

Constraining Aspect Usage

Shmuel Tyszberowicz

School of Computer Science,
Tel-Aviv University and the
Academic College of Tel-Aviv
Yaffo

tyshbe@tau.ac.il

ABSTRACT

Aspect-Oriented Programming (AOP) is a promising approach for handling concerns in software-development that crosscut the conventional modular structure of the system. Such concerns are, for example, tracing, memory management, fault-tolerance, etc. Aspects are class-like elements in which developers encapsulate concerns that cut across classes, the natural unit of modularity in OOPs. Aspects are similar to classes: they have a type, state, and behavior, and they can have fields, methods, and types as members.

We have proposed a methodology for aspect development using the UML notation [8].

In this position paper we introduce ways to constrain the use of aspects using Aspect Constraint Language (ACL), an OCL like language. This allows project managers to control the development while avoiding undesired effects.

Research on AOP has mainly focused on the implementation phase of software development. Furthermore, researchers have not paid much attention to the way aspects themselves are to be developed, and have not attempted to separate the aspect development from that of the other parts of the system. According to our approach, an aspect is considered as a separate application that crosscuts the application under development (AUD), yet must not merge into it, nor change the AUD logic and structure, i.e. the aspect has to keep the AUD untouched and consistent. The aspect application has its own lifecycle, and a defined process should support aspect development.

Currently, the usage of aspects is uncontrolled, and no limits or constraints are implied during the aspect design and implementation phases. Not constraining the usage of aspects, weakens the added value of AOP. Instead of simplifying the development process, aspects might cause code tangling in various sources, spread functionality, and produce inconsistent and unpredictable behavior throughout project lifecycle. Therefore, in order to gain control, there is a need to impose constraints on aspect's usage during the analysis and design phases and to enforce the constraints during aspect implementation phase.

1 ASPECT ANALYSIS AND DESIGN

The development methodology presented here relates aspects to the *application under development* (AUD), using the following point of view. An aspect is a separate application that interfaces the AUD without merging into it, nor changing its logic and structure, keeping it untouched and consistent. Figure 1 illustrates an *aspect* using UML notation.

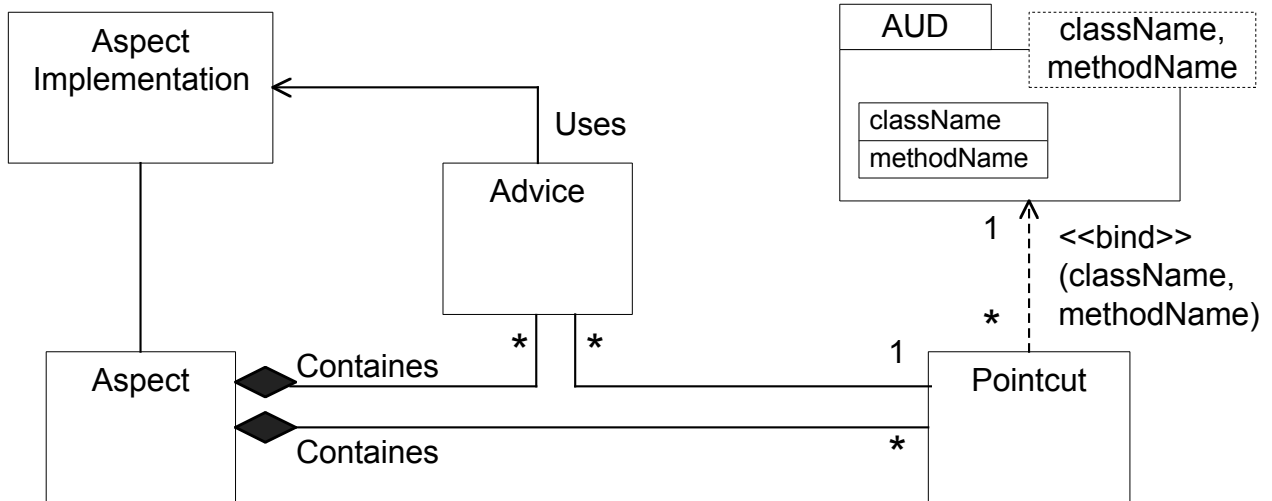


Figure 1: Aspects object diagram

The diagram contains:

- A UML package element that defines the *AUD* as a UML template.
- The parameters to the template are `<className, methodName>` placed in a dotted box.
- The parameters of templates must be bound to actual values. This is achieved by using the *bind* relationship which is represented by a dependency association with the keyword `<<bind>>`. A dashed arrow represents this relationship. The arguments are shown on the association arrow in parentheses after the keyword `<<bind>>`, i.e. `<<bind>> (className, methodName)`. When an aspect crosscuts many classes and methods, more than one pair of names (`className, methodName`) may replace the parameters. In order to keep the diagram simple and intelligible, the list of explicit names that replaces the parameters will be defined in a *bind* constraint in *Aspect Constraint Language* (ACL) rather than in the object diagram (see also Sections 3.2 and 4).
- A class named *aspect* represents the aspect definition which contains *pointcuts* and *advices*. Each *pointcut* can have zero or more *advices* (in AspectJ up to three *advices*: *before*, *after*, *around*). Each *advice* is associated with one *pointcut*.
- An *aspect implementation* class is associated with an *aspect* class. The *aspect* class contains the definition of the interface between the aspect and the AUD, whereas the *aspect implementation* is the actual realization of the aspect.

According to our approach, the interaction between an aspect and the AUD is defined by *where* and *when* the aspect runs. The object diagram of the aspect describes *where* it runs, in addition to describing the regular object model of the aspect implementation. A sequence diagram defines *when* the aspect runs: *before*, *after*, or *around* the *pointcut*. The UML template specification `<className, methodName>` is used in the sequence diagram in the same meaning as in the object diagram.

2 CONSTRAINING ASPECT USAGE

2.1 Rationale

The usage and implementation of aspects is totally unrestricted. The methods and variables of the AUD are vulnerably accessible by aspects. The execution and the static structure of the AUD might be badly modified by aspects at any point in runtime. For example, *Introduction* in AspectJ can change

the static structure of the AUD by adding fields and methods to classes. The examples in Section 1 highlight the problem of causing the logic of a class (*Point*) to be tangled in various places in the code. Multiple field definitions might produce a conflict. For example: assume a class *myClass* is defined in a package other than the aspect's. A field *myField* of type *myType* in class *myClass* is defined private in the aspect:

```
private myType otherPackage.myClass.myField;
```

This field is only accessible from the code inside of the aspect. Assume now that the same field is defined in the class *myClass* as public. A conflict is produced when the aspect accesses the field, since it is ambiguous whether the introduced *myField* or the public *myField* should be used.

AUD data structures can be modified by any aspect that sets new values into them (e.g. `appClass.Field=new_value`). Any method of any class can be called by an aspect and thus might change the AUD runtime data (e.g. `className.myMethod (type1 args1, type2 arg2)` where *myMethod* modifies the AUD data). An *around advice* can override any method functionality.

The above-mentioned issues might make the management of a project a difficult task. Unlimited aspect usage might ruin code reliability, stability, and logical continuation, and might produce conflicts. The rationale for restricting aspects usage is similar to the rationale for restricting access to memory (pointers) or restricting control flow expressions (*goto*) in Java: programs are easier to understand, maintain, and extend without the full power of the feature. We suggest constraints that form a development strategy that enables the avoidance of uncontrolled aspect usage.

2.2 Introducing Aspect Usage Constraints During the Analysis and Design

Defining *Aspect Usage Constraints* (AUC) is an additional stage required to complete the analysis and design of an aspect.

ACL offers the power to define constraints, and to enforce their use. Enforcement of constraints usage can be achieved during both compile and run time.

Following are five major aspect usage constraints. The arguments to the constraints are given in `<>`.

- The *const* constraint defines that the aspect is not allowed to modify the AUD run time data. An aspect that is defined as *const* can not do the following: a) Override a method by using *around advice*. b) Use the *introduction statement*. c) Modify any AUD data structure during runtime using a *set* command, and d) Call any method of the AUD since it might modify AUD run time data.

A *const* aspect can run at a join point and read the parameters. A *const* constraint is useful for aspects like logger, trace, etc.

- The *timeLimit* `<time>` constraint. At the aspect level it defines the upper limit of the execution time of the aspect application. At the *pointcut* level, i.e. when reaching a pointcut and executing its advice, it defines the upper limit of the *advice* execution time. This constraint is useful for real-time applications. The *Real* type `<time>` argument defines the upper limit in milliseconds.
- The *singleton* constraint declares that the aspect class has exactly one instance that cuts across the entire AUD. That instance is available at any time during AUD execution. Because an instance of the aspect exists at all join points in the running of the AUD, its advice will have a chance to run at all such join points. A singleton design pattern [3, Chapter 3 p. 127] “ensures a class only has one instance, and provide a global point of access to it”. Similar to the design pattern usage this constraint is useful to define the aspect class as a global point for central processing and analysis for the aspect. A singleton constraint is useful for aspects like performance analysis that collects information analyzes it and produces results.
- The *timing* constraint defines *when* the aspect starts its execution. The aspect can start *before*, *after*, or *parallel* to a *pointcut*. Executing before or after a *pointcut* corresponds to the *before advice* and the *after advice* respectively. A *parallel* aspect runs in a thread. The *parallel* characteristic is useful when the AUD is time critical and should not be influenced by the aspect processing time.

- The *bind* constraint describes *where* aspect runs. It specifies the list of classes and methods in the AUD which the aspect crosscuts. The bind replaces the template specification <className,methodName> that is used in the object diagram and in the sequence diagram . The constraint is defined at the *pointcut* level.

3 THE ASPECT CONSTRAINT LANGUAGE - ACL

In order to formulate the AUC, we need a language like *Object Constraint Language* (OCL). OCL is adapted by the UML standard. It is a language for specifying invariants, pre-conditions, post-conditions, and other kinds of constraints. “Being free of side effects is at the heart of OCL definition “ [5, Chapter 5 p.86]. Consequently the AUC can not be expressed by OCL because the constraints do affect the execution of both the AUD and the aspect.

Therefore the constraints should be expressed by another language. This position paper presents *Aspect Constraint Language* (ACL).

ACL is used during aspect analysis and design phases for several purposes:

- To express aspect usage constraints as described in Section 3.2.
- To express the bind relation that substitutes the template specification <className,methodName> by explicit names list, simplifying both the object diagram and the sequence diagram.
- To express the sequence diagram information. The sequence diagram illustrates when the aspect executes during the execution stage of the AUD. It is possible to show the information provided by a sequence diagram by using ACL expressions instead.

ACL uses syntax similar to OCL. This section describes the syntax for constraint definition. Constraints such as *const*, *singleton*, and *parallel* can be defined as aspect characteristics. Constraints such as *timeLimit*, *bind*, and *timing* are aspect attributes. We distinguish between *aspect* level constraints and *pointcut* level constraints. The context of an ACL expression is one of the following: a super type of aspects, an aspect, or a pointcut. This is elaborated on in the following subsections.

3.1 A Super Type Definition

An *aspect* is defined as a class in the object diagram, consequently the aspect is a *model element* in ACL terms. Every ACL model element has an *aclType*. A *supertypes* attribute declares the *aclType* of aspects. The aspects inherit the constraints that are defined in the context of the super type. A designer would define common characteristics of aspects in the super type. The ACL super type for aspects is named *AspectObj*. For example, if the designer restricts all aspects in the project to be *const*, the *const* constraint would be defined in the super type *AspectObj*. Then every aspect with a super type *AspectObj*, inherits the *const* constraint.

The following example defines a super type named *AspectObj* for aspects.

```
AspectObj
```

```
aclType = AspectObj
```

```
characteristics = aclIsConst
```

```
    // all inherited aspects are const
```

Assume we have a performance analysis aspect named *PerformanceAnalysis*, that inherited from *AspectObj*. The attribute *supertypes* is the set of all direct super types of the *aspect* class. The following ACL defines that the super type of the aspect performance analysis is *AspectObj*: `PerformanceAnalysis.supertypes = Set{AspectObj}`

3.2 Aspect Level Characteristics Constraints

The following example defines the performance analysis AUC.

PerformanceAnalysis

```
supertypes = Set{AspectObj}
    // The super type of the aspect is AspectObj.

characteristics = set {aclIsSingleton}
    // The aspect is a singleton, hence it can
    // finalize and
    // process the gathered performance analysis
    // data.

characteristics = set {aclIsParallel}
    // The aspect should run as a thread parallel to
    // the AUD.

timeLimit = 20
    // Specifies a time limit of 20 milliseconds on
    // the
    // aspect total processing time. The parameter
    // type is Real.
```

3.3 Pointcut Level Constraints

The *pointcut* level constraints are aspect usage constraints that are imposed on the *pointcut* and on the *advice* code that is performed when reaching the *join point* during execution. Such constraints are, for example, *bind*, *timing*, and *action*.

The *action* constraint is an additional constraint that defines two optional actions at the join point, *aclsCalled* and *aclsExecuted*. The *aclsCalled* defines that the aspect execution starts when a join point method is called, the aspect then runs in the execution context of the caller. The *aclsExecuted* defines that the aspect runs when the method of the join point is being executed. The aspect then runs in the execution context of this method.

The following ACL example describes constraints for two *pointcuts* in *PerformanceAnalysis* aspect. The constraints are *bind*, *action* and *timing*. The *pointcut* named *myPointcut* specifies the *bind* to the explicit names of classes and methods that gets performance analysis. The *pointcut* named *mainPointcut* specifies the runtime point to finalize performance data processing and analysis.

PerformanceAnalysis

```
pointcut=set{myPointcut, mainPointcut}
    // pointcuts of the aspect
```

The context of the pointcut *myPointcut* describes *bind* and *action* constraints. A performance analysis aspect measures the time performance for methods *call* and *pickup* in class *customer*, with any *signature* (specified by *(..)*).

myPointcut

```
action = aclIsExecuted
    // In the context of the called routine

bind = Set{customer.call(..), customer.pickup(..)}
    // Replace the template parameters with explicit
    // classes and
    // methods names, specifying method signature.

timeLimit = 3
    // Specifies the limit on this pointcut's
    // processing time in
    // milliseconds.
```

The Pointcut *mainPointcut* defines the runtime point to finalize the performance analysis and to print reports.

mainPointcut

```
bind = Set{main(..)}
      // The aspect runs for main
action = acliIsExecuted
      // The aspect runs in the context of main.
timing = set{acliIsAfter}
      // The aspect runs after main completes
      execution and
      // before exit.
```

4 ADAPTATION OF ACL BY UML

The stage of aspect usage constraints' definition is very important during the aspect analysis and design phases. This phase should be part of the aspect development methodology and even part of aspect pattern methodology. Like in design patterns, a catalog of patterns can be developed for aspects. An aspect pattern should define the aspect usage constraints. A developer then, could apply the pattern of aspect and its usage constraints, to creating a specific aspect application. Such reusable aspects could be for example: trace, fault tolerance, concurrency and synchronization, object interaction, error handling, logger etc.

Adapting ACL into UML, as well as OCL, provides an added value to software best practices [7], and enables integration with case tools and code generators.

Given aspect's object diagram, sequence diagram and ACL, code generator can produce mature code for the aspect definition file and for the aspect implementation class file. A project leader will specify ACL constraints on a project level or per aspect. Supported by a case tool that generates code, constraints' enforcement becomes a practice.

Examples and full documentation of the aspect analysis, design, and implementation phases can be found in [8].

REFERENCES

- [1] S. Clarke R.J. Walker, *Composition Patterns: An Approach to Designing Reusable Aspects*, Proceedings of ICSE 2001.
- [2] J. Suzuki and Y. Yamamoto, *Extending UML with Aspects: Aspect Support in design phase*, Proceedings of Aspect-Oriented Programming Workshop at ECOOP, 1999.
- [3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- [4] Xerox Corporation, *AspectJ 1.01 The AspectJ Programming Guide*, 2001, <http://aspectj.org/servlets/AJSite/>
- [5] J.B. Warmer and A.G Kleppe, *The Object Constraint Language. Precise Modeling with UML*, Addison-Wesley, 1999.
- [6] OMG, *The Unified Model Language Specification*, Version 1.4, September 2001, <http://www.omg.org/>
- [7] Software Program Managers Network, Secretary of Defense for Science and Technology, *16 Critical Software Practices*, <http://www.spmn.com/>.
- [8] Naomi Sapir, *Introducing Aspect Usage Constraints in Aspect Development Methodology*, Masters Thesis, Tel Aviv University, 2002.

- [9] Naomi Sapir, Shmuel Tyszberowicz, Amiram Yehudai, *Extending UML with Aspect Usage Constraints in the Analysis and Design Phases*, Aspect-Oriented Modeling with UML workshop, Dresden, Germany, September 2002.