AspectMaps: A Scalable Visualization of Join Point Shadows

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

Simon Denier
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 will execute. 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 visualization tools 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 does scale 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 three small examples and present the results of a user study that shows that AspectMaps clearly 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

1. Introduction

Aspects modularize cross-cutting concerns by encapsulating not only their behavior but also where and how they are invoked. As a result, the other modules of the system, called the *base code*, perform *implicit invocations* to the behavior of the aspects. First, the flow of execution of the base application is reified as a sequence of *join points*. Second, the specification of the implicit invocations is made through a *pointcut* that selects at which join points the aspect executes. The behavior specification of the aspect is called the *advice*. An aspect may contain various pointcuts and advices, where each advice is associated with one pointcut.

The concepts of pointcuts and advices 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 [17]. One well-documented case of this is the so-called *fragile pointcut problem* [19, 21]. It states that – upon evolution of the base code of an aspect-oriented system – seemingly innocent changes to that base code can lead to unintended and erroneous behaviour. 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 [24].

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 runtime 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 a visualization that clearly shows how aspects crosscut 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 also implemented as an open source tool, downloadable from its website http://pleiad.cl/aspectmaps, which furthermore features screencasts of the tool in action.

[*] 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)

The remainder of the paper is structured as follows: we next give an overview of software visualization, detailing typical pitfalls as well as providing an evaluation of the currently most complete aspect visualization. Section 3 introduces the AspectMaps visualization, detailing what is shown at each zoom level. In Section 4 we show how the use of AspectMaps aids program comprehension, using two examples and an initial user study. We consider future work in Section 5, followed by an overview of related work in Section 6. Finally, Section 7 concludes.

2. Software Visualization

Software visualization is defined as the use of graphic means (typography, graphic design, animation, . . .) to facilitate human understanding and effective use of computer software [26]. The idea of using visualizations to aid in program comprehension is not new. For example, within the reverse engineering community, software visualizations are a well-established means of supporting various software comprehension tasks [22, 27, 28].

One of the major advantages of software visualizations is that they are able to convey a large quantity of information to a user [32]. The human brain can easily combine complex information from visual cues, making them a suitable means for understanding complex software systems. Furthermore, a well-chosen visualization allows users to pre-attentively process the information: rather than having to search for specific information (*e.g.*, by extracting it from the source code), visualizations can immediately draw a user's attention to specific parts of the system.

2.1 Visualization Pitfalls

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. However the user should not be overwhelmed by the visualization, making the extraction of any meaningful information impossible. In cognitive sciences, the topic of data visualization has been well studied [5, 31], which has produced different guidelines to follow to design a successful visualization. In what follows, we discuss a number of common pitfalls of software visualizations and distill from this a set of requirements for our visualization of aspect-oriented systems.

- **Amount of colors:** The visualization should not overwhelm the user with the number of colors that are used. The human brain is only able to distinguish between a limited number of colors (the threshold often mentioned in literature is 10) in a meaningful way [5, 30, 32]. If more colors get used, the meaning that is conveyed by them gets lost.
- **Complexity:** If the visualization is overly simplistic, it becomes hard to convey meaningful information to the user. Conversely, overly complex visualizations become hard to interpret. Instead of being able to extract information by glancing at the visualizations, a user needs to make a conscious effort to interpret the visualization. As a result he loses the mental context of the development activity in progress [12].
- **Mapping to reality:** There should be a clear mapping between the entities that are present in the visualization and the actual domain the visualization represents. For a user, the representation used should feel natural for the particular domain concepts [13].
- **Information density:** All visual elements of the visualization should aim at conveying some meaning to the user. This is also known as Tuftes' data-ink rule [31]. Elements that are without such meaning clutter the visualization, thus making it more complex and should therefore be avoided.
- **Scalability:** The visualization should be able to work on small data samples, as well as large quantities of data. When applied to large amounts of data, the visualization should still be comprehensible. One metric that is often applied is that the information is best represented on one or two screens, thus minimizing the amount of scrolling that is required of the user [5].
- **Interactivity:** A good visualization is not limited to providing a static picture of the system but also provides a means for user interaction. By adding such functionality to the visualization, the user gets more involved in the process of interpreting the visualization. Additionally, such interactivity might be an ideal candidate to improve the scalability of the visualization and to deal with complexity issues. Interactions can be added to the visualization (*e.g.*, pop-ups) to convey additional information to the user, or to limit the scope of the visualization to a particular subset of the software system that is visualized [29].

2.2 Assessing Existing Aspect Visualizations

Our work is not the first to study the subject of aspect visualization. In this section we discuss what is arguably the currently most complete visualization: the visualization of the AspectJ Development Toolkit. We provide a brief evaluation of how this visualization performs with respect to the principles we previously treated. To the best of our knowledge, this is the first time that such an evaluation has been performed, and we perform it here to provide context for the remainder of this paper.

AspectJ Development Toolkit The AspectJ Development Toolkit (AJDT) for Eclipse [7, 9] is arguably the most mature toolkit for Aspect-Oriented Programming. 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 [23]. The gutter markers only show one extremely fine-grained view. When looking at the source code of one specific class, the developer can see where aspects apply in the code currently being displayed, and only there. To look at large classes requires multiple scrolling operations, to view multiple classes requires opening their code in the editor one by one. The cross-references view is, in essence, a textual representation. It lists the signatures of methods where aspects apply. It therefore cannot provide the advantages of a good visualization. It is for example impossible to tell at a glance whether an aspect affects a given class. Instead the developer needs to interpret all of the text that is being displayed.

Evaluating the AJDT Visualization AJDT also offers a visualization tool [7], and again it can be argued that this tool provides the most complete visualization for software that uses aspects. In Figure 1 we show the visualization of AJDT applied to the Spacewar example project from AJDT[1]. This visualization is a continuation of the AspectBrowser work by Shonlhe *et al.* [25]. It offers a Seesoft-inspired [16] view 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. An alternative organization (not shown here) is ordering the classes/aspects by package: it shows one bar for each package, by essentially vertically ordering the bars of the classes and aspects of that package. Each aspect in the project is assigned a specific color, and colored stripes represent lines of code affected by aspects. Bars that are black indicate that no aspects apply to the class or aspect. Multiple colors in a stripe show that multiple aspects apply, and the same color repeated that the aspect applies multiple times. Note

[1] We omit the legend, which associates colors to aspects.

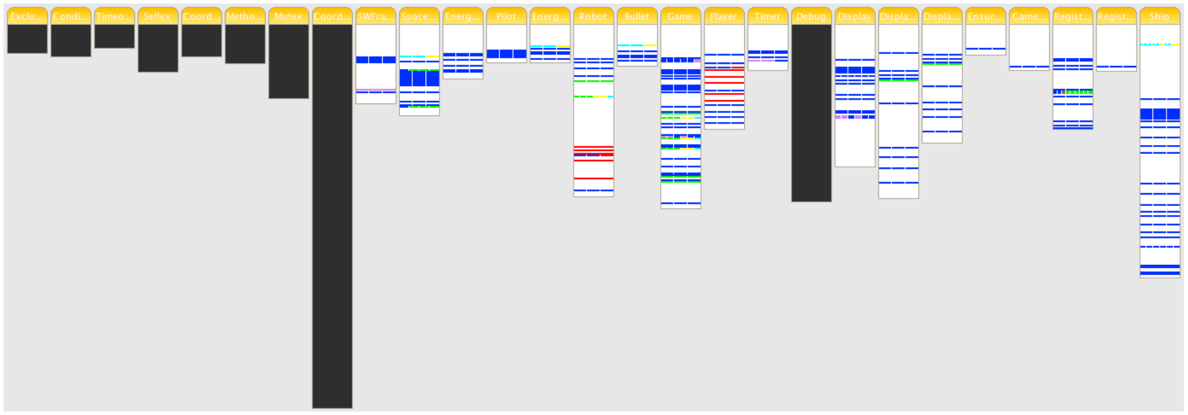


Figure 1. The AspectJ development tools visualization, showing the Spacewar example

that, strangely, sometimes there are more segments for an aspect than the number of times that it applies. By double-clicking on a stripe, the code for that class/aspect is shown with the relevant line highlighted. By then inspecting gutter marks or the cross-references view the user can obtain more detailed information on what advice applies, as well as its nature (before, around, . . .).

Unfortunately however, the AJDT visualization suffers from four of the five pitfalls we discussed above, as we discuss next.

Overly simplistic. First, this visualization is *overly simplistic*. It offers a simple lines-of-code oriented view of the project source code. This does not convey enough relevant information to the user. For example, the visualization does not contain sufficient information to aid a developer in understanding what happens when multiple aspects apply at a single join point shadow. This low-level information can however be vital to a developer: subtle interactions at the join point shadow level can result in erratic behaviour. As we will show in Section 4.2, visualizations can be useful in such cases to help identify such low-level problems.

Furthermore, the visualization fails to show the inherent structure of the code. What we have here is a set of packages that contain classes or aspects, where each class or aspect is subdivided in methods or advices. The first example of structural information that is lacking is what a bar represents: when looking at a bar we are unable to determine whether the visualized entity is a class or an aspect. As a second example of the importance of showing this structure, consider the aspect with the dark blue color. This aspect is called Debug. What we can see here is that it applies in many places in the source code. What is however not immediately obvious is that it applies *at the beginning and at the end of each method or constructor* of classes (amongst other locations). To discover this, for each stripe we need to go to the source code and investigate the gutter marks there. This takes a number of seconds per stripe, so to verify this for all the code is prohibitively time-consuming.

High complexity due to context switching. In general, obtaining any information of an aspect beyond the approximate source code location of its application requires to navigate to the source code representation. This first requires a mental context switch of the user to the source code. Secondly, aspect-related information such as whether the advice is before, after or around, requires looking at additional information that is not shown in the revealed source code. While the visualizer does provide adequate support for navigation, performing all these actions require more time and effort than a simple glance. Therefore it is clearly beneficial to display more information in the visualization itself.

Information Density. Surprisingly, the visualization also suffers from problems of too much *information density* by showing black bars for non-affected classes or aspects. The absence of colored stripes is sufficient to convey this, so the user naturally wonders what the additional meaning of the black color is. Anecdotal evidence of this is that when discussing this visualization with colleagues, this question invariably was one of the first to be posed.

Scalability and Interactivity. The visualization also suffers from the pitfalls of low *scalability* and *interactivity*. 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, which is what the user would expect, resulting in low *interactivity*. Conversely, zooming out does not give a higher level of abstraction on the data, leading to *scalability* problems on large code bases. Hovering over a stripe does produce a pop-up, but this only details the name of the aspects that apply there. This information is already conveyed by the color of the stripe (and legend of the diagram), and therefore this pop-up is useless, not adding any *interactivity*.

Other Aspect Visualization Tools To the best of our knowledge there are only two other tools that provide for a visualization of how aspects cross-cut the code, namely Asbro by Pfeiffer and Gurd [23] and ActiveAspect by Coelho and Murphy [8]. These however do not visualize as much information as the AJDT does. Asbro does not show elements at a granularity finer than classes, and does not reveal the information that multiple aspects apply to one class or package. ActiveAspect does scale down to method level but does not differentiate between multiple aspect applications within the body of a method. As all aspects that apply within one method are gathered together in one visualization element, information is lost. It is for example impossible to see if multiple aspects apply at one line of code, nor to see whether one aspect applies multiple times within the method. Alternatively, ITDVisualiser by Zhang *et al.* [33] and AspectScope by Horie and Chiba [18] provide information on structural modifications and extended module interfaces, respectively. However we do not consider these as proper visualization tools as they use a textual tree-based representation to show the data. More detailed information on this related work can be found in Section 6.

We believe that it is possible to construct a better visualization for aspect-oriented code that does not suffer from the limitations of none of the above mentioned tools. This paper details our attempt to build such a visualization, and we introduce it next.

3. The AspectMaps Visualization

AspectMaps is a visualization that offers users a detailed overview of implicit invocation. It visualizes:

1. where aspects are specified to apply in a system, based on visualizing join point shadows
2. how aspects possibly interact at each join point shadow
3. 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 [20] and Java code. In this case we ignore inter-type declarations as well as advices that applies to fields. The AspectMaps tool is however not restricted to the AspectJ/Java combination, more detail is in Section 3.5.

Following the guidelines of *scalability* and *interactivity* discussed in 2.1, 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.

Scalability: Selective Structural Zoom AspectMaps provides visualization of 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 in turn without needing to zoom in and out.

Scalability: Aspect Identification A second factor that enables scalability is the selection of aspects to be displayed as well as the colors that identify them. AspectMaps implements the *amount of colors* guideline: it by default visualizes the join point shadows of up to 10 aspects simultaneously, each using distinctive colors. This is however not a hard-coded limit. 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 same time (at the cost of more difficult identification).

Scalability: The Fine-Grained Join Point Shadow View 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 can be obtained this way.

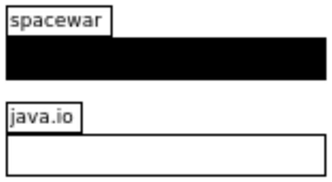


Figure 2. Compact visualization of spacewar, java.io.

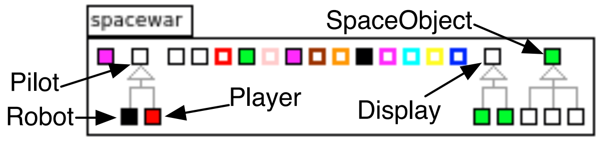


Figure 3. Extended visualization of the spacewar package (annotated with selected class names).

Detailing AspectMaps 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 mention how it follows the guidelines outlined in 2.1. To illustrate the tool we use a number of examples in this section. Specifically, for Sections 3.1 and 3.2 we use the Spacewar example from AJDT where we visualize the aspects Coordination in green, EnsureShipIsAlive in red and Debug in blue. In section 3.3, we use an additional artificial example, as Spacewar does not suffice to show all AspectMaps features.

3.1 Package Level

When opened, AspectMaps provides an overview of all the packages in the system. At this level 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 Figure 2, which shows two packages, named spacewar and java.io. Multiple aspects apply in the spacewar package. The contents of packages, be it classes or aspects 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 Figure 3 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 [22], which we extended with support for the visualization of aspects. Polymetric Views display entities as boxes and box dimensions 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 Figure 3, 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 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.

3.2 Class and Aspect Level

The visualization at this level is similar to that at the package level. Here for classes instance variables 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. Method squares have a gray border to more easily distinguish them from classes, decreasing *complexity*. Pop-ups of instance variables reveal their name and type.

For aspects, the advices are 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 advices show the line number and signature, and pop-ups for pointcuts show the name of the pointcut.

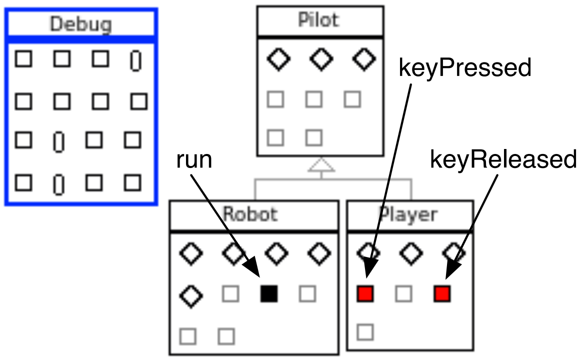


Figure 4. Pilot hierarchy and the Debug aspect.

In Figure 4 we show a zoomed in view 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.

3.3 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.

If we consider only one join point, an 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 advices 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.

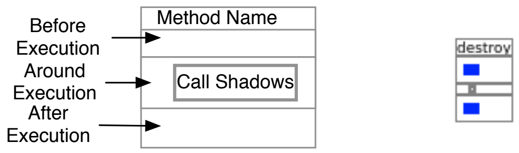


Figure 5. Template for visualization of execution join point shadows (left), and an example destroy method (right). Figure 7 shows call visualization.

Figure 5 shows a template for the visualization of method execution join point shadows. On the right, the destroy method is displayed using this template: we see that the blue aspect (Debug) applies before and after. (We detail call visualization in Section 3.3.3). The figure shows how AspectMaps provides the method name and shows before, after and around advice in their separate divisions. We detail next how advice execution within such a division is visualized.

3.3.1 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. Figure 6 shows two examples of this.

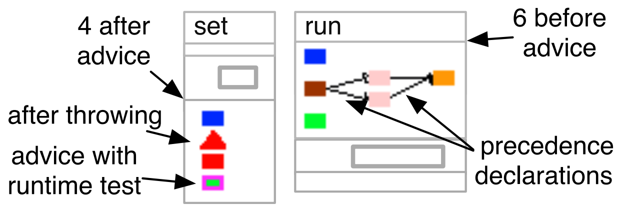


Figure 6. Example after and before execution advices on a set and run method, respectively.

If multiple advices of one aspect apply, for each of these a figure is drawn. In the example this occurs for the red aspect (left) and light red aspect (right). AspectMaps currently has two kinds of figures: a triangle for after throwing advice and a rectangle for all other advices. 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 cflow pointcut) the figure has a thick border in a contrasting color. In the example this is evident in the green aspect (left). This allows easy identification of advices 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 advices apply at the same join point shadow, the order of their application may be specified by the programmer, *e.g.*, using the declare precedence construct in AspectJ [20]. When such an order is specified, AspectMaps indicates this by drawing an arrow between the advice execution figures that indicates the order in which the advices will be executed, as well as attempting to layout these figures in a horizontal sequence[2]. This increases *information density* and maintains a good *mapping to reality*. An example is shown in Figure 6 (right). Here the brown code is run before the two light red advices, and then the orange advice code is run. There is no ordering specified for the blue and green aspects, hence no arrows are drawn and no claims can be made about the order in which these advices will be executed.

Note that AspectMaps shows the order of execution of advices, 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*.

3.3.2 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 in Figure 5, and illustrated in Figure 6.

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.

3.3.3 Call Join Point Shadows

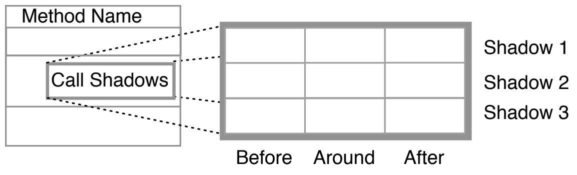
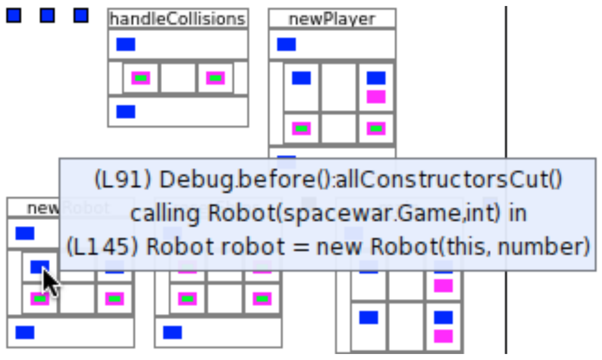


Figure 7. Template for call join point shadows

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 Figure 7. 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 Figure 8 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 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*.

[2] As advice ordering is a partially ordered set, a one-line layout is not always possible.



(L91) Debug.before():allConstructorsCut()
calling Robot(spacewar.Game,int) in
(L145) Robot robot = new Robot(this, number)

Figure 8. A selection of methods in Spacewar showing call join point shadows, as well as a pop-up of one advice execution (in the newRobot method of Game).

3.3.4 Summary: Method Level Visualization

At method level, AspectMaps provides a visualization that shows both call and execution join point shadows at that method, concisely visualizing a large amount of data. For each shadow it shows the execution of before, after and around advices, as well as the order of execution. Lastly, for each advice execution it indicates whether the pointcut depends on a run-time value, as well as highlighting after throwing advice.

3.4 Quick Zoom Options

Tools implementing the AspectMaps visualization are expected to also provide support for the developer by implementing a number of quick zoom options as well as context-specific zoom options. Our implementation, which we discuss in Section 3.5, features all of these zoom options.

The following predefined zoom operations should be provided:

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 being 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 should be supplied, typically contained in a right mouse button menu:

On pointcuts revealing all the join point shadows.

On advices 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.

3.5 The AspectMaps Tool

The AspectMaps visualization is also implemented as a tool with the same name. This tool is built on top of Moose [14], a platform for software analysis and reverse engineering. As a result, it is

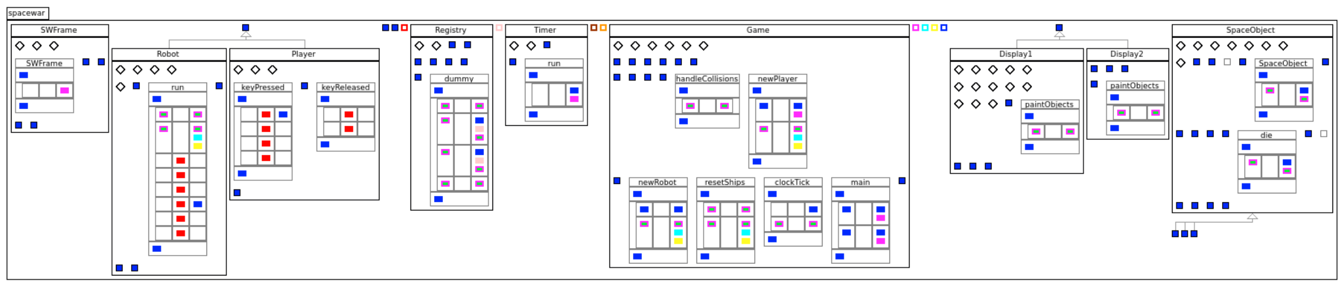


Figure 9. AspectMaps visualization of Spacewar, fully zoomed in on all join point shadows, except for those of the Debug aspect.

relatively independent from the programming languages used both for the base code as well as for the aspect code. AspectMaps does not consider the actual source code of the program, but instead uses its own model. This model is an instance of the ASPIX meta-model, a new member of the FAMIX meta-model family [14] which we implemented for the AspectMaps tool. ASPIX (ASPects in famIX) consists of a generic class-based object-oriented meta-model enriched with information of join point shadows, advice ordering and aspect structure. The idea is to have language-specific importers that generate model instances. These can *e.g.*, obtain the required information from the source code.

The AspectMaps tool currently only has one importer for a base and aspect language combination: Java and AspectJ. We have first implemented an Eclipse plugin that exports join point shadow information as generated by AJDT, as well as structural information about the aspects. The importer imports an Eclipse project into the AspectMaps tool as follows: First the source code of the project is parsed and the corresponding FAMIX model is generated. Secondly the join point shadow and aspect structural information is imported, extending the FAMIX model into an ASPIX model that is subsequently visualized in the AspectMaps tool. Currently the import process needs to be set in motion manually. Automating this, which would result in 'live updates' of the visualization on each compile is a straightforward implementation task. Similarly, further Eclipse integration *e.g.*, presenting the visualization as a pane within the Eclipse development environment, is an implementation effort that does not impact the AspectMaps visualization itself. As a first report on the AspectMaps visualization, this paper solely focuses on the concepts of the visualization. Constructing a tool with full Eclipse integration is out of the scope of this paper, and we leave this as future work.

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 pre-loaded.

4. Program Understanding with AspectMaps

To show how AspectMaps aids in software development and maintenance activities, we show three examples where we use AspectMaps on Java and AspectJ code and conclude with a user study. As a quick foretaste of AspectMaps diagrams compare Spacewar in AJDT, shown in Figure 1, with Spacewar in AspectMaps, shown in Figure 9. Note that for a fair comparison, the scale of both figures is the same.

In this section we use AJDT as a base of comparison with AspectMaps because, to the best of our knowledge, AJDT is the only visualization that provides the same amount of information as AspectMaps. Other visualizations fall short on various points, as indicated in Section 2.2 and discussed in more detail in in Section 6. Therefore we consider AJDT as the only other tool that can be used as a basis for a fair comparison.

4.1 Case 1: Unintended Join Point Capture

We first show how AspectMaps allows the developer to avoid the typical AspectJ pitfall of infinite loops due to unintended join point capture [4]. Consider the following scenario: a payroll application is being developed for a large organization. The payroll database is replicated over multiple redundant data warehouses.

Consequently, database objects are accessed over the network and a networking package is developed, implementing all database networking operations. Persistent objects are required to implement the dummy Persistent interface, and their state may only be accessed and modified through getter and setter methods. An aspect named TransparentProxy is created. It intercepts all getter and setter accesses to database objects, and hands these to the network package, using the execution(public * Persistent+.get*(..)); pointcut.

Visualizing the code in AspectMaps at package level view immediately reveals a suspicious situation. We see that the network package itself is colored with the TransparentProxy color (dark blue). In other words, the network proxy code itself will be intercepted by the aspect that redirects the call to the network proxy. This may lead to an infinite loop. Opening the network package, we establish that one Config class is the culprit. We examine the popups of its methods, an example of which is shown in Figure 10. This reveals that it contains the *persistent* configuration settings that are used to establish a network connection. Therefore making a network call is intercepted by the aspect, which leads to a network call being made, which leads to an infinite loop.

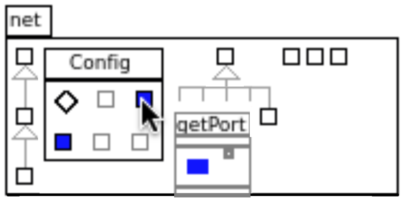


Figure 10. The cause of an infinite loop.

Considering the same scenario, the AJDT visualization tool does not permit such immediate feedback in all cases. The standard visualization only shows package information as a pop-up. The user needs to scrub over all bars, wait for the pop-up to appear, and read the text of the pop-up. The package view is a better visualization for this case, but it still falls short. Firstly the package names are not shown in full in the bars, which requires the user to again scrub, looking for the right package. Secondly, in larger packages all classes of the package quickly fail to fit on one screen, requiring a scrolling operation of the user.

To summarize: in AspectMaps a quick glance is sufficient to raise suspicion, and further exploration quickly reveals the nature of the problem. The AJDT visualization tool needs much more manual intervention before suspicion can be raised.[3]

4.2 Case 2: Aspect Interactions

The second evaluation of the use of AspectMaps considers interactions between aspects as it is deemed an important research challenge for the future of Aspect-Oriented Software Development [6]. More specifically, we focus on the execution of multiple advices at one join point shadow. Dependencies and interactions with aspects is a large and complex area, and we do not claim that AspectMaps addresses all of the issues. Instead we show that in some cases the use of AspectMaps allows one to quickly understand interactions at a given join point shadows.

In the payroll application above, three more aspects are added: a Timing aspect for timing all network operations, a Logging aspect for logging selected network operations, and a RepStats aspect that gathers statistics of replication operations. In the design phase it is determined that these aspects, in some cases, will apply at the same call join points. A precedence order is determined: Logging should be performed at the end of the call, and Timing should include the work of RepStats. If the precedence order is omitted, the system will behave erroneously.

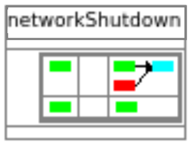


Figure 11. AspectMaps reveals a missing precedence declaration

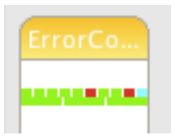


Figure 12. No precedence is shown in AJDT.

Using the interaction zoom button (see Section 3.4) of AspectMaps to visualize join point shadows where all three aspects apply, such an omission is immediately clear. We show this in Figure 11. It shows one method with two relevant join point shadows, where Timing is in green, Logging is in cyan, and RepStats in red. At the second join point shadow only Timing applies, while at the first all three aspects apply. The precedence order should be one line of red, green and cyan rectangles with precedence declaration arrows. This is clearly not the case. Instead the visualization shows us that there is a precedence declaration between timing and logging, and a second precedence declaration between statistics gathering and logging. Investigating the source code of all the aspects, we can see that the precedence declaration for RepStats is missing.

Performing the same analysis with the AJDT visualization tool is simply impossible, as revealed in Figure 12. The only information that the tool gives us is that the Timing, Logging and RepStats apply at one join point shadow. It segments a stripe in a seemingly random number of green, red, and blue parts, in an unclear order. We need to navigate to all the different aspects and examine all their source code to build a mental map of precedence ordering.

Remark that with AJDT we must investigate *all the aspects in the system*. The reason for this is that AspectJ allows a precedence declaration for two aspects to be declared in *any* aspect in the system, not solely in the affected aspects. We therefore cannot restrict our investigation to the aspects involved in the precedence relation. Lastly, this investigation process is made even more time-consuming as the visualizer does not provide a means to navigate to the source code of an aspect. We consider it unlikely that such an investigation will be carried out by developers. This would be especially the case in a large application with multiple development teams, where no developer has the overall picture of aspect precedence, and responsibility for finding these issues may not be clear cut. This will probably cause the problem we show here to be found only in the testing phase, if at all.

To summarize: AspectMaps immediately gives the developer an insight in the execution order of aspects. The AJDT visualizer does not give such information. This requires a whole-source analysis by the developer to obtain this information, which is an unlikely scenario in large applications.

[3] Note that the AJDT crosscutting view does not provide any relief here either as it fails to show the package names of the affected classes.

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 14. The code comprehension questions.

4.3 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 indicated in Section 2.2, and discussed in more detail in Section 6. 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 Figure 14. 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. Question three and four concern aspect interactions, with question four stressing scalability issues for when multiple aspects apply. Question five is a straightforward scalability question considering a large amount of join point shadows. Considering other visualization tools, they do not provide enough information to answer all of these questions: Asbro [23] cannot be used to provide answers to *any* of the questions, ActiveAspect [8] cannot provide answers to questions 2, 3 and 4.

The user study was performed on 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 and have at least basic knowledge of AOSD. To introduce them to AspectMaps the subjects were presented with a preprint of Section 3 of this paper and were shown the screencasts on the AspectMaps website. Also they were given the paper that explains the AJDT visualization [7]. Lastly, before being given tasks to perform using the tools, they had five minutes to familiarize themselves with the tools, asking questions to the authors 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	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 13. User survey results. Global is total time, mean accuracy and overall tool evaluation questions.

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 Figure 13, giving the average result of all participants for each entry in the table[4]. 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 clearly evaluate 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.

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[10]). 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 Figure 14) can be perceived as artificial and tailored towards demonstrating superiority of our approach. Each task however is grounded in a realistic setting and aims at generalizing a typical comprehension scenario. The first question simulates the setting where a new developer needs to get to know the application, and wishes to focus on the cross-cutting concerns. The second question 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 [19, 21]. Question three addresses the issue of interactions between aspects. 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 with a specific focus on scalability: testing for a large number of aspects and affected classes. The wording of question five is arguably the most artificial, as it is a straightforward scalability question. It can however 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.

5. 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 [11], which consists of 374 classes and 31 aspects. Note that the AJDT visualizer is unable to process this example: it crashes on start-up. A full report of the exploration of the AJHotdraw code is outside of the scope of this paper.

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

One striking property of the AspectMaps visualization, especially at the most fine-grained zoom level, is the "boxes within boxes" syndrome which could confuse the user. While it might seem that we have superfluous boxes here, we strictly adhere to the *information density* data-ink rule [31]: every box has a specific meaning. The syndrome is due to the deep nested structure we are showing: the most inner boxes are call join point shadows, located within a method body, located within an around advice, located within a method, located within a class, located within a package. Not visualizing boxes would mean hiding structural information. We have instead chosen to mitigate the syndrome and reduce *complexity* through two strategies: the use of color and the ordering of boxes. All structures at method level and below have a gray border, above method level a black border. Execution join point shadows are shown in a vertical order, while call join point shadows are shown horizontally. None of the user study subjects reported problems with understanding the visualization due to the "boxes within boxes" syndrome, therefore we believe that this is not an issue.

In AspectJ a precedence declaration for two or more aspects may be specified in a totally unrelated aspect. As we have mentioned in Section 4.2, this means that to know the precedence of any given two aspects, the developer needs to investigate all the source code of all the aspects in the system. Using AspectMaps this question is not only resolved much more quickly but also less prone to errors. This is shown by the results of question 4 in our user study, which is a testament to the power of a good visualization.

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 as future work.

Currently, the AspectMaps tool is not integrated into any development environment, running instead in a stand-alone fashion. As mentioned in Sections 3.5 and 4.3, 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 straightforward implementation task that we consider as future work.

A last limitation of AspectMaps we discuss here is the limits of the Java and AspectJ importer. Due to our reliance on a source code importer for FAMIX, we only are able to parse fully compliant Java source code (and not able to import java files that also include AspectJ code). As for our Eclipse plugin, it does not yet provide information on structural modifications made by the aspects, also known as static cross-cuts or inter-type declarations. As a result, the visualization does not show inter-type declarations, nor the aspects that apply there. Secondly, the plugin does not provide all information on the internal structure of aspects, it omits the methods and attributes that they contain. Consequently, there is no complete visualization of aspects, nor any visualization of whether aspects apply within other aspects. As the above features are orthogonal to the core visualization concepts of AspectMaps we have not yet implemented support for this, and leave this as future work.

6. Related Work

Arguably the most complete tool suite for aspect-oriented programming is the AspectJ Development Toolkit [9]. We discussed this toolkit in 2.2, and used it as a basis for comparison with AspectMaps. AJDT is a tool suite for AspectJ in the Eclipse IDE. 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 [23] 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 [8]. 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, as we have detailed in Section 2.2, 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 [33]. 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 [18] 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 [15]. 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.

7. 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 [8, 9, 23]. 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 avoids the common visualization pitfalls, discussed how it improves on existing work, and applied it to two example case studies. 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 not only on the amount of time required to perform each task, but also on the users opinion of which tool is better and their willingness to use the tool again for similar tasks in the future.

Downloads, Additional Information

The AspectMaps tool is available on the AspectMaps website `http://pleiad.cl/aspectmaps`. This page also contains more information, including screencasts and the complete user study results.

Acknowledgments

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 helping out with Mondrian. We thank the user study participants for their time and effort. We are grateful to Theo D'Hondt for supporting this research. This research is supported by the IAP Programme of the Belgian State.

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] Aspectj programming guide, chapter 5: Pitfalls. http://www.eclipse.org/aspectj/doc/released/progguide.

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

[6] Ruzanna Chitchyan, Johan Fabry, Shmuel Katz, and Arend Rensink. *Transactions on Aspect Oriented Software Development V*, volume 5490 of *LNCS*, chapter on Dependencies and Interactions with Aspects. Springer Verlag, 2009.

[7] Andy Clement, Adrian Colyer, and Mik 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.

[8] Wesley Coelho and Gail C. Murphy. Presenting crosscutting structure with active models. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 158–168, New York, NY, USA, 2006. ACM.

[9] Adrian Colyer, Andy Clement, George Harley, and Matthew Webster. *Eclipse aspectj: aspect-oriented programming with aspectj and the eclipse aspectj development tools*. Addison-Wesley Professional, 2004.

[10] 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)*, pages 100–109. IEEE Computer Society, 2009.

[11] Arie Van Deursen. Ajhotdraw: A showcase for refactoring to aspects. In *In: Workshop on Linking Aspect Technology and Evolution. (2005*, 2005.

[12] 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)*, pages 33–38. IEEE Computer Society, 2005.

[13] 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*, pages 94–103, Oct. 2007.

[14] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, and Lukas Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 8(1):5–19, February 2009.

[15] Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006.

[16] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, 1992.

[17] Wilke Havinga, Istvàn Nagy, and Lodewijk Bergmans. Introduction and derivation of annotations in AOP: Applying expressive pointcut languages to introductions. In *First European Interactive Workshop on Aspects in Software*, 2005.

[18] Michihiro Horie and Shigeru Chiba. Aspectscope: An outline viewer for aspectj programs. *Journal of Object Technology, Special Issue: TOOLS EUROPE 2007*, 6(9):341–361, October 2007. http://www.jot.fm/issues/issue_2007_10/paper17/.

[19] 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)*, number 4067 in LNCS, pages 501–525, 2006.

[20] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

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

[22] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–796, September 2003.

[23] J.-Hendrik Pfeiffer and 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*, pages 146–157, New York, NY, USA, 2006. ACM.

[24] 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.

[25] Macneil Shonle, Jonathan Neddenriep, and William Griswold. Aspectbrowser for eclipse: a case study in plug-in retargeting. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 78–82, New York, NY, USA, 2004. ACM.

[26] J. Stasko, J. Domingue, M.H. Brown, and B.A. Price, editors. *Software Visualization - Programming as a Multimedia Experience*. MIT Press, 1998.

[27] Margaret-Anne D. Storey, Kenny Wong, F. D. Fracchia, and Hausi A. Müller. On integrating visualization techniques for effective software exploration. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis '97)*, pages 38–48. IEEE Computer Society, 1997.

[28] Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Müller. How do program understanding tools affect how programmers understand programs? In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 12–21. IEEE Computer Society, 1997.

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

[30] E. Tufte. *Envisioning Information*. Graphics Press, 1990.

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

[32] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[33] 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.