STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LX**, Number 2, 2015

A STARUML PLUGIN FOR INCLUDING ASPECTS IN A UML CLASS DIAGRAM

BRISTENA VRÂNCIANU AND GRIGORETA S. COJOCAR

Abstract. Aspect oriented programming (AOP) is a programming paradigm that complements the existing programming paradigms in order to be able to clearly separate all the concerns from a software system in analysis, design and implementation phases. One of the main difficulties when using the aspect oriented paradigm is that the control flow of the system is difficult to follow and understand just inspecting the source code since not all the relevant data about a piece of code can be seen at that code. Some additional information may exist in the aspect that affect that part of code. In this paper we propose a set of notations for including aspects in an UML class diagram and we present a StarUML plugin that allows the use of these notations. The aspects, their relationships with other aspects, classes or interfaces, and the visualization of the classes that will be modified dynamically or statically by includind the aspects into the final system can be represented with the plugin. This may ease the understanding of the overall static structure of a software system and may highlight the consequences of adding aspects to a software system.

1. Introduction

Separation of concerns [15] is always an important factor in designing easily maintainable and evolvable software systems. However, practice has shown that it is not easy to clearly separate all the concerns from a software system. Most concerns can be clearly separated using just one programming paradigm, however there still are some concerns whose design and implementation are entangled with other concerns. Other programming paradigms, extending the existing ones, have been developed in order to allow better separation of concerns that are still entangled. Aspect oriented paradigm (AOP) is one of these paradigms and it usually extends the object-oriented paradigm [13].

Received by the editors: .June 26, 2015.

2010 *Mathematics Subject Classification.* 68N19, 68N99.

1998 *CR Categories and Descriptors.* D.2.2[**Software Engineering**]: Design Tools and Techniques – *Computer-aided software engineering*; D.1.m [**Software**]: Programming Techniques – *Miscellaneous*.

Key words and phrases. aspect oriented paradigm, design, UML class diagram.

Even though there are already a number of aspect oriented languages such as AspectJ [3] and AspectC++ [2] that provide new language constructs for implementing the crosscutting concerns, there is no generally accepted design notation that supports the design of aspect oriented systems.

Having a graphical design notation would make the understanding of an aspect oriented system much easier and it would also serve as a basis for assessing the impact of crosscutting concerns on their base classes (core classes).

The Unified Modeling Language (UML) is a graphical language for visualizing, constructing, specifying and documenting the artefacts of a software system. The goal of UML is to provide tools for analysis, design, and implementation of software based systems to all parties involved: developers, system architects, etc [5, 17, 22]. One of the main advantages of using UML is that it has defined a set of modeling concepts that are now generally accepted, and it also contains visual representation of the defined concepts that are easy to understand and interpret by humans. The concepts defined by UML can be used to represent the static structure (such as classes, components, node artifacts) or the behavior (such as activities, interactions, state machines) of the software system [21]. The concepts can be used to build different kind of diagrams (i.e., class diagrams for the static structure and sequence diagrams for the behavior). Even though UML contains a very large set of concepts, it does not include all the concepts that may appear during the development of different kinds of software systems, mainly because some of the concepts are specific to a certain application domain. That is why, the language also contains extension mechanisms that allow the addition of new modeling elements, modify the specification of the existing ones or change their semantics [5].

The main contributions of this paper are the proposal of a new notation for representing introductions in an UML class diagram and the development of a StarUML plugin that allows developers to include aspects in UML class diagrams. Aspects relationships with other elements from a UML class diagram can also be displayed using the proposed plugin.

The paper is structured as follows. In Section 2 we present the new concepts introduced by the aspect oriented paradigm. The existing proposed notations for including aspects in an UML class diagram are presented in Section 3. The proposed notations are given in Section 4 and the StarUML plugin is described in Section 5. A small example of using the proposed notations is presented in Section 6. Some conclusions and future research directions are given in Section 7.

2. AOP Concepts

In order to design and implement crosscutting concerns, the aspect oriented paradigm introduces new concepts: *join point*, *pointcut*, *advice*, *aspect* and *introduction*, and *weaving* for building the final software system. An important characteristic of aspect oriented programming is that it can only be used for crosscutting concerns, the core concerns are still designed and implemented using the base programming paradigm, that usually is object-oriented programming, but it can be any other programming paradigm. In the following the concepts introduced by AOP are briefly presented.

Join point. A *join point* is a well-defined point in the execution of a program. Any software systems can be seen as a sequence of execution points like: assignments, conditional statements (if, switch), loop statements (for, while, do-while or repeat), function/method calls, function/methods executions, etc. regardless of the programming paradigm used for developing the system. Aspect oriented programming only uses some of these points, called join points, in order to add new behavior.

Pointcut. The execution of a software system consists of many join points. However, not all of them are necessary for the design and implementation of crosscutting concerns. A *pointcut* selects join points, and exposes some of the values in the execution context of those join points.

Advice. A pointcut allows selecting join points from the software system, however they do not change its behavior. An *advice* defines crosscutting behavior and it is defined in terms of pointcuts. The code of an advice runs at every join point selected by its pointcut. There are different options as to when the code of the advice is executed relatively to the corresponding join point(s):

- *Before*: the advice code is executed before the selected join point. This type of advice does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).
- *After*: the advice code is executed after the selected join point. There can be three situations, depending on the execution of the join point:
  - *After returning*: the advice code is executed only if the join point execution completes normally.
  - *After throwing*: the advice code is executed only if the join point execution ends by throwing an exception.
  - *After (finally)*: the advice code is executed regardless of the means by which the selected join point exits (normal or exceptional return).

- *Around*: the advice code surrounds the selected join point. It can perform custom behavior before and after the selected join point. It can also decide whether the selected join point should still be executed or not, or it may cause multiple executions of the selected join point.

Introduction. It is sometimes necessary to modify the static structure of a type (by adding new members - attributes/methods or by modifying its inheritance hierarchy) in order to design and implement a crosscutting concern. Even though advices add new behavior to existing types, they do not modify their static structure. An *introduction* allows developers to extend the static structure of existing types. New methods and/or attributes can be added, or the type inheritance hierarchy can be modified (by adding new interfaces or by modifying the base type of the existing type).

Aspect. An *aspect* is a new kind of type specified by the aspect oriented paradigm that is used to implement one crosscutting concern in a modular way. An aspect is similar to a class, it can contain attributes and methods declarations but it also encapsulates pointcuts, advice and introductions. In some aspect oriented languages (i.e., AspectJ [3], AspectC++[2]) aspects can inherit from other classes, implement some interfaces or even inherit from other aspects. However some constraints must be followed when inheriting from another aspect. For example, in AspectJ an aspect can inherit only from an abstract aspect.

Weaving. When the aspect oriented paradigm is used for developing software systems, the core concerns are developed independently of the crosscutting concerns. However, in the end, they still have to be put together in order to obtain the final executing system. *Weaving* is the process that produces the final system, and the *weaver* is the tool used to produce it. The weaver takes some representation of the core concerns (source code or binaries), some representation of the crosscutting concerns (source code or binaries) and produces the output, which is often a binary representation. The approach used for weaving depends on the aspect oriented language: AspectJ uses byte-code modification, Spring AOP uses dynamic proxies, while AspectC++ uses source code preprocessing.

3. Existing UML-based Notations and Plugins for Aspects

Since the appearance of aspect oriented programming, many attempts to identify an appropriate notation of aspect oriented design have been made. Most approaches (Suzuki and Yamamoto [20], Aldawud et al. [1], Kande et al. [12, 11], Zakaria et al. [23], Pawlak et al. [16], Stein et al. [19], Jacobson and Ng [10], Basch and Sanchez [4], and Zhang [24]) focused on introducing new

modeling elements for the concepts defined by AOP: aspect, advice, pointcut and the relevant relationships, while other approaches (like the one proposed by Herrero et al. [9]) focused on a particular crosscutting concern and tried to introduce special notations for the elements needed to design that crosscutting concern.

All the proposals have considered a way of representing the aspect concept into the class diagram. Most of them also considered representing the advice [12, 16, 19, 20, 23, 24] and the pointcut [10, 12, 16, 19, 24]. Very few proposals considered representing introductions [10, 19, 20] and even fewer considered representing join points [4, 19]. There is very little consensus related to the appropriateness of a chosen representation for the concepts introduced by AOP:

- *Aspect* - The aspect is usually represented starting from the *Class* classifier enhanced with the *aspect* stereotype [1, 16, 19, 23] or starting directly from the *Classifier* [10, 12, 20]. Only a few proposals considered using a non classifier as a starting element, namely the package with two compartments for pointcuts and advices [4, 24].
- *Join point* - There are only two proposals for representing them, consisting in links [19] and a notation in the form of a circle with a cross inside [4].
- *Pointcut* - There is very little overlapping between the proposals for representing pointcuts. In [12] and [19] pointcuts are represented using the *pointcut* stereotype, while Pawlak et al. [16] propose a representation based on an association from an aspect class towards a classifier stereotype with *pointcut*. Zakaria et al. [23] model a pointcut based on the *Class* classifier and by providing a link to its aspect by a *has pointcut* association. Zhang sees it as a stereotype package [24]. Jacobson and Ng [10] proposed representing pointcuts as operations in their own compartment of the aspect stereotype.
- *Advice* - Stein et al.[19] and Zakaria et al.[23] model an advice as an UML operation of a stereotyped named *advice*; while Suzuki and Yamamoto [20] provide a suggestion of using a constraint for the corresponding weave and Kande et al. [12] suggest to represent it as a compartment in the new *aspect* classifier.
- *Introductions* - There is only one approach, proposed by Stein et al. [19], in explicitly modeling introductions that uses parameterized templates. The class extensions compartment introduced by Jacobson et Ng [10] in their proposal can also be used to represent introductions, however from their representation it is not very clear how the extensions will be realized (introduction or advice).

Only a few proposals (Aldawud et al. [1], Suzuki and Yamamoto [20], Zakaria et al. [23], Kande et al. [12] and Stein et al. [19]) have taken into consideration the relationships that the aspects can have with the other elements from the class diagram (classes, interfaces). Aldawud et al. [1] proposed to use the *control* relationship to represent which other classes the aspect code controls. Suzuki and Yamamoto [20] proposed the usage of the already existing *realize* relationship to represent an aspect and the classes that the aspect affects. Zakaria et al. [23] proposed to use one of their newly introduced *control, track, report, customize, validate, save, handleerror, handleexception* relationships to represent the relation between an aspect and a class. Each aspect should have at least one association with a class in order to affect the system. Kande et al. [12] introduced the *binding* relationship to specify what class of objects an instance of the aspect can be bound to. Stein et al. [19] introduced the *crosscut* relationship between an aspect and a class to specify that the aspect affects the class. The crosscut relationship also implies that the aspect requires the presence of the class in order to behave as expected.

Very few proposals considered explicilty vizualizing the parts that will be affected by introducing one or more aspects, however for some proposals the structure of the aspect notation or the relationships introduced can be used for determining the parts of the software system that are affected by the aspects presence. Jacobson and Ng proposal [10] display the affected parts in the *Class extensions* compartment that contains all the classes from the system that will be affected by the aspect (either statically by introductions or dynamically through advice). The aspect notation proposed by Kande et al. [12] contains compartments that display introductions, meaning that those classes will be affected by the aspect. Also, the binding relationship introduced by them show other affected parts (through dynamic crosscutting). The aspect-class relationships can also be considered as relationships that show the affected parts of the software system.

Plugins. Even though there were many proposals for including aspects into an UML class diagram, only a couple plugins were developed (Suzuki and Yamamoto [20] and Herrero et al. [9]) in order to actually use the proposed notations. These plugins were developed for Rational Rose CASE tool.

4. UML Aspect Notations Proposed for Designing

We consider that not all the concepts introduced by the aspect oriented paradigm can and should be represented in a class diagram. The join points from a software system do not provide any relevant information about the static structure of the system. However, the visual representation of other concepts can provide useful information to the developers. Adding aspects to

the class diagram may ease the understanding of the overall static structure of a software system and may highlight the consequences of adding aspects to a software system. The information that should, in our opinion, be represented in a class diagram are:

- The aspects that are used for building the system, and their type (abstract or concrete). The internal static structure of an aspect is important as it will show, besides the normal fields and methods, the defined pointcuts together with the collected context, the type of the pointcut (abstract or concret), and the defined advice and their type (before, after, around).
- If and how they change the static structure of other existing elements from the class diagram (classes, interfaces). It should represent the type whose static structure will be modified either by introducing new members (fields, methods, etc.) or by modifying the inheritance hierarchy of the type (adding a base class or implementing interfaces).
- Relationships with other elements from the class diagram. We consider important the following relationships:
  - Aspect-aspect: An aspect may inherit from another aspect, or an aspect may have precedence over another aspect during weaving.
  - Aspect-interface: An aspect may implement one or more interfaces.
  - Aspect-class: An aspect may inherit (or extend) from a class, it may modify the static structure of an existing class, or it may modify the behavior of a class through one or more advices.

4.1. **Concepts represented.** In the following we describe the notations that we propose to be used to represent the AOP concepts in a class diagram:

- **Aspect**. An aspect is a stereotype of the UML *Class* model element, like in [1]. We use the stereotype <<aspect>> to distinguish between the aspect and base class (see Figure 1).
- **Pointcut**. A pointcut is a stereotype of the UML *Operation* model element, like in [19]. A pointcut is displayed as an operation with the << *pointcut* >> stereotype (Figure 1).
- **Advice**. An advice is a stereotype of the UML *Operation* model element. We define a new stereotype for each type of advice: << *before* >>, << *after* >> and << *around* >> (Figure 1).
- **Introduction**. Usually, there are two types of introductions that can be performed using AOP: addition of members (attributes or methods) to a class, and the modification of class hierarchy (inheriting from another class or implementing one or more interfaces). We propose the usage of our newly defined << *introduces* >> relationship in order

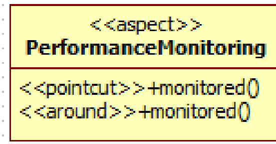

FIGURE 1. UML notation for aspect, pointcut and advice.

to specify that the introduction is done through the aspect. For each situation, the << *introduces* >> relationship is linked with different model elements:

- *Slice.* A *slice* is a stereotype of *Class* model element, and it contains the attributes and the methods added to a class through an aspect, like in AspectC++ [2]. The slice is linked with the class that will contain its members, and with the aspect, as shown in Figure 2. The end of the relationship corresponding to the class is a plus sign to represent that the slice members will be added to it.

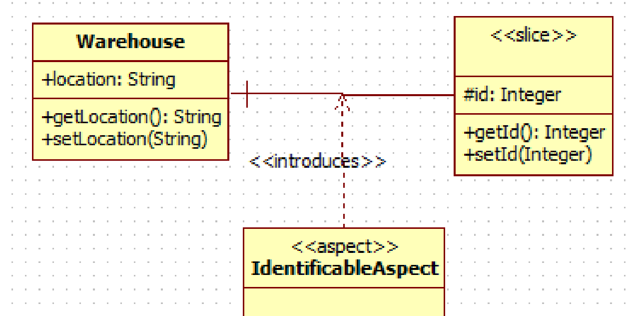

FIGURE 2. Members introduction with slice.

- *Generalization.* The << *introduces* >> relationship is linked with a generalization relationship between two classes to represent that the inheritance is introduced through the aspect at weave time (see Figure 3). If the aspect is removed from the system, the generalization relationship between the two classes will not exist anymore.
- *Interface realization.* The << *introduces* >> relationship is linked with an interface realization relationship between a class and an interface to represent that the realization is introduced through the aspect at weave time (see Figure 4). If the aspect

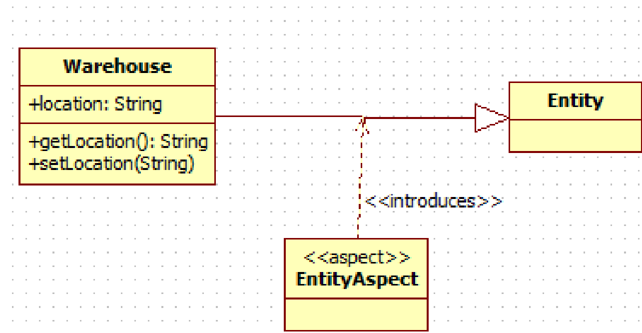

FIGURE 3. Class inheritance introduction.

is removed from the system, the interface realization relationship between the class and the interface will not be defined anymore.

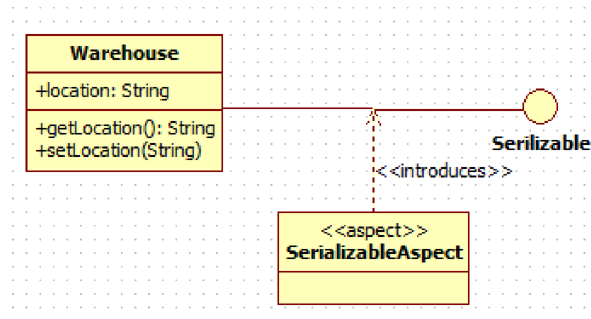

FIGURE 4. Interface implementation introduction.

- **Crosscutting**. A << *crosscut* >> relationship between an aspect and a class is defined, as in [1], to signify that the aspect will modify the behaviour of one or more source code parts from the class (usually methods) through an advice (before, after, or around) (see Figure 5).

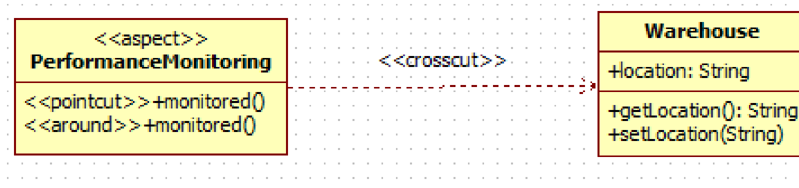

FIGURE 5. Crosscutting relationship

- **Aspect generalization**. An aspect may inherit from another abstract aspect (it may contain abstract pointcuts). A generalization relationship is defined between two aspects (as in Figure 6), with the constraint that the base aspect should be abstract.

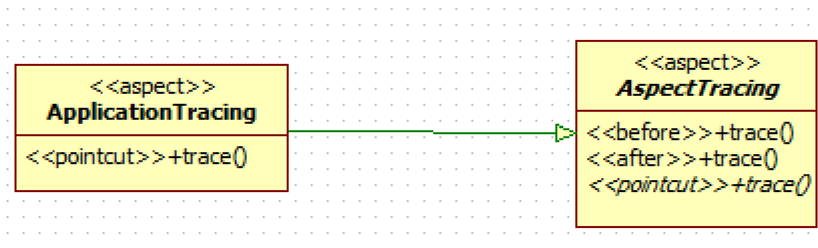

FIGURE 6. Aspect generalization

5. StarUML Plugin

StarUML [18] is a software modeling platform that supports UML (Unified Modeling Language). It is an open source software and it provides extensibility, and flexibility. It accepts UML 2.0 notation and it offers many types of diagrams, but you can also create your own type of diagram. StarUML has two important versions:

- StarUML1 which appeared around 1996, and since then suffered many changes until 2005. It was implemented in Delphi, C/C++, JavaScript, C# and has 11 types of diagrams, including: class diagram, use-case diagram, sequence diagram, etc. The 2005 version can still be used today.
- StarUML2, appeared at the beginning of 2015. There were no changes made on the first version from 2005 until 2015. Because this version of StarUML appeared only in 2015, its documentation is not complete, especially the part about extending its features.

We have developed a plugin for StarUML version 1 because it has a developers guide that allows us to make the extensions.

This plugin can be used for representing the set of notations described in Section 4.1: aspects, pointcuts and advice and the relationships between aspects, classes or interfaces. StarUML offers a special concept for creating user defined elements: Notation Extension Technology (NXT) that is a dialect of the Lisp programming language. The profile is implemented in XML.

Code generation. StarUML also offers the possibility of generating an XMI file, that is an XML-based type of file, associated to a diagram. Starting from this file, we can automatically generate the AspectJ (or other aspect oriented language) source code corresponding to the aspects from the diagram and their relationships (*introduces*, *aspect generalization*, etc.). We cannot generate code for all the relationships that can be defined in the diagram. For example, the *crosscut* relationship is too ambiguous to automatically generate code for it. The current version of our plugin can automatically generate source code only for the aspects included in the diagram. The code corresponding to other elements such as classes and interfaces is not generated.

In Listing 1 and Listing 2 are shown the AspectJ automatically generated code corresponding to the aspects from the diagram presented in Figure 6.

```
public abstract aspect AspectTracing{
  public abstract pointcut trace();
  before(): trace(){
  }
  after():trace(){
  }
}
```

Listing 1. AspectTracing generated code.

```
public aspect ApplicationTracing extends AspectTracing{
   public pointcut trace();
}
```

Listing 2. ApplicationTracing generated code.

6. Example

Hannemann and Kiczales [8] have studied the effects of using aspect oriented techniques in the structure of the design patterns introduced by Gamma et al. in [7], and implemented them in Java and AspectJ. Their results have shown that aspect oriented techniques improve the implementation of many patterns. In some cases the improvement is reflected in a new solution structure with fewer or different participants, in other cases, the structure remains the same, only the implementation changes. Patterns assign roles to their participants, for example *Subject* and *Observer* for the *Observer* pattern. These roles define the functionality of the participants in the pattern context. They found that patterns with crosscutting structure between roles and participant classes gain the most improvement. The improvement comes primarily from modularizing the implementation of the pattern. This is directly reflected in the implementation being textually localized. An integral part of achieving

this is to remove code-level dependencies from the participant classes to the implementation of the pattern.

In order to show the usage of the proposed notations and plugin, we use the Weather Monitoring application designed and implemented as an example for the *Observer* design pattern in *Head First Design Pattern* [6]. The Weather Monitoring application tracks from a weather station the current weather conditions (temperature, humidity, and barometric pressure), and provides three display options: current conditions, weather statistics and a simple forecast which are all updated in real time as the station acquires the last measurements. The design of the application without using AOP is shown in Figure 7. The weather conditions are kept in a `WeatherData` object and there are three different displays for each option: `CurrentConditions`, `StatisticsDisplay` and `ForecastDisplay`. The `WeatherData` is the subject and the three display options are the observers from the *Observer* pattern.

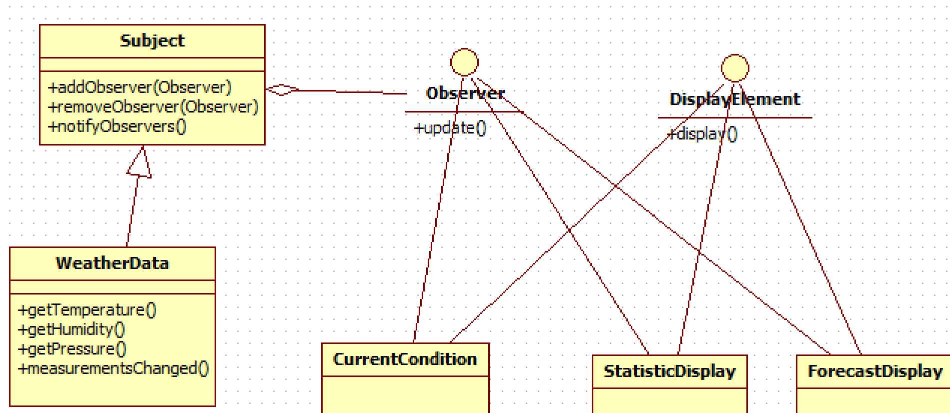

FIGURE 7. Weather Monitoring architecture with Observer pattern.

The AOP implementation of the *Observer* pattern is similar with the one described in [14] and removes the dependency between the concrete subject class and the abstract `Subject` class, and the dependency between the concrete observers and the `Observer` interface from the source code. The dependencies are made using introductions in a separate aspect. The new architecture of the application using AOP is shown in Figure 8 (the methods from the classes are not shown).

The diagram shown that the dependencies between `CurrentConditions`, `StatisticsDisplay` and `ForecastDisplay` and the `Observer` are introduced by the `ObserverAspect`. Without the aspect, these dependencies do not exist

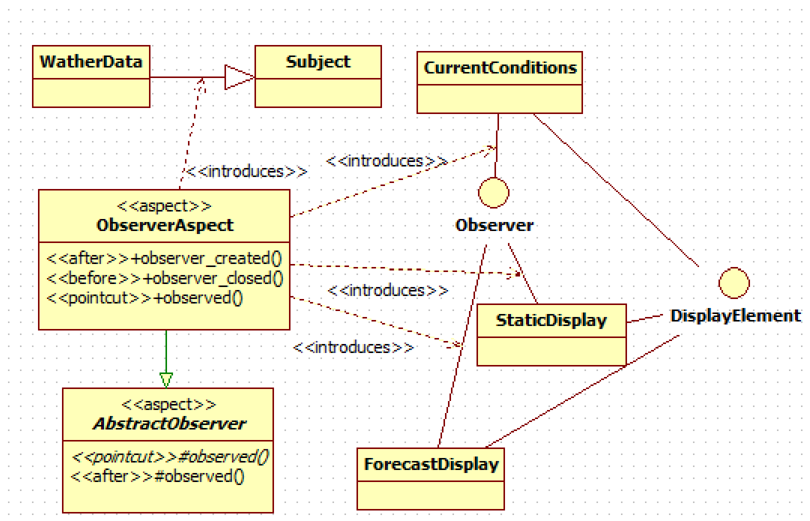

FIGURE 8. Weather Monitoring architecture with AOP based Observer.

anymore. The `WeatherData` generalization of the `Subject` class is also introduced by the aspect. The `ObserverAspect` also crosscuts the `WeatherData` class, as each time the `measurementsChanged` method is executed, an advice is executed that will automatically notify the observers.

All these changes to the static structure of the Weather Monitoring application can be vizualized from the AOP-enhaced class diagram.

7. CONCLUSIONS AND FURTHER WORK

We have proposed in this paper a set of notations to be used in order to model aspects in a UML class diagram. Some notations were previously introduced by other authors, and a few of them are newly defined (the *slice*, the *introduces* relationship, *aspect generalization*, etc.). These notations can ease the understanding of an AOP-based software system by graphically representing the parts that will be affected by introducing the aspects into the final software system. The notations were included in a StarUML plugin that we have developed in order to ease their usage and acceptance. We have also presented an example of how these notations can be used for modeling the AOP-based architecture of a small application.

Further work should be done in the following directions:

- To consider other AOP elements that should be represented in the diagram, like the precedence relationship between two aspects.

- To propose a more general version of the *crosscut* relationship, as for large software systems it may be difficult the vizualize all the classes that will be modified by the aspect. For *Logging* crosscutting concern which affects a big number of classes from the software system, the usage of the *crosscut* relationship may burden the understanding of the design.

References

[1] Omar Aldawud, Tzilla Elrad, and Atef Bader. A UML Profile for Aspect Oriented Modeling. In *Aspect Oriented Programming Workshop at OOPSLA 2001*, pages 1–6, 2001.
[2] AspectC++ Homepage. http://www.aspectc.org/.
[3] AspectJ Project. http://eclipse.org/aspectj/.
[4] Mark Basch and Arturo Sanchez. Incorporating aspects into the uml. In *Proceedings of the Aspect Oriented Modeling Workshop at AOSD*, 2003.
[5] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide (2nd Edition)*. Addison-Wesley Professional, 2005.
[6] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.
[7] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, March 1995.
[8] Jan Hannemann and Gregor Kiczales. 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*, pages 161–173, New York, NY, USA, 2002. ACM Press.
[9] J.L. Herrero, F. Sanchez, F. Lucio, and M. Torro. Introducing Separation of Aspects at Design Time. In *Aspect-Oriented Programming (AOP) Workshop at ECOOP 2000.* 2000.
[10] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases.* Addison Wesley, 2004.
[11] Mohamed M. Kande, Jorg Kienzle, and Alfred Strohmeier. From AOP to UML- A Bottom-Up Approach. In *Proceedings of the 1st International Workshop on Aspect-Oriented Modeling with UML*. Enschede, The Netherlands, 2002.
[12] Mohamed M. Kande, Jorg Kienzle, and Alfred Strohmeier. From AOP to UML: Towards an Aspect-Oriented Architectural Modeling Approach. Technical report, Swiss Federal Institute of Technology, Lausanne, Switzerland, 2002.
[13] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume LNCS 1241, pages 220–242. Springer-Verlag, 1997.
[14] Russell Miles. *AspectJ Cookbook*. O'Reilly, March 2004.
[15] David L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
[16] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. A uml notation for aspect-oriented software design. In

Proceedings of the Aspect Oriented Modeling with UML workshop at AOSD (2002), 2002.

[17] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[18] StarUML - The Open Source UML/MDA Platform. http://staruml.sourceforge.net/v1/about.php.

[19] Dominik Stein, Stefan Hanenberg, and Rainer Unland. An UML-based Aspect-Oriented Design Notation for AspectJ. In *AOSD 2002*, pages 1–7, 2002.

[20] Junichi Suzuki and Yoshikazu Yamamoto. Extending UML with Aspects: Aspect Support in the Design Phase. In *Aspect-Oriented Programming (AOP) Workshop at ECOOP'99*, pages 14–18. 1999.

[21] UML 2.4.1 Superstructure. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[22] Unified Modeling Language(UML). http://www.uml.org/.

[23] Aida Atef Zakaria, Hoda Hosny, and Amir Zeid. A uml extension for modeling aspect-oriented systems. In *Second International workshop on Aspect-Oriented Modeling with UML at UML 2002*. 2002.

[24] Gefei Zhang. Towards aspect-oriented class diagrams. In *Proceedings of 12th Asia-Pacific Software Engineering Conference*, pages 763–768, 2005.

Babeş-Bolyai University, Department of Computer Science, 1, M. Kogalniceanu Street, Cluj-Napoca, Romania

E-mail address: vrancianu_bristena@yahoo.com

E-mail address: grigo@cs.ubbcluj.ro