AdapPE: An Architectural Pattern for Structuring Adaptive Applications with Aspects

Ayla Dantas[*] and Paulo Borba[†]
Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7851 - 50.732-970 Recife, PE
{add,phmb}@cin.ufpe.br

Abstract

This paper presents an architectural pattern for structuring adaptive applications using aspect-oriented programming in order to obtain separation of concerns. It is composed of known and novel patterns organized so as to provide good maintainability and modularity. [1]

Intent

This architectural pattern is intended to show how to use aspects [13] in order to better structure adaptive applications, which are able to change their behavior in response to context changes [10].

Context

Adaptability has become a common requirement [8] and its implementation usually affects many parts of the code. Most implementations of this requirement lead to tangled code, mixing different concerns such as application business rules, GUI code, and adaptive behavior implementation. It is sometimes hard to include the adaptability concern in new and existing applications in easily maintainable way. Besides that, the mechanisms for accessing contextual information change frequently, which usually demands regular modifications to the application. This is a problem because new contextual information should lead to new application behavior.

[*]Supported by CNPq.

[†]Partially supported by CNPq, grant 521994/96–9.

[1]We followed the POSA (Pattern-Oriented Software Architecture) structure to present our pattern, including an Intent section and distributing the contents of sections to be called *Variants* and *Resolved Example* throughout the pattern.

Problem

Develop reusable and easily maintainable adaptive applications and support flexible mechanisms for obtaining contextual information in different ways.

Forces

In order to solve the problem, *AdapPE* balances the following forces:

- Separate adaptability concerns from other concerns.
- The adaptability functionality might be either plugged in/out and also turned on/off.
- Developers do not need to know a particular Aspect-Oriented Programming (AOP) language.
- The application should be easy to maintain.
- The application can be implemented in any platform, from embedded devices, such as cellular phones, to enterprise applications.
- The kind of contextual information might change and this should not cause a significant impact on the system.

Solution

Aspects [13] are used to make the application adaptive in a modularized and not invasive way. They define how the behavior of the core application functionalities should be changed in order to support adaptability. They interact with a module for monitoring the context and interact with another module responsible for obtaining dynamic data specifying how the application should adapt in a specific situation. Auxiliary classes are also used to avoid requiring all developers to know an AOP language and to avoid code tangling mixing different concerns such as application business rules, GUI code, and adaptive functionality implementation.

Structure

The *AdapPE* architectural pattern presents five elements or modules:

- **Core Application**: The core application functionalities, such as business and GUI code, and possibly persistence and distribution code, but no adaptability code.
- **Adaptability Aspects**: The aspects implementing the adaptability concern. They specify how the behavior of the core application functionalities should be changed to adapt to contextual changes. This element delegates several tasks to the auxiliary classes.

- **Auxiliary Classes**: Classes used by the aspects to provide the adaptive behavior. Its isolation from the aspect is intended to improve reuse. They communicate with the adaptation data provider module in order to obtain dynamic data for the adaptation. Besides that, for developing the auxiliary classes, the developers of this module do not necessarily need to know an AOP language. The Adaptability Aspects developer or the system architect may simply specify the interfaces of these classes and what they should do. Then, from these specifications, these classes can be built and have their methods invoked by the aspects.

- **Context Manager**: Module responsible for analyzing context changes and triggering adaptive actions implemented by the aspects. It can also be called by the aspects to obtain information about the context. Its implementation can be based on a variation of the Observer pattern [4], or on its implementation with aspects [6]. In this way, new mechanisms for accessing the context can be easily supported without significant impact on the application (see the Example section).

- **Adaptation Data Provider**: Classes responsible for providing data for dynamic adaptations according to context changes. This means that the same context change can lead to different behaviors in different moments according to the data provided by this module. These classes can be organized as an Adaptive Object-Model (AOM) [14].

These elements and their inter-relation are shown in Figure 1. They are represented there using the UML package notation. Each package represents a logical part of the code, but each of these parts can be implemented using several programming language-specific packages. The arrows represent the dependency between the packages.

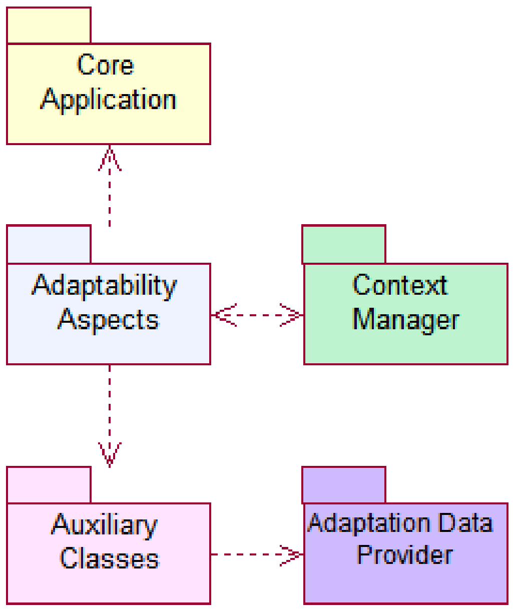

Figure 1: *AdapPE* elements.

Dynamics

The following scenarios depict the dynamic behavior of the *AdapPE* pattern.

Scenario I illustrates the application behavior when the aspects request information about the context at specific points in the execution flow and the application changes its behavior at those points according to the response obtained:

- The application starts executing one of its core functions.
- The adaptability aspects detect points in the execution flow (join points) of the core function where an adaptation might be performed (secure points). This is implemented using AOP constructs such as pointcuts, which are a way of identifying join points. Then, these aspects lead to behavior changes in the application before, after or around those points using advice.
- Those aspects then query the context manager whether any adaptation should be performed, which will depend on the environment state.
- The actions implementing the adaptation are then delegated to the auxiliary classes. These actions are performed before, after, or around the join points.
- In order to access dynamic data specifying how an adaptation should be performed, the auxiliary classes access the objects of the adaptation data provider, which can be based on the Metadata and Active Object-Model pattern language [3], also called Adaptive Object-Model Architecture [14].
- The application is then changed according to this dynamic data.

  The sequence diagram in Figure 2 shows the interaction between the pattern elements. Instead of representing objects, each diagram box represents a collection of objects from the correspondent pattern module.

Scenario II shows how the elements of the *AdapPE* pattern interact when a context change triggers an adaptation on the application behavior.

- The application starts execution.
- The context manager begins to monitor the context continuously.
- When a registered environment change is detected, the adaptability aspects are notified.
- The adaptability aspects detect a point in the execution flow and then use auxiliary classes to perform the changes in the application.
- The auxiliary classes use the adaptation data provider in order to access dynamic information on how the adaptation should be performed.
- The aspects change the application behavior.

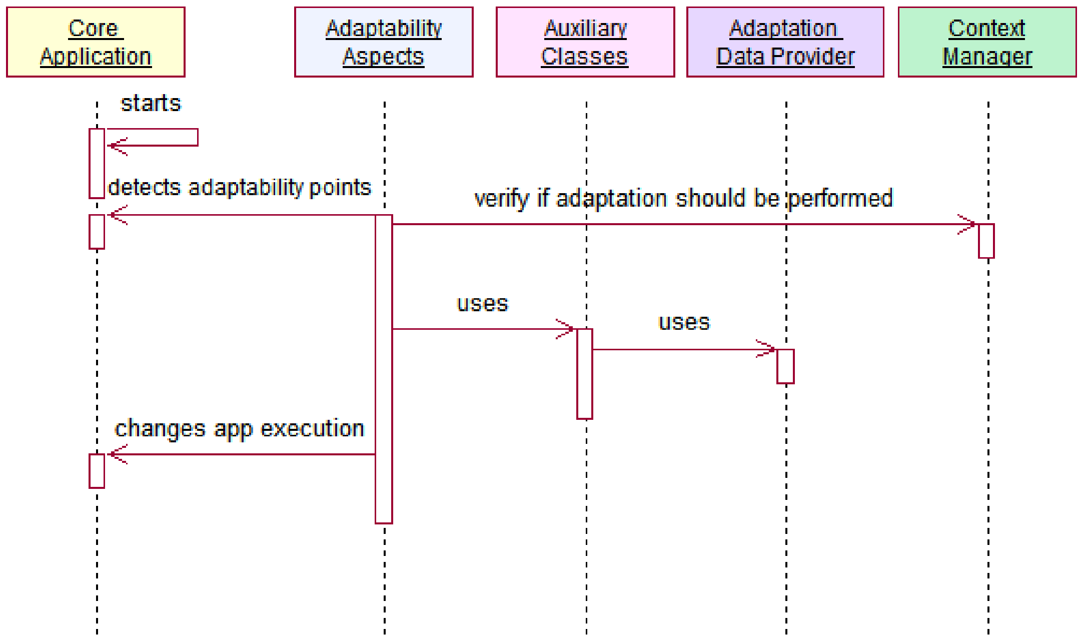

Figure 2: Dynamics of *AdapPE* pattern (scenario I)

When the adaptability aspects are notified about a context change, the action to be performed can execute either immediately, as for example, an invocation to the garbage collector or not, as it is more usual. In the latter case, a field representing the adaptation state can be directly set, but the adaptive behavior is introduced on the core application just when a specific join point is reached.

Consequences

The *AdapPE* pattern provides the following benefits:

- *Modularity.* The core application is isolated from the adaptability aspects and from the classes that actually execute actions at the identified adaptation points.

- *Reuse.* The use of auxiliary classes improves reuse, since these classes can be used by many aspects.

- *Extensibility.* As the aspects code is isolated from the core and auxiliary classes code, it becomes easier to maintain each part of the application and also to extend the system. The context manager should also be internally organized to promote extensibility, making it simpler to achieve and minimizing the impacts caused by the inclusion of a new context element. Nevertheless, the use of AOP in some modules must be done carefully and the use of visual tools is strongly recommended, because AOP languages are considerably powerful, and a change in one part of its code can affect the entire system.

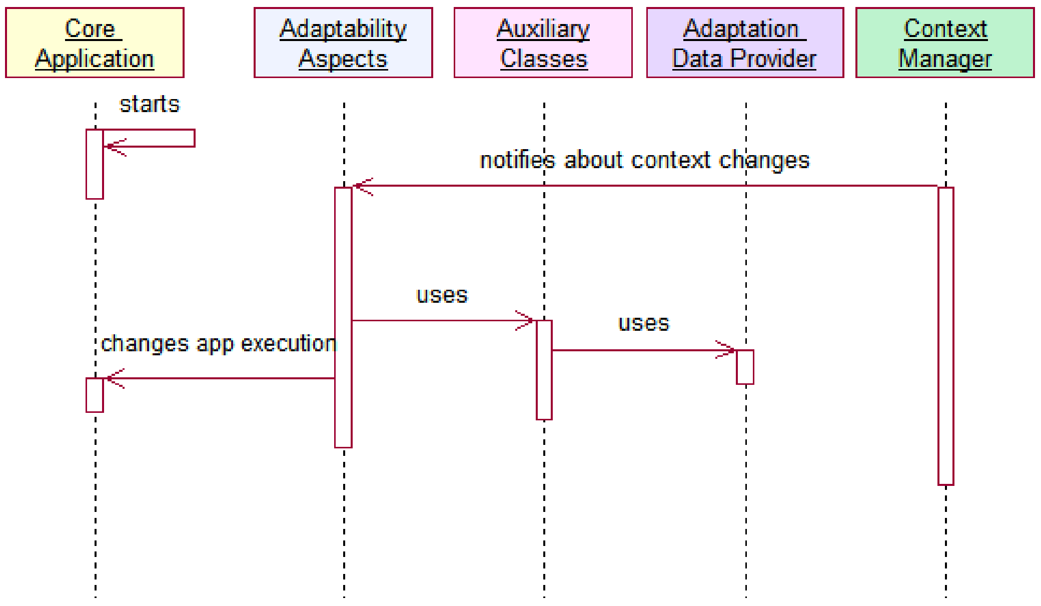

Figure 3: Dynamics of the *AdapPE* pattern (scenario II)

- *Platform Independence.* Its general structure can be applied to a wide range of systems, from embedded systems (such as the example we will present shortly) to enterprise applications. This follows from it not imposing a burden on efficiency and not requiring a significant amount of resources.

- *Dynamic changes.* With the adaptation data provider module, dynamic changes in the application behavior can be performed, and those changes do not need to be expressed by new code; they may be expressed in metadata (XML files, for example).

The *AdapPE* architectural pattern also has some liabilities:

- *Code size.* Implementations of the pattern are bigger than alternative non-modular solutions that tangle adaptation code with core application code. The implementations proposed on the Implementation section for this architectural pattern can also increase the code size if more flexibility, dynamicity, and reuse are required. This liability can be a problem for embedded systems, in which resources are restricted, but for enterprise applications it would not be an issue.

- *Efficiency.* Due to the number of elements required by the architecture, the proposed pattern imposes a burden on efficiency when compared to other non-modularized solutions. Nevertheless, such burden would be a minor problem for an enterprise architecture.

- *Dynamic loading.* An important requirement related to adaptability nowadays is dynamicity (the extent to which the adaptation should be dynamic). Our architectural pattern provides dynamic changes using the adaptation data provider element, which presents new application behaviors in metadata. However, it may not be possible to dynamically load code in order to include new kinds of adaptation mechanisms (ways of triggering new adaptations) without stopping the application, which will depend on the application platform and on the AOP language being used. AspectJ [7], an AOP language in widespread use, does not allow dynamic loading of new aspects in its latest stable version.

- *New programming paradigm.* Although the use of AOP requires learning a new programming paradigm, only one pattern element necessarily uses aspects. The developers of the other elements do not necessarily need to know a specific AOP language.

Implementation

The following guidelines help implementing an adaptive system using the *AdapPE* pattern.

1. *Develop the core application.* The application is developed with its main functionalities, preferably organized in a way to help its evolution. If the application has already been developed, it is occasionally suggested to do some refactoring in order to provide internal modularity to this module.

2. *Develop the context management module.* This module must present ways of detecting which elements from the context should change, which elements should be informed when these changes occur, and which actions should take place at those moments. We propose the implementation of this module using the *Observer* pattern, and specially its implementation using aspects, proposed in another work [6], and adapted to the observation of the context, which increases the flexibility of this module in relation to new environment elements to be observed. This is illustrated by Figure 4. Although this figure illustrates the Context Manager module, the `SpecificAdaptabilityAspect` is part of the Adaptability Aspects part and is shown here just to illustrate the interaction between these pattern elements. The `SpecificAdaptabilityAspect` and `SpecificAdaptationProtocol` aspects and the `SpecificContextElementVerifier` class of this figure are used to represent the elements used to manage a specific context change. For example, if we want a different application behavior, such as the changing of languages used for translation in a dictionary, when the device localization changes, we would replace these classes by the following ones respectively: `DictionaryCustomization`, `LanguageAdaptationProtocol` and `LocalizationVerifier`. Figure 7, in the next section, better illustrates this example and the inter-relationship among the pattern elements in this situation.

3. *Develop the adaptation data provider module.* Dynamic adaptation is generally desirable. It is not good practice to fix the adaptations in code. One option is to have these changes in metadata as XML files which can be obtained locally or remotely (the Bridge [4] pattern is indicated for providing this flexibility). Such an idea is adopted by the Adaptive Object-Model architecture [14]. The use of AOM is not mandatory for the development of this module, but presents the advantage that, once implemented, one of its parts can be reused by many systems; only the interpreter of this AOM remains to be implemented according to the wanted changes,

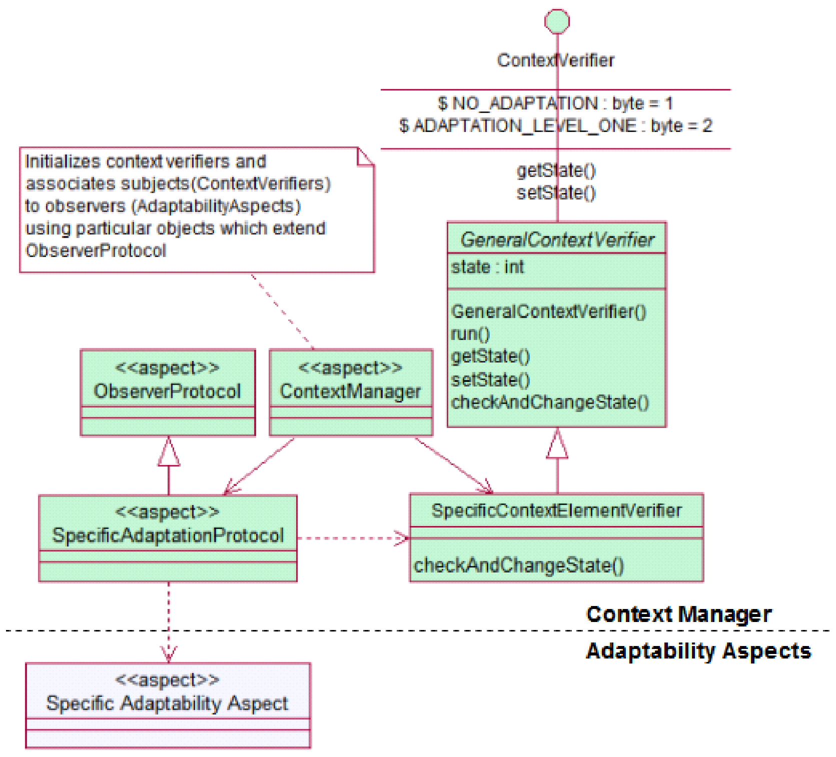
ContextVerifier
$ NO_ADAPTATION : byte = 1
$ ADAPTATION_LEVEL_ONE : byte = 2
getState()
setState()
Initializes context verifiers and associates subjects(ContextVerifiers) to observers (AdaptabilityAspects) using particular objects which extend ObserverProtocol
GeneralContextVerifier
state : int
GeneralContextVerifier()
run()
getState()
setState()
checkAndChangeState()
<<aspect>> ObserverProtocol
<<aspect>> ContextManager
<<aspect>> SpecificAdaptationProtocol
SpecificContextElementVerifier
checkAndChangeState()
Context Manager
Adaptability Aspects
<<aspect>> Specific Adaptability Aspect

Figure 4: Context Manager Possible Implementation

which is a task that can be performed by auxiliary classes. An AOM partial implementation that can be reused by many kinds of adaptive applications is shown in Figure 5. The application elements can be represented as `EntityType` elements, which present `Entities` with their `Properties`. The interpreter of the AOM will translate these kinds of elements into application properties or algorithms. The auxiliary classes used by the adaptability aspects will have access to the adaptation data provider module by the `AppAOMManager` class, using its `getEntityType` method.

4. *Identify which kind of adaptations will be included.* It is necessary to analyze what will really constitute an adaptation (an application change caused by the environment) and which context changes would lead to those adaptations. Example adaptations are changes in the application language due to the detection of a device location modification, memory management, changes and inclusion of screens in the application due to user inputs or server responses, for example.

5. *Transform each kind of adaptation into an aspect.* Use AOP constructions such as pointcuts and advice to access the application components that can change and the execution points (join points) where the adaptations should be performed. These aspects should contain the least amount of non-aspect code as possible, delegating necessary actions to auxiliary classes that will be developed in a later step. Besides that, these classes should be able to access the context manager module in order to verify whether an adaptation should be done. Each aspect can also be a

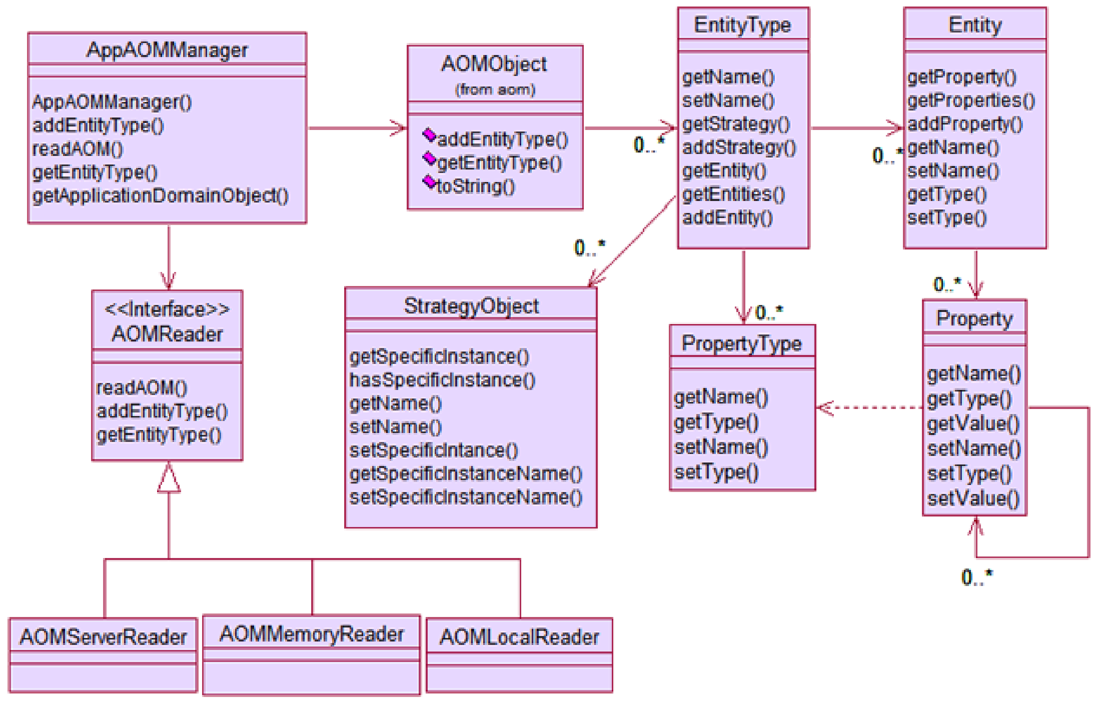

Figure 5: **Adaptation Data Provider** using AOM

`SpecificAdaptabilityAspect` in the situation illustrated by Figure 4.

6. *Develop the auxiliary classes.* These classes will actually present the code to be executed at the points selected in the aspects module. Their design should promote reuse by many aspects. In order to perform dynamic adaptations, this module should communicate with the adaptation data provider module. One suggestion is to include in these classes interpreters of AOM for the application being developed.

Example

To exemplify the *AdapPE* pattern, and specially its second scenario, we will now consider a dictionary application developed in Java [5], for the J2ME (Java 2 Micro Edition) platform. This platform accommodates consumer electronics and embedded devices [11]. The AOP language used is AspectJ [7], a general purpose aspect-oriented extension to Java in widespread use.

Our **Core Application** will be composed by this dictionary, which is a simple MIDP (Mobile Information Device Profile)-based application (also known as MIDlet [11]), without any adaptations. Its basic functionality is to translate a given word from English to Portuguese. It follows the MVC [1] architectural pattern and its main classes are illustrated by Figure 6: the controller part is being represented by the `DictionaryMIDlet` and the `DictionaryController`; the view part is represented by the `DictionaryScreen`; the model is represented by the `InputSearchData` class, which stores some information about the dictionary, such as the word being translated and the source and destination languages of the translation.

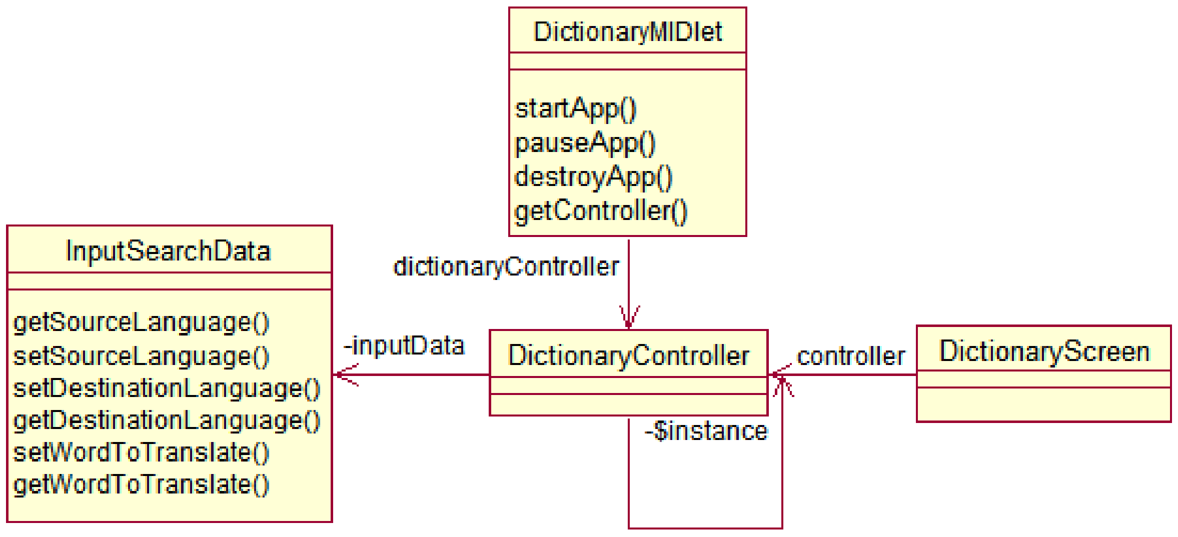

Figure 6: A **Core Application** example

In order to make the example easier to understand, the adaptation we will include will be the change of the application translation language when the device location changes. The main classes and aspects used for this adaptation are illustrated by Figure 7, which is a UML diagram that uses the << `aspect` >> stereotype to identify aspects. Other stereotypes are also used in some dependency relationships to represent classes whose behavior is monitored or changed by an aspect (<< `affects` >> stereotype), or those which are just used as auxiliary classes (<< `uses` >> stereotype) by an aspect.

The aspects which will compose the **Adaptability Aspects** element of the pattern are organized in a small framework, composed of two aspects: the `Customization` aspect (which can be reused by any J2ME application) and the `DictionaryCustomization` aspect (which is application specific and extends the former). They are responsible for changing application properties, using a captured `MIDlet` instance at the moment the `startApp` method is executed. The following code from the `Customization` aspect illustrates that.

```
pointcut MIDletStart(MIDlet midlet):
  execution(void startApp()) && target(midlet);
before(MIDlet midlet): MIDletStart(midlet) {
  adaptBefore(midlet);
}
```

The `Customization` aspect is general because any J2ME application should present a class implementing `MIDlet` with a `startApp` method. Its `before` advice invokes the `adaptBefore` abstract method, which is defined in the `DictionaryCustomization` aspect. At its invocation, this aspect stores the `MIDlet` instance, which makes it able of changing application properties in its methods.

The **Context Manager** module will follow the structure shown in Figure 4. The code

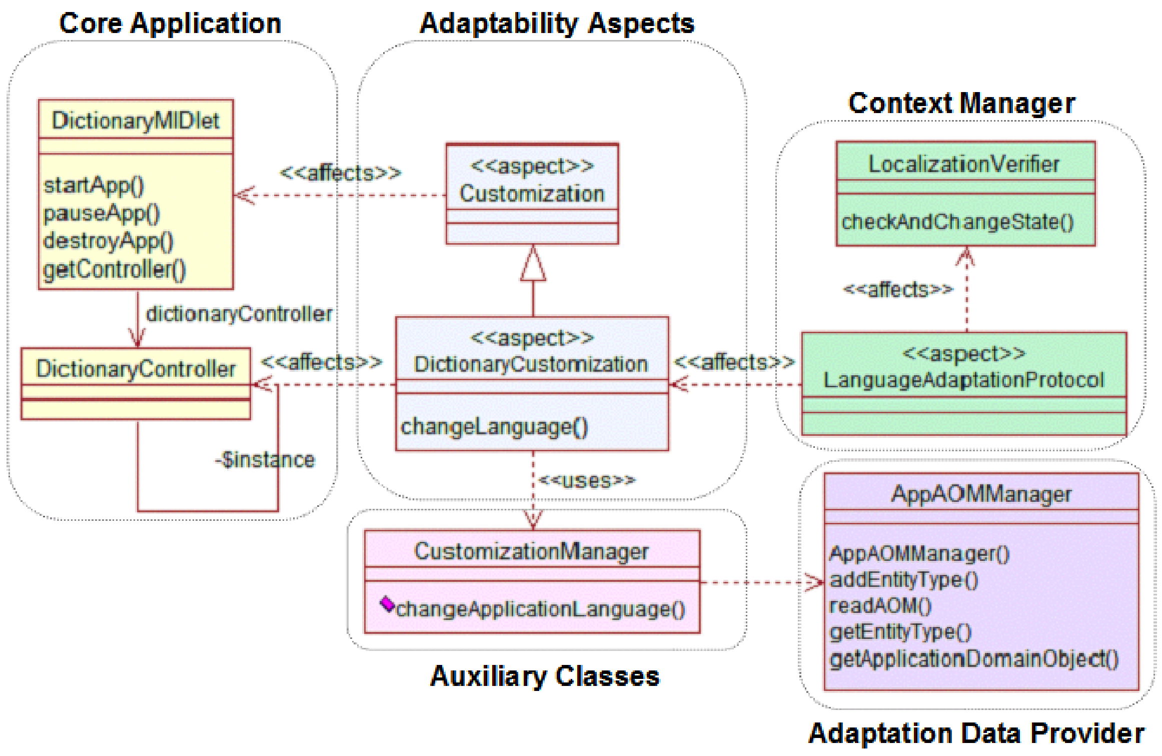

Figure 7: An **AdapPE** pattern example

for the `ObserverProtocol` is shown elsewhere [6]. Our `SpecificAdaptationProtocol` aspect will be the `LanguageAdaptationProtocol`, which is shown below:

```
public aspect LanguageAdaptationProtocol extends ObserverProtocol {
  declare parents: LocalizationVerifier implements Subject;
  declare parents: DictionaryCustomization implements
    Observer;
  protected pointcut subjectChange(Subject s):
  call(void LocalizationVerifier.setState(int))
    && target(s);
  protected void updateObserver(Subject s, Observer o) {
    LocalizationVerifier lv = (LocalizationVerifier) s;
    if (lv.getState()!=ContextVerifier.NO_ADAPTATION) {
      ((DictionaryCustomization)
        o).changeLanguage();
      ((LocalizationVerifier)
        s).setState(ContextVerifier.NO_ADAPTATION);
    }
  }
}
```

In this aspect we identify the `Subject` (`LocalizationVerifier`) and the `Observer`

(DictionaryCustomization aspect) of the pattern using AspectJ introduction (lines 2 and 3). Then, we define the pointcut subjectChange, inherited from ObserverProtocol, identifying the execution points that characterize the Subject change (lines 5, 6, and 7), which is the call of the setState method on a LocalizationVerifier object in this example. Finally, we define the also inherited method, updateObserver, indicating what should happen to the Subject and the Observer when there is a change (lines 8-16). In this case, a method from the DictionaryCustomization aspect is invoked to change the application language.

The GeneralContexVerifier from Figure 4 extends the Thread class and invokes an abstract method called checkAndChangeState at regular intervals. This class can be reused by other applications. The LocalizationContextVerifier extends the GeneralContextVerifier and should only specify its abstract method, verifying the device localization as the following code illustrates.

```
  private LocalizationObject lo = new LocalizationObject();
  public void checkAndChangeState() {
    if (lo.hasChanged())
      this.setState(ContextVerifier.ADAPTATION_LEVEL_ONE);
  }
```

We use here an abstraction of a LocalizationObject, which is responsible for verifying if the device localization has changed. If this happens, the setState method is called, the LanguageAdaptationProtocol aspect identifies that the Observer should be notified and executes the updateObserver method, which will call the changeApplicationLanguage method from the DictionaryCustomization aspect.

As Figure 4 illustrates, the ContextManager initializes the context verifiers and associates subjects to observers. The following code extracted from it shows how it works for this example.

```
  public aspect ContextManager {
    pointcut MIDletStart(MIDlet midlet): execution(void startApp()) &&
      target(midlet);
    after(MIDlet midlet): MIDletStart(midlet){
      LocalizationVerifier lv = new LocalizationVerifier();
      lv.start();
      LanguageAdaptationProtocol.aspectOf().addObserver(lv,
        DictionaryCustomization.aspectOf());
...
    }
  }
```

In order to avoid non-aspect code in aspects, the changeLanguage method from the DictionaryCustomization aspect delegates the application language change to an auxiliary class, the CustomizationManager, which will be part of the **Auxiliary Classes** element of the *AdapPE* pattern. Nevertheless, this element is not mandatory and another possible technique would be to use non-aspect code in methods of the aspect and not in the advice body when extensive code is necessary to provide the change. However, to illustrate the **Auxiliary Classes** module use and its interaction with **Adaptation Data Provider** module (see Figure 5), we will show one of the `CustomizationManager` methods implementation, the `changeApplicationLanguage`:

```
public void changeApplicationLanguage(){
  EntityType et = aomManager.getEntityType("DictionaryProperties");
  Entity e = et.getEntity("InputSearchData");
  String sL = e.getProperty("sourceLanguage").getValue().toString();
  String dL = e.getProperty("destinationLanguage").getValue().toString();
  InputSearchData isd = this.midlet.getController().getInputData();
  isd.setDestinationLanguage(dL);
  isd.setSourceLanguage(sL);
}
```

In this code we can notice the relation of the **Auxiliary Classes** component with the **Adaptation Data Provider** module. Here, this module uses the *Adaptive Object-Model* pattern language. As we can see, the application properties and behavior, described in metadata (in our case an XML file), are translated into the object structure illustrated by Figure 5. The `changeApplicationLanguage` method then interprets this object model and changes the application using the `MIDlet` instance.

Known Uses

The *AdapPE* architectural pattern has been used in two experiments we have developed with J2ME: a Dictionary application for a cellular phone and an album for pictures, also for small devices. Besides that, some parts of the patterns have already been referred to. An example of this is a work [9] presenting some experiments using aspects. One of the conclusions is that it is an interesting architecture to separate an AOP system in three parts: the base code, the aspects and the auxiliary code. In this structure, the aspects part functions as glue between the other two. Another example is the implementation of the Observer aspect using aspects proposed by Hanneman [6] which we modify here in the **Context Manager** module.

See Also

- The *Reflection* architectural pattern [1], which provides a mechanism for changing structure and behavior of software systems dynamically, is related to our pattern. It is intended to make applications adaptable, that means, able to easily evolve due to requirement changes, which is also one of our requirements. Although it provides a lot of flexibility, it seems to increase the complexity of the system more than our solution and even to be less efficient.

- *Adaptive Object-Model* systems represent its attributes, classes, and relationships as metadata [14]. Although AOMs can be used to represent all the system structure, they were used in our pattern to represent just its part that should dynamically change. Using just AOM, the adaptation code is spread through out all the application, which does not happen in our solution; it is also harder to turn off the adaptive behavior feature and to maintain the system.

- The *PADA(Pattern for Distribution Aspects)* [12], even dealing with Distribution, is also related to this work as it provides a solution to the lack of modularity and maintainability using AOP.
- The *Observer* pattern [4] is very useful in the implementation of the **Context Manager** component (see Figure 4), which actually characterizes the adaptive behavior. Its implementation with aspects [6] is interesting as a dynamic way (using pointcuts) of identifying subject changes.
- Another work [2] describes some practices incorporated to this pattern and presents the implementation of some adaptive concerns into J2ME applications using AspectJ. The implementation shown there does not follow this pattern and does not use Adaptive Object-Models yet.

Acknowledgments

We would like to give special thanks to Paulo Masiero, our shepherd, for his important comments and suggestions, helping us to improve our pattern. Thanks to Rossana Andrade for her work as mediator and her attention, and also to Vander, for his valuable suggestions.

References

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.

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

[3] Brian Foote and Joseph Yoder. 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, September 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, second edition, 2000.

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

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[8] Kalle Lyytinen and Youngjin Yoo. Introduction. *Communications of the ACM*, 45(12):62–65, 2002.

[9] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten Kersten. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, 2001.

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

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

[12] Sérgio Soares and Paulo Borba. Pada: A pattern for distribution aspects. In *Second Latin American Conference on Pattern Languages of Programming — SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, August 2002.

[13] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st international conference on Software engineering*, pages 120–130. IEEE Computer Society Press, 1999.

[14] Joseph W. Yoder, Federico Balaguer, and Ralph Johnson. Architecture and design of adaptive object-models. *ACM SIGPLAN Notices*, 36(12):50–60, 2001.