Programming context-aware applications

Andreas Seelinger

Bonn-Aachen International Center for Information Technology (b-it)
Prof. Dr. Albrecht Schmidt
b-it, Dahlmannstr. 2, 53113 Bonn, Germany

1 Abstract

Context-aware programming should enable an application design, which can change its behaviour depending on the context in which it is running. Many approaches exist already, which deal with this topic. Reflex or Distributed Tracking and Interception Of Service (Ditrios) are two elaborated frameworks, but there exist a lot of other frameworks. I will introduce three different approaches in my paper. All have their advantages and disadvantages and fit in some situations better than in others, because they were designed for different tasks. Some examples for the usage of context-aware applications are given in section 3. The introduction and description of the Mixing Layers is written in section 5 and the Aspect-oriented programming (AOP) approaches in section 6. Later on I will analyze shortly the advantages and disadvantages of all methods in section 7. My own statement is written in section 8.

2 Introduction

Before you can discuss the necessity of context-aware frameworks and context-aware programming, you should understand what context-aware programming is and which associated requirements are important for this kind of programming languages. Context-awareness is the ability of objects to behave according to a context in which they are running. I give an example with an simplified cellular phone to illustrate this specific context-aware behaviour. The cellular phone knows two different contexts. First, if the cellular phone battery level is low only contacts which are marked as VIP will be put through. Second, if a call arrives the cellular phone between 22 and 7 o'clock, this call will be redirected automatically to the telephone answering machine. The cellular phone is able to react context-aware in these two defined contexts. One main problem in this context, is the combination of more than one context at the same time.
How should the cellular phone react, if both, the battery level is low and the time is

between 22 and 7 o'clock?
Programming languages which support such programming of devices like this cellular phone, must be able to analyze Sensor-information and depending on this information change the program structure in the executing program. Current object-oriented programming languages often reach their limit in such situations. Such implementations often become unclear and additional changes like adding new contexts for example can affect nearly all parts of the program. It is very hard and time-consuming for a programmer to deal with all these problems. Therefore a new programming concept, the context-aware programming have to be used, that allows programmers to break through the strict program flow of object-oriented program language and change a program in a dynamical way at run-time. I will introduce the static as well as the dynamic representatives of this context-aware programming concepts. But there exist quite a lot of other reasons for context-aware programming languages, for example one side-effect is the support of crosscutting concerns. This includes the effect that if a special context occurs, changes at many points in the program are accomplished at the same time. Hence we have a good reason to analyze context-aware programming.

3 Motivation Example

I will explain the usage of context-aware programming on a simple example of a PDA, which acts as a tourist guide. This special PDA could be lent by a tourist company in towns which are often visited by tourists. This PDA is equipped with a specific map of this town and its neighborhood. The map contains useful information about places of interest, bus and taxi stations, restaurants and other important places. This information is encapsulated in the `Map` class. Also this PDA provides simply navigation algorithms, so that it can compute bus and train paths between different places. The navigation functionality is embedded in the `Navigation` class. This PDA includes a GPS and a bluetooth sensor and is able to communicate with other bluetooth devices. Furthermore it is able to connect to any bluetooth device if available and adopt its functionality. So you can make phone calls with the PDA instead of using a mobile phone or surf in the internet with the Universal Mobile Telecommunications System (UMTS) receiver of an other bluetooth device. The device is then only used as an antenna to the Base Transceiver Station or the UMTS Stations. This functions are implemented in the `Bluetooth Device` class. Finally, the main task of this PDA is to show all these information on its Display and allow the user to explore the city. This functionality is provided in the `Display` class. These different functionalities constitute the *application core logic* of the PDA. The behaviour of the application core logic can change at runtime according to the context. I figured out three different contexts conditions.

TouristGuideConcern if the user enters a defined area of a place of interest or other important places, information like explanations or price lists are shown on the display. The current position is detected by a Global Positioning System (GPS) sensor and the PDA analyzes if the user is in a defined place, for example if the user is in a five meter radius of a sight. Also it depends on the kind of place which

information is shown. So that if the user is in the near of a statue, the history or the tale of this relict is denounced. But if the user is in front of a restaurant, the price list and comments of this restaurant are shown. And of course the user can choose different routes to explore the city, which are hard-coded in the system and show the most and important sights.

UpdateConcern if the PDA is connected to an UMTS device, it can connect to the main server of the tourist company. There it can download available new information of some sights, so that the user has always the newest information about the actual sights in front of him/ her. Even new tourist routes through the town will be uploaded or the existing routes will be updated, when for example a museum is closed. Other base functions are the search of information on the internet.

BatteryLowConcern if the battery level is getting too low, all functionality excepting the navigation functionality is shut down. The PDA will only navigate the user to a predefined place like a hotel, where the user can charge the PDA or spend the night. It is very important that the battery level is high enough to navigate the user to the predefined place.

Although the three context conditions can be active at the same time, the behaviour cannot be freely combined. It can happen that different conditions might conflict with each other or that the system does not know which of these conditions to execute first. In case of a contradiction the designer must take measures to avoid a system-crash. The designer must define the program behaviour, if more than one condition occurs. In this example the following combinations of conditions are available, and it is described how contradictions should be solved.

Rule I All conditions can exist individually

Rule II TouristGuideConcern and UpdateConcern can coexist. If the user selected a tourist route, nothing on this route will be changed. But if the user visits a sight or other important places, which have a new information available, the new information will be shown.

Rule III TouristGuideConcern and BatteryLowConcern can coexist. The actual routes will be stopped and a new route will be calculated to the predefined place. But should the user come close to an important place, its information will still be shown.

Rule IV UpdateConcern and BatteryLowConcern cannot coexist. BatteryLowConcern has priority.

Rule V TouristGuideConcern, UpdateConcern and BatteryLowConcern cannot coexist. Same behaviour as in Rule III.

4 Potential Techniques

In the following sections, I will present a selection of techniques supporting context-aware programming. It is hard to find fully developed techniques for this topic, mainly because this topic is new that just a few works are dealing with it. I will introduce the mixin layer approach as well as the Reflex and Ditrios framework. But first, I will specify requirements for context-aware programming languages, which allow comfortable programming of context-aware applications. Later on I will introduce some techniques, which satisfy most of the requirements. But first I need to explain the meaning of a context on a programming language layer. The notion **context** describes a special situation. This situation must be known by the system and the system needs all required information to analyze if the defined situation is reached or not. If the situation is reached, the context is occurred and the system change its behaviour.

4.1 Requirements / Techniques

- First of all, the context-aware programming language needs access to the **raw sensor layer** or other information, which describes context information. This access is important for the system, because depending on this information the system can analyze and decide, if it is running currently in a special context or is entering a new context.

- Then the programming language must support constructs, which allow to change the running program depending on the given information. These changes must be done synchronously in all relevant classes, so that applications can change their behaviour immediately at runtime. These constructs represent the **context concerns**. These concerns are added to several parts in the program at the same time.

- **Encapsulation of context-depending behaviour** should be supported. Encapsulation is important because more than one context concern could be evaluated at the same time, and the system must react according to all context conditions. For example if the tourist-PDA from section 3 has an UMTS connection and is in front of a place of interest at the same time, the tourist-PDA has to behave according to both contexts. The tourist-PDA must show information from this place of interest and simultaneously download new map information. The programming language must be able to handle such situations.

- Context-aware programming languages should also support **context aggregations**. Context aggregation means, that if more than one context condition occures, that the system mixes the different behaviours in the right way. For example if the tourist-PDA has an UMTS connection and at the same time the battery level is low, the tourist-PDA would just navigate the tourist to his/her predefined place and would not at the same time download new map information (section 3 rule IV). This example shows that context aggregation leads to a different behaviour

than simple parallel execution of both concerns. Context behaviour can depend on other context conditions.

- One important issue is the **context management**. Programming languages should provide a rule engine, where you can define rules like in section 3. This rule system could aggregate context behaviour depending on the defined rules. This allows to implement context constructs considering other contexts. The context aggregations would then be done by the context management with the help of the rule system.
- It should be possible to have information about **past contexts**. This information could be used to implement context conditions which depend on past contexts. For example if a tourist visits a place of interest and has downloaded the latest information and then visits the same place ten minutes later again, it would make sense, that depending on the past events no other download of the same information is done. It is unlike that new information became available in the last ten minutes and the download would cost energy and money needlessly.
- Context-aware programming languages should not only support changing behaviour depending on context information, but also reactions on **context-change events**. Sometimes it make sense to destroy all created objects when a context condition is not true anymore or other actions depending on a context-change event have to be done. It allows a more flexible programming, because the system can react on context-change conditions.

That are the basic requirements desired from a programming-language, which can be used for context-aware programming.

5 Mixin Layers[?]

The notation of mixin layer[?] was introduced by Smaragdakis as an implementation technique to support refinement of collaboration-based designs. mixin layers is a modified variant of a Feature-oriented programming (FOP) language. The basic idea of FOP languages is to separate features from the application core logic by allowing the implementation of mixin classes. Mixin classes are abstract classes, which differ from general classes in the fact, that abstract classes allow the use of parameterized superclasses. In principle FOP languages can implement mixin classes (mixins), that inherit more than one abstract class. Moreover mixins can be executed as instances of various superclasses. I will clarify this in one example[?]:

Example: Door

```
interface IDoor {
    boolean canOpen(Person p);
    boolean canPass(Person p);
}
```

```
class Door implements IDoor{
    boolean canOpen(Person p){ return true; }
    boolean canPass(Person p){ return true;}
}
```

Special doors

```
class LockedDoor extends Door {
    //implements the canOpen method
    boolean canOpen(Person p) {
        if (!p.hasItem(theKey))
            return false;
        else return super.canOpen(p);
    }
}

class ShortDoor extends Door {
    //implements the canPass method
    boolean canPass(Person p) {
        if (p.height() > 140)
            return false; // too tall
        else return super.canPass(p);
    }
}
```

For creating a class which includes both the LockedDoor and the ShortDoor implementation, both classes must be mixed together. Problem: A class for a Short Locked Door is only possible by coping code from LockedDoor class to the ShortDoor implementation.

```
class LockedShortDoor extends ShortDoor {
    boolean canOpen(Person p){
        if (!p.hasItem(theKey))
            return false;
        else return super.canOpen(p);
    }
}
```

Solution with mixins which implement parts of the interface IDoor.

```
mixin Locked extends IDoor {
    boolean canOpen(Person p) {
        if (!p.hasItem(theKey))
            return false;
        else return super.canOpen(p);
```

```
    }
}

mixin Short extends IDoor {
    boolean canPass(Person p) {
        if (p.height() > 140)
            return false; // too tall
        else return super.canPass(p);
    }
}
```

Because mixins usually implement only parts of an interface they are not an adequate class. Therefore a class which implements all method of this interface is needed to create classes from mixins.Combinations of mixins to create a class for a Short Locked Door.

```
class LockedDoor = Locked(Door); \\equal to the implementation above
class ShortDoor = Short(Door);
\\multiple inheritance allows to combine this two mixins
class LockedShortDoor = Locked(Short(Door));
```

Such programming supports reusability and modularization. One important feature of mixin layers is building of collaborations. These collaborations exist between different mixin classes with the same context. These classes are implemented as self-contained single classes, and so each mixin class can be changed independently. Furthermore mixin layers can refine other mixin layers by using the basic idea of multiple inheritances from the FOP languages. The functionality of mixin layers is shown graphically in Figure 1[?], while the mixin layers are presented horizontally and the affected classes and interfaces vertically.Figure 1[?] shows how different mixin layers can be merged together and which functionality is offered by the mixin layer approach. The fundamental idea is that mixins inherit from other classes and can implement those methods. This is shown by mixin1a and interface1, also mixins can inherit from other mixins as well, so that mixin2b inherits from mixin1b and extends the methods of interface2 as well. One other base idea of the mixin layers approach is the possibility of the multiple inheritance, and so mixin2b and mixin2d inherit from two different classes. Mixin2d from interface4 and class and mixin2b from mixin1b and class. This presentation describes two main properties of the mixin layers architecture. On the one hand mixin layers architecture has a good interaction and support of the crosscutting concerns, because layers can affect multiple classes. This means that all modifications from one context condition are combined to one mixin layer, but affect many other classes (*Code Tangling*), and that this code is only implemented in this one layer, so it did not need to be implemented in other classes(*Code Scattering*). On the other hand mixin layers are hierarchically organized, mixin layers refine existing behaviour instead of modifying them. This includes the effect that mixin layers can refine other mixin layers. For this reasons the mixin layers technique is very suitable for programming context-aware applications. I will show this by illustrating the

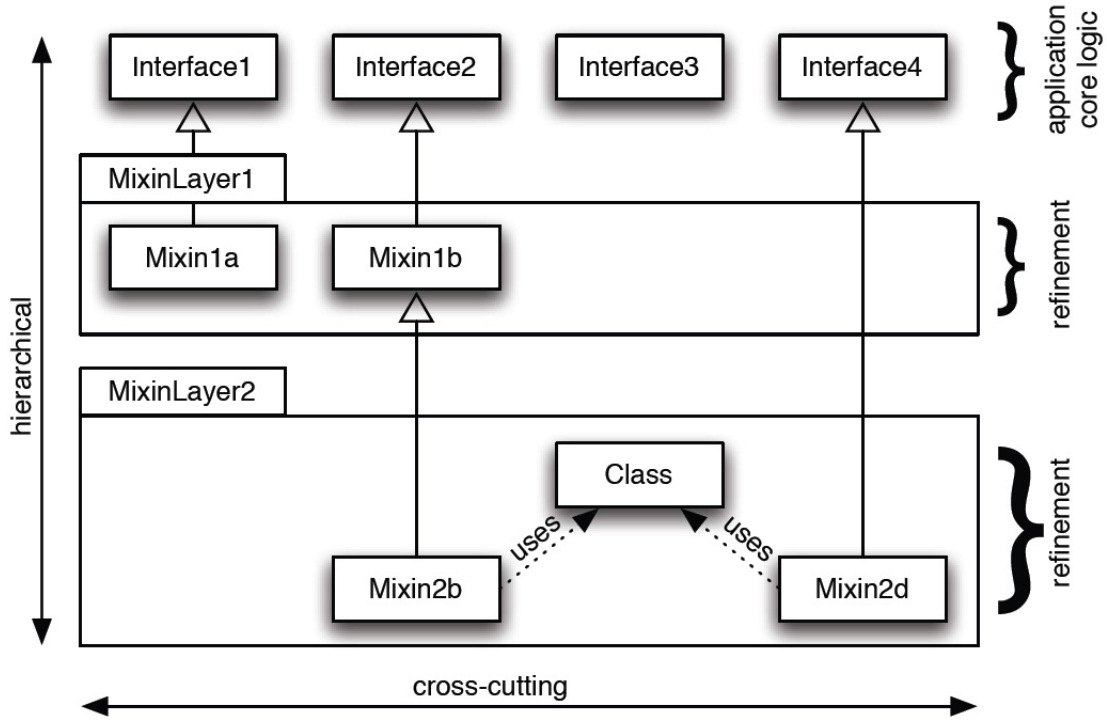

Figure 1: mixin layers

example of Section 3. Figure 2 shows how the TouristGuideConcern can be realized. The NavigationMixin takes the map information from the Map and searches for the right information of the place, which is located in the Navigation class. And finally shows the received information on the display by sending all information to the Display class. Here you can see the basic idea of mixin layers, namely the separation on context-dependent behaviour from the application core logic and the modulation of them using mixins. A correct hierarchical order is very important to avoid system errors, because the system has to know how to combine different layers with each other. For this reason different rules have to be designed like in Section 3. But there are many other reasons for using mixin layers for context-aware programming. One important feature is a high pluggability of this technique, in fact dynamically at runtime. This high pluggability is supported by *dynamic activation* and *dynamic composition* mechanism. *dynamic activation* describes the fact, that behaviour changes are realized by activating and deactivating mixin layers at runtime depending on context changes. Furthermore dynamic activation allows to refine classes at runtime by updating them automatically. In addition activation and deactivation of mixins of the same layer should be done synchronically and it should be an atomic operation. Good base languages for such programming are CLOS[1] or Smalltalk[2].

[1] Common Lisp Object System
[2] http://www.whysmalltalk.com/

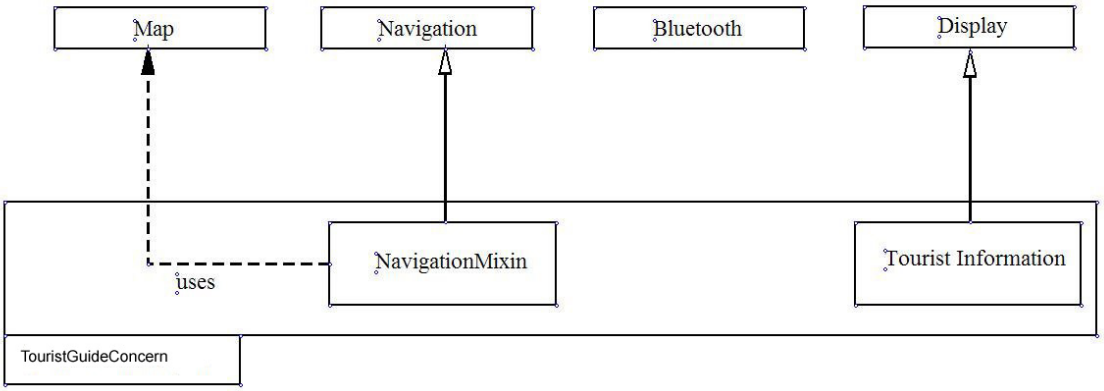

Figure 2: System design if user is standing in front of a place of interest

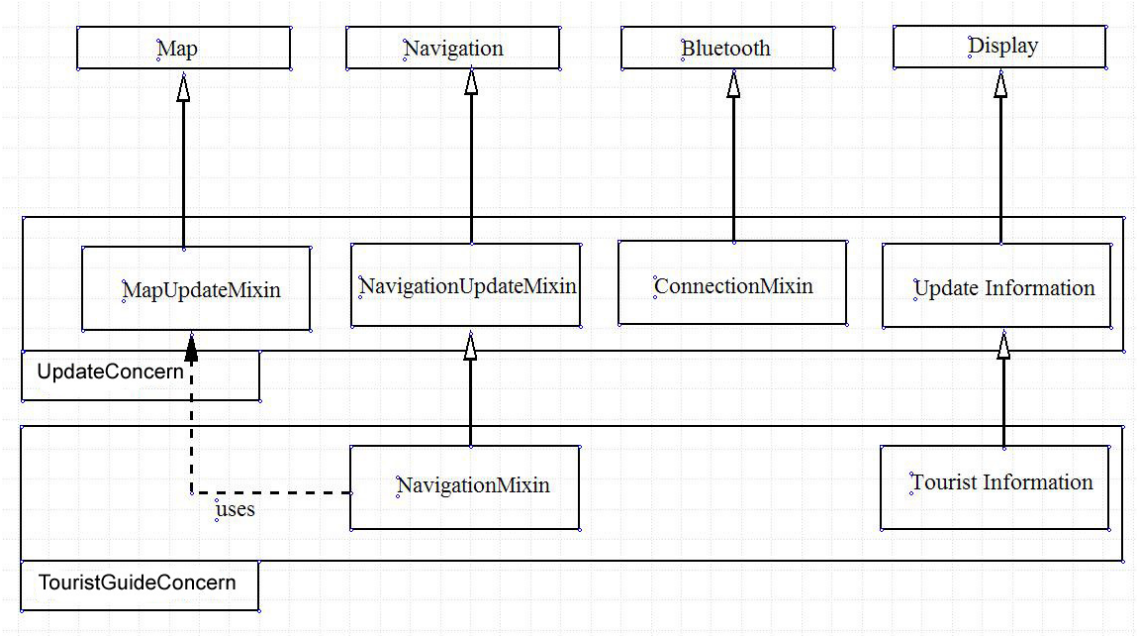

Figure 3: System design if both user is standing in front of a place of interest and PDA has a connection to the Update Center

Dynamic composition mechanism explains the algorithm of how to create composition at runtime. First, all mixin layers that are part of the same composition are computed automatically based on context information. And here is the difference between mixin layers and the FOP languages. In FOP this selection must be done manually at design time. Therefore mixin layers are much better suitable for context-aware application than other FOP languages. Second, the composition of mixin layers changes over time as the context changes. In FOP compositions cannot change at runtime and therefore must be implemented at design time, which causes a loss of flexibility and pluggability. I show

this technique with the Tourist-PDA example from Section 3. The user stands in front of a place of interest and the systems-design is represented by Figure 2. This means that the NavigationMixin takes the map information from the map class and together with the information of the users position from the navigation class the TouristGuideConcern layer is able to show all possible information on the display. Now the situation described by rule II appears and the user connects the UMTS mobile phone via Bluetooth to the Tourist-PDA. The UpdateConcern layer is activated as described by the dynamic activation. It is the responsibility of the dynamic composition mechanism to fulfill this request by adding the UpdateConcern layer to the current set of mixin layers at runtime. This situation is shown in Figure 3. Now the MapUpdateMixin moves between the map class and the NavigationMixin. This means that all map information is updated by the UpdateConcern layer and that the NavigationMixin gets its information from now on from the Update Center Server. All information are shown on the display as before and the navigation route does not change, too. But the new route will be available for the NavigationMixin. It is very important that the system knows how to react when different combinations of contexts become active. Therefore the rule from Section 3 is implicit needed. There are many reasons why mixin layers are suitable for context-aware programming. The important reasons are the flexible and pluggable use of mixins at runtime and the support of different crosscutting concerns, because it allows to make changes at many parts of the program at the same time by using inheritance. The mixin layer approach supports the encapsulation of context-depending behaviour as well as context aggregations by using multiple inheritance and mixin layer which are a set of mixin classes.

6 Aspect-Oriented Programming Techniques

6.1 Aspect-Oriented Programming[?]

AOP is a technique which use aspects for separating feature-specific program code and general requirements like logging and security aspects. Therefore the main aims of AOP is the separating and modularizing of dynamic and static crosscut concerns. It allows to implement aspects, that combine equal and interrelated concerns, in just one class. But before I can introduce the aspects, some other important terms must be described. Dynamic Crosscutting allows to leave the normal program flow and mix other special code in-between. But the normal program flow can only be interrupted on special points, called **advices**. Advices interrupt the program flow and link this point in the execution-path. Some examples for advices are the `before-`, `after-` and `around-`methods. The `around` advice allows to replace the method and execute complete different code. **Join points** now define all possible advices in a programming language. Possible join points could be the call or the execution of an operation or a method, the access to an attribute or a variable of an objective or the throw of an exception. Join points are everywhere in the program code, but to define specific join points **pointcuts** are needed. Pointcuts merge join points and advices by defining special join point advices. For example a pointcut

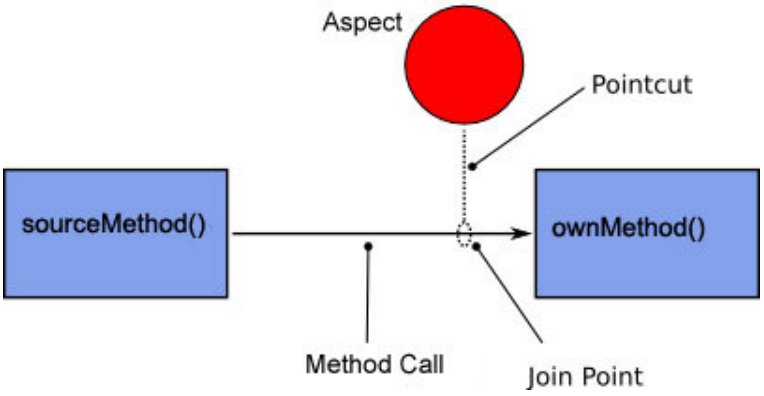

Figure 4: Functionality of aspects

could be the execution of every hallo-method in the class Test.

```
private pointcut testPointcut():
    execution ( * Test.hallo(..));
```

It could happen that a pointcut defines many join points. For example if a pointcut defines every execution of a method in the Test class, this pointcut exists of many join points, namely as many as the number of methods in the Test class. Then again one join point could be part of many pointcuts. For example if you would take the described of both pointcut-examples, the pointcut which defines every execution of a method from the Test class and the testPointcut which only exists of one join point, the call of the Test.hallo method, you would note that the hallo-method execution join point is part of both pointcut-definitions.
Pointcuts select a set of join points and advices represent the code which is executed at the join points, this pointcut refers to. All together represent an **aspect**, which merges the functionality of pointcuts and advices. An aspect can contain many advices, that refer to different pointcuts. This allows a high dynamic support of crosscutting concerns. Figure 4 illustrates the principle of AOP.

One example which shows how aspects works:

Logging Example

```
public class MessageCommunicator {
    public static void deliver(String message){
        System.out.println(" Enter static void deliver(String message)");
        System.out.println(message);
        System.out.println(" Leaving static void deliver(String message)");
    }

    public static void deliver(String person,String message){
        System.out.println(" Enter static void deliver(String person,String message)"
```

```
        System.out.println(person+","+message);
        System.out.println(" Leaving static void deliver(String person,String message
    }
}
```

This example shows a small logging functionality. All methods must be modified to guaranty logging functionality by inserting an output at the beginning and at the end of each method. You can easily see, that this approach leads to a high complexity. Each new method must be modified in the same way (Code Tangling).

Aspect Approach

```
public aspect Logging{
    private pointcut toBeLogged():
        execution ( public * MessageCommunicator.*(..));

    before() : toBeLogged() {
        System.out.println("Enter "+thisJoinPoint.toShortString());
    }

    after() : toBeLogged() {
        System.out.println("Leaving "+thisJoinPointStaticPart.toLongString());
    }
}
```

MessageCommunicator class

```
public class MessageCommunicator {
    public static void deliver(String message){
        System.out.println(message);
    }

    public static void deliver(String person,String message){
        System.out.println(person+","+message);
    }
}
```

Now the logging functionality is encapsuled from the real functionality of the class MessageCommunicator. In this example the aspect Logging adopts the logging functionality. The pointcut toBeLogged includes all join points which fit to this description. That means all executions of all public methods in the MessageCommunicator-class with any kind of parameters. This aspect includes two advices, both refer to the pointcut toBeLogged. The `before`-advice inserts the code before the actual method is executed and signalizes the beginning of the method execution. The `after`-advice inserts the code after the execution of the current method and signalizes the ending of this method.

6.1.1 Kinds of join points

In principle join points can occur everywhere in the program-flow. It depends on the framework which defines all possible join points. The existing aspect-oriented systems only provide a special set of join points. But new approaches offer other join points to extend the application range and so it is hard to say which kind of join points already exist. I will just introduce a few exposed join points.

Method Execution This join point is reached when a concrete implementation of a method is executed. The `before execution` -advice is processed before the method is executed, the `after execution`-advice after the method execution and the `around execution` -advice replaces the execution of the method.

Operation Call This join point is accessed when a concrete method is called, but before it is executed. The advices behave like in the Method Execution join point description.

Constructor Execution This join point includes all constructor calls.

Field Access This join point includes the reading and writing as well as creating and deleting of Data-elements.

6.1.2 Aspect-Oriented Programming for context-aware applications

Aspects based on join points are able to change the program-flow by adding additional code in-between at all join points defined by a pointcut at the same time. But for context-aware programming, context depending join points are needed as well as extended pointcut languages. I will introduce two approaches that extend AOP languages by adding different mechanisms for context checking. The main reasons, why AOP languages are not enough to solve the problem of context-aware programming, are that aspect-oriented frameworks have no tools for saving past contexts and that AOP has bad support for mixing behaviour when more than one context condition is fulfilled. AOP offers limited hierarchy for aspects. That means that if two or more advices, defined in different aspects, are linked to the same join point, no fixed order of passing all aspects can be guaranteed unless an explicit order is defined by the program designer. But AOP on its own offers a good basic framework, because it supports many crosscutting concerns.

6.2 Context-Aware Aspects[?]

Context-aware applications have the ability to change behaviour depending on different contexts. Context-aware aspects introduce the idea of aspects whose behaviour depends on contexts, so that they are able to solve the problems of context-aware applications. Context-aware aspects provide these by designing pointcuts which depend on different contexts, so that advices would only be executed in specific context conditions. Current AOP languages are limited regarding to context condition expression. First, they are not able to consider past contexts, so that for example if a context changes application settings and the actually context have to reset them to execute its changes, aspects have

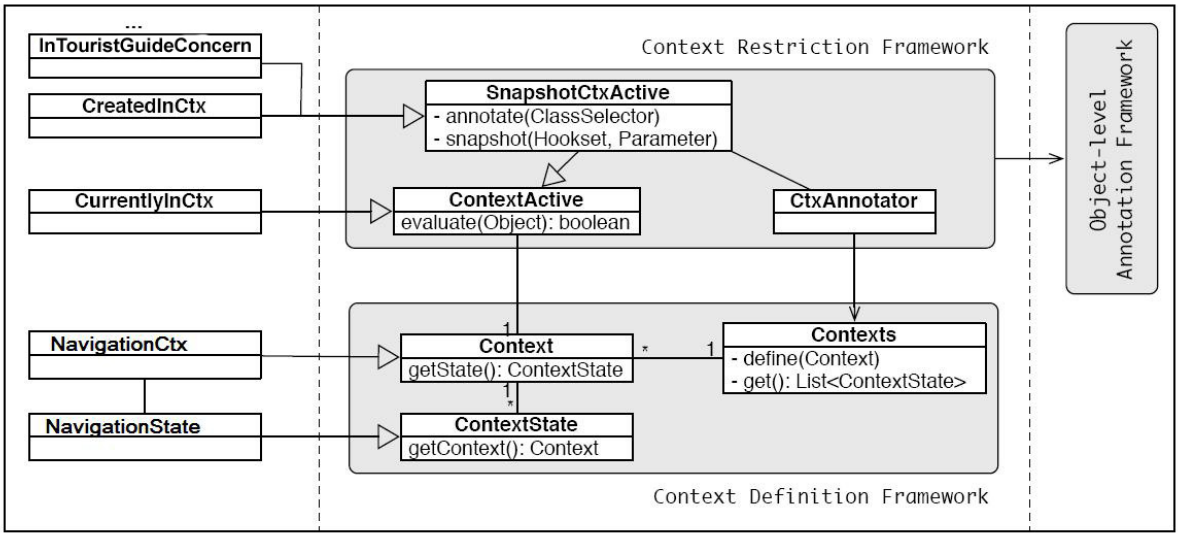

Figure 5: The Reflex-based framework for context-aware aspects: the Context Definition Framework for defining when contexts are active and the Context Restriction Framework for defining context-dependent activation conditions

no functionality to check this. Second, they are not able to express context-dependencies in aspects. Designing aspects that become active when particular contexts are verified, require the possibility to refer to a context definition in a pointcut construction. That means join points like `BeInContext(Context NavigationCtx)` should be provided by the framework. An other important ability of a framework should be giving an overview about all actual and past activated contexts, so that pointcuts can be designed which base on this information. Therefore the framework **Reflex**[3] is used for designing the context-aware aspect approach. Reflex is an open framework for context-aware aspects, that is based on AOP languages for Java like AspectJ. The structure of this framework is illustrated in Figure 5[?]. I will introduce the main functionality of Reflex, for more Information please read the paper[?]. Reflex supports a set of new join points which allow to construct extended pointcuts. One important join point is called `CurrentlyInCtx(Context ...)` and allows to create a new kind of aspect. These restricting aspects base on a context condition which is evaluated on the context which was previously activated or is still active. An other join point allows to construct pointcuts which base on past contexts. The `CreatedInCtx(Object ...)` pointcuts become verified when a special object was created during a past context-dependency. But Reflex allows to create own pointcuts, for example the `InTouristGuideConcern(Object ...)` pointcut condition. This pointcut condition is evaluated when an object has been created while the TouristGuideConcern condition from Section 3 was active. This means that Reflex needs to keep track of past context conditions and their associated states. This is called *context snapshotting*,

[3] http://reflex.dcc.uchile.cl/confluence/display/RFX/Home

and the saved state of one context condition at a given point of time is called *context snapshot*. A *global context snapshot* is therefore a snapshot of all context conditions at a given point in time. Context snapshots are only made at a special point of time, because otherwise it would lead to high memory problem. So the main problem is to define the right points of time to take such content snapshots. The actual solution is to take snapshots of context conditions only if necessary. Thus if an object is affected by an aspect that is subject to a creation-in-context dependency, this context condition state is saved by a context snapshot. All this functionality is provided by the framework Reflex and I will explain the illustrated classes from Figure 5[**?**].

Context Reflex represents context as objects with a method `getState()` which returns null if the context is not active and otherwise returns a snapshot of the context as a ContextState object.

ContextActive To check if an object is in an active context, Reflex designed the ContextActive class. The evaluate-method returns true if a given object is in the prompted context and false otherwise. In the constructor, the context on which the activation condition depends is stored in an instance variable. The activation condition evaluates to true if the associated context is active, or is determined to be active by getContextState.

SnapshotContextActive To providing the support of snapshotting the class SnapshotContextActive is designed. It offers the ability to define conditions depending on past context snapshots.

With such construction options a context-aware application can be designed. What AOP languages do not support, now is supported by Reflex in the form of context-aware aspects. The ability of context-aware aspects to depend on actual and past context conditions make it lucrative to design context-aware aspects with the help of the Reflex Framework. With Reflex it is possible to design constructs which handle the rule II from Section 3. In this example the UpdateAspect defines many independent pointcuts.

```
aspect UpdateAspect{
    \\Rule I
    pointcut independent() : execution( * *.*(..)) &&
        inNoOtherCtx(UpdateCtx);

    before(): independent() { updateMap(); }


    \\Rule II
    pointcut touristGuide(long, lat) :
        execution(* Map.getInformation(lang,lat) &&
        inContext(TouristGuideConcern);

    before(): touristGuide(long,lat) { updateMap(long,lat); }
```

```
    ...
}
```

The touristGuide pointcut represents the situation where both the UpdateConcern condition and the TouristGuideConcern condition are active like in rule II. The aspect now updates the map at the needed place before other map specific methods are executed. If no other context than the UpdateCtx is evaluated the independent-pointcut becomes evaluated and the map is updated itself by downloading the newest information from the server (rule I). Therefore it is possible to design context-aware applications with context-aware aspects.

6.3 Context-Sensitive Service Aspects[?]

This approach takes the idea of snapshotting and extends it by introducing context-sensitive services. Services are the offered features of an application and are the stable part of a system, because they do not change during the application runs. It is possible that not all of them are offered at the same time. Therefore a mechanism must select the needed services. I take the example from Section 3. If the battery level of the PDA is low, only the service Navigation-Home is offered and all other services like Updating-Map are stopped. This prevents the system of getting out of energy. This selection is done by service aspects, the unstable part of the system. They change the settings of services depending on the actual contexts. The idea of context sensitive service aspects is introduced by the Ditrios[4] framework. Ditrios is a Service-Oriented Architecture (SOA)[5] framework based on Open Services Gateway Initiative (OSGi)[6]. Therefore Ditrios combines two different approaches, aspect-oriented programming and the service-oriented architecture. Services are selected by aspects and weaved together by a proxy. The functionality of Ditrios is illustrated in Figure 6[**?**]. This Figure shows how this Client-Server principle is structured. When a client asks for available services, first it must establish a connection to the *ClientService*. The ClientService is a special interface that provides access to the Ditrios framework by established connections between Ditrios and client applications. Clients must ask for a request object which is mainly a Lightweight Directory Access Protocol (LDAP) search string. After a request reached the ClientService, a proxy instance wraps all found services from the Service pool identified by the asked search string and sends it back to the client. The *proxy* instance wraps all services belonging to the same request. They are selected by service aspects which change the selection set accordingly to current contexts. Because of this proxy indirection clients do not have to care about stale references, no longer available services and other imaginable situations. Every status change of the requested services is recognized by an event listen on the side of the client.

[4] http://www.ditrios.org/
[5] http://www.xml.com/pub/a/ws/2003/09/30/soa.html
[6] http://www.osgi.org/

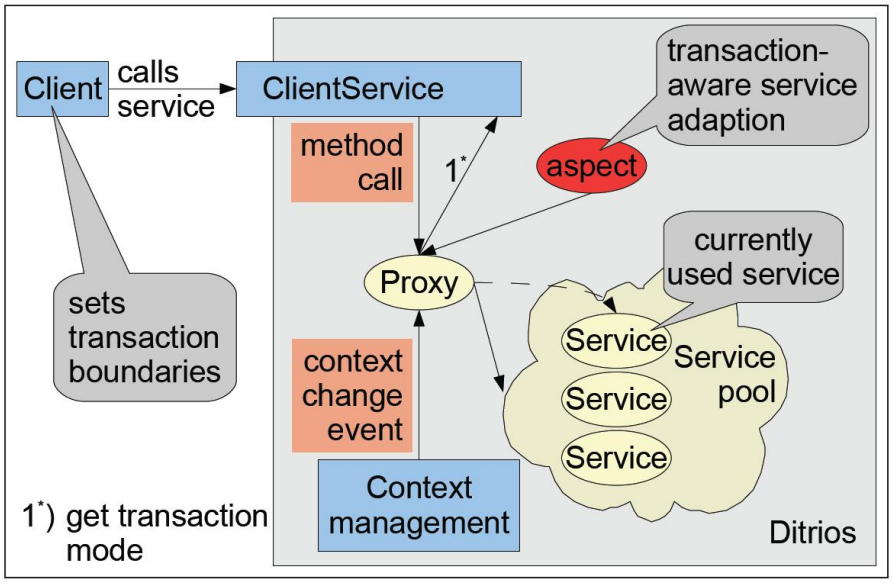

Figure 6: Ditrios functionality: Separating services and aspects. Combine selected services by a Proxy.

Service aspects are the heart of the Ditrios architecture.

[?] They operate on proxies and services as described in Figure 6[?] and provide the means to control dynamic composition of services. They use new join point models and an extended pointcut language to allow such functionality, although to implement this the CSLogicAJ language is used. Service aspects are enclosed in service holding states and by simply activating and deactivating them, services are weaved or unweaved by the proxy. Service aspects support two different kinds of advices synchronous and asynchronous advices. *Synchronous advices* need a program-execution pointcut as described in Section 6, they just put code in-between the program-flow or provide intercepting call for services. `Before`, `after` or `around` advices are common representatives of the synchronous advices. Asynchronous advices do not need program-execution pointcuts. They only react on context changes indicated through context providers. Context information is provided by the Context Management. The *Context Management* tracks and analyzes contexts with the help of different sensors or profile information. But the context management uses the approach of snapshotting, too, as described in Section 6.2. This allows not a view of past context states, but is used to analyze in which context states the system is at a special point of time. This snapshot is deleted afterwards and therefore no other context condition can depend on it. Recapitulating all information services are selected by service aspects which gets their information form the Context Management and uses the CSLogicAJ language. Furthermore the proxy weaves all found services and sends them back to the asking client. I will give a short example of an aspect structure in CSLogicAJ.

```
aspect UpdateAspect {
```

```
// Rule I
    pointcut noDependencies() :
        currentAspect(?aspect) &&
        aspectsActive([?aspect]);
    onchange() :
        noDependencies()
    {
        updateMap();
    }

// fail:

    onchange(?aspect) :
        !noDependencies()
    {
    stopUpdateMap();
    }

// Rule II
    pointcut touristGuideActive(?long, ?lat) :
        aspectActive("org.cs3.TouristGuideConcern") &&
        execution( * Map.getInformation(..)) &&
        position(?long, ?lat,?alt);
    before(): touristGuideActive(?long, ?lat) {
        updateMap(?long,?lat);
    }
}
```

This example illustrates how a service aspect implements the rules I and II for the UpdateConcern. The pointcut noDependencies is evaluated only if no other context condition is active and with the help of the asynchronous advice *onchange* the map information will be updated while the noDependencies-pointcut is evaluated [rule I]. Rule II is satisfied as well by this aspect, course the touristGuideActive pointcut becomes evaluated if in addition the TouristGuideConcern is active, too, then the information of the current place is updated before the map information is shown. Therefore the service aspect approach allows the implantation of contexts as well.

7 Conclusion

In this Section, I will analyze which requirements from Section 4.1 are fulfilled by the individual approaches.

- First, the different approaches need information to decide if a special context condition is evaluated or not. Access to the raw sensor level is needed. The service

aspect approach retrieves context information, like hardware sensor or profile information from a OSGi service. Sensors or other available information have to implement the IContextProvider[?] interface and register the implementing class as an OSGi service. Once registered Ditrios can access the context information via the context management system and decide if a special context condition is fulfilled. The context-aware aspect approach uses context-awareness toolkits such as WildCAT[?] to get access to the raw data level. These context awareness toolkits manage sensors and other evaluable information so that the context-aware aspects can access this data and implement conditions depending on them. The mixin layer approach does not consider this topic. It is assumed that all needed context information is available for the system. Therefore I have no information how the mixin layer approach handles this topic.

- All approaches provide constructs for implementing context concerns. The context-sensitive service aspect approach allows the use of service aspects. Context-sensitive service aspects are an advancement of aspects. These support the change of many classes at the same time and allow the change of the program-flow at runtime. All this advantages of AOP languages are described in section 6.1.2. Service aspects also allow to implement conditions which can use information from the raw data level and decide if a context condition is evaluated or not.
  The context-aware aspect approach also uses aspects to realize context concerns. Two important differences between those two is that service aspects use the CSLogicAJ language to implement aspects and context-aware aspects the "normal" AspectJ language. CSLogicAJ allows the use of meta-variables, which for example allows to design generical calls of methods and have a greater expressiveness than the normal aspect languages. One other difference is that context-aware aspects get their evaluable information from context-awareness toolkits. Context conditions are implemented as an instance of the *context class*.
  The mixin layer approach uses mixin layer to allow implementations of context concerns. Because a mixin layer is a set of mixins, it is possible that many mixins change the program at many parts at the same time. Also the change of a situation which lets the program run in a special context and is implemented by a mixin layer, is separated according its functionality in different mixins, so that each mixin class represents only one change in the program-structure. This supports a clear program design and the program designer knows where to search if an error occurs. How the mixin layer approach decides, if it is running in a special context or not, is not specified and therefore I have no information about it.

- Encapsulating of different contexts is in principle possible in all approaches. Service aspects can be freely combined, because they are independent from each other and should one join point be defined by more than one service aspect, these service aspects would make their changes independently after each other. That means that if two different service aspects each define a different context are evaluated at the same time and change the same method, they would do this in a strict order.

For example in a lexical order or a self-define order as it is possible in many AOP languages. Sure, these kinds of ordering can course functionality problems and do not have the respected result every time. For example if one aspect adds a security request before one special method and an other aspect overwrites the same method by using the *around* advice, the first aspect has no effect any more, because the added code is removed by the second aspect.
Also the context-aware aspects have the same advantages and problems as the service aspects, because both are AOP languages and have the same sources.
The mixin layer approach supports the encapsulation of mixin layers, moreover it has a main focus in this topic. When a mixin layer becomes active the mixins depending to this mixin layer implement there functionality to the inherited classes. Should now a second mixin layer becomes active and its mixins want to implement their functionality to the same classes as the mixin layer before, these mixins do not inherit from the original class, but inherit from the mixins of the other mixin layer, which implement these classes before. The result is that the resulting class supports both functionalities. For example if the mixin1 of a mixin layer inherits from the interface1 and implements it by adding security aspects to it and a mixin2 from an other mixin layer wants to implement its functionality like logging aspects to the interface1, too, it would inherit from the mixin1 which implements this interface before and added its security aspects to it. The result is that the logging functionality is added to the implementation of mixin1 by mixin2. But the mixin layer approach has the same problem as the other approaches, if all mixin layers are implemented independently from each other without knowing each other, it could happen that one mixin from a mixin layer overwrites the functionality of an other mixin from an other mixin layer. This could happen because mixins inherit from other mixins and therefore can overwrite existing methods of these mixins.

- Context aggregation is a topic where the mixin layer approach can score against the other two approaches. With its multiple inheritance of classes and mixins the mixin layer approach has an advantage. Thus that mixins can inherit from other classes, they are able to manipulate their code in many ways, for example extending a method or replacing it or mixing the functionality of two mixin classes together. This allows a flexible design of context aggregations. One other important issue is that mixin layers support hierarchical structures, thereby the designer can make a ranking list which helps the system to combine and aggregate mixin layers. For example the BatteryLowConcern-mixin layer from section 3 would have the highest priority in this hierarchical structure, so that it would dominate all other mixin layers. This mixin layer can overwrite the changes of all other mixin layers like it is required from the rules in section 3. A correct aggregation is possible by this approach. But this hierarchical structure has a disadvantage. Mixins from a layer with lower priority could never change mixins from a higher priority mixin layer. This face limits the possible combinations of satisfiable context aggregations, so that the mixin layer approach could never implement each combination of context aggregations.

The context-aware aspect approach supports correct aggregations as well but with a higher cost. All context-aware aspects have to know all context conditions and in every context-aware aspect additional queries are needed to control the set of current context conditions. For this reason the *CurrentlyInCtx* class exists which is able to control if a current context-aware aspect is running in a special context. This allows to implement pointcuts which depend on a set of context conditions. But with an increase of context conditions the number of different pointcuts would increase exponential, because the possible combinations of these contexts increase exponential. The same problem has the service aspect approach, too, because in both approaches the hierarchical order is not supported. All service aspects have to know every single set of context conditions. Depending on that they have to change their behaviour. It is plausible that with an increase of different context conditions the possible set of context conditions increases exponential. That means that the AOP approaches have a big problem with a muchness of context conditions.

- No approach provides a rule system or a concern collaboration management which mixes different context behaviours in a defined way. But the mixin layer approach with its hierarchical structure of mixin layers allows something like an light rule system, but here the program designer has to create settings at designing time. An automatical mechanism does not exist. Also the service aspect approach with its proxy server could use this proxy server to mix behaviours before the system reacts on a context condition change. The context-aware approach does not have a rule system, either, and I do not see how it could be implemented in this approach.
  But at all it is for each approach hard to implement such a construct, because this construct must be able to change mixin classes or aspects and thereby change the context behaviour of single contexts. Here is a need of action in every approach.

- The only approach that supports the use of information of past contexts is the context-aware approach. The Reflex framework spends a lot of energy to support such functionality. This functionality is possible, because the snapshot technique provides information of past context conditions. But this means that the system has to know when to make a snapshot, because making snapshots of every point in time would cost enormous capacity to save them all. Therefore the system has a complex mechanism to spot after such points. For example a snapshot is taken of a situation if this situation is required in a pointcut condition somewhere in the programme. The service aspect approach does make snapshots, too. But here they are taken only for guaranteing a stable situation while scanning the situation, and depending on that deciding if the programme is running in a special set of context conditions. If a mechanism as in the context-aware approach would be developed for the service aspect approach, it could use past context information, too. Snapshotting or other techniques for saving sets of past context conditions are not supported by the mixin layer approach.

- The only approach which deals with asynchronous advices is the service aspect approach. It supports actions on context-change events and has therefore a clear

	mixin layer	context-aware aspects	context-sensitive service aspects
Access to the raw sensor layer	*	++	++
Context concerns	++	++	++
Context encapsulation	++	+	+
Context aggregation	+	-	-
Concern collaboration management rule system	o	-	o
Information of past contexts	–	++	o
Context-change events	–	o	++

Table 1: Overview about all Requirements and how the different approaches fulfill them. [*: no near information available]

advantage about the other two approaches. This advice is called *onchange* and can be used to execute code on a context-change event. The context-aware aspect approach does not support such functionality, because it is not possible to design such advices without having a modified AOP language like CSLogicAJ, where you can get context information with the help of meta-variables. This context information helps to define whether an object is part of a set of context information or not.
The mixin layer approach does not support such functionality, too, because mixins only inherit from other classes or not, and the execution of code while inheriting is not provided or supported.

I summarized all requirements and how the different approaches fulfill them in the table 1. A ++ is the best support and a – is the worst support of a requirement. This table shows the main focus of each approach. Mixin layers have a good support for context aggregation and capsulation and do not support additional functionality like the information about past context conditions and the possibility to execute code on a context-change event. Context-aware aspects have their main focus on offering information about past contexts, but they have limited support for context aggregation and the possibility to implement a rule system. One other point is that context-aware aspects do not support context-change advices. Service aspects have their focus on context-change events and have a limited support for context aggregation. In all other topic service aspects are average.

8 Statement

Context-aware programming is a big topic. It requires many considerations from a program designer to construct such a context-aware language. Considerations about context aggregation and context evaluation must be done. The program must be able to change its code and structure at runtime. And all approaches are far away from being

perfect for every requirement. Therefore no approach I introduced fulfills all described requirements perfectly. I think that all these approaches can learn from each other, because every approach has its strength and weakness. Other approaches which combine the functionality of AOP and FOP languages currently are developed[**?**]. Every approach can learn from the strengths of the other approaches to eliminate its weaknesses. I think a topic where all approaches have a need for action is the context aggregation, because no approach solves this problem in a perfect way. But at all, context-aware programming will become an important issue, because it offers the functionality for the future. It allows the program designers to break through the borders of object oriented program languages and construct dynamic and versatile programs.

List of Acronyms

AOP	Aspect-oriented programming
Ditrios	Distributed Tracking and Interception Of Service
FOP	Feature-oriented programming
GPS	Global Positioning System
LDAP	Lightweight Directory Access Protocol
mixins	mixin classes
OSGi	Open Services Gateway Initiative
SOA	Service-Oriented Architecture
UMTS	Universal Mobile Telecommunications System

References

[1] URL http://www-pu.informatik.uni-tuebingen.de/oopl-0506/material/mixins-4.pdf.

[2] Gregor Rayman Bernhard Lahres. *Praxisbuch Objektorientierung*. Galileo Computing, 2006. ISBN 978-3-89842-624-4. URL http://www.galileocomputing.de/openbook/oo/oo_07_aspekteinoo_000.htm.

[3] Stijn Mostinckx Brecht Desmet, Jorge Vallejos Vargas and Pascal Costanza. *Using Mixin Layers for Context-Aware and Self-Adaptable Systems*. Pleinlaan 2, 1050 Brussels, Belgium, May 2006. URL http://sam.iai.uni-bonn.de/ot4ami/attach?page=Doc%2FDesmet_OT4AmI_2006.pdf.

[4] Thomas Ledoux Pierre-Charles David. WildCAT: a generic framework for context-aware applications. Grenoble, France, November 2005. URL http://portal.acm.org/ft_gateway.cfm?id=1101483&type=pdf&coll=GUIDE&dl=GUIDE&CFID=20401896&CFTOKEN=63341331. In Proceeding of MPAC 05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing.

[5] Marcus Denker Alexandre Bergel ic Tanter, Kris Gybels. Context-Aware Aspects. Springer-Verlag LNCS, March 2006. URL http://www.iam.unibe.ch/~scg/Archive/Papers/Tant06aContextAspectsSC2006.pdf. In Proceedings of the 5th International Symposium on Software Composition (SC 2006).

[6] Yannis Smaragdakis and Don Batory. *Implementing layered designs with mixin layers*, pages 550–570. Springer-Verlag LNCS 1445, 1998. URL http://portal.acm.org/citation.cfm?id=679703. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP).

[7] Gunter Saake Sven Apel, Thomas Leich. Aspectual mixin layers: aspects and features in concert. ACM Press, 2006. ISBN 1-59593-375-1. URL http://portal.acm.org/ft_gateway.cfm?id=1134304&type=pdf&coll=GUIDE&dl=GUIDE&CFID=24936694&CFTOKEN=54986672. International Conference on Software Engineering, Proceeding of the 28th international conference on Software engineering.

[8] Mark Schmatz Tobias Rho. *Towards Context-Sensitive Service Aspects*, April, 15th 2006. URL http://sam.iai.uni-bonn.de/ot4ami/attach?page=Doc%2FRho_OT4AmI_2006.pdf.