AspectMaps: Extending Moose to visualize AOP software

Johan Fabry, Andy Kellens, Simon Denier, Stéphane Ducasse

Science of Computer Programming 79 (2014) 6–22

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

AspectMaps: Extending Moose to visualize AOP software

Johan Fabry [a,*], Andy Kellens [b], Simon Denier [c], Stéphane Ducasse [c]

[a] PLEIAD Laboratory, Computer Science Department (DCC), University of Chile, Chile[1]
[b] Software Languages Lab, Vrije Universiteit Brussel, Belgium[2]
[c] RMoD Team, INRIA Lille - Nord Europe, France[3]

ARTICLE INFO

Article history:
Received 24 January 2011
Received in revised form 13 November 2011
Accepted 23 February 2012
Available online 10 March 2012

Keywords:
Aspect-oriented programming
Visualization
Moose

ABSTRACT

When using aspect-oriented programming the application implicitly invokes the functionality contained in the aspects. Consequently program comprehension of such a software is more intricate. To alleviate this difficulty we developed the AspectMaps visualization and tool. AspectMaps extends the Moose program comprehension and reverse engineering platform with support for aspects, and is implemented using facilities provided by Moose. In this paper we present the AspectMaps tool, and show how it can be used by performing an exploration of a fairly large aspect-oriented application. We then show how we extended the FAMIX meta-model family that underpins Moose to also provide support for aspects. This extension is called ASPIX, and thanks to this enhancement Moose can now also treat aspect-oriented software. Finally, we report on our experiences using some of the tools in Moose; Mondrian to implement the visualization, and Glamour to build the user interface. We discuss how we were able to implement a sizable visualization tool using them and how we were able to deal with some of their limitations.

Note: This paper uses colors extensively. Please use a color version to better understand the ideas presented here.

1. Introduction

Aspect-oriented programming [18] is a novel programming paradigm that aims at modularizing what are called *crosscutting concerns*. Due to the "tyranny of the dominant decomposition" [27], particular concerns, *e.g.*, logging, tracing, transaction management, and so on, cannot be captured in individual modules using classical software engineering techniques. Instead their implementation is scattered throughout various modules, hence crosscutting the implementation of the entire system.

To tackle this problem, aspect-orientation introduces the concept of an *aspect*: a novel kind of module that captures such crosscutting concerns. Key to the notion of aspects is the fact that they introduce an *implicit invocation* mechanism. Rather than invoking the crosscutting behavior *explicitly* from within the code of a system, this base code no longer contains these calls. To allow implicit invocation, conceptually, each execution step of the application is reified as a *join point*, *e.g.*, a method call, an assignment, … An aspect then defines the behavior to be executed, an *advice*, to occur on certain join points. These are identified by defining a *pointcut*: a predicate over join points. In other words, while an advice[4] implements the actual

* Corresponding author.
E-mail address: jfabry@dcc.uchile.cl (J. Fabry).

1 http://pleiad.cl.
2 http://soft.vub.ac.be.
3 http://rmod.lille.inria.fr.
4 In the AOSD literature the term 'advice' is used instead of 'a piece of advice' and the plural of 'advice' remains 'advice'.

0167-6423/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2012.02.007

crosscutting behavior, pointcuts select at which points in the execution of the system the advice should be executed. Finally, the static location in the source code that gives rise to a join point at execution time is called a *join point shadow*. In summary, the following process (conceptually) takes place for each execution step of the software. When an expression in the source code, *i.e.*, a join point shadow, is executed, it causes a join point to be emitted. All pointcuts of the system are automatically called with this join point as argument, and for the pointcuts that return true the advice associated to this pointcut is run.

Although aspects open up new possibilities in terms of modularization of crosscutting concerns, the presence of this implicit invocation mechanism can make it more difficult for a developer to understand the behavior of the system. In literature, various problems have been documented that are caused by this implicit invocation mechanism. One such problem is the *fragile pointcut problem* [17,19]. Due to the limited expressiveness of pointcut languages, such pointcut expressions are often tightly coupled to the structure of the base code. As a consequence, seemingly safe changes to this base code can have an unexpected and erroneous impact on the behavior of the aspects that are defined in the system. Similar problems can occur when multiple aspects intervene at a single join point. As it is not immediately clear which aspects intervene at a single point in the execution, and as these aspects potentially interact with each other, this can lead to erratic behavior [26].

To alleviate these problems and aid developers to understand such software we have proposed the *AspectMaps* visualization [14]. AspectMaps aims to ease code comprehension of software using aspects by offering a scalable visualization of implicit invocation. The AspectMaps visualization is also implemented as a tool, on top of the Moose [9] reverse engineering platform, leveraging the facilities offered by this platform.

This paper builds upon our previous work [14], and features two novel contributions: On the one hand it presents a detailed discussion of the AspectMaps visualization tool, showing how it can be used to explore aspect-oriented code of a fairly large application, yielding some interesting observations. On the other hand it provides a discussion of the implementation of the tool. In this part we first introduce our extension to Moose that allows it to also operate on aspect-oriented code. This extension is called ASPIX; it augments the FAMIX [28] meta-model that underlies the Moose platform such that its instances can model aspect-oriented software. Second, we discuss our experiences of using the tools offered by Moose for building visualizations: Mondrian [24], and for (specific kinds of) user interfaces: Glamour [5]. We report how we were able to use these tools to build a reasonably large visualization tool, while avoiding some of their drawbacks. For an in-depth discussion regarding the concepts underlying our visualization, the pros and cons of the visualization and an initial empirical validation of our approach we refer the interested reader to [14].

The intended audience of this paper is twofold, echoing the two main contributions of the tool. On the one hand we have the developers of aspect-oriented software that want to use AspectMaps to gain a better understanding of the software on which they are working. On the other hand we have developers using Moose that may want to build their own analysis tool for aspect-oriented software, or that may gain from our lessons learned building a large analysis tool using the features of Moose.

This paper is structured as follows: We start with giving an introduction to the AspectMaps visualization, and in Section 3 we present the tool, followed by an example case study showing how it can be used. Section 4 introduces ASPIX, our extension to the FAMIX meta-model that allows Moose to be used to analyze aspect-oriented software. This is followed in Section 5 by a report of our experiences on using Moose to build the AspectMaps tool. Section 6 gives an overview of related work, and Section 7 discusses future work and concludes.

2. The AspectMaps visualization

AspectMaps is a visualization that offers users a detailed overview of implicit invocation. It visualizes where aspects are specified to apply in a system, based on visualizing join point shadows, and how aspects possibly interact at each join point shadow. Moreover this is performed in a scalable way, thanks to a multilevel selective structural zoom. Note that this paper does not focus on the visualization itself: a discussion of the design rationale of our visualization, its merits and limitation, and a small user study can be found in [14]. We therefore here only give a summary of the AspectMaps visualization, adapted from [14], before discussing the tool.

For the sake of this discussion, we define that an aspect applies at a certain source code element (a package, class, or method) if for at least one pointcut that is associated with an advice of that aspect, at least one of its join point shadows belong to that element.

AspectMaps supports the traditional pointcut–advice model of aspects applied to object-oriented class-based languages. The join points we visualize are method calls and method executions. Advice can execute before, around or after a join point, and we distinguish between after returning and after exception throwing. Aspects may contain various advice, and an execution order may be specified between aspects. The above effectively allows us to visualize a subset of AspectJ [18] and Java code (which is arguably the most popular aspect language combination). In this case we ignore inter-type declarations as well as advice that applies to fields. The AspectMaps tool is however not fundamentally restricted to the AspectJ/Java combination and more detail on this is given in Section 4.

The key feature of AspectMaps is having the ability to selectively zoom in on the source code at different levels of granularity. Zooming in from a coarser level to a more fine-grained level reveals more detail. The behavior is analogous to street map applications, *e.g.*, Google Maps, hence the name AspectMaps.

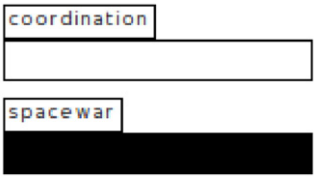

Fig. 1. Compact visualization of packages: coordination and spacewar.

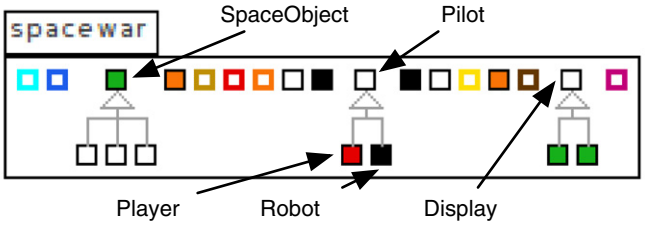

Fig. 2. Extended visualization of the spacewar package (annotated with selected class names).

To detail the AspectMaps visualization, the remainder of this section is structured following its different levels of granularity. For each level we show how it is visualized, and illustrate the tool using a number of examples. Specifically, for Sections 2.1 and 2.2 we use Spacewar, an example program that is provided with the AspectJ Development Toolkit (AJDT) [7] and is also included as example in the AspectMaps tool. Spacewar defines multiple aspects. For this discussion we visualize the Coordination aspect in green and the EnsureShipIsAlive aspect in red. In Section 2.3, we use an additional artificially created example, as Spacewar does not suffice to show all the features of AspectMaps.

Additional material is available at the AspectMaps website http://pleiad.cl/aspectmaps *e.g.*, featuring some screencasts of the tool in action.

2.1. Package level

When opening the tool, AspectMaps provides an overview of all packages in the visualized system. Each package is represented by a *compact* visualization that shows the name of the package and the aspects that apply to this package. The color of the rectangle of each package corresponds to the color of the aspect that applies, if that aspect is currently enabled for visualization. In the case that multiple aspects apply to a single package, this is indicated by the color black (which never can serve as the color of an aspect). This package visualization is shown in Fig. 1. This figure shows two packages, namely coordination and spacewar. As multiple aspects intervene in the spacewar package, it is indicated in black. Package contents is not shown at this moment.

A user of AspectMaps can obtain an *extended* visualization, revealing the package contents, by zooming in on a selected package (clicking in a package). Fig. 2 shows this extended visualization applied to the spacewar package. Inside of the package, all different classes and aspects belonging to that package are shown. This same extended visualization is shown in a pop-up when the user of the tool hovers the mouse pointer over a package that is represented in the compact visualization.

Our extended package visualization is inspired by the work of Lanza et al. on Polymetric Views [21]. We have extended this work to also support the visualization of aspects. Key to Polymetric Views is that the dimensions of the entities reflect particular properties of these entities (e.g., LOC, number of methods). At this level of detail, our visualization shows the following information:

- **Classes:** rectangles with black borders. Inheritance relations are visualized using UML notation.
- **Aspects:** rectangles with thick colored borders. The border color is the color for the aspect (never black).
- **Where aspects apply:** class rectangles have the color of the aspect that applies, black for multiple aspects.
- **Class and aspect metrics:** the user selects which dimension reflects which metric, or none. For the figures in this section we select no metric, for clarity of the discussion.

Similar to the original Polymetric Views visualization, AspectMaps does not display the names of the classes, in order to prevent cluttering the visualization. These names are instead revealed at the class level, where classes and aspects show their extended visualization. This visualization is shown as a pop-up when the mouse hovers over a class or aspect, or when the user performs a zoom operation on a class or aspect by clicking on it.

2.2. Class level

Classes are represented in our visualization using rectangles. At the top of the rectangle, and separated from the rest of the rectangle by means of a horizontal line, the name of the class is shown. Below the horizontal line we represent the instance

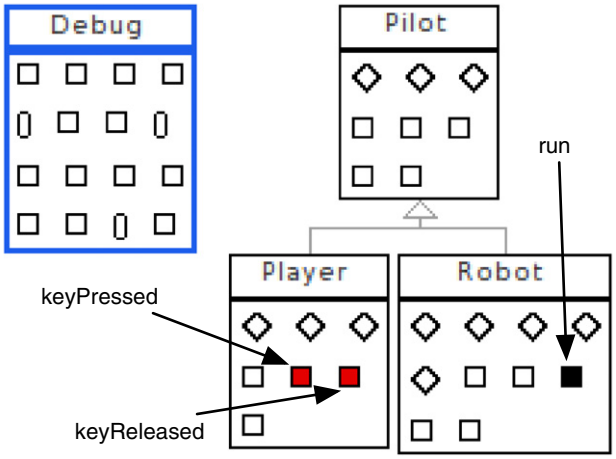

Fig. 3. The Debug aspect and the Pilot hierarchy (annotated with selected method names).

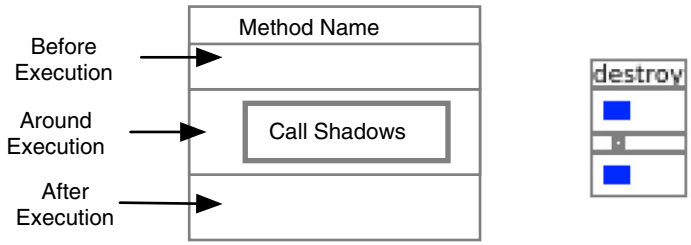

Fig. 4. Template for visualization of execution shadow points (left), and an example destroy method (right). Fig. 7 shows call visualization.

variables (as diamonds) and methods (smaller rectangles) of the class. The height and width of methods can be determined by a metric. The methods are colored based on the applicable aspects, taking into account that multiple aspects intervening at a single entity are colored black. By means of a pop-up on the class name, the user can get additional information of the class (its superclass, the names and types of the instance variables).

Aspects are visualized in a similar fashion, where the name of the aspect is indicated above the horizontal line. Below the line we can find the advice (shown as rectangles) and pointcuts (shown as ovals). The dimensions of the advice can also be customized by a user-selected metric. Pop-ups for the aspect show the superaspect; for pointcuts they show the name of the pointcut. Pop-ups for advice show their signature and line number in the aspect, as an advice signature does not uniquely identify an advice.

Fig. 3 shows our visualization zoomed in on all the classes in the hierarchy of Pilot, and also shows the Debug aspect. The visualization shows that there are multiple aspects applying to method run defined on the class Game. Furthermore, we see that the aspect EnsureShipIsAlive applies to methods keyPressed and keyReleased.

2.3. Method level

The method level is the finest level of detail offered by AspectMaps. At this level, our visualization shows how aspects intervene at the shadow points within the method. Note that multiple aspects can intervene at a single join point, and that advice can be declared to execute before, after or around the join point. To correctly reflect the semantics of such a join point model, our visualization of shadow points must represent this information accordingly.

Fig. 4 (left) shows the various regions of our visualization of methods. From top to bottom, our visualization shows the name of the method followed by separate boxes containing before, after and around advice applied to execution shadow points. Advice applicable to call shadow points is displayed within a separate box (also see Section 2.3.2). On the right of the figure, our visualization is illustrated on a method named destroy. We can see that the blue aspect applies before and after the execution of this method.

By means of a pop-up on the method name, users of the tool can access the parameter list of the method.

2.3.1. Advice execution, Run-time tests, Ordering

For each applicable advice, our visualization draws a small figure representing the shadow point in the corresponding division of our method-level visualization. The color of this rectangle corresponds to the color of the aspect. Within a single

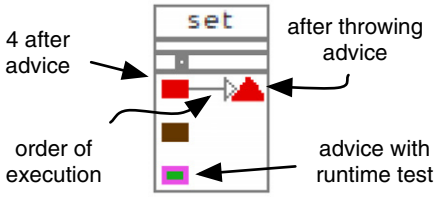

Fig. 5. Four after execution advice of a set method.

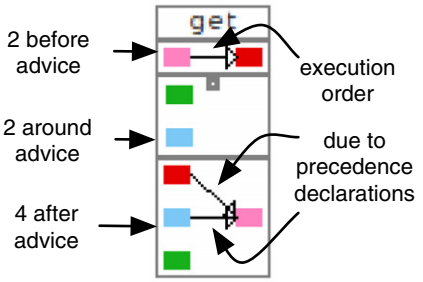

Fig. 6. Eight execution advice for a get method, with two precedence declarations yielding three ordering arrows.

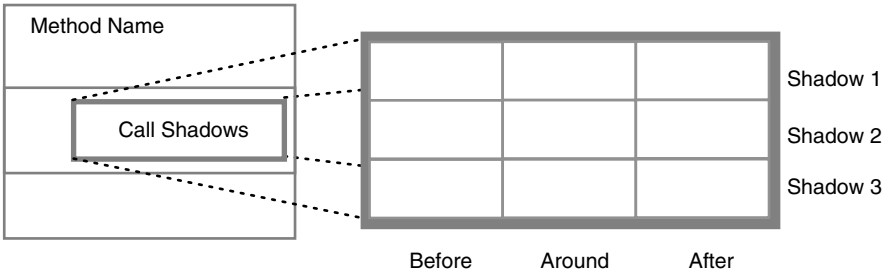

Fig. 7. Template for call shadow points.

division of our visualization, these rectangles are stacked vertically. As an illustration of this, consider Fig. 5 that shows the after execution advice of an example set method.

AspectMaps makes the distinction between after throwing advice (shown using triangles) and all other kinds of advice (represented by rectangles), in order to emphasize the special nature of after throwing advice: it executes when the method terminates by throwing an exception. If there is a run-time test involved in evaluating the pointcut for an advice execution (*e.g.*, an if-test or a control-flow pointcut) the figure has a thick border in a contrasting color (for example, the green aspect in Fig. 5).

Note that it is possible for multiple advice of the *same* aspect to apply at a single shadow point. This is represented in AspectMaps by visualizing a figure for each application of the advice. The order in which these advice will be executed is indicated by means of a gray arrow. For example, in Fig. 5 the regular after advice of the red aspect occurs before the after throwing advice. When multiple advice of *different aspects* apply, the order of their application may be specified by the programmer. If such an order is specified, AspectMaps indicates this order by means of a black arrow. An example of this is shown in Fig. 6. Considering the after advice, the cyan and red code is run before the pink advice. There is no ordering specified between the green aspect and any of the other aspects, nor between the cyan and red aspects, hence no arrows are drawn. Fig. 5 does not show any black arrows, indicating no ordering is specified between the green, brown and red aspects and therefore no claims can be made about the order at which the aspects will be executed at run-time.

2.3.2. Call shadow points

The body of a method may contain multiple call shadow points, sequentially ordered by the source code of the method. We visualize advice execution in this same order, aligning them vertically. The visualization of call shadow points uses the same visualization as execution shadow points. It however orders the before, around and after divisions horizontally instead of vertically. A template of this is shown in Fig. 7.

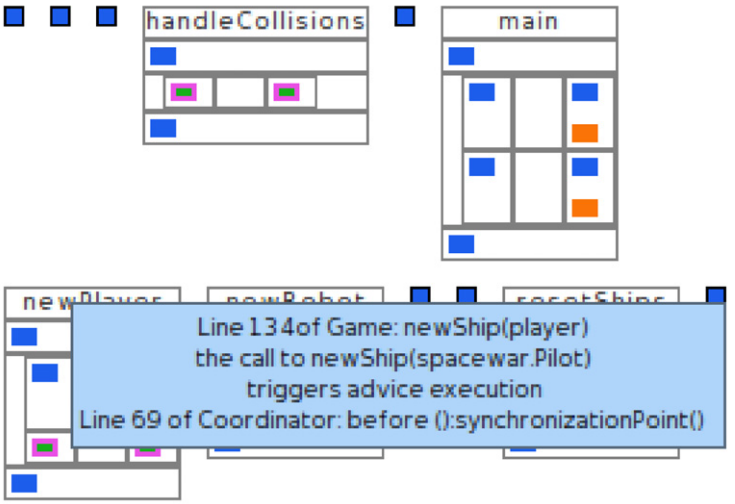

Fig. 8. A selection of methods in Spacewar showing call shadow points, as well as a pop-up of one advice execution (in the newPlayer method).

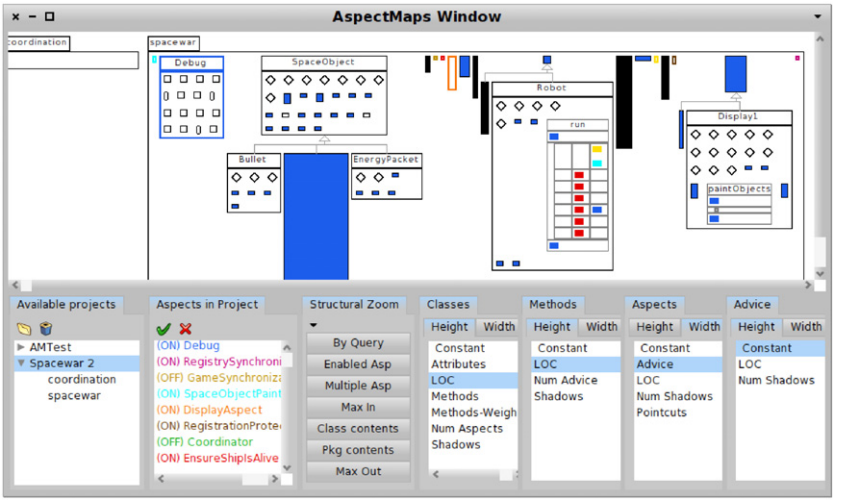

Fig. 9. The AspectMaps tool showing a part of Spacewar. Above is the visualization, below is project selection, visualization of aspect selection, zoom shortcut buttons and metrics selection. Nonvisible metrics choices: class width is Attributes, aspect height is LOC (lines of code).

An example of the visualization of call shadows – applied to a number of methods of the Spacewar example – can be found in Fig. 8. This figure also demonstrates the pop-up information for each advice execution that contains the signature of the advice, the line number of the aspect in the source code and the source code of the shadow point.

3. Using AspectMaps

The AspectMaps visualization has been implemented in Pharo, a version of Smalltalk, using Moose [9], a platform for software analysis and reverse engineering. The resulting tool is also called AspectMaps. It is a standalone tool that in its current incarnation can visualize AspectJ/Java code developed with the Eclipse AspectJ Development Toolkit [7].

We introduce the tool here and how it is linked to Eclipse before describing a software exploration session using the tool. The implementation of the AspectMaps tool is discussed in Sections 4 and 5.

3.1. The AspectMaps tool user interface

The AspectMaps tool consists of a single window, shown in Fig. 9. The top part of the window shows the AspectMaps visualization, and the bottom panel contains multiple controls that determine what is visualized and how. We present these controls next, followed by a description of the context menu of the visualization.

Available projects. AspectMaps allows for multiple Java projects to be visualized. In the figure these are AMTest and Spacewar 2. Selecting a project shows the visualization in the top part of the window, and populates the *Aspects in Project* list. The hierarchy of packages in a project is also available as a flattened list, by clicking on the expansion triangle shown before the project name. In the figure this is shown for Spacewar 2. Selecting a package from this list restricts the visualization to only show this package.

As the AspectMaps tool is not integrated in an existing Java/AspectJ development environment, the code that is visualized needs to be imported into the tool, which adds it to the list. Importing is required because AspectMaps does not operate on the original source code, but builds its internal model in order to visualize it. (This process is discussed in Sections 4.1 and 4.2.) A project can be removed from the AspectMaps tool repository, no longer making it available for visualization, by clicking on the trash icon. Clicking on the folder icon launches an import wizard that will add the project to the model repository. Importing is a two-step process: first the aspectual information of the project is obtained by asking for an .xcr file of the project. Second the OO structure of the project is retrieved, either by indicating the source code directory of the project or by providing a .mse Moose exchange file for the project. How to generate both .xcr and .mse file is discussed in Section 3.2.

Aspects in project. Two settings for the aspects in the project that is currently being treated are specified here. Firstly, each aspect name is shown in the color that represents it in the visualization. A context menu on the entries in this list allows their color to be changed by selecting it from a list of colors. Secondly, visualization of join point shadows of each aspect can be turned on or off. For example, in Fig. 9, visualization of Debug is turned on, while for Coordinator it is off. The context menu on the entries in the list allows this visualization state to be toggled. Clicking the check icon turns on visualization of all aspects in the list, except for the selected aspect, if any. Clicking the cross icon performs the inverse.

Structural zoom. The structural zoom buttons perform seven different zoom operations on the entities being visualized. In order these are:

By Query: Given a query, which may contain wildcards, zooms in maximally on entities of which their names match.
Enabled Asp: Zooms in maximally on all join point shadows of aspects that are turned on for visualization.
Multiple Asp: Zooms in maximally on all join point shadows where multiple aspects that are turned on are present.
Max In: Maximal zoom level on all entities in the visualization, *i.e.*, all entities show their extended representation.
Class contents: Level showing contents up to classes and aspects, all methods showing their compact representation.
Pkg contents: Zoom level where the packages show their contents, and lower levels show the compact representation.
Max out: Zoom level where all entities show their compact representation.

Classes, Methods, Aspects, Advice. These four tabs allow for a specification of the dimensions of the rectangles that represent these entities, when drawn in the compact representation. This is taken from Polymetric Views [21], as discussed in Section 2.1. By default the value here is constant, yielding small squares for each entity. Selecting a metric for a dimension, *e.g.*, LOC for the height of classes in Fig. 9, specifies that the size of this dimension is equal to the value of this metric for the entity that the rectangle represents. Further considering the classes in the example, the width of classes is set to the Attributes metric, indicating the number of attributes. The tall class rectangles hence are large classes, and the wide classes have a large number of instance variables. Note that metrics only apply on the compact representation, as in the expended representation the size of an entity rectangle may need to expand to show its contents.

Context menu of the visualization. The last control of the visualization is the context menu that is available on selected entities that are being visualized. We have the following context menu items:

On pointcuts: Revealing all join point shadows of this pointcut by zooming in on the methods containing them.
On advices: Revealing all join point shadows of the pointcut of this advice.
On advice execution: Revealing the aspect of the advice and revealing all the executions of this advice.

3.2. Generating import files for AspectMaps

As said previously, to visualize a project it first needs to be imported into AspectMaps. This import process takes on the one hand the OO structure of the application, *i.e.*, what is defined by the Java code, and on the other hand the aspectual information, *i.e.*, what is defined by the AspectJ code. The import process itself is discussed in more detail in Sections 4.1 and 4.2, we describe here how to obtain the files required to perform an import.

To import the OO structure of the project the AspectMaps tool requires a .mse file that contains the relevant data of the project being imported. As .mse is the standard file format for Moose data interchange, there are multiple tools available that produce a .mse file from a Java source directory. One option is the inFamix tool by intooitus,[5] a second possibility to

[5] http://www.intooitus.com/products/infamix.

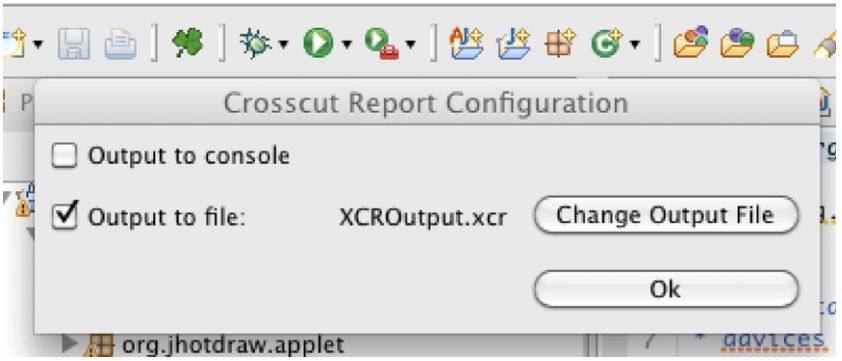

Fig. 10. The crosscut report plugin toolbar button (the shamrock) opens the configuration dialog.

generate such files is to use VerveineJ.[6] AspectMaps has support for both tools (limited to UNIX/MacOSX): it can use them to generate the .mse file on the fly when importing, thus streamlining the process.

A notable drawback of using Java source code to generate the .mse is that the code being examined must be correct and fully compliant to the Java standard. This is relevant as the AspectJ compiler, used in Eclipse AJDT, also accepts .java files where aspects, pointcuts and advice are defined. Therefore some projects that compile successfully with the AJDT cannot be imported into AspectMaps. To solve this, the developer must ensure that all aspectual code is defined in AspectJ .aj files, **not** in .java files.

The aspectual information of a project is generated by an Eclipse plug-in we implemented. The plugin, called XCut report, is triggered by a successful AspectJ compilation. After an AspectJ project is compiled, it produces the relevant aspectual information. The plugin has a straightforward configuration dialog box, shown in Fig. 10. Output can be turned off (the default state, which effectively disables the plugin), shown on the Eclipse console or written to an .xcr file (or both of the latter). The .xcr file format and the XCut report plugin are discussed in more detail in Section 4.2.1.

3.3. Example case: AJHotDraw

To show how the AspectMaps tool can be used to explore a given application, we now present a study of AJHotdraw [8] using AspectMaps. We chose this application as it is a widely known example of an aspect-oriented application. AJHotdraw is a fairly large application, consisting of 374 classes and 31 aspects, which serves to illustrate the scalability of our tool. A full investigation of the AJHotdraw code is outside of the scope of this paper. We restrict our discussion here to showing how the tool was used to reach some notable conclusions from our exploration.

- A first assessment of the location of classes and aspects, using the Pkg contents button shows that all the aspects are within the ccconcerns package hierarchy.
- Looking at the names of aspects in the Aspects in Project list, we see that of the 31 aspects, at least 15 are dedicated to undo operations.
- Selecting the ccconcerns.tools.undo package in the Available projects (restricting the visualization to only show that package) shows that it contains five undo aspects. Performing a Max in zoom operation reveals that none of these contain a pointcut nor an advice. (Source code inspection reveals that these aspects only perform inter-type declarations, which are not visualized in AspectMaps.)
- Investigating the contents of the other ccconcerns package in the same way we find that the other 10 undo operations are located in the package ccconcerns.commands.undo. These have a name that ends in Undo, except for UndoRedoActivity. All of the former only have one pointcut. Using the context menu item that reveals the shadows of the pointcut we learn that each of these aspects solely affects the execute method of a class with the corresponding name, without the Undo suffix. All of the affected classes are subclasses of AbstractCommand. This is shown by the inheritance edge for subclasses in the same package, and also by the class name pop-up for subclasses in a different package than AbstractCommand.
- Using the Num Shadows metric we see that one of the two most important aspects is CommandContracts. Enabling visualization of this aspect only using the toolbar icon and performing a Enabled Asp zoom, shows that it affects the execute method that is present in almost all classes of the AbstractCommand hierarchy. Manual zooming and exploration of the classes in the hierarchy that are not affected reveal that these do not contain an execute method. According to the same metric, the other important aspect is UndoableCommand. Investigating this aspect in an analogous fashion, we see it only affects the AbstractCommand class and 6 more classes, each of which is a subclass of a Swing component: 4 menu-related classes, and the button and combo-box classes.

[6] http://www.moosetechnology.org/tools/verveinej.

In addition to a study of AJHotdraw, AspectMaps has been used to explore Spacewar [14]. We also have performed a user study that reveals that on an exploration of Spacewar, AspectMaps far outperforms the visualization of AJDT [14], which is arguably the most competitive software visualization for aspect-oriented software. Again, a discussion of these results is outside of the scope of this paper, as here we focus on the tool, and we refer to the published article [14] for more information.

4. Adding aspect support to Moose

The AspectMaps tool is built on top of Moose [9], a platform for software analysis and reverse engineering. Moose offers various facilities for building software engineering tools, ranging from importing data, source-code modeling and querying, to building software visualizations. In the past, Moose has been successfully leveraged for building a wide variety of tools, such as Torch [29] (a tool supporting source-code change integration), SmallDude [4] (a duplication detector), and eDSM [22] (a tool to detect cyclic dependencies). Within the context of this work, we opted to base our tool on Moose for the following reasons:

- **FAMIX.** FAMIX [28] is a language-independent meta-model for class-based programming languages that underpins the Moose tool suite. Using this model allows AspectMaps to achieve a certain level of base and aspect language independence. A downside is that FAMIX however does not provide concepts for representing aspect-oriented constructs.
- **Mondrian.** Mondrian [24] is a domain-specific language for scripting visualizations that is part of the Moose tool suite. It is particularly well-suited for implementing AspectMaps as it does not only offer constructs for declaring the different nodes and edges that make up a visualization, but also provides a means to create highly-interactive visualizations.
- **Glamour.** Similar to Mondrian, Glamour [5] is a domain-specific language for scripting browsers. It provides a convenient way for declaring and composing the user-interface components of AspectMaps.
- **Integration with other tools.** By implementing AspectMaps on top of the Moose platform, we can also leverage the other tools that are part of the Moose platform (to calculate metrics, …).

As said above, Moose does not provide support for analyzing aspect-oriented software as the FAMIX meta-model family lacks the required concepts. To address this shortcoming, we provided an extension to FAMIX that adds this support, called ASPIX. ASPIX (ASPects In famiX) is a generic class-based object-oriented meta-model to which the essential aspect-oriented concepts have been added. The ASPIX model is generic such that it can be used by all of the Moose tools, *i.e.*, enabling Moose to be used to analyze aspect-oriented software. Moreover, basing AspectMaps on ASPIX allows our tool to be used for other combinations of base languages and aspect languages. This as long as they follow the, arguably prevailing, model of aspects being composed of pointcut and advice.

In this section we provide a description of ASPIX and detail how model instances can be created, *i.e.*, how aspect-oriented code is represented in our Moose extension and how this code can be imported for analysis.

4.1. ASPIX : ASPects In famiX

To perform visualization, AspectMaps does not consider the actual source code of the program, but instead uses its own model, as do the other tools of Moose. This model is an instance of the ASPIX meta-model, a member of the FAMIX meta-model family [9,12,28], which we created such that Moose is able to work with AOP code. ASPIX (ASPects In famiX) consists of a generic class-based object-oriented meta-model enriched with information of aspects, pointcuts, advice, advice ordering and shadow points. Analogously to FAMIX, the idea is to have language-specific importers that generate model instances. These can obtain the required information from the source code. AspectMaps currently only has one importer for a base and aspect language combination: Java and AspectJ. In this section we first give an overview of FAMIX, before we detail our aspect-oriented extension ASPIX. The next section details how instances of the model are built based on the data made available by the importer.

The core of FAMIX is a language independent meta-model that describes the static structure of object-oriented software systems. The FAMIX core can describe the structure of object-oriented systems from the namespace level down to variable accesses and method invocations. Fig. 11 shows relevant parts of FAMIX core (in gray) that are directly involved in AspectMaps. Namespace is the entity with the highest granularity: it is used to represent Java packages in the case of Java systems. Type is the meta-description for entities containing methods in FAMIX (as indicated by its methods attribute) and Class is a specialization of Type. BehaviouralEntity is a generic meta-description for behaviors, such as its specialization Method. It can contain outgoingInvocations (for example, method calls) which themselves are described by Invocation.

ASPIX extends the FAMIX core with meta-descriptions specific to the structure of an aspect-oriented system. Fig. 11 shows (in black) the meta-descriptions, attributes, and inheritance relationships which constitute ASPIX as an extension to FAMIX. Hence, Aspect is a specialization of Type and contains namedPointcuts and advices. An Advice has a kind (before, after, or around), a precedenceOver relationship to indicate precedence of the advice over other advices, and a relation to its pointcut (which can be a named pointcut of the aspect or its own anonymous pointcut). Pointcut holds a boolean flag indicating whether evaluating it requires a runtime test. It also points to joinpointShadows captured by this pointcut. The meta-description JoinPointShadow holds the opposite relation in its matchingPointcuts attribute. In the current version of ASPIX, two kinds of join point shadows are described: ExecutionJoinPoint which links to a method and CallJoinPoint which

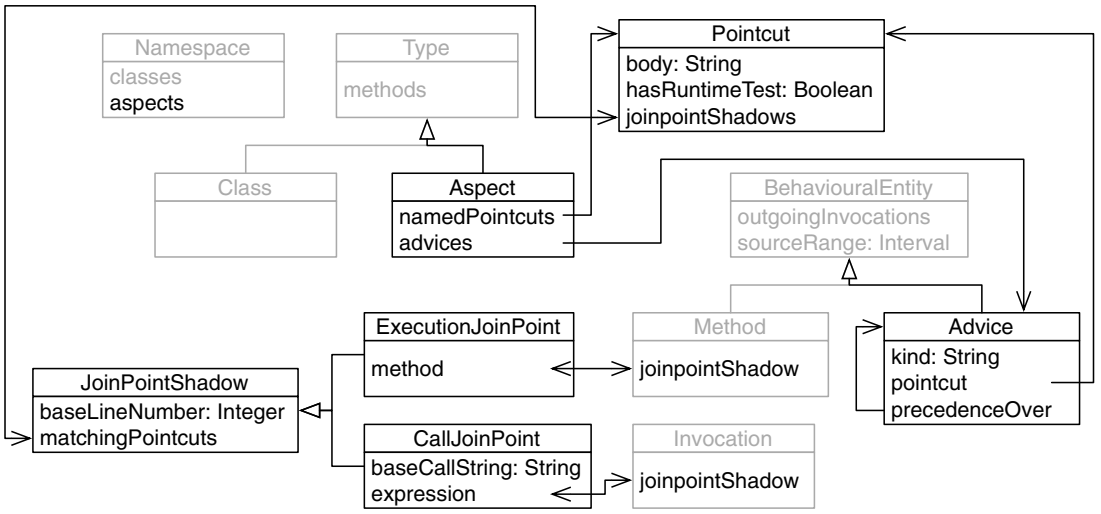

Fig. 11. Class diagram of the ASPIX meta-model, with relevant parts of the FAMIX core meta-model shown in gray.

links to an expression (an instance of Invocation). Note that both Method and Invocation are extended by ASPIX to link back to their respective joinpointShadow, while Namespace provides a link to its contained aspects.

4.2. Constructing an ASPIX model

ASPIX models for Java and AspectJ code are created using a two-step process as outlined in Section 3.2. First the Java and AspectJ code are analyzed and the relevant data is exported to an exchange format: an .mse file for Java and an .xcr file for AspectJ. Second the ASPIX model is created based on joining the information in these two files. The rationale for this two-step process using exchange formats is twofold. On the one hand we maintain a loose coupling between the different parts of the loading operation. On the other hand, in the implementation we are able to reuse as much existing infrastructure as possible.

The former is important as it allows for language independence of the import process by using standardized interchange formats. To allow Moose to work with different base and aspect language combinations, *e.g.*, to let AspectMaps to visualize these, it simply suffices to be able to export this code to a combination of .mse and .xcr file.

To clarify the creation of both these files and how these are joined to form an ASPIX model, we now discuss both steps of the process in more detail.

4.2.1. Exporting Java and AspectJ code to .mse and .xcr files

We have chosen to use the standard Moose .mse exchange format [20] to import Java code into Moose, yielding FAMIX model instances. This as it allows us to reuse existing Java to .mse export tools, as well as the standard Moose file import feature. As said in Section 3.2, when loading in software to be visualized the AspectMaps tool can automatically generate a .mse file using the inFusion tool and an alternative way to generate them is to use VerveineJ.

To the best of our knowledge, there is however no tool that takes AspectJ code and provides structural information of aspects and join point shadow information. Neither are we aware of any standard file format for this information. We therefore built our own export tool and defined a straightforward file format. As mentioned in Section 3.2, the tool we built is an Eclipse plugin, called XCut Report. When active, it uses AJDT [7] to produce an .xcr file whenever AspectJ code is compiled. XCut Report hooks into AJDT to receive a notification whenever compilation ends successfully. This notification includes a complete structural model of the program that was compiled. The plugin then traverses this structural model, writing the relevant data to the corresponding .xcr file.

The structure of an .xcr file is a self-describing list of tuples, where each tuple is tab-delimited and separated from the next tuple by a newline. Tuples are grouped according to the nature of the element they describe and groups are separated by an empty line. The first two lines of a tuple group describe the group as follows: a title line describes the nature of the group and a description line states the structure of the tuples, *i.e.*, the semantics of each field. This file format was chosen because it is straightforward to parse when importing, and moreover it can be easily read when opened by a spreadsheet application (by importing as tab-separated text).

For example, in Fig. 12 we show part of the tuple group of join point shadows of the Spacewar example (with each tuple wrapped to fit the size of the page). The example shows that for join point shadows, tuples contain the name of and line number in the advised class, the type of join point, the name of the aspect and line number of the advice, as well as whether the shadow point has runtime tests. For call join points, the signature of the call as well as the source code including the call (Call source code) are also reported.

```
Shadows
Base FQN   Base Line Number   Type of Pointcut   Aspect FQN   Pointcut with Test  Advice Line Number   Base Call String//
   Call target method   END
[...]
spacewar.Registry   64   execution   spacewar.Debug   false   131   -   -   END
spacewar.SpaceObject   97   call   spacewar.Debug   false   152   getGame().getRegistry().unregister(this)//
   void spacewar.Registry.unregister(spacewar.SpaceObject)   END
spacewar.SpaceObject   97   call   coordination.Coordinator   true   69   getGame().getRegistry().unregister(this)//
   void spacewar.Registry.unregister(spacewar.SpaceObject)   END
[...]
```

Fig. 12. Part of the .xcr file of Spacewar, showing join point shadows. // indicates the tuple continues on the next line and [...] omitted tuples.

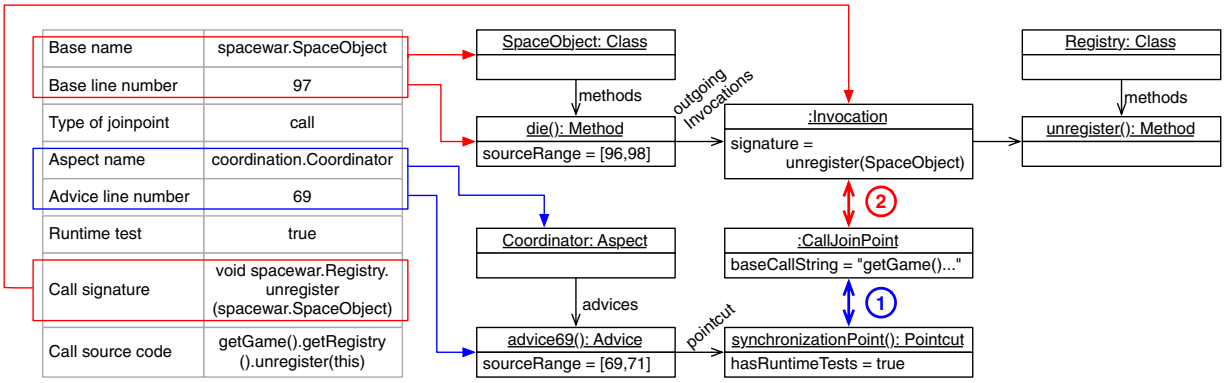

Fig. 13. Binding an ASPIX join point shadow (left-hand side table) to a pointcut and an invocation (right-hand side UML object diagram).

Special to the .xcr file format is its reliance on line numbers to identify different elements. This is for two separate reasons. Firstly, recall that an advice signature does not uniquely identify an advice. Therefore we use the line number where the advice definition starts as a unique identifier for the advice. Secondly, the file does not contain the complete structure of the program, but solely a partial specification of aspect-related elements. This requires it to refer unambiguously to elements outside the file: the methods where join point shadows are located. To do so, for each join point shadow we include the fully qualified name of the advised class and the line number of the shadow. As we show next, both these uses of line numbers are sufficient to allow us to construct the complete ASPIX model from .mse and .xcr files.

4.2.2. Importing .mse and .xcr files to an ASPIX model

As the .mse file format is a standard Moose format, Moose straightforwardly creates a FAMIX model instance from a .mse file. Therefore we solely discuss here how processing the .xcr file complements this existing model instance, producing an ASPIX model of the software that is to be visualized.

When the data in the .xcr file is imported in Moose, first the corresponding ASPIX entities are created using the retrieved information. The second part of the import process merges the ASPIX entities into the original FAMIX model and binds entities together: an ASPIX Aspect points to its superaspect, an Advice has its precedence information linked, and most importantly JoinPointShadow entities are bound both to their matching pointcuts and to the FAMIX entity which they represent (either a Method or Invocation)

Binding a JoinPointShadow is somewhat more intricate. Fig. 13 shows which information is used to bind an ASPIX JoinPointShadow instance with respectively 1) a pointcut (blue binding) and 2) the advised entity (here an invocation with the red binding). The concrete example used is the last tuple of Fig. 12.

To bind the pointcut with the join point shadow, we use first the aspect name *coordination. Coordinator* to retrieve the aspect entity and second, the line number 69 to identify the corresponding *advice69* entity. (Recall that we use the line number where the advice definition starts as a unique identifier for the advice in ASPIX.) This allows us to link back a shadow point to its advices as above. We navigate to the advice pointcut (*synchronizationPoint*) and add the shadow point to the list of joinpointShadows in the pointcut, and reciprocally add the pointcut to the list of matchingPointcuts in the shadow point.

The process is a bit more complex to bind the shadow point to the advised entity. We use the base name *spacewar.SpaceObject* to retrieve the entity of the advised class. Then the line number 97 allows us to detect the method *die()* where the join point shadow resides, as the range of source code for *die()* starts at 96 and ends at 98. If it is an execution join point, the process stops here and the binding is made between the shadow point entity and the method entity. However, if it is a call join point, we iterate over the outgoingInvocations of the *die()* method to detect one matching the call signature *void spacewar.Registry.unregister (spacewar.SpaceObject)* of the shadow point. The shadow point entity and the invocation entity are then bound together.

Other shadow point information is used to fill-in the model: *Type of join point* indicates whether to instantiate a CallJoinPoint or an ExecutionJoinPoint, *Runtime test* is used to set the flag Pointcut.hasRuntimeTests in the corresponding pointcut, and *Call source code* is stored into the field CallJoinPoint.baseCallString.

5. Experiences using Moose to build AspectMaps

AspectMaps is arguably one of the most complex software visualization tools built using Moose. We experienced this while building the tool, encountering that we were stretching the different Moose tools to their limits. We report here on our experiences such that subsequent implementations of complex visualization tools may also benefit from them.

Firstly, we discuss the Mondrian [23] visualization, reporting on how we avoided the drawbacks of monolithic scripts, introducing the implementation of the structural zoom feature, and presenting how the visualization is kept separate from the ASPIX model. Second we present how we were able to use Glamour [5] to build the user interface of AspectMaps, overcoming the restricted focus of Glamour to building one specific type of user interfaces.

5.1. Visualizing the model with Mondrian

AspectMaps uses the Mondrian [23] information visualization engine, also a part of Moose, to render its diagrams. To draw diagrams in Mondrian the programmer declaratively specifies a graph of nodes, a layout algorithm for these nodes and (optionally) edges to be drawn between nodes. Such specifications are written in Smalltalk, and called Mondrian scripts. In a script for each node the shape to be used in the visualization can be specified, along with a context menu, pop-up and mouse-over actions, if any.

In our experience, Mondrian is a good solution to build software visualizations as it offers sufficient functionality, allows for rapid prototyping and with subsequent refinements can yield the finished visualization. Moreover it is integrated with the other tools in Moose which enables rapid construction of the complete tool, *e.g.*, adding browser-like user interfaces using Glamour [5]. In building AspectMaps we found that we were taxing the Mondrian visualization engine, pushing its limits. We found various bugs, *e.g.*, popups not working correctly, and performance issues, *e.g.*, slow scrolling. The current maintainer of Mondrian was very responsive in addressing all the issues we raised, even adding the node replace feature we needed to implement the structural zoom. In our experience this responsiveness was key in realizing AspectMaps, without it the visualization and tool would probably not have been realized.

To the best of our knowledge, the AspectMaps visualization is one of the largest (if not the largest) Mondrian script in existence. We discuss salient points of the script here to provide pointers to other Mondrian users that need to write large scripts or big visualizations. We first describe how we structured the Mondrian script according to the existing graph of nodes contained in the ASPIX model, second how the node replace feature added to Mondrian allows for the structural zoom, and third how the combination of class extensions, FAMIX properties and a trait installer make the Mondrian script for AspectMaps a modular extension to the ASPIX model.

5.1.1. Structuring the Mondrian script

Early prototypes of what would become AspectMaps used monolithic scripts to visualize an ASPIX data structure. These scripts however turned out to relatively quickly become unmanageable. Therefore when setting out to implement the complete functionality of AspectMaps we decided early-on for a modular approach. In our experience, the resulting modularity of the script, as described below, has made it significantly easier to develop and maintain than if it would be a monolithic entity.

AspectMaps visualizes an ASPIX data structure, which is a tree containing packages, classes, aspects, methods, pointcuts and advice. When declaring what is to be visualized we traverse this graph, passing the respective elements to the Mondrian engine along with their visualization specification. We chose not to write one large script that defines the traversal and visualization of each entity. Instead we let each entity itself specify how it is visualized and delegate visualization of its contained entities. This is a natural mapping, and moreover is in accordance with good OO practice. The AspectMaps Mondrian script, *i.e.*, the logic of the visualization is therefore defined as methods in the FAMIX and ASPIX entities Namespace, Aspect, Class, Method, Advice and Pointcut. Consequently we can consider the script itself as a composition of modules, each module a Smalltalk method.

We found that this modularization led to the appearance of several cross-cutting concerns in the script. Some examples are: specifying a context menu for each entity, declaring that entities may not be dragged around the canvas, and the core functionality of the structural zoom feature (discussed in more detail in Section 5.1.2). A straightforward way to address such code duplication is to extract the duplicated code to a method in the superclass. The different methods that comprise the script are however contained in classes that do not have a common superclass. The extraction process resulted in this behavior to be placed in a common ancestor that is relatively far up in the inheritance hierarchy. We do not consider this a clean solution. Arguably the biggest drawback of this is that it adds this behavior to other entities where it does not belong. We have therefore chosen to refactor this common behavior to traits [11]. Put briefly, traits are partial class descriptions, only containing the definition of methods. These can be applied to classes, in effect adding the behavior defined in the trait to the class. Using traits allowed us to effectively group methods that implement the cross-cutting functionality in one

module, and selectively compose these modules where required. Moreover we have made good use of the feature of traits that methods defined in the class take priority over likewise-named methods in a trait. We defined generic behavior in the trait, *e.g.*, for drawing the contents of a compact representation of an entity, and let the different entities override this if required, *e.g.*, Aspect has a specific compact representation (not showing anything in its contents).

5.1.2. Node contents replacement for structural zooming

Arguably a key feature of AspectMaps is its selective structural zoom. This zoom allows a user to selectively drill down to parts of the application of interest. As a result, the visualization is more scalable than if it would use a uniform zoom level. Since the selective structural zoom can also benefit other visualizations, we describe its implementation here.

To enable the structural zoom feature Aspect, Class and Method have both a compact and an extended representation, as discussed in 2. As a general rule, a compact representation gives an overview of the aspects that apply within the element, while the extended representation visualizes the elements at the next level of granularity, each child element according to their own representation. This means that a zoom in operation on these entities consists of specifying that it should show its extended representation while a zoom out operation specifies that the compact representation should be shown.

Each Namespace, Aspect, Class and Method possesses a shape attribute that maintains whether they should be visualized in the compact or the extended representation. To draw itself, each entity first declares how to visualize its perimeter. For example, a class specifies that it should be drawn as a black box with minimum height and width according to the metrics selected by the user. This is followed by stating the inner visualization of the entity, according to what is stipulated by the shape attribute. In general, the compact visualization consists of filling the perimeter with the color of the aspect that applies in that entity, or the multiple aspect color if more than one aspect applies. The extended visualization collects all the elements at the next level of granularity, *e.g.*, getting the methods and instance variables of a class, and instructs these entities to visualize themselves within the perimeter. Note that as a consequence of this implementation, a zoom out of a root element, *e.g.*, a Namespace, does not affect the zoomed state of its children as their shape attributes are not modified. When zooming in again the view is restored to what it was previously, each child represented in the same way (either compact or extended) as it was before.

The above mechanism allows the visualization to draw each entity in its compact or extended state, according to the zoom level indicated by the user. To make zooming in and out on a given element interactive an extra step is however required: we need to dynamically replace the visualization of contents of the entity by new contents. At our request this feature was added to Mondrian by its maintainer. The forNode:do: feature takes a node in the visualization graph and replaces its current visualization with the results of executing a Mondrian script (given as a Smalltalk block). After such a replace operation, triggering a redraw of the visualization replaces the affected shapes and causes the layout of the figure to be adapted, if necessary. Note that the dynamic replacement of parts of the visualization differs from redrawing the entire diagram in more than just performance. Dynamic replacement respects the state of scrolling inside the visualization, while a complete redraw resets the view to the top left part of the visualization. The latter is clearly undesirable behavior for when performing a zoom operation.

Implementing the structural zoom is thus a straightforward combination of the above. The Mondrian scripts for Namespace, Aspect, Class and Method register a handler on left mouse button clicks: a Mondrian script that will be executed on a click within (specific parts of) their visualization. This script first negates the shape attribute, second uses forNode:do: to replace its contents being visualized, and third triggers a redraw.

5.1.3. The Mondrian script as a modular extension of ASPIX

ASPIX, being an aspect-oriented extension of the FAMIX meta-model, was conceived as a generic meta-model for aspect-oriented code. The goal for ASPIX is therefore not only to be used for AspectMaps, but also for other cases where AO software needs to be analyzed using Moose. Hence the ASPIX model should be free of AspectMaps-specific code. The Mondrian script of AspectMaps however distributes responsibility of visualization of the different entities in an ASPIX model to these entities themselves. This is achieved by adding methods, attributes and traits to them. If this was performed naively it would pollute the ASPIX meta-model with AspectMaps-specific code, negating the generic nature of ASPIX. As this issue potentially applies to all Mondrian visualizations using the modular structure we outlined in Section 5.1.1, we now show how we avoided this pollution through a combination of Smalltalk class extensions, FAMIX properties and programmatically installing traits.

Smalltalk allows for methods of one class to be specified in more than one package thanks to its class extension feature. For example, we can perform a class extension of the class named Aspect defined in the package ASPIX: in the package AspectMaps we define a number of methods for visualization to the Aspect class. When the AspectMaps package is loaded, these methods are added to the Aspect class but remain part of the AspectMaps package. The Mondrian script of AspectMaps is defined in this way: all the methods added to the different entities are class extensions defined in the package AspectMaps. As a result the ASPIX package is not polluted by AspectMaps-specific methods. This solution however falls short on 2 points: the shape attribute and the application of traits. We discuss both next.

Smalltalk class extensions only specify new methods to be added to an existing class, they are unable to add extra state through instance variables. We however need to keep the zoom state of Namespace, Aspect, Class and Method in a shape attribute that therefore needs to be added to these entities. FAMIX allows for an alternate way in which attributes can be kept for model entities: the use of properties [9]. The advantage of the properties system is that getting and setting of properties

can be performed purely by adding methods to the different FAMIX entities. As all ASPIX entities are also FAMIX entities we elected to keep the shape attribute as a FAMIX property. Consequently we successfully overcome the limitation of class extensions not being able to add extra state to a class.

Recall that to avoid code duplication in the Mondrian script, common behavior is refactored to a trait. This trait is then composed with the relevant entities in the ASPIX model to yield the full functionality. Trait composition however, like adding state, is not supported by class extensions. We therefore are required to perform trait composition in a separate step. To achieve this we implemented a small trait installer utility[7] that allows for the installing (and uninstalling) of traits from within application code. As part of the startup routine, the AspectMaps tool uses this installer to ensure that the required trait is composed with the different entities in the ASPIX model. This results in the complete functionality of the AspectMaps visualization being present after the tool is opened.

To summarize, we were able to maintain a separation between ASPIX as a generic meta-model for AO code and the AspectMaps visualization while maintaining the modularity of the visualization code. To achieve this, we used Smalltalk class extensions to separately specify part of the behavior of the visualization, implemented a trait installer to compose the behavior that was refactored as traits on startup, and FAMIX properties to store the shape attribute.

5.2. Building the user interface with Glamour

Moose contains a variety of tools to analyze software, for example FAMIX model exploration tools, a tool to build custom visualizations in Mondrian, a tool to build wizards, et cetera. Moose however does not provide any tool for building user interfaces and neither is such a tool available in Pharo. The tool offered by Moose that comes the closest to such functionality is Glamour [5]. It provides for defining only one specific kind of user interface, called browsers. In this section we present how we were still able to use Glamour to build the user interface for the AspectMaps tool. We show how the fundamental paradigm of Glamour can be adapted to apply outside of its intended context, making it usable as a more generic user interface tool.

5.2.1. Browsers according to Glamour

Glamour defines a browser as a user interface dedicated to manipulating models. Browsers visualize the model, allow to navigate in this model and perform actions on model elements where appropriate. One example is the Smalltalk code browser. It shows a fixed structure of a list of packages, a list of classes, a list of categories, a list of methods and a source code field. The browser allows to navigate from packages to classes, from classes to categories of methods, from method categories to methods, and from methods to their source code.

Visually, browsers consist of a collection of panes where each pane shows information specific to the data being browsed. For example, in the Smalltalk Browser the panes are the lists and the source code field being shown. For Glamour, the fundamental paradigm underlying navigation in browsers is the flow of one pane to the next pane. This is realized by having a pane continuously transmit its status to dependent panes. A Glamour specification, called a script, specifies the layout of the different panes of the browser, the contents shown in the different panes, and the transmissions between the different panes. A selection of, or an operation on an element in a pane produce a change in the transmission that is sent to dependent panes. This new transmission then contains the relevant data, *e.g.*, the item that was selected. Dependent panes update their contents on such changes, showing the required data. For example in a Glamour implementation of the Smalltalk browser a selection of a package transmits this selected package to the classes list pane. This pane then shows the classes present in that package.

5.2.2. AspectMaps in Glamour

The AspectMaps user interface however does not straightforwardly conform to this flow paradigm. Selecting an aspect to be visualized in the available projects list immediately shows the visualization in the Mondrian pane, without the other panes being involved. A button press in the Structural Zoom pane does not correspond to a selection of a model element. Moreover, an interaction in any of the other panes updates the Mondrian pane as well. Finally, the information shown in the Classes, Methods, Aspects and Advice panes is a fixed set of metrics that does not originate from the list of available projects. Nonetheless, by using multiple sources of information flow, allowing empty transmissions and generating falsified selection events, we were able to build the AspectMaps tool using Glamour.

In the AspectMaps tool, the flow of transmissions between the different panes is as follows:

- The Mondrian pane is the destination of transmissions of all the other panes in the user interface except for the Aspects in Project. All of these therefore transmit their selection to the Mondrian pane.
- The Aspects in Project and Structural Zoom panes are also the destination of transmissions from the Available Projects pane, receiving selection events.
- The Available Projects pane is one source of information flow, showing the models present in the repository.
- The Classes, Methods, Aspects and Advice panes are a second source of information flow, showing the list of available metrics (which are available for all models).

[7] Available at http://www.squeaksource.com/TraitsApplication.html.

The Mondrian pane thus refreshes itself whenever a transmission sent by any of the other panes in the user interface changes. Upon such a refresh, the behavior of the Mondrian script is to first process all its incoming transmissions before redrawing the visualization. This processing first ensures that at least a project has been selected for visualization. Second it enables that when another pane, *e.g.*, a metric pane, has no selection the model can still be visualized. Third it ignores the contents of selection events of the Structural Zoom pane, the reason for which we explain next.

The Structural Zoom pane is subject to a limitation in the current version of Glamour: buttons can only transmit selection events. However performing the zoom operations of the buttons do not correspond to a selection in the model. Instead, clicking on a button launches a visitor on the model, which toggles the shape attribute of the different entities accordingly. Therefore to update the visualization when the visit action is finished, a fake selection event is produced. This changes the transmission to the Mondrian pane, causing the visualization to be redrawn.

Recall that the Aspects in Project pane does not transmit its selection events to the Mondrian pane. This is so because it would entail that a selection of an aspect in the list would entail that the visualization is redrawn. This is however not the expected behavior, the visualization should only be redrawn in three cases: activation or deactivation of an aspect or if the aspect color is changed. To achieve this, we use an alternate method for redrawing the Mondrian pane as well as the Aspects in Project pane. This alternate method is the use of Announcement, which can be considered as the Pharo implementation of model–view–controller. Both the Mondrian and the Aspects in Project pane specify that they are dependent on a specific type of announcement that is posted when one of the three above events occurs. As a result both panes refresh themselves, taking into account the new status of the aspect.

The use of announcements is arguably more generic than transmissions, and the AspectMaps tool could possibly be re-implemented using only the former. Glamour however strongly favors the use of transmissions, making them much easier to specify and use. We therefore restricted the use of announcements to a minimum, using them only when necessary. Finding the correct trade-off between the use of transmissions and announcements is out of the scope of this paper.

In our experience Glamour is an outstanding tool to build user interfaces, but only if the architecture of the user interface can be made to fit inside of the flow paradigm. Glamour scripts allow a wide range of functionality to be specified in a simple fashion. As a result implementing basic browsers is straightforward. Glamour is however not just limited to basic browser construction. It is also suited to build advanced browsers through more complex scripts, as shown by the AspectMaps tool.

6. Related work

Although AOSD is a relatively young field of research, there exist some tools that use visualizations of aspect-oriented programs to support various development tasks.

The AspectJ Development Toolkit (AJDT) [7] is arguably the most complete and mature tool suite for aspect-oriented programming. AJDT integrates tightly with the Eclipse IDE and offers various facilities for understanding the interactions between aspects and base code. For example, AJDT features the use of gutter markers in the source-code editor to indicate join point shadows for affected entities. Furthermore, AJDT offers a textual ''Cross-references View'' that lists the signatures of methods affected by an aspect. While these features can provide developers useful information, they do so at a very fine-grained level of granularity, making them less suited for e.g. providing developers a global overview of where a particular aspect intervenes in a system. AJDT also offers a dedicated visualization based on the AspectBrowser tool of Griswold et al. [15]. This Seesoft-inspired [13] visualization shows the classes and aspects in the entire project as bars, placed side by side. The height of the bar is proportional to the number of lines of code that are present in the entity. Within each bar, each line of source code of that entity is represented by a stripe. The color of this stripe depends on the aspects that intervene at this line of code. In contrast to AspectMaps, this visualization only provides information at this level of granularity. As a result it has scalability issues when considering a large amount of classes and does not reveal structure at a finer-grained level than classes, *e.g.*, showing which methods an aspect affects. Finally, it does not visualize precedence relationships.

Other development environments also have some form of tool support for AspectJ. However this support is usually limited to a weaver (*e.g.*, for Netbeans [2], IntelliJ [3] and JBuilder [1]) and a view similar to the Cross-references view of AJDT (*e.g.*, for Netbeans and JBuilder).

Pfeiffer and Gurd [25] propose Asbro, a visualization tool based on Treemaps. A Treemap maps the nodes of a tree to rectangles in a plane, using a space-filling layout. Asbro represents classes or packages as rectangles that are colored with an aspect color if an aspect applies there. The authors assess their tool as being beneficial for obtaining a high-level overview of aspect application, and state that it is scalable up to on average 2100 classes. In contrast to AspectMaps, Asbro does not reveal aspect application at finer levels than types nor does it visualize aspect interactions.

Coelho and Murphy have introduced the ActiveAspects tool [6]. This tool uses an extension of UML to represent aspects, method execution advice and method call advice. As such a graph notation scales poorly when representing a large number of classes, ActiveAspects employs a number of heuristics to limit the number of visualized entities. As their user study has indicated, such a use of heuristics does not always align with the user's wishes. In contrast, our approach also allows developers — by means of the zooming functionality of AspectMaps — to only visualize part of the system. Rather than automating this process, we rely on users of our tool to manually determine which parts of the system are visualized. Furthermore, ActiveAspects does not provide information at a sub-method level, making it impossible to differentiate multiple join point shadows within one method or visualize interactions at one given shadow point.

Zhang et al. [30] present ITDVisualizer, an analysis toolkit for assessing the impact of structural modifications through AspectJ inter-type declarations on the behavior of the system. Due to the inherent obliviousness of such declarations, it can become increasingly difficult for a developer to understand how a program will behave. They propose analyses to assess how the declarations impact the method lookup of the base program, and to identify how particular base-code entities are shadowed by inter-type declarations. To present the results of the analyses to a developer, an integration with Eclipse is offered by means of visual clues (markers) and dedicated views that represent the lookup impact and shadowing impact. This approach is complementary to ours: AspectMaps focuses on the visualization of shadow points while ITDVisualizer aids in comprehending inter-type declarations.

Finally, the AspectScope work by Horie and Chiba [16] considers aspects as extensions to classes and displays the extended module interfaces of these classes. This however uses a textual tree-based representation, and therefore faces the same scalability issues as the AJDT cross-cutting view.

While software visualization is an active research field, to the best of our knowledge, there are no approaches that aim at visualizing the modularity or advanced decomposition of software systems, with the notable exception of Distribution Maps [10]. A Distribution Map is a generic visualization that shows the distribution of a particular property across a reference partition of a software system (e.g., package structure, files, and so on). This visualization provides information regarding how the property is spread across the partition. While not originally intended to do so, Distribution Maps can in principle be used to visualize how aspects cross-cut a particular software system. However, as a Distribution Map only displays a single property per entity, this approach is ill-suited to represent situations where multiple aspects intervene at a single entity.

7. Conclusion and future work

Despite the possible advantages of aspect-oriented software development in terms of modularizing a software system, the use of such technology places a burden on a developer's ability to understand the system. Especially the presence of an implicit invocation mechanism — that lies at the heart of aspect-oriented programming, and that introduces an extra level of indirection — can make it difficult to understand overall system behavior.

The AspectMaps tool was introduced to alleviate this problem. AspectMaps offers a scalable visualization that shows the implicit invocations in the source code by visualizing join point shadows where aspects are intervening. Key features of this visualization are a selective structural zooming functionality that progressively reveals more information as a user drills down into the structure of the code, and a fine-grained representation of individual join points that reveals which aspects apply at that join point and in which order.

While AspectMaps is a stand-alone tool, it was conceived as an extension of the Moose reverse engineering platform. The main contribution of this paper is a discussion of the architecture of AspectMaps and the lessons learned from building our tool on top of Moose. Firstly, we introduced ASPIX (ASPects In famiX), which is our aspect-oriented extension of the FAMIX meta-model, and discussed how our tool constructs such ASPIX models. Secondly, AspectMaps leverages some of the reusable frameworks and components that are offered by Moose. Our visualization is implemented using the Mondrian visualization engine; the interface of the tool itself was scripted using the Glamour language. Although these frameworks eased the development of AspectMaps, they have an impact on how the implementation of the tool itself is structured. We discussed the impact of this structuring and our experiences in overcoming the limitations that this caused.

There are multiple avenues of future work on AspectMap, both in terms of improving our visualization as well as providing better tool support. Currently, our visualization uses the color black to indicate that multiple aspects intervene at a single source-code entity. We plan to improve our visualization by using a distribution map in such cases that provides a user of our tool with more information. Furthermore, our tool does not support inter-type declarations. As for example illustrated in Section 3.3 this can result in odd visualizations containing a collection of seemingly empty aspects. To tackle this our Eclipse plugin needs to also export this information, the ASPIX model needs to be extended to include inter-type declarations, and the visualization needs to be augmented to represent them. Similarly, our tool does not show join point shadows that apply to fields, which would require an analogue process as above.

In terms of tool support we plan to tackle the limits of the Java and Aspect importers. Due to our reliance on a source code importer, we only are able to parse fully compliant Java source code (and not able to import Java files that also include AspectJ code). Moreover, we do not fully support all of the intricacies of Java, *e.g.*, anonymous inner classes. AspectMaps could be extended to support other combinations of base and aspect language by implementing export tools that export their aspectual information to the `.xcr` format. In combination with (existing) exporters to the `.mse` format this allows these programs to be imported into Moose as ASPIX models and hence visualized in AspectMaps. Finally, AspectMaps currently does not provide a tight integration with Eclipse. Consequently, users have to manually reload the visualization when the source code has changed, and cannot access this source code directly from within AspectMaps. As future work we foresee tighter integration with the Eclipse IDE in order to alleviate these issues.

Availability, Additional information

The AspectMaps tool is open source and available on the AspectMaps website http://pleiad.cl/aspectmaps.

Acknowledgments

We wish to thank Éric Tanter, Jacques Noyé, Alexandre Bergel, Awais Rashid, Thomas Cleenewerck, Kris De Schutter, Kim Mens, and Andrew Eisenberg for their invaluable feedback when discussing early versions of AspectMaps. Thanks also go to Andrew Eisenberg for helping us understand the AJDT crosscutting model and Alexandre Bergel for helping out with Mondrian. We are grateful to Theo D'Hondt for supporting this research. This research is partially supported by the IAP Programme of the Belgian State and the INRIA Equipe Associée PLOMO. Johan Fabry author is partially funded by FONDECYT project 1090083.

Appendix. Supplementary data

Supplementary data to this article can be found online at http://dx.doi.org/10.1016/j.scico.2012.02.007.

References

[1] AspectJ for jBuilder. http://aspectj4jbuildr.sf.net/.
[2] AspectJ for NetBeans. http://aspectj-netbeans.sf.net/.
[3] The AspectJ plugin for IntelliJ IDEA. http://intellij.expertsystems.se/aspectj.html.
[4] Mihai Balint, Tudor Gîrba, Radu Marinescu, How developers copy, in: Proceedings of International Conference on Program Comprehension, ICPC 2006, 2006, pp. 56–65.
[5] Philipp Bunge, Scripting browsers with Glamour. Master's Thesis, University of Bern, April 2009.
[6] Wesley Coelho, Gail C. Murphy, Presenting crosscutting structure with active models, in: AOSD '06: Proceedings of the 5th International Conference on Aspect-oriented Software Development, ACM, New York, NY, USA, 2006, pp. 158–168.
[7] Adrian Colyer, Andy Clement, George Harley, Matthew Webster, Eclipse AspectJ: Aspect-oriented Programming With AspectJ and the Eclipse AspectJ Development Tools, Addison-Wesley Professional, 2004.
[8] Arie Van Deursen, AJHotDraw: a showcase for refactoring to aspects, in: Workshop on Linking Aspect Technology and Evolution, 2005.
[9] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, Lukas Renggli, Meta-environment and executable meta-language using Smalltalk: an experience report, Journal of Software and Systems Modeling (SOSYM) 8 (1) (2009) 5–19.
[10] Stéphane Ducasse, Tudor Gîrba, Roel Wuyts, Object-oriented legacy system trace-based logic testing, in: Proceedings of 10th European Conference on Software Maintenance and Reengineering, CSMR'06, IEEE Computer Society Press, 2006, pp. 35–44.
[11] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, Andrew Black, Traits: a mechanism for fine-grained reuse, ACM Transactions on Programming Languages and Systems (TOPLAS) 28 (2) (2006) 331–388.
[12] Stéphane Ducasse, Sander Tichelaar, Dimensions of reengineering environment infrastructures, Journal of Software Maintenance and Evolution: Research and Practice (JSME) 15 (5) (2003) 345–373.
[13] Stephen G. Eick, Joseph L. Steffen, Eric E. Sumner Jr., Seesoft-a tool for visualizing line oriented software statistics, IEEE Transactions on Software Engineering 18 (11) (1992) 957–968.
[14] Johan Fabry, Andy Kellens, Stéphane Ducasse, AspectMaps: a scalable visualization of join point shadows, in: IEEE International Conference on Program Comprehension, ICPC, 2011, pp. 121–130.
[15] William G. Griswold, Jimmy J. Yuan, Yoshikiyo Kato, Exploiting the map metaphor in a tool for software evolution, in: Proceedings of the 23rd International Conference on Software Engineering, ICSE '01, IEEE Computer Society, Washington, DC, USA, 2001, pp. 265–274.
[16] Michihiro Horie, Shigeru Chiba, AspectScope: an outline viewer for AspectJ programs, Journal of Object Technology 6 (9) (2007) 341–361. Special Issue: TOOLS EUROPE 2007, http://www.jot.fm/issues/issue_2007_10/paper17/.
[17] A. Kellens, K. Mens, J. Brichau, K. Gybels, Managing the evolution of aspect-oriented software with model-based pointcuts, in: European Conference on Object-Oriented Programming, ECOOP, in: LNCS, vol. 4067, 2006, pp. 501–525.
[18] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William Griswold, An overview of AspectJ, in: Jorgen L. Knudsen (Ed.), Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP 2001, in: in Lecture Notes in Computer Science, vol. 2072, Springer-Verlag, Budapest, Hungary, 2001, pp. 327–353.
[19] C. Koppen, M. Stoerzer, Pcdiff: attacking the fragile pointcut problem, in: European Interactive Workshop on Aspects in Software, EIWAS, 2004.
[20] Adrian Kuhn, Toon Verwaest, FAME, a polyglot library for metamodeling at runtime, in: Workshop on Models at Runtime, 2008, pp. 57–66.
[21] M. Lanza, S. Ducasse, Polymetric views — a lightweight visual approach to reverse engineering, IEEE Transactions on Software Engineering 29 (9) (2003) 782–796.
[22] J. Laval, S. Denier, S. Ducasse, A. Bergel, Identifying cycle causes with enriched dependency structure matrix, in: Proceedings of the Working Conference on Reverse Engineering, 2009, pp. 113–122.
[23] Adrian Lienhard, Adrian Kuhn, Orla Greevy, Rapid prototyping of visualizations using mondrian, in: Proceedings IEEE International Workshop on Visualizing Software for Understanding, Vissoft'07, IEEE Computer Society, Los Alamitos, CA, USA, 2007, pp. 67–70.
[24] M. Meyer, T. Girba, M. Lungu, Mondrian: an agile visualization framework, in: Symposium on Software Visualization, SoftVis'06, ACM, 2006, pp. 135–144.
[25] J.-Hendrik Pfeiffer, John R. Gurd, Visualisation-based tool support for the development of aspect-oriented programs, in: AOSD '06: Proceedings of the 5th International Conference on Aspect-oriented Software Development, ACM, New York, NY, USA, 2006, pp. 146–157.
[26] M. Rinard, A. Salcianu, S. Bugrara, A classification system and analysis of AO programs, in: Twelfth International Symposium on the Foundations of Software Engineering, 2004.
[27] P. Tarr, H. Ossher, W. Harrison, S. Sutton, Degrees of separation: multidimensional separation of concerns, in: International Conference on Software Engineering, ICSE, ACM, 1999, pp. 107–119.
[28] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, Oscar Nierstrasz, A meta-model for language-independent refactoring, in: Proceedings of International Symposium on Principles of Software Evolution, ISPSE'00, IEEE Computer Society Press, 2000, pp. 157–167.
[29] V. Uquillas, S. Ducasse, T. D'Hondt, Visually supporting source code changes integration: the torch dashboard, in: Proceedings of the Working Conference on Reverse Engineering, WCRE'10, IEEE, 2010, pp. 55–64.
[30] D. Zhang, E. Duala-Ekoko, L. Hendren, Impact analysis and visualization toolkit for static crosscutting in AspectJ, in: International Conference on Program Comprehension, ICPC, 2009.