AspectMaps: A Scalable Visualization of Join Point Shadows

Preprint of ICPC 2011

Johan Fabry *
PLEIAD Laboratory
Computer Science Department (DCC)
University of Chile
http://pleiad.cl

Andy Kellens †
Software Languages Lab
Vrije Universiteit Brussel
Belgium
http://soft.vub.ac.be

Stéphane Ducasse
RMoD,
INRIA Lille - Nord Europe
France
http://rmod.lille.inria.fr

***Abstract*—When using Aspect-Oriented Programming, it is sometimes difficult to determine at which join point an aspect executes. Similarly, when considering one join point, knowing which aspects will execute there and in what order is non-trivial. This makes it difficult to understand how the application will behave. A number of visualizations have been proposed that attempt to provide support for such program understanding. However, they neither scale up to large code bases nor scale down to understanding what happens at a single join point. In this paper, we present AspectMaps – a visualization that scales in both directions, thanks to a multi-level selective structural zoom. We show how the use of AspectMaps allows for program understanding of code with aspects, revealing both a wealth of information of what can happen at one particular join point as well as allowing to see the "big picture" on a larger code base. We demonstrate the usefulness of AspectMaps on an example and present the results of a small user study that shows that AspectMaps outperforms other aspect visualization tools.**

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

***Keywords*-aspect-oriented programming; visualization; join point shadow**

I. Introduction

Aspect-oriented programming introduces a novel kind of module, *aspects*, as a means to modularize *cross-cutting concerns*: concerns whose implementation is scattered throughout the system under development. Aspects encapsulate not only the implementation of the cross-cutting behavior but also the specification of where and how they are invoked. As a result, the other modules of the system, called the *base code*, do not contain explicit calls to the functionality implemented in the aspects. Instead the base code performs *implicit invocations* to the behavior of the aspects, as specified in the aspects themselves.

To realize this, the flow of execution of the base application is reified as a sequence of *join points*. Second, the specification of the implicit invocations, in the aspect, is made through a *pointcut* that selects at which join points the aspect executes. The behavior specification of the aspect is called the *advice*[1]. An aspect may contain various pointcuts and advice, where each advice is associated to a pointcut.

The concepts of pointcut and advice open up new possibilities in terms of modularization, allowing for a clean separation between base code and crosscutting concerns. However this separation makes it more difficult for a developer to assess system behavior. In particular, the implicit invocation mechanism introduces an additional layer of complexity in the construction of a system. This can make it harder to understand how base system and aspects interact and thus how the system will behave.

Various well-documented issues within the aspect-oriented community serve as a testimony to this problem. For example, inherent limitations of the expressiveness of pointcut languages have an impact on the ease with which the correct set of join points can be captured in a pointcut expression [1]. One well-documented case of this is the so-called *fragile pointcut problem* [2], [3]. It states that seemingly innocent changes to the base code of an aspect-oriented system can lead to unintended and erroneous behaviour upon evolution of that base code. A similar problem that may arise is that in complex systems, multiple aspects can intervene at the same join point. If a developer is not aware of the interactions of multiple aspects intervening at the same join point, this again can result in erratic application behavior [4].

Consequently, there is a need for tools that allow software developers to easily assess the impact of aspects on the base system to aid in the detection and prevention of the problems we discussed above. While software visualization is known to be a good approach to achieve this, current visualizations fall short on various points, as we show in this paper.

We present a visualization to aid the understanding of aspect-oriented software systems, called *AspectMaps*. It provides a scalable visualization of implicit invocation. AspectMaps renders selected *join point shadows*: locations in the source code that at run-time produce a join point. AspectMaps shows the join point shadows where an aspect is specified to execute, and if multiple aspects will execute, the order in which they are specified to run. This results in

* Partially funded by FONDECYT project 1090083.
† Funded by a research mandate provided by the "Institute for the Promotion of Innovation through Science and Technology in Flanders" (IWT Vlaanderen)

[1] AOSD literature uses the term 'advice' instead of 'a piece of advice'.

a visualization that clearly shows how aspects cross-cut the base code, as well as how they interact at each join point. AspectMaps is a scalable visualization mainly due to its use of selective structural zooming. The structure of source code is shown at different levels of granularity, as determined by the user. AspectMaps is implemented as an open source tool, downloadable from http://pleiad.cl/aspectmaps.

The remainder of the paper is structured as follows: we next give an overview of the design choices underlying our visualization, and motivate our selective structural zooming mechanism. Section II introduces the AspectMaps visualization, detailing what is shown at each zoom level. In Section III we show how the use of AspectMaps aids program comprehension, using illustrative examples and an initial user study. We consider future work in Section IV, followed by an overview of related work in Section V. Finally, Section VI concludes.

II. The AspectMaps Visualization

The idea of using visualizations to aid in program comprehension is not new. Within the reverse engineering community, software visualizations are a well-established means of supporting various software comprehension tasks (*e.g.*, [5], [6], [7]). Despite the advantages of software visualizations, designing a good visualization is not a trivial task. A visualization must be sufficiently rich such that it can convey the correct information in a single glance, yet not overwhelm the user. In cognitive sciences, the topic of data visualization has been well studied [8], [9], which has produced different guidelines to follow to design a successful visualization. We summarize six guidelines here: 1) The visualization should not overwhelm the user with the *number of colors* that are used [8]. 2) The visualization should not be too *complex* making it hard to interpret, nor too simplistic, not conveying information [6]. 3) There should be *a clear mapping* between the entities that are present in the visualization and the actual domain the visualization represents [7]. 4) The right *information density* has all visual elements of the visualization conveying some meaning to the user [9]. 5) The visualization should *scale*, working on small data samples, as well as large quantities of data [8]. 6) A good visualization has *interactivity*, providing a means for user interaction [10].

AspectMaps is a visualization that was built specifically with these six guidelines in mind, offering users a detailed overview of implicit invocation. It visualizes:

- where aspects are specified to apply in a system, based on visualizing join point shadows
- how aspects possibly interact at each join point shadow
- in a scalable way, thanks to a multilevel selective structural zoom.

We define that an aspect applies in 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 on an object-oriented class-based language. The join point model consists of method calls and method executions. Advices can execute before, around or after a join point, and we distinguish between after returning and after exception throwing. Aspects may contain various advices, and an execution order may be specified between aspects. The above effectively allows us to visualize a subset of AspectJ [11] and Java code. In this case we ignore inter-type declarations as well as advice that applies to fields.

Selective Structural Zoom: Following the guidelines of *scalability* and *interactivity*, 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.

AspectMaps visualizes code at the level of granularity of packages, classes and methods. In contrast to mapping applications, however, in AspectMaps the level of granularity is not a global setting: within one single diagram, various levels of granularity can be used. For example, certain packages can be shown at the package level, while others are zoomed in at the class level. Likewise, for certain classes, the visualization can be further zoomed in to depict the system at the level of individual methods. This allows the user to selectively zoom in and out to elements of interest. Furthermore, hovering the mouse pointer over a given element produces a tooltip style pop-up that shows the element at the next higher zoom level if available. This allows the user to skim over a number of elements, getting more information of each without zooming in and out.

A second factor that enables scalability is the selection of aspects to be displayed as well as the colors that identify them. AspectMaps allows users to respect the *number of colors* guideline. The user can for each aspect choose a specific color and turn visualization of join point shadows on or off, visualizing as much aspects as needed (at the cost of more difficult identification).

AspectMaps also scales down to a very fine level of granularity. At the most detailed zoom level on a join point shadow, it shows a wealth of information at a single glance. The user can see the specification of the kind of advice (before, after, ...), how different aspects are specified to interact (due to precedence declarations), and whether the pointcut has a run-time test or not. More detailed information is available as pop-ups: *e.g.*, advice signatures.

Detailing AspectMaps: To detail the AspectMaps visualization, the remainder of this section is structured following the levels of granularity that can be shown. For each level we show how it is visualized, and mention how it follows the six guidelines we outlined. To illustrate the tool we use

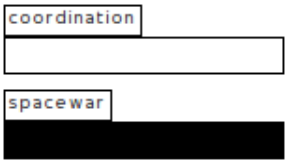

Figure 1. Compact visualization of coordination, spacewar.

Figure 2. The spacewar package extended view (annotated).

two examples in this section. Specifically, for Sections II-A and II-B we use the Spacewar example from the AspectJ Development Toolkit (AJDT) [12]. This as it is a small example application that is easy to obtain and suffices to show many features of AspectMaps. In Spacewar we visualize the aspects Coordinator in green, EnsureShipIsAlive in red and Debug in blue. In section II-C, we use an additional artificial example, to illustrate AspectMaps features that are not revealed by Spacewar.

A. Package Level

When opened, AspectMaps provides an overview of all the packages in the system. For each package AspectMaps shows a compact visualization that details the names of packages as well as which aspects apply in this package. AspectMaps colors the package rectangle with the color of the aspect that applies, if it is currently enabled for visualization. If multiple aspects apply in the package this is indicated by using the color black (which is never a color of an aspect). An example of this is shown in Fig. 1, which shows two packages: coordination and spacewar. Multiple aspects apply in the spacewar package, indicated by the black color. The contents of packages is not shown at this point.

The extended visualization of packages, enabled by performing a zoom operation on a selected package, reveals package contents. In Fig. 2 the package spacewar has been zoomed in on, showing the different classes and aspects that it contains.

The extended package visualization is a version of the work of Lanza and Ducasse on Polymetric Views [5], which we extended with support for the visualization of aspects. Polymetric Views display entities as boxes the dimensions of which reflect entity properties (LOC, number of methods, ...). A variety of different types of information is shown at this level:

- **Classes:** rectangles with black borders. Inheritance relations are visualized using the standard UML notation, a conventional *mapping to reality*.
- **Aspects:** rectangles with thick colored borders. The color is the aspect color (which is never black).
- **Where aspects apply:** class rectangles have the color of the aspect that applies, or black for multiple aspects.
- **Class and aspect metrics:** the user selects which dimension reflects which metric. This increases *information density*. For the figures in this paper we select no metric, as this feature of polymetric views is not the focus of our work.

Note that the polymetric views visualization does not display the names of classes. This is chosen to avoid clutter, which would increase *complexity*. We are faithful to this feature of polymetric views, class and aspect names are instead revealed in their respective pop-ups (which is the class level visualization discussed next).

In Fig. 2, we see three class hierarchies with as roots Pilot, Display, and SpaceObject, along with eight aspects where Debug is in blue and EnsureShipIsAlive in red. The Coordinator aspect (in green) is not part of this package but applies at least in four classes (SpaceObject, Display1, Display2, Registry). In the *Pilot* hierarchy, multiple aspects apply in the subclass Robot, and only the EnsureShipIsAlive aspect (in red) applies in the subclass Player.

B. Class and Aspect Level

The visualization at this level is similar to that at the package level. Here for classes fields are shown as diamonds and methods are shown as rectangles with a gray border, the height and width of which can be determined by a user-selected metric. Methods are colored according to the aspects that apply. Pop-ups of fields reveal their name and type.

For aspects, advice is shown as rectangles of which the dimensions are determined by a user-selected metric, and the named pointcuts are drawn as ovals. No further zoom level is available for aspects, therefore pop-ups for advice show the line number and signature, and pop-ups for pointcuts show the name of the pointcut.

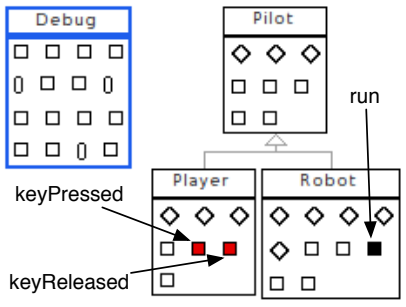

Figure 3. The *Debug* aspect and the *Pilot* hierarchy (annotated with selected method names).

In Fig. 3 we show a view that is zoomed in on all classes in the Pilot hierarchy, as well as on the Debug aspect. This reveals that multiple aspects apply on the method run of Robot and that the EnsureShipIsAlive aspect applies in the methods keyPressed and keyReleased of Player.

C. Method Level

Method level is the finest level of granularity offered by AspectMaps. At this level a wealth of information is presented, and hence the visualization is more complex. At method level, methods have a gray border to more easily distinguish them from classes, decreasing *complexity*.

If we consider only one join point, advice can be specified to execute before, around or after this join point. Therefore a visualization of its join point shadow needs to separate showing before, after and around advice. Also, at one join point multiple aspects may apply, so the visualization must be able to show the execution of various advice at that point. Considering the method level, we can have a join point shadow for the execution of the method, and within the method body various join point shadows for method calls.

Fig. 4 shows a template for the visualization of method execution join point shadows. On the right, a set method is displayed using this template (and we discuss this next). The figure shows how AspectMaps provides the method name and shows before, after and around execution advice divisions. We detail next how advice execution within such a division is visualized.

1) Advice Execution, Run-time Tests, Ordering: To show that an advice applies at a given division of a join point shadow, AspectMaps draws a small figure in the color of the corresponding aspect. This is done for all aspects that apply, aligning the figures vertically. Fig. 4 shows four after execution advice in the example set method.

If multiple advice of *the same aspect* apply at the same join point shadow, for each of these a figure is drawn and these are ordered horizontally. A gray arrow between them indicates the order in which these advice will be executed, *e.g.*, in AspectJ this is determined by the line numbers of the advice. In the example in Fig. 4 this occurs for the red aspect. AspectMaps currently has two kinds of figures: a triangle for after throwing advice and a rectangle for all other kinds of advice. This is to emphasize the special nature of after throwing advice: it executes when the method terminates by throwing an exception. In the example this is again the red aspect. Moreover, 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. In the example this is evident in the green aspect. This allows easy identification of advice that will always run at this join point shadow: these have no border. Note that each figure shows three different data points: the aspect, if it is an after throwing, and if there is a run-time test. This increases the *information density*, however without overly increasing the *complexity*.

When multiple advice of *different aspects* apply, the order of their application may be specified by the programmer, *e.g.*, using the declare precedence construct in AspectJ [11]. When such an order is specified, AspectMaps indicates this by attempting to order them horizontally and drawing a black arrow between the advice execution figures, indicating the order in which the advice will be executed. This increases *information density* and maintains a good *mapping to reality*. In other words: gray arrows indicate execution order as specified by the semantics of the aspect language, while black arrows indicate programmer-specified execution order. An example of the latter is given in Fig. 5. 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. 4 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.

Note that AspectMaps shows the order of execution of advice, and not a declaration of aspect precedence, as defined in *e.g.*, AspectJ. The difference lies in that advice execution of after advice runs in the *reverse* order than that of before advice. This makes the visualization easier to understand: what is shown is more directly connected to the behavior of the resulting application. We do not require the programmer to perform a context switch and mentally invert the advice execution order being shown. In other words, we have a better *mapping to reality* and reduce *complexity*.

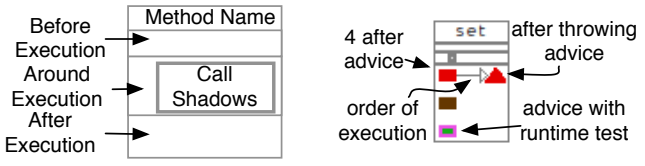

Figure 4. Template for visualization of execution join point shadows (left), and an example set method (right). Fig. 6 shows call visualization.

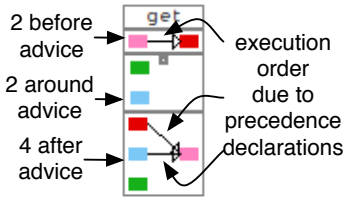

Figure 5. Eight execution advice for a get method, with two precedence declarations yielding three ordering arrows.

2) Execution Join Point Shadows: For execution join point shadows the groups of figures detailing advice execution are placed in the locations as given by the template and examples in Fig. 4 and 5.

Recall that all entities provide extra pop-up information when the mouse pointer hovers over them, and that this information is the visualization of the next zoom level if available. As there is no finer grained zoom level here, we instead provide relevant textual information on the advice execution element being hovered over (increasing *interactivity*). Specifically, we show the signature of the advice, including its line number in the aspect source code.

3) Call Join Point Shadows: The body of a method may contain multiple call join point shadows, sequentially ordered by the source code of the method. We visualize advice execution in this same order, aligning them vertically as a suitable *mapping to reality*. The visualization of call join point shadows uses the same visualization as execution join point shadows. It however orders the before, around and after divisions horizontally instead of vertically. A template of this is shown in Fig. 6. The horizontal layout was chosen to minimize unused space when visualizing (increasing *scalability*), as well as to avoid confusion of what advice execution belongs to which join point shadow (decreasing *complexity*).

In Fig. 7 we show a number of methods of the Spacewar example that demonstrate the visualization of call join point shadows. This figure also illustrates the pop-up information

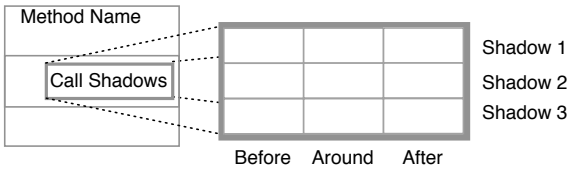

Figure 6. Template for call join point shadows

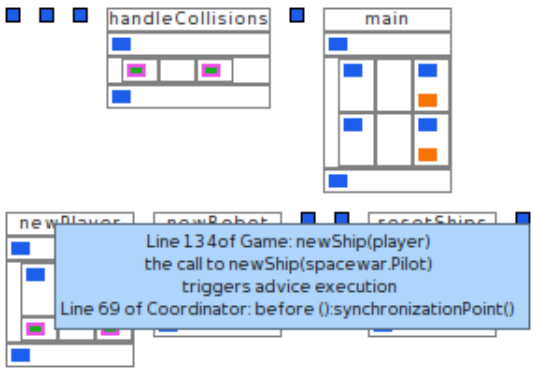

Figure 7. A selection of methods in Spacewar showing call shadow points, as well as a pop-up of one advice execution (in the newPlayer method).

for each advice execution. It consists of the signature of the advice (including its line number) as well as the signature of the method being called and the base code expression containing the call (including its line number). This again increases *interactivity* and *information density*.

D. The AspectMaps tool and Quick Zoom Options

The AspectMaps visualization is implemented as a standalone tool. A discussion on the implementation of this tool is however the topic of a separate publication. We restrict our discussion here to the quick zoom and navigation options provided to ease interaction with the visualization.

The AspectMaps tool provides a number of predefined zoom operations, we highlight four here:

- **Max Zoom Out** Zooms all elements out *i.e.*, for each element specifying that its compact representation should be shown.
- **Max Zoom In** Zooms in maximally on all join point shadows where an aspect that is visualized applies.
- **Interactions Zoom** Zooms in maximally on all join point shadows where more than one of the aspects that are being visualized apply.
- **Query Zoom** Given a query, which may contain wildcards, zooms in maximally on classes or methods of which their names match.

Furthermore, context-specific zoom options are present on the following elements:

- **On pointcuts** revealing all the join point shadows.
- **On advice** revealing all the join point shadows.
- **On advice execution** revealing the aspect.
- **On advice execution** revealing all other executions.

The advantage of these zoom options is that they save developer time and effort. There is no time wasted in manually exploring the visualization and zooming in or out, *e.g.*, looking for a place where two specific aspects interact, or finding all places where a given advice applies.

The AspectMaps tool is open source and available from the website http://pleiad.cl/aspectmaps. This site also provides an executable version with the Spacewar example preloaded, as well as other examples ready for visualization.

III. Program Understanding with AspectMaps

Our validation of AspectMaps is two-fold. First, we show how our tool aids in software development and maintenance activities by a more detailed exploration of the SpaceWar example. Second, we present a user study where we compared AspectMaps with the AJDT tool suite.

A. Exploring the Spacewar Example

We now show how AspectMaps allows the developer to gain insight of existing code, by performing a larger study of the Spacewar code we have been using as a running example. In Fig. 8 we show one possible zoom state of Spacewar In this figure, we can see that the Debug aspect (in blue),

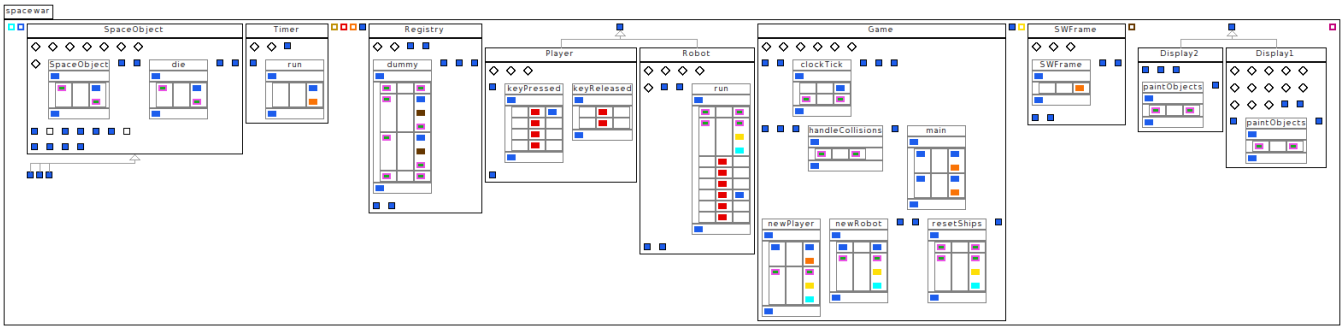

Figure 8. AspectMaps visualization of Spacewar, fully zoomed in on all join point shadows, except for those of the *Debug* aspect.

applies at the beginning and at the end of each method and constructor declaration that is zoomed in, and also applies in all other methods except for two in SpaceObject. Zooming in, or using the popups on zoomed out entities, we can see that this pattern of the Debug aspect is omnipresent. Note that, while we show a lot of information in this figure, achieving a high *information density*, it does not come at the price of too high *complexity*. For example, it is straightforward to deduce the above mentioned behavior of the Debug aspect.

Examining Fig. 8, four interesting elements are further revealed by AspectMaps:

Firstly, no precedence declarations have been declared, as there are no arrows between advices. We find this remarkable as there is interaction between the Coordinator aspect (in green), and other aspects. In other words, advice of other aspects execute at a number of coordination points. However there is no specification of whether this behavior should be also be coordinated.

Secondly, the DisplayAspect aspect (in purple) has four different pointcut-advice combinations: one specific combination for each method where the aspect applies. This is revealed by looking at the pop-ups for each advice execution rectangle: the line number given is specific for each method.

Thirdly, the EnsureShipIsAlive aspect (in red) applies only in the Pilot hierarchy. Only one around advice of the aspect is called. Lastly, when the join point shadow is ship.fire(), the Debug aspect also executes an after returning advice that uses the named pointcut allConstructorsCut(). (Again, this detailed information is obtained by looking at pop-ups.)

Lastly, at every join shadow where SpaceObjectPainting1 applies (yellow), SpaceObjectPainting2 (cyan) also applies, and vice-versa.

Considering *scalability*,when zoomed in completely it is impossible to present the visualization on one screen. We however consider AspectMaps to be used differently: using a different zoom level for different elements, depending on the focus of the developer. This is the case in Fig. 8. We ignore the details of the Debug aspect and solely focus on the remaining aspects in the system, only zooming in where these apply. This still yields four relevant observations, stated above, attesting to the scalability of AspectMaps.

1 What are the names of the aspects and in what packages are they located?
2 At which join point shadows do which advices of the aspect Coordinator apply?
3 At which join point shadows does an advice of Coordinator **and** SpaceObjectPainting1 apply?
4 At which join point shadows where multiple advices apply is the precedence order of all these adviced not explicitly specified, also for which is it not specified?
5 What methods are not affected by any aspects?

Figure 9. The code comprehension questions.

B. User Study: Understanding Existing Code

As part of the evaluation of AspectMaps we have performed a user study. The goal of the study was to provide an initial comparison between AspectMaps and AJDT, establishing their usefulness for code comprehension of AOSD code. Other aspect visualization tools were not considered because none of these provide as much information as AspectMaps and AJDT, as discussed in more detail in Section V. This has significant impact on being able to perform the typical code comprehension tasks we considered for our study, as we show next.

Study setup: For our study, five code comprehension questions were created, listed in Fig. 9. The first two questions treat basic code comprehension with aspects, of which the second one requires visualization at sub-method level, *i.e.*,scalability to a very fine-grained level. This simulates the setting where a new developer needs to get to know the application, and wishes to focus on the cross-cutting concerns, first finding the corresponding aspects. The second question then establishes whether an aspect applies where it is supposed to *i.e.*, whether its pointcuts are correct, which is directly linked to the fragile pointcut problem [2], [3]. Question three and four concern aspect interactions, with question four stressing scalability issues when multiple

Task	AM Time	AM Correct	AJDT Time	AJDT Correct	Prefer AM	Use AM	Use AJDT
1	1m 19s	88%	1m 34s	86%	3.5	3.9	2.7
2	5m 32s	88%	7m 55s	71%	4.1	4.3	2.4
3	2m 22s	100%	3m 41s	71%	4.3	4.3	2
4	5m 18s	88%	9m 17s	14%	4.7	4.4	1.7
5	2m 44s	100%	5m 3s	71%	4.5	4.3	1.9
Global	17m 14s	93%	27m 31s	63%	4.5	4.2	2

Figure 10. User survey results. Global is total time, mean accuracy and overall tool evaluation questions.

aspects apply. Multiple aspects applying at one join point can cause bugs if their execution order matters and this order is incorrect in the actual application. Question four addresses the same issue of question three, but tests for a large number of aspects and affected classes. Question five is a straightforward scalability question considering a large amount of join point shadows. Although the wording of this question arguably is somewhat artificial, it can be considered as a question similar to question two, which considers verifying whether a pointcut picking out particular join point shadows is correct. In question five the programmer verifies whether a more broad pointcut of an aspect is correct, by determining where it does not apply.

Considering other visualization tools, none allow the users to answer all of these questions: Asbro [13] cannot be used to provide answers to *any* of the questions, ActiveAspect [14] cannot provide answers to questions 2, 3 and 4.

The user study was performed with 15 subjects (PhD students, postdocs and professors), volunteers from the three different research groups of the authors. All work in the field of software engineering, have at least basic knowledge of AOSD, AspectJ and the AJDT, but none had knowledge of the Spacewar example. To introduce them to AspectMaps the subjects were presented with a preprint of Section II of this paper and were shown the screencasts on the AspectMaps website. Also they were given the paper that explains the AJDT visualization [15]. Lastly, before being given tasks to perform using the tools, they had five minutes to familiarize themselves with the tools, asking questions if necessary.

For each subject one of the two tools was randomly selected (8 started with AspectMaps; 7 with AspectJ). The subject then performed the five code comprehension tasks sequentially, on the Spacewar example. Each task was timed, with a maximum of ten minutes, and verified for correctness when the subject deemed the task done or if timed out. To obtain a subjective impression of both tools, the subject then performed the same five tasks on the other tool. This was not timed nor verified for correctness to rule out learning effects. After having finished working with both tools, the subjects filled in a questionnaire.

For each task, the questionnaire asked whether the first tool is better than the second tool for that task (phrased in those terms to reduce acquiescence bias) and if they would want to use AspectMaps resp. AJDT for these kinds of investigations in the future. Then the survey inquired whether the subject globally considers AspectMaps outperforming AJDT, and if they would use AspectMaps resp. AJDT in the future for similar code comprehension tasks. Grades were given on a five-point Likert scale, and a space was allowed for final remarks.

Study results: An overview of the results is given in Fig. 10, giving the average result of all participants for each entry in the table[2]. It shows that AspectMaps outperforms AJDT for each of the tasks, both in objective measurements of time and accuracy as well as the subjective opinion of the test subjects.

Considering time taken to perform the different tasks, AspectMaps is only slightly faster in the first, most basic task. For the other tasks, time differences vary from one minute to almost four minutes. The biggest difference is for task four, arguably the most complex task, where using AJDT takes 175% of the time needed when using AspectMaps. Furthermore, in addition to this speedup, results with AspectMaps are more correct than with AJDT. In task one the difference is negligible (2%), but in the other, more complex tasks the difference vary from 17% up to an important difference of 74% in favor of AspectMaps. Again the biggest difference is obtained in task four. Lastly, AspectMaps is the only tool where 100% accuracy is obtained, and this for 2 questions.

The users evaluated AspectMaps as being a better tool for supporting comprehension of AOSD software than AJDT. With 3 being a neutral answer and 5 the strongest preference, AspectMaps rates more than 4 on all but the most basic task. For future code comprehension tasks, the users completely discard the AJDT visualization. They however state that they would use AspectMaps, with scores of 4.3 and 4.4, except for task 1, with 3.9. This is confirmed by the overall evaluation that AspectMaps scores better than AJDT, with a 4.5 average.

Some positive observations of the users are: “AspectMaps works nicely as a code comprehension tool.”, “Clearly AspectMaps is more intuitive, I think this is because it uses a spacial metaphor to show the data and not only text/bars”. The most frequent negative observation of the users is that AspectMaps does not show the source code, or that some IDE integration is needed. We consider this as future work.

[2]The complete results are available on the AspectMaps website.

Threats to validity: The sample size of 15 persons can be considered the weakest point of the study, but it is in line with sample sizes of published visualization research (*e.g.*, 24 subjects in[16]). The user study we performed is at a small scale and does not allow us to generalize about the superiority of our tool over AJDT. Nonetheless it is worthwhile to remark that the numbers we obtained are unambiguous and consistently in favor of AspectMaps. This is both in the quantitative as qualitative results.

A second threat lies in the use of colleagues as test subjects for our study. First, as researchers, our test subjects might not be considered typical developers. They might favor more complex tools and might not possess the same set of skills as developers working in industry. This is however compensated due to the various backgrounds of our test subjects, their different levels of acquaintance with AOSD and the differences in programming experience, as we involved a mix of 12 PhD students (being in various stages of their PhD), 1 postdoc and 2 professors. Second, the test subjects might be biased in favor of our work. This might indeed influence the users subjective opinion. However the time taken and accuracy for each task are not influenced by such a bias (if present), and solely based on these numbers AspectMaps already is a considerable improvement on AJDT.

A final threat lies in the definition of the tasks that were performed in the user study. More specifically, the five tasks (see Fig. 9) can be perceived as artificial and tailored towards demonstrating superiority of our approach. We however have shown that each task is grounded in a realistic setting and aims at generalizing a typical comprehension scenario.

IV. Discussion and Future Work

The Spacewar example we have used in this paper as a basis to explain the AspectMaps visualization is but a small piece of software. We use it as an example because it illustrates the majority of the features of AspectMaps, therefore obviating the need to introduce many examples to introduce the visualization. We have successfully used AspectMaps to visualize larger applications, for example AJHotdraw [17], which consists of 24 packages, 374 classes and 31 aspects. A full report of the exploration of the AJHotdraw code is outside of the scope of this paper.

Considering interactions between aspects, AspectMaps has a weak point in this setting. This is due to its focus of being a visualization of advice execution at join point shadows. This weakness is visualization of interactions at method call and method execution join point shadows. Consider for example the pointcuts call(* * AClass.aMethod()) and execution(* * AClass.aMethod()). The join point shadows for the former are visualized at all calls to aMethod(). This is a different place in the figure than the visualization of aMethod() (unless the call is a recursive call). Nonetheless, advice execution at the call side interacts with advice execution at the execution side. It would be beneficial to visualize these interactions as well. We have not yet encountered a suitable visualization for this, and consider this future work.

Currently, the AspectMaps tool is not integrated into any development environment, running instead in a stand-alone fashion. As mentioned in Section III-B, a possible target for integration is the Eclipse IDE. This would, *e.g.*, allow the user to easily navigate to the source code of the entities being visualized or allow the visualization to update itself automatically on a recompile. Such functionality is however additional to the core visualization concepts presented here. These were developed and validated separately to assess their inherent benefits, avoiding ambiguity of whether any advantages are gained though the visualization or though other means. Eclipse integration is a implementation task that we consider as future work.

A last limitation of AspectMaps we discuss here is the lack of information on structural modifications made by the aspects, also known as static cross-cuts or inter-type declarations. Currently the visualization does not show inter-type declarations, nor the aspects that apply there. As this feature is orthogonal to the core visualization concepts of AspectMaps we have not yet implemented support for this, and leave this as future work.

V. Related Work

Arguably the most complete tool suite for aspect-oriented programming is the AspectJ Development Toolkit [12]. Amongst other features, AJDT adds gutter markers in the code editor to indicate join point shadows for affected code entities, and also provides a textual "Cross-References View". While these features provide useful feedback, they do not scale to a large code base [13]. AJDT also offers a visualization tool [15] that shows the classes and aspects in the entire project as bars, placed side by side. The name of the class or aspect is printed in the top of each bar. The height of the bar is proportional to the number of lines of code that are present in the entity and colored stripes represent lines of code affected by aspects. This visualization however is overly simplistic, not showing the inherent structure of the code, nor the detailed information that AspectMaps offers as described in Section II-C. Hovering over a stripe does produce a pop-up, but this only details the name of the aspects that apply there. In general, obtaining any information of an aspect beyond the approximate source code location of its application requires to navigate to the source code. Lastly, the tool does not scale. While the tool has a 'zoom in' and 'zoom out' function, all that this does is to make the bars bigger or smaller. No more detailed (structural) information is revealed upon a zoom in action. Conversely, zooming out does not give a higher level of abstraction on the data, leading to scalability problems on large code bases.

Pfeiffer and Gurd [13] propose a visualization tool that is based on the concept of Treemaps. A Treemap maps the nodes of a hierarchical structure to rectangles in a plane, using a space-filling layout. In contrast to graph-based layouts of tree nodes, this does not waste any screen space. Their tool is called Asbro and provides for a tree map visualization of where aspects apply in packages and types. Rectangles representing classes or packages 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. However, Asbro does not scale down: it does not reveal aspect application at finer levels than types. Furthermore, it does not provide any information of aspect interaction at a given join point shadow. Additionally, the tool does not have a feature which shows that multiple aspects apply in one class or package.

Coelho and Murphy take a different approach to scalability in their ActiveAspect tool [14]. The tool shows an automatically selected subset of the elements in the code, depending on the current focus of the developer. The visualization that is used is an UML extension with a representation of aspects, method execution advice and method call advice. An important issue with such a graph notation is that it scales poorly with a large number of classes. ActiveAspects includes a number of abstraction operations to lessen clutter in these cases. The power of the approach lies in the ability of the tool to automatically perform such abstraction operations, as well as the automatic selection of elements to be visualized. However Coelho and Murphy note that their user study shows that the heuristics they are using often do not correspond with the users wishes. In contrast, in AspectMaps the user selects what is visualized and what is not, hence there are no heuristics issues. A further downside of ActiveAspects is that all aspects that apply within one method are gathered together in one visualization element. Because of this, ActiveAspects reveals no information of aspect interactions at one given join point shadow.

Zhang et al. have presented an analysis toolkit for assessing the impact of structural modifications through AspectJ inter-type declarations on the behaviour of the system [18]. Due to the inherent obliviousness of such declarations, it can become increasingly difficult for a developer to understand how a program will behave. To present the results of their 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 join point shadows while ITDVisualizer aids in comprehending inter-type declarations.

Lastly, the AspectScope work by Horie and Chiba [19] 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.

Software visualization is a very active field with numerous research results. However, few of them have a clear relevance in the context of aspect understanding. The most straightforwardly applicable is Distribution Map [20]. Distribution Map is a generic visualization that shows how a given phenomenon or property is distributed across a reference partition of a large software system (packages organization, files...). In particular, Distribution Map reveals the spread and focus of a phenomenon. Spread: how much does a property spread across the reference partition: is it local or global? Focus: how close does a property match the reference partition: is it well-encapsulated or cross-cutting? The goal of Distribution Map is not to display aspects but more general properties like code owners, commits, symbolic information. Also, Distribution Map can only display one property per node which prohibits visualizing interacting aspects. Consequently, it could be used to represent aspects, but lacks the AspectMaps abilities to visualize information at a sub-method level.

VI. Conclusion

Program understanding is a complex task that is made more difficult when using aspects because the base code implicitly calls aspect code. Implicit invocation is specified by pointcuts, adding an extra level of indirection that makes it difficult to understand total system behavior.

A common way to aid program understanding is the use of visualization tools that extract relevant information from the code under study. A number of visualizations for code using aspects have been developed [12], [13], [14]. However all of these have visualization-specific shortcomings, as we have discussed in this paper. Most noticeably neither of these tools scale both up to a large code base and down to a very fine-grained level.

In this paper we presented a new visualization for code using aspects, called AspectMaps. AspectMaps shows implicit invocations in the source code by visualizing join point shadows where aspects are specified to execute. For a given join point shadow, AspectMaps reveals very fine grained information at a glance: it shows the type of advice (before, after, ...) as well as specified precedence information (if any). Furthermore, AspectMaps scales to a large code base thanks to a selective structural zooming functionality (*i.e.*, a map metaphor) that progressively reveals more information as a user drills down into the structure of the code.

To argue for the merits of our visualization, we have shown how AspectMaps follows visualization guidelines and applied it to an example case study. We furthermore performed an initial user study, comparing the AspectMaps tool with the only other visualization tool that allows as much information to be obtained from the code: The AspectJ Development Toolkit (AJDT). For five different code comprehension tasks, AspectMaps consistently outperforms

AJDT on the amount of time required to perform each task and also on the users opinion of which tool is better and their willingness to use it again for similar tasks in the future.

Downloads, Additional Information

The AspectMaps tool, extra information and screencasts are available on the website http://pleiad.cl/aspectmaps.

Acknowledgment

We wish to thank Éric Tanter, Jacques Noyé, Alexandre Bergel, Awais Rashid, Thomas Cleenewerck, Kris De Schutter, Kim Mens, Andrew Eisenberg and Romain Robbes for their invaluable feedback when discussing early versions of AspectMaps. Thanks also to Andrew Eisenberg for helping us understand the AJDT crosscutting model and Alexandre Bergel for aid with Mondrian. We especially thank the user study participants. 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 Associated team PLOMO.

References

[1] W. Havinga, I. Nagy, and L. Bergmans, "Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions," in *First European Interactive Workshop on Aspects in Software*, 2005.

[2] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, no. 4067, 2006, pp. 501–525.

[3] C. Koppen and M. Stoerzer, "Pcdiff: Attacking the fragile pointcut problem," in *European Interactive Workshop on Aspects in Software (EIWAS)*, 2004.

[4] M. Rinard, A. Salcianu, and S. Bugrara, "A classification system and analysis of AO programs." in *Twelfth International Symposium on the Foundations of Software Engineering*, 2004.

[5] M. Lanza and S. Ducasse, "Polymetric views — a lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, no. 9, pp. 782–796, September 2003.

[6] S. Ducasse, M. Lanza, and R. Robbes, "Multi-level method understanding with microprints," in *2nd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*. IEEE Computer Society, 2005, pp. 33–38.

[7] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui, "Package surface blueprints: Visually supporting the understanding of package relationships," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, Oct. 2007, pp. 94–103.

[8] J. Bertin, *Graphische Semiologie. Diagramme, Netze, Karten*. Gruyter, 1974.

[9] E. Tufte, *The Visual Display of Quantitative Information*, 2nd ed. Graphics Press, 2001.

[10] M. Storey, F. Fracchia, and H. Müller, "Cognitive design elements to support the construction of a mental model during software exploration," *Elsevier's Journal of Systems & Software*, vol. 44, pp. 171–185, 1999.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., no. 2072. Budapest, Hungary: Springer-Verlag, Jun. 2001, pp. 327–353.

[12] A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.

[13] J.-H. Pfeiffer and J. 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*. New York, NY, USA: ACM, 2006, pp. 146–157.

[14] W. Coelho and G. C. Murphy, "Presenting crosscutting structure with active models," in *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2006, pp. 158–168.

[15] A. Clement, A. Colyer, and M. Kersten, "Aspect-oriented programming with AJDT," AAOS 2003: Analysis of Aspect-Oriented Software workshop at ECOOP 2003, http://www.comp.lancs.ac.uk/˜chitchya/AAOS2003/AAOS_Home.php, 2003.

[16] B. Cornelissen, A. Zaidman, A. van Deursen, and B. Van Rompaey, "Trace visualization for program comprehension: a controlled experiment," in *International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, 2009, pp. 100–109.

[17] A. V. Deursen, "Ajhotdraw: A showcase for refactoring to aspects," in *In: Workshop on Linking Aspect Technology and Evolution. (2005*, 2005.

[18] D. Zhang, E. Duala-Ekoko, and L. Hendren, "Impact analysis and visualization toolkit for static crosscutting in aspectj," in *International Conference on Program Comprehension (ICPC)*, 2009.

[19] M. Horie and S. Chiba, "Aspectscope: An outline viewer for aspectj programs," *Journal of Object Technology, Special Issue: TOOLS EUROPE 2007*, vol. 6, no. 9, pp. 341–361, October 2007, http://www.jot.fm/issues/issue_2007_10/paper17/.

[20] S. Ducasse, T. Gîrba, and A. Kuhn, "Distribution map," in *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*. Los Alamitos CA: IEEE Computer Society, 2006, pp. 203–212. [Online]. Available: http://scg.unibe.ch/archive/papers/Duca06cDistributionMap.pdf