 2002]. But with the used 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. However, as a future work, we will develop several versions of the same application(s) using or not aspects and using or not patterns and will observe with more details, besides code/bytecode size, aspects such as adaptive capability, modularization, load time, new adaptations inclusion impact, used memory, and code quality.

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 to be a problem and the proposals presented here will be able to be applied for dynamically adaptable software systems too. We could also obtain more adaptability avoiding the use of the `ClassLoader` by using Adaptive Object Models [Yoder et al. 2001], by which we can represent the application structure and behavior using metamodels that can be described as an XML file, for example.

The lack of some J2SE classes was also a problem in the definition of the aspects, because some AspectJ constructions, such as `cflow`, required other non available classes, and were consequently avoided. Nevertheless, such constructions, if allowed, would increase the application size considerably, since their required classes would have to be included in the JAR file too.

Despite the pointed drawbacks, AspectJ seems 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, more experiments, involving several different developers and application domains, are necessary to better observe the accuracy of our evaluation and the usefulness of our patterns.

Acknowledgements

We would like to thank the anonymous referees for making several suggestions. Special thanks go to Rodrigo Rebouas, Franklin Ramalho, Sérgio Soares, Rohit Gheyi and Vander Alves for their important comments. This work was supported by CNPq, a brazilian research agency.

References

[Gamma et al. 1994] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software"; Addison-Wesley (1994).

[Glosh 2002] Glosh, S. "J2me record management store: Add data storage capacities to your midlet apps". (2002). `http://www-106.ibm.com/developerworks/java/library/wi-rms/?dwzone=java`

[Gosling et al. 2000] Gosling, J., Joy, B., Steele, G., and Bracha, G. "The Java Language Specification"; Addison-Wesley, second edition (2000).

[Kiczales et al. 2001] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. "Getting started with AspectJ"; Communications of the ACM, 44, 10 (2001), 59-65.

[Kienzle and Guerraoui 2002] Kienzle, J. and Guerraoui, R. (2002). "AOP: Does it Make Sense? The Case of Concurrency and Failures"; European Conference on Object–Oriented programming, ECOOP'02, LNCS 2374, Málaga, Spain, Springer–Verlag (June 2002), 37-61.

[Mahmoud 2002] Mahmoud, Q. "J2ME MIDP and WAP complementary technologies".(2002). `http://wireless.java.sun.com/midp/articles/midpwap`

[Murphy et al. 2001] Murphy, G. C., Walker, R. J., Baniassad, E. L. A., Robillard, M. P., Lai, A., and Kersten, M. A. K. "Does aspect-oriented programming work?"; Communications of the ACM, 44, 10 (2001), 75-77.

[Oreizy et al. 1999] 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"; *IEEE Intelligent Systems.* 14, 3 (1999), 54-62.

[Piroumian 2002] Piroumian, V. "Wireless J2ME Platform Programming"; Sun Microsystems Press (2002).

[Soares et al. 2002] Soares, S., Laureano, E., and Borba, P. "Implementing distribution and persistence aspects with AspectJ"; Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications. ACM Press, (2002) 174-190.

[Team 2002] Team, A. "The AspectJ Programming Guide" (2002) `http://aspectj.org`

[Walker et al. 1999] Walker, R. J., Baniassad, E. L. A., and Murphy, G. C. "An initial assessment of aspect-oriented programming"; Proceedings of the 21st international conference on Software engineering. IEEE Computer Society Press, (1999) 120-130

[Yoder et al. 2001] Yoder, J. W., Balaguer, F., and Johnson, R. "Architecture and design of Adaptive Object-Models"; ACM SIGPLAN Notices. 36, 12 (2001), 50-60.