Programming for Context-based Adaptability
Lessons learned about OOP, SOA, and AOP

Holger Mügge[1], Tobias Rho[1], Daniel Speicher[1], Pascal Bihler[1], and Armin B. Cremers[1]

Institute of Computer Science III, University of Bonn
Römerstr. 164, 53117 Bonn, Germany
{muegge|rho|dsp|bihler|abc}@iai.uni-bonn.de

Abstract. Context-sensitive applications are a key issue to exploit the benefits of mobile devices. For many applications planned adaptation starts to become status quo. However, the huge potential of adapting to unanticipated situations still remains a research issue. The general question "how to program for adaptivity" in a reasonable way is not answered yet. Current methods relying on statically typed languages either introduce a high level of complexity, or rely on detailed anticipation, or both. First, we describe how a statically typed Object-Oriented language can with the help of some design patterns to cope with basic requirements of adaptivity. Then, we show how adopting a Service-Oriented Architecture (SOA) can leverage coding. Finally, we present the Aspect-Oriented (AOP) language CSLogicAJ (compatible to AspectJ) which includes a logic notation and direct access to context data. We discuss how AOP on top of SOA can help to reduce the level of anticipation of runtime adaptation. Finally, we discuss which of the main obstacles can be overcome with our approach and what remains to be tackled in the future.

1 Introduction

Mobile applications running on a portable devices are the natural setting for context sensitive applications. Loosely attached to the user's clothing the device, together with its owner, is exploring the world and gets confronted with a lot of different context situations. Adapting mobile applications to predefined context changes in order to support the nomadic user in his everyday actions is good practice by today. For example mobile phones can change their signaling method based on the currently selected workspace profile.

But with adapting to different predefined context situations just half the battle is won. Often, the user and his device are confronted with new and unforeseen situations. Adapting to such settings where just a low degree of anticipation is given defines the challenge for context sensitive intelligence research.

As example of use, we selected a personal information management application (*PimPro*) on a mobile device. The *PimPro* application gives its users prompt access to exactly that part of their personal data which is relevant for the

current context. To determine currently relevant documents it applies context-dependent filtering and sorting strategies on typical business data like e-mail, links to websites, meeting minutes, or financial data.

The remainder of this paper is structured as follows: In section 2, we outline the basic requirements for our adaptation scenario and deduce general conditions from this special case. In section 3, we discuss a purely object-orientated approach to meet these requirements, using different design patterns. In section 4 we show how a service-oriented architecture can be applied to improve these solutions and discuss the remaining limitations. To overcome these, we introduce the benefits of aspect-orientated development in section 5 by discussing two concrete aspect-oriented solutions for the adaptation scenario. Finally, section 6 gives a short summary and explains briefly our future plans.

2 Requirements

A Scenario for Context-Sensitive Adaptation In our scenario a business user visits a trade fair. He keeps a lot of documents on his mobile device and manages them by different applications, as illustrated in figure 1. On the fair he will be engaged in meetings and presentations and thus needs an efficient way to access the relevant documents at each particular event.

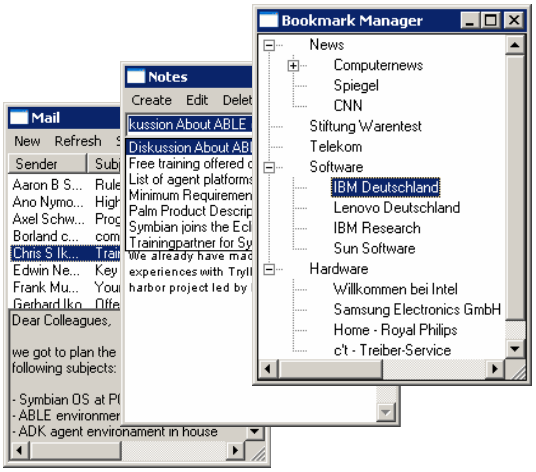

Fig. 1. Simple document management tools for mail, minutes and bookmarks.

When he enters the fair and is about to be engaged in meetings and negotiations, his device recognizes the new situation automatically and scans for adequate adaptations. The fair organization offers some special services for document management support: a *document indexing service*, which creates an index

for searching and classifying documents; a *vicinity explorer* calculating the fair stands closest to your current location; an *index matcher*, which determines how similar two index lists are; a *sorter*, a *filter*, and a *tree flattener* for list data manipulation. These services can be combined to offer a real benefit for the user while he is on the fair, as shown in figure 2.

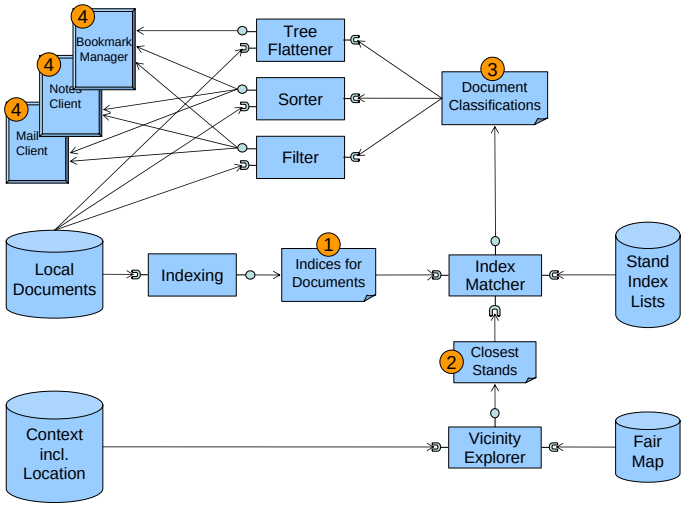

Fig. 2. Combining services leads to a beneficial adaptation

These adaptations work together in the following way: First, the indexing service creates for each locally stored user document a (potentially weighted) index characterizing the content of the document. Second, the vicinity explorer calculates the distances between the user and all stands on the fair (given by the Fair Map) and determines which stands are close to the user's current position. In the third step for each of the stands in the user's vicinity the document set gets ordered with respect to their relevance. Therefore, each exhibitor provides an index list describing his company (Stand Descriptions). The index matching service estimates the relevance of each document to a stand by comparing both index lists and though produces a document classification. Finally in step four the relevance classification for the documents are used to display them in a more convenient way. Therefore three services can be used for sorting, filtering and flattening document entries in list- and tree-like structures. Figure 3 illustrates the adapted client tools, providing prompt access to those documents relevant for the closest stands.

Requirements Elicitation What requirements can be elicitated from the given scenario? First, the adaptation should be done at runtime, since the user will probably have his tools already started before he enters the fair. Second, the

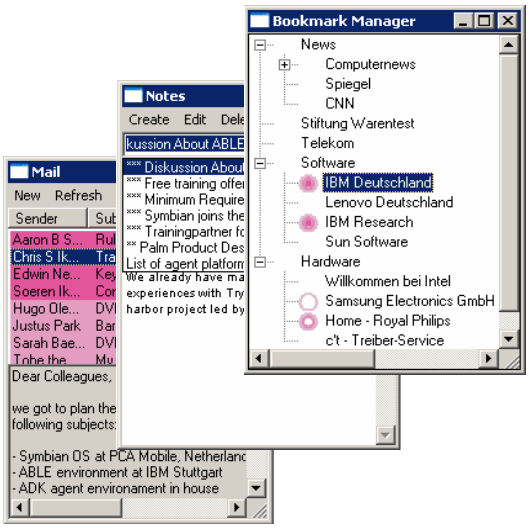

Fig. 3. Adapted tools provide prompt access to currently relevant documents.

situation should be recognized automatically, since the user will not manually specify each situation change without knowledge about a possible benefit through adaptation. Third, the services popping up at the fair, should be detected automatically, since a busy user won't be able to scan manually for new services every now and then. Fourth, we assume that there will be a large cloud of services offered. The user needs support for selecting appropriate services. This holds for services that are separately useful (e.g. the vicinity explorer service might adapt a map application to show stands), in cases where it is tedious to find the service in a long list. But it becomes definitely necessary when the adaptations benefit is provided by combinations of services as shown in our scenario. Fifth, the adoption of new services must be tightly integrated into the functionality of the user's applications. In our example service adaptations are spread across three different client applications, and more complex scenarios are easy to think of, i.e. adaptation should allow for cross-cutting changes.

General Requirements for Context-Sensitive Adaptivity Most of the requirements deduced from the scenario in the last sections, can be abstracted to a general form and occur frequently in context-sensitive settings. Additionally we found some basic technical requirements regarding the tight integration of the adaptation into the existing applications. I.e. we need to care about replacing existing functionality with more appropriate alternatives, we must allow for extending the set of given functionality by new elements, and finally we also need to be able to remove functionality, which has been added before. The following list summarizes the requirements and gives a first short comment about how we are going to accomplish each of them. Thus, we tackle

- *replace existing functionality* by enhancing the strategy pattern
- *enhance given functionality* by enhancing the decorator pattern
- *add new functionality* by enhancing the visitor pattern (not in this paper)
- *adaptation at runtime* by enhancing design patterns, using SOA, and applying runtime aspect weaving
- *automatic service detection* using a service-oriented architecture
- *cross-cutting adaptations* by aspect quantification
- *minimizing anticipation* by applying aspects and thus introducing details about the variants at runtime
- *automatic detection* by a context management system based on logic descriptions and reasoning (not in this paper)
- *support for service retrieval* by applying IR techniques combined with ontological descriptions of services and the context (not in this paper)

3 Pure Object-Orientation — Patterns for Adaptivity

On our way towards development tools for adaptive software, we start with pure object-oriented methods. Design patterns are a well-known means for introducing flexibility to software (c.f. [1]). For the field of product line engineering Svahnberg discusses in [2] several patterns for introducing systematic variability. Hence we investigated what could be achieved applying appropriate patterns.

Design patterns mostly address statical flexibility, i.e. adaptivity during the software evolution process. Although this is fundamentally different to our setting, where adaptations occur at runtime, some selected patterns seem to be a good starting point. In particular we applied the strategy pattern to exchange functionality and the decorator pattern to enhance functionality at runtime.

The strategy pattern provides an infrastructure for dynamically exchanging a certain functionality at runtime. While exchanging strategies (and thus functionality) at runtime is supported by the pattern, two problems remain: first, we need to be able to detect strategies and load them at runtime. The latter is relatively easy to achieve by extending the strategy pattern with a dynamic strategy repository that is able to load new strategies at runtime and offer them to the business logic part of the application. Second, detecting available strategies and in particular discovering appropriate adaptations remains a complex problem in its own right, calling for abstractions on the architecture level of the software.

Enhancing functionality by decorators is even more complicated. One scenario is that a decorator wraps a given object so that calls to its methods will first be executed by the decorator, potentially specifying additional behavior before calling the original object. The decorator could also specify to enhance the behavior after the original object's functionality has been executed or even replace it completely with new functionality.

At least when multiple decorators come into play, the client who configures the decoration of objects, needs detailed knowledge about possible or reasonable

combinations. This is in a static setting feasible since flexibility then means to have the option of convenient re-specification of decorator combinations at development-time. In our setting, we need a dynamic decorator in the sense that we can robustly add or remove decorators to objects at runtime without explicitly taking care of permitted combinations.

We achieved this by extending the decorator pattern with a configuration logic that automatically cares for reconfiguration of the object's decorations when it is changed. This basically means that combination of decorators are self-managed by an "intelligent"decorator manager.

Further patterns can be applied enhancing adaptivity: we used a variant of the adapter pattern to allow for flexible connections between not quite fitting interfaces and the observer pattern for dynamically recombining functional units. The visitor pattern could be used to allow adding new functionality at runtime without changing class code or compilation.

While design patterns can provide for basic adaptivity, relying on them as the only means can not reasonably cope with the requirements of context-sensitive adaptivity. For example, cross-cutting concerns will lead to a proliferation of similar structures within the whole software and introduce a high level of maintenance complexity. Detection of available and discovery of appropriate adaptations will lead to very specific implementations with a high level of complexity. Hence, using separate abstractions for coping with these issues seems appropriate, as we describe when using services in section 4. The general problem of aiming for least anticipation can also not be reached solely by applying patterns. According to this, we use Aspect-Orientation as discussed in section 5.

4 SOA and Object-Orientation - Patterns for Adaptivity

Service-oriented architecture simplifies dynamic adaptation because components are low coupled. The use of a component, like viewer categorization or tree decoration, must be completely anticipated in a pure OO solution. A configuration class is necessary to organize the dynamic change of decorations and categorizations. In a service-oriented approach a *service registry* is responsible for organizing the available services. *Service Providers* register services and *Service Consumers* request services. Figure 4 depicts how this architecture can be used to implement a Strategy Pattern. Consider the components *Context, Strategy I* and *Strategy II*. The *Context* component requests a service *IStrategy* which is implemented by services registered by the component *Strategy I* and *Strategy II*. We call code that responsible to select one or several strategies based on the availability and properties *tracking code*.

The adaptations discussed in the reminder of this paper are based on the service-oriented platform OSGi [3]. OSGi (open services gateway initiative) is a Java based component platform. It's purpose is mainly the management (install, start, stop, update, uninstall) of the components running on this platform. These standardized components are called *bundles*, the platform is named *framework*. Bundles can be installed and uninstalled at run time. They are capable to

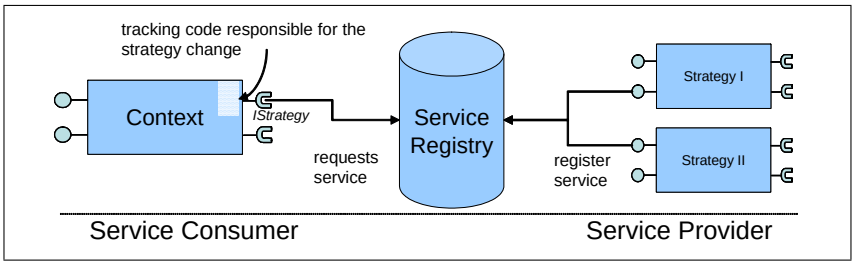

Fig. 4. Strategy Pattern in a service-oriented architecture

dynamically provide services and use services of other bundles, and statically import other bundles and export own interfaces. Dependencies are dissolved upon installation by the framework.

Service-oriented version of the strategy pattern We used the strategy pattern in the *PimPro* application to adapt the document categorization of all three applications: Notes-, Mail- and Bookmarks-Tool (c.f. figures 1 and 3). All three applications play the role of the context in the strategy pattern as depicted in figure 4 and are implemented as bundles requesting a DocumentCategorization service. We implemented three concrete strategies: a *vicinity categorizer* indicating that a document is relevant for the closest stand, a *gradual categorizer*, reflecting the distance of the next stand for which the document is relevant , and a *constant categorizer* yielding the same category for all documents.

To be able to react to changes concerning the availability of services, service tracking code was introduced to all bundles. The user is responsible for activating and deactivating bundles containing categorization services. Only if no service is available the default service is activated, namely the constant categorizer.

Service-oriented version of the decorator pattern The dynamic tree decoration also takes advantage of SOA. Here the configuration of the bookmark tree decorator is realized via services. All available tree decorators are tracked and added to the dynamic decorator implementation in the *Bookmarks View* component. The general concept of a decorator pattern as described in section 3 is preserved.

Limitations Several downsides remain. Concerning the strategy solution the necessity of tracking code itself is problematic. The selection criteria of the concrete service still is anticipated in the tracking implementation. For a context dependent selection the bundles must be context-aware. This is a strong restriction since a bundle should be usable in arbitrary settings.

The main limitation of the decorator solution is the anticipation in the *Bookmarks View* component. Every service request that should be decorated must be extended with the relatively complex dynamic decorator pattern structure.

For both patterns an inherent OSGi problem, the uncertainty about the service reference, must be taken into account in the concrete implementation.

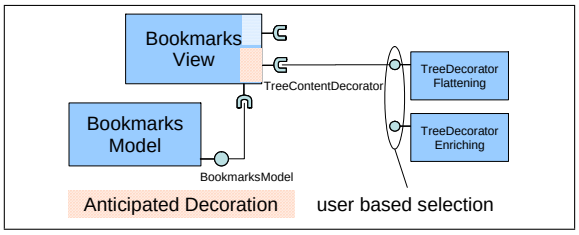

Fig. 5. The anticipated decorator pattern for the bookmark tree decoration.

While using the reference and passing it to the internals of the component, the service may become unavailable. This problem must be anticipated by the programmer. There is no means in OSGi to replace a service reference on the platform level.

5 SOA and Aspect-Orientation - Patterns for Adaptivity

Generic aspect orientation (c.f. [4,5]) has shown to simplify and improve the implementation of most of the GoF design patterns [1]. Here we show how aspect-oriented techniques combined with a context-aware service-oriented architecture can even go further. With the help of *service aspects* [6] we are able to remove most of the tracking code - and thereby the anticipation of possible contexts - from the components.

Service aspects enable replacement or decoration of services and can refer to arbitrary context information. By this service tracking can be moved to the architecture level making dynamic changes of services transparent for the application. The same is true for the dynamic decorator preparation described in section 3. Bundles are only responsible to specify their service dependencies via service requests. The service aspects organize the reconfiguration of the composition.

The following section gives a short introduction to the aspect language CSLogicAJ [6], section 5 and 5 illustrate how the language enables dynamic strategy change and decoration.

CSLogicAJ (Context-aware Service-oriented LogicAJ) is an aspect language for the SOA framework Ditrios [6,7] which is based on OSGi [8]. This section only sketches the main concepts of the language. A thorough introduction can be found in [9]. An advice is composed of two parts: a selection part, selecting points in program execution, and a code body being executed at the selected points. Dynamic aspects enable unanticipated adaptation cross-cutting a series of program parts. Our approach comes with a highly flexible and expressive pointcut language which incorporates external context information in the adaptation steps. The context-awareness is built on a context management system which is part of Ditrios. Context information like hardware sensors, profile information and service tracking information is attachable via *context provider*

services and is queryable in CSLogicAJ's pointcut language. Context depended adaptation - in our terms - means adapting services by calling weaving processes triggered through the change of context information like e.g. heart rate or GSM position.

The CSLogicAJ language differentiates between two advice concepts:

Synchronous advice is applicable on the service message level. Such advice intercept calls to services taking the program flow and available context information into account. The commonly known *before*, *after* or *around* advice can be used to execute additional code. Furthermore, transparent (re)binding and (re)composing of services is possible due to the proxy architecture. Synchronous advice corresponds to the common dynamic weaving.

Asynchronous advice react exclusively on context changes indicated through *context providers*. Such advice is comparable to event-condition-action (ECA) rules known from relational databases. The *event* is the change of the context, the *condition* is represented by a pointcut and the *action* by the advice code. Asynchronous advice does not need program-execution pointcuts. The pointcut expression is reevaluated every time the context information changes. These advice constructs are marked with the keyword *onchange*.

Dynamic Strategy Change This example realizes the dynamic change of a categorization service based on user preferences. Lets assume that a context provider is available which represents the user preferences on a certain device. Since the context management is not in the focus of this paper we skip the definition and registration of the provider.

Once the context provider is registered a pointcut *userPreference(Key, Value)* is available via the context management system. Based on this pointcut and two more predefined pointcuts it is now possible to define a dynamic strategy change via CSLogicAJ aspects. Figure 6 depicts how the *CategorizationAspect* selects one of the available categorization types based on the user preferences. If the preferences or the set of available services change the aspect selects a different categorization service. If no service is available the default service is selected.

The onchange advice in Figure 7 facilitates two predefined pointcuts to query currently available services and requests for services. The pointcut *serviceAttr(Service, Key,Value,Type)* provides access to the attributes of all available services. The pointcut *serviceRequest(Request, Interface, Attributes)* provides access to all service requests. The advice strategyChange is re-executed when the aspect is activated and every time at least one of the advice parameters (`?Service` or `?Category`) changes. The disjunction in the pointcut description ensures that if the user preferences can not be fulfilled the default service is selected. The advice body accesses the Ditrios facade via the built-in field `ditrios`.

The setActiveService method performs the actual strategy change. All requests for the services with interface IDocumentCategorization that were bound to `?Req` are set to the selected service `?Service`. Here the crosscutting application of the aspect on all bundles becomes evident. This change is transparent for the using bundles. They keep their references to requested services.

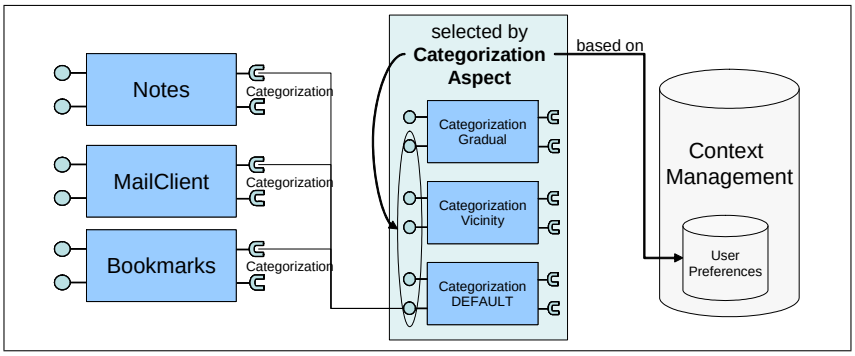

Fig. 6. Aspect-oriented Strategy Pattern based on user preferences.

Dynamic Service Decoration is here illustrated with the TreeFlatteningAspect. In the PimPro example this aspect is responsible for flattening the model of the bookmarks model. This is realized by around advices on all TreeContent-Provider services like the model of the Bookmarks applications, see Figure 8. Every method call to the Bookmarks model is intercepted and modified to flatten the tree structure.

The aspect implementation is shown in Figure 9. We concentrated on the most relevant advice that intercepts the getElements(Object) method. The tree content provider object and its arguments are bound in the pointcut expression. The helper method *getAllElementsRecursively* retrieves all elements of the provider and returns a list. The list is converted to an array and returned as the flattened element list. The proceed call represents the delegation to the original service method or the next advice that was woven, e.g. by another aspect that decorates the tree.

If more than one decorator aspect is woven the order of weaving is important. E.g. an aspect that adds new bookmarks to the tree must be executed before the flattening takes place. Otherwise the result is a mixture of flattened original bookmarks and a new tree of added bookmarks. A first solution to this problem is explicitly annotating the dependencies between the aspects.

Limitations The implementation of bundles becomes much easier with AOP but in concrete adaptation scenarios anticipation is still needed in the bundles. The strategy change in Section 5 is only possible if the service change is possible at any time. We proposed a lightweight transaction concept in [6] to overcome this problem, which results in the necessity to define transaction start and end points in the bundle.

The ordering problem makes parallel use of independently developed aspects error prone. We are currently exploring a more sophisticated solution than explicit ordering using semantic annotations and dependency inference.

Still, the aspect-oriented solution is more flexible and extensible and does not require change the application bundles in many cases.

```
aspect CategorizationAspect {
  onchange strategyChange(?Service, ?Category) :
   userPreference("CategorizationKind",?Category) &&
   (
     serviceAttr(?Service,"CategorizationKind",?SelCategory,?Type) &&
     equals(?Category, ?SelCategory)
   ) || (
     !serviceAttr(?Service,"CategorizationKind",?SelCategory,?Type))&&
     equals(?Category,"DEFAULT")
   ) &&
    serviceRequest(?Req,"org.cs3.csi.IDocumentCategorization", ?_)
  {
     try {
        ditrios.setActiveService(?Req, ?Service);
     } catch(Exception ex) {
         throw new RuntimeException(ex);
     }
  }
}
```

Fig. 7. Dynamic strategy change

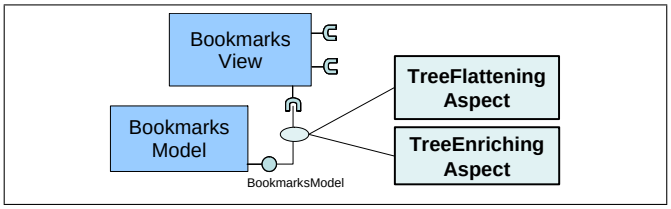

Fig. 8. Aspect-oriented decorator pattern for the bookmark tree decoration

6 Summary and Outlook

This papers addresses the question how adaptive software for context-sensitive scenarios could be developed based on a statically typed language like Java or C++. We first outlined a scenario and deduced the most crucial requirements this kind of software faces. Solely relying on Object-Oriented abstractions can not reasonably cope with the problems of not fully anticipated runtime adaptation, detecting, discovering and integrating adaptations. These requirements call for more sophisticated abstractions, as provided by a Service-Oriented Architecture. As we illustrated rich adaptation frequently will make cross-cutting changes necessary. Aspects can cover this issue to a large extend. They address also the general issue of reducing the level of needed anticipation.

We are currently porting a more complex scenario to the described technique. In this case further reduction of anticipation is aimed at. We therefore plan to integrate an ontological description of services and context for systematic retrieval of appropriate services.

```
aspect TreeFlatteningAspect {
  Object[] around flattenElementTree
        (ITreeContentProvider provider, Object inputElement) :
      execution(* ITreeContentProvider.getElements(Object)) &&
      target(provider) &&
      args(inputElement)
  {
       List result = getAllElementsRecursively(provider,
                          proceed(provider,inputElement));
       return result.toArray();
  }
   // ... around advices that intercept the other
   //     methods of ITreeContentProvider and return
   //     null or false for the other methods
}
```

Fig. 9. Bookmarks Flattening Aspect

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)
2. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques: Research articles. Softw. Pract. Exper. **35** (2005) 705–754
3. OSGi Alliance: OSGi Service Platform Service Compendium - Release 4. (2005)
4. Hannemann, J., Kiczales, G.: Design pattern implementation in java and aspectj. In: OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2002) 161–173
5. Rho, T., Kniesel, G.: Independent evolution of design patterns and application logic with generic aspects - a case study. Technical Report IAI-TR-2006-4, Computer Science Department III, University of Bonn (2006)
6. Rho, T., Schmatz, M., Cremers, A.B.: Towards context-sensitive service aspects, workshop on object technology for ambient intelligence and pervasive computing, in conjunction with 20th european conference on object oriented programming (ecoop 06), july 3-7, nantes, france (2006)
7. Ditrios: Webpage, http://www.ditrios.org. (2006)
8. OSGi Alliance: Listeners Considered Harmful: The "Whiteboard" Pattern - Revision 2. (2004)
9. Rho, T., Kniesel, G.: Uniform genericity for aspect languages, technical report iai-tr-2004-4, computer science department iii, university of bonn. Technical Report IAI-TR-2004-4 (2004)
10. Fortier, A., Gordillo, S., Rossi, G.: Engineering pervasive services for legacy software. SEPS Workshop at IEEE International Conference on Pervasive Services 2006 (ICPS'06) Lyon (2006)