Coordinated Visualization of Aspect-Oriented Programs

Álvaro F. d’Arce, Rogério E. Garcia, Ronaldo C. M. Correia

1Faculdade de Ciências e Tecnologia
Universidade Estadual Paulista “Júlio de Mesquita Filho” (UNESP)
Rua Roberto Simonsen, 305 – 19.060-900 – Presidente Prudente – SP – Brazil
alvaro@darce.com.br, {rogerio,ronaldo}@fct.unesp.br

Abstract. Software Visualization provides support to program analysis by visual presentation, improving cognition to comprehension tasks. Focusing specifically on aspect-oriented source code, it’s needed to map features of Aspect-Oriented Program (aspects and advices) into visual presentations, providing suitable visualization for understanding how aspects crosscut usual classes. In this paper we present a proposal of visual mapping for aspect-oriented programs aimed to support Program Comprehension by improving cognition.

1. Introduction

Aspect-Oriented Programming (AOP) supports the modularization of crosscutting concerns by mechanisms that make possible the addition of behavior to selected elements of the programming language semantics, thus isolating implementation that otherwise would be spread and tangled throughout the base code [Kiczales et al. 2001]. However, AOP new features can increase the structural complexity of software systems, increasing the difficulty to comprehend their architecture and structure, bringing up challenges to the comprehension of Aspect-Oriented Programs.

Software Visualization is an alternative approach to help software engineers to cope with structural complexity due to fragmentation and the need to compose fragments. However, the visualization must be able to deal with an Aspect-Oriented Program effectively. The visualization process deal with gathering data, its organization and mapping into visual structures to generate visual representations [Card et al. 1999]. Following such process, it is needed not only to have specific visual mapping for Aspect-Oriented Programs, but also have mechanisms to gather and to organize data representing AOP features.

Different techniques and tools have been proposed to support software visualization. For example, hierarchical structures can be visualized using Treemaps [Johnson and Shneiderman 1991, Bederson et al. 2002, Pfeifer and Gurd 2006], Polymetric Views [Lanza 2004, Carneiro et al. 2008], and Hyperbolic Trees [Munzner 1998] (this last one for large structures); Bars and Stripes might be used for inter-relational structures visualization [Baldwin et al. 2009]. Besides, some extensions have been proposed, like UML 3D to visualize packages, classes and methods [Gall and Lanza 2006] and Dependence Graphs to inter-dependency level visualization [Wurthinger et al. 2008] can also be used to build visual representations to support comprehension process. These visualization techniques can represent, by themselves, only the aspects before the weaving – they cannot represent the resulted code of an Aspect-Oriented Program after the weaving process (tangled and spread code). A Treemap extension was proposed to show
advised relations of AOP [Pfeifer and Gurd 2006], but it is a user based mining by searching regular expressions representing aspects and classes elements.

We have developed a visual mapping tackling Aspect-Oriented Program. To present our approach, the remainder of this paper is organized as follow. In Section 2 it’s presented some considerations about Aspect-Oriented Programs and features to its visualization. In Section 3 is presented the visual mapping for Aspect-Oriented Programs and the tool implemented. And in Section 4 the final remarks are presented.

2. Aspect-Oriented Program

AOP tackles crosscutting concerns problem supporting the definition units of implementation (aspects) that cut across the system units (base code), providing the behavior expected [Kiczales et al. 2001]. A generic AOP language should define elements enough to combine aspects and code base. The following elements were pointed out: a model to describe base code points where additional behavior may be defined, named joinpoints; a mechanism to identify these joinpoints; units that encapsulate both joinpoint specifications and behavior enhancements; and a process to combine both base code and aspects, named the weaving process [Elrad et al. 2001]. Aspects and how they crosscuts a base code are important elements that should be mapped into visual structures.

There are several programming languages extensions to support AOP, one of them is AspectJ: an extension of the Java language to support AOP. In AspectJ are defined new constructions to aspect, advices, and pointcuts. Aspects are units that combine joinpoint specifications and pieces of advice. Also, aspects can declare members – attributes and methods – to be owned by other types, what is called inter-type declarations. Advices are pieces of code joined to specified points in the program. Before, after and around advices are method-like constructs that can be executed before, after and in place of the joinpoints, respectively. The AOP runtime invokes advice automatically when the pointcut matches the joinpoint defining the crosscutting behavior.

Listing 1. Piece of source code aspect-oriented

```java
public class Account {
    private float funds;
    public Account(float funds) { this.funds = funds; }
    public float getFunds() { return funds; }
    public void withdraw(float value) { funds -= value; }
    public void deposit(float value) { fundso += value; }
}

public aspect Register {
    pointcut withdraw(Account acc, float value):
        target(acc) && args(value) && call(* Account.withdraw(float));
    pointcut deposit(Account acc, float value):
        target(acc) && args(value) && call(* Account.deposit(float));
    after(Account Account, float value) returning(): withdraw(Account, value) {
        System.out.println("Withdraw - Value: " + value);
    }
    after(Account Account, float value) returning(): deposit(Account, value) {
        System.out.println("Deposit - Value: " + value);
    }
}
```

In Listing 1 is shown pieces of source code of an Aspect-Oriented Program. In this example is shown a bank account class and an aspect modifying it by two pointcuts
and two *advices* to register the transaction after calls to \texttt{withdraw()} and \texttt{deposit()} methods. The AspectJ *advice* weaver statically transforms the program so that at runtime it behaves according to the language semantics, as show in Figure 1.

![Figure 1. Execution model after weaving](image)

To visualize Aspect-Oriented Programs, specifically AspectJ programs, it’s interesting that a Software Visualization tool highlights (in more than one visual scenario) program characteristics like *aspects*, *advices* and *pointcuts*, with the aim to let the user view how an *aspect* crosscuts one or more classes or how much a class is modified by one or more *aspects*, or how the code is tangled and spread by the weaver.

After the weaving and compilation processes, the specific data of an Aspect-Oriented Program are tangled and spread in bytecode, identified by specific signatures. By these signatures, it’s possible to obtain data about program *aspects* and *advices* (two important items in an Aspect-Oriented Program visualization) and organize them in a graph, along with the other program elements (like packages, classes and methods) to obtain data enough to generate the visual representations.

### 3. Aspect-Oriented Program Visualization

We proposed a multiple view approach to visualize AOP, as depicted in Figure 2. *Structural Presentation* aims to show how the source code is organized in classes and *aspects*; *Inter-Units Presentation* aims to show crosscutting between method and *advices*; and *Intra-Method Presentation* aims to show the behavior intra-method after weaving process. One might observe in Figure 2 that visualizations are coordinated to allow exploring different levels of view.

![Figure 2. Presentations Schema](image)

For each presentation was chosen suitable technique according to its purpose: for example, there are several techniques for hierarchical views, and we have chosen Treemap to represent the program’s hierarchical structure including *aspects* and *advices* (*Structural Presentation*). Hyperbolic view was chosen to represents classes dependences coupled with *aspects* crosscuts (*Inter-Units Presentation*), and Control Flow Graph (CFG) to visualize the advised source code representing the behavior of a piece of code, showing *advices* over the code after the weaving process (*Intra-Method Presentation*). By the coordination of these three visual presentations, Aspect-Oriented Program can be visualized.
in hierarchical structure, classes dependence, aspects crosscuts, tangled code and spread code. A Software Visualization tool has been developed according to the visual mapping proposed, and it is shown in the following.

3.1. SoftVizOAH: The Visualization Tool

The tool SoftVizOAH consists in a standalone desktop software, and its architecture, shown in Figure 3, is organized onto three layers: Dataset, Control and Visualization. In addition, the tool allows visualizing test case results as described in the following. The Dataset layer is aimed to static and dynamic analyses creating data sets necessary to generate the visualizations. The Reading Module reads a program’s bytecode and analyzes classes, aspects and test cases. The Static Analysis reads classes, aspects and test cases (using JUnit). During the reading, the data flow stream of each unit is analyzed and put into a data structure representing a CFG, while new instructions are inserted in each unit code (instrumentation technique, containing methods and advices calls, providing feedback to allow monitoring test cases by Dynamic Analysis), and it’s built a list with all classes, aspects and their information obtained by the instrumentation. From this list, it’s built a hierarchical structure, which will be used to generate the Treemap visual presentation. Also, informations like superclass references, variable types, methods parameters and returns, and others are grouped into a node table and an edge table to mount a Dependence Graph (by the interpretation of these tables and the organization of the links among the nodes), which is used to generate the Hyperbolic visual representation. While doing a Structural Test, Dynamic Analysis monitors test cases and registers each visited portion of code of each code unit, making a execution path. From the path, the coverage criteria are verified to determine the test’s Result – each criterion has its own standard to verify whether the test has been completely or partially met. During a test execution, the instrumentation sends to the tool information about each visited portion of code. This information is stored to be used by each visual representation. By these two steps analyses, the tool gathers data to generate visual representations [Martins 2007, Trevisan 2010, Dutra 2010].

At Control layer the Integration Module organizes data obtained from Reading Module to generate visual presentations (all three presentations are generated at the same time) and provides mechanisms to coordinate them. These mechanisms capture interaction with visual presentations to reflect them into another view. An event, caused by an user interaction in a specific visualization, is captured. From this event, the Integration Module obtains data about the selected unit of the program (class, method, aspect or advice). The data structures stored in the Dataset layer (i.e., the graph data structure, the list containing information from the instrumentation, the hierarchical structure and the node and edge tables) are used to do the coordination among the visualizations. Then, an event is sent to each other visualization in the Visualization layer, informing the new state of the visual representation (i.e., highlighted items), performing, this way, the coordination.

The Visualization layer provides CFG, Treemap and Hyperbolic visual presentations, as explained in Section 3. In each visual presentation, aspects are highlighted. In the CFG visual representation, basic code blocks are represented by rectangles outlined by a continuous line, code blocks advised by aspects are represented by rectangles outlined by a dashed line and method returns are represented by rectangles outlined by a double line. Inside each rectangle there is a number indicating the code block execution
sequence. Normal code sequences are represented by continuous lines, and exceptions are represented by dashed lines. All code blocks belong to codes inside methods and aspects. If a test case (created by the user) is applied, each rectangle is colored according to a gradient from red to green. The more the code block has failed in the test, the redder it’s colored. The more the code block has succeeded, the greener it is colored. Such color mapping allows viewing how succeeded (or failed) a test case covered the source code.

Also, Treemap presents hierarchical structures in constrained and nested rectangles. Each rectangle can represent packages, classes, methods, aspects and advices. A rectangle size represents the number calls to the represented method or advice, and the color represents test case results, just like the referred gradient. In the Hyperbolic visual presentation, nodes represent whole program classes and aspects, and edges represent dependency between each class or aspect, i.e. method calls and joinpoints. Each edge is colored according to their structural tests results. Clicking on an edge, the participant methods in its test are shown in Treemap. Clicking on a node, it is positioned in the center of the graph and its representation is highlighted in Treemap. All visual presentations use the same color mapping, but to highlight aspect selection, a different color is used.

Listing 2. Method calling two “advised” methods

```java
public void transferring (int sourceNumber, int destinationNumber, float value) {
    try {
        account sourceAccount = getaccount(sourceNumber);
        account destinationAccount = getaccount(destinationNumber);
        if (sourceAccount.getBalance() >= value) {
            sourceAccount.withdraw(value); // ADVICED METHOD
            destinationAccount.deposit(value); // ADVICED METHOD
        } else {
            throw new InsufficientFundsException(sourceAccount.getNumber(), value);
        }
    } catch (AbsentAccountException e) {
        System.out.println(e.getMessage());
    } catch (InsufficientFundsException e) {
        System.out.println(e.getMessage());
    }
}
```
Figure 4. **CFG and Interaction with Treemap projection**
In Figures 4(d) and 4(a) are shown the CFG and the Treemap views from transfer-Test test case, which calls transferring method – the transferring method shown in Listing 2 calls withdraw and deposit methods, which are “advised” as depicted in Figure 1. Each node in CFG represent a code block executed, and their connection represents possible path execution. All code blocks, advices and methods are colored according to their test results, using the same color mapping to CFG and the Treemap. When code block is selected in the graph visualization, its respective method is selected in the Treemap visualization, and when code block selected contains an advice, the respective advice is also selected in the Treemap visualization. The selected rectangle in Figure 4(d) represent a code block containing the calling to withdraw method and its advice, and both method and advice are highlighted in the Treemap visualization. The graph visualization is showing two advices (represented by rectangles with dashed lines) with their respective pointcuts (represented by their sequence in the graph) crosscutting a test case with the Account class. The global view of the graph visualization represents the tangled and spread code. In the example depicted in Figure 4(d), the code is tangled (advices among the code) and the aspect is thinly spread (advices are close to each other).

Nodes in Hyperbolic view represent classes and aspects, distinguished by different colors. Edges represent method calling or aspect crosscutting. By interacting with Hyperbolic view, one might select classes, aspects and edges between them. When a class or aspect is selected in the Hyperbolic visualization, the respective class or aspect is also selected in the Treemap visualization, and its respective methods or advices are shown in a separated panel, depicted in Figure 4(b). By selecting an edge in the Hyperbolic visualization, the participating methods or advices are also selected in the Treemap visualization. In Figure 4 is shown the selected class Terminal – moved to center of projection –, its selection in Treemap visualization, and the list of methods. By interacting with those visual presentations, one might observe how aspects crosscut classes.

4. Final Remarks

We proposed a visual mapping to present features of Aspect-Oriented Programs using three coordinated visual presentations. Those visualizations aim to externalize structural organization, the relations among classes and aspects, and the advised code. By coordination proposed it is possible to highlight selected elements on different detail levels, allowing the user to gather information about aspect and its spreading to the source code.

The visual mapping proposed has been evaluated using some AOP source code. One of them is the GanttProject\(^1\), originally written in Java that we modified to an AspectJ project by factoring some functionalities to aspects (to visualize them in the tool), resulting in 51305 lines of code at all. According to preliminary evaluation, the tool functionalities are useful to support Aspect-Oriented Program understanding. However, to assess the effectiveness and efficiency of the visual mapping proposed in more realistic way, it is needed to conduct a controlled experiment, which will be conducted using an experimentation process. Such controled experiment will be conduct aimed to reveal defects previously inserted into the GanttProject source code.

\(^1\)An open source project obtained at [http://www.ganttproject.biz/](http://www.ganttproject.biz/)
References


