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

Stéphane Ducasse
RMoD Team
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.

This paper makes heavy use of colors in the figures. Please obtain and read a color version of this paper to better understand the ideas presented here.

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 pointcut and advice, where each advice is associated with one pointcut.

The concepts of pointcuts 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, when writing pointcut definitions, it is not always clear for a developer where the aspect will intervene in the base code. This can then lead to situations where the pointcut either captures too many join points (false positives), or where certain join points that were intended to be captured, are not (false negatives). One variant of this problem is the so-called *fragile pointcut problem* [15, 17]. It states that seemingly innocent changes of the base code can lead to unintended and erroneous behaviour. Since it is easy to lose track of the global picture of how the base code and the aspects interact, it can become difficult to identify the source of such unanticipated behavior. A prominent example is found in the work of Coelho *et al.* [8]. The authors investigated a number of applications that use aspects to determine erroneous exception handling behavior. They found that even in applications where the developers are experts in the use of aspects false positives and false negatives occur. Specifically"mistakes on [sic.] pointcut expressions" were found in the Health Watcher [21] and the Mobile Photo [14] application, which are both well-known case studies for AOSD.

Another similar problem that may arise is that in complex systems with lots of aspects, 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.

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.

In this paper we present a novel approach based on software visualization to aid the understanding of aspect-oriented software systems. Our tool is called *AspectMaps*, and it provides a scalable visualization of implicit invocation. AspectMaps visualizes selected join point shadows (a.k.a. sha-

[*]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)

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'10, Submission.
Copyright 2010 ACM TBD ...$5.00.

dow points): locations in the source code that at run-time produce a join point. AspectMaps visualizes the shadow points 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 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.

The remainder of the paper is structured as follows: we next give an overview of software visualization, detailing typical pitfalls as well as discussing existing work on 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 three case studies. We provide a discussion and avenues for 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 [22]. 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 [23, 24, 18].

One of the major advantages of software visualizations is that they are able to convey a large quantity of information to a user [28]. The human brain can easily combine complex information from visual cues, making visualizations a suitable means for understanding complex software systems. Furthermore, a well-chosen visualization allows users to pre-attentively process the visual 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 [6, 27], 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 [6, 26, 28]. 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 [10].
- **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 [11].
- **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 [27]. 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 [6].
- **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 [25].

2.2 Existing Aspect Visualizations

We are not the first to study the subject of aspect visualization. In this section, we discuss previous work and illustrate how they can be improved upon.

AspectJ Development Toolkit.

Description. The AspectJ Development Toolkit (AJDT) for Eclipse [4] is arguably the most mature toolkit for Aspect-Oriented Programming.

Amongst other features, AJDT adds gutter markers in the code editor to indicate shadow points 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 [19]. 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

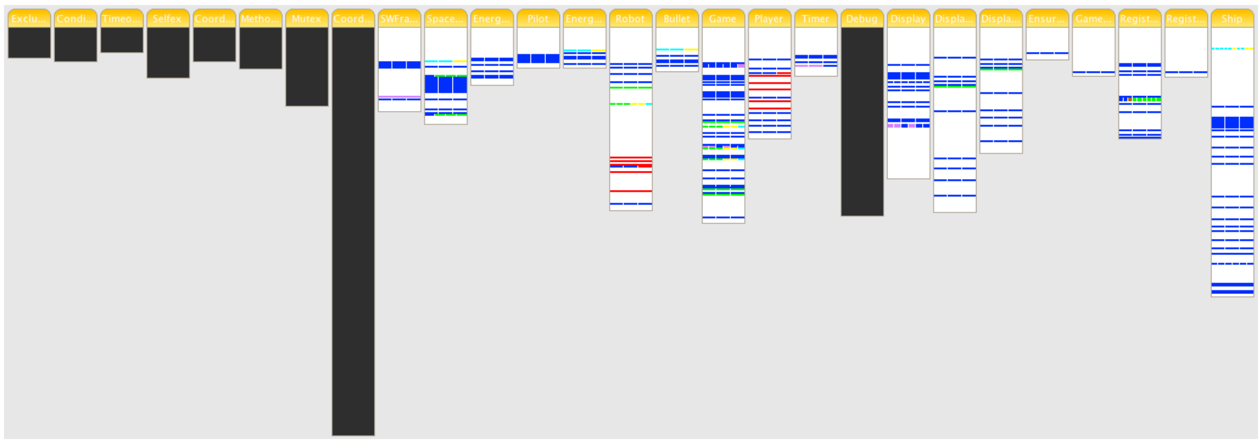

Figure 1: The AspectJ development tools visualization, showing the Spacewar example

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.

AJDT also offers a visualization tool. This tool is a continuation of the AspectBrowser work by Shonlhe *et al.* [20]. In Figure 1 we show the visualization of AJDT applied to the Spacewar project. It offers a Seesoft-inspired [13] 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 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 a number of the pitfalls we discussed above, as we shall 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 shadow point. This low-level information can however be vital to a developer: subtle interactions at the shadow point level can result in erratic behaviour. As we will show in Section 4.1, 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. 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.

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 effect is that when discussing this visualization with various 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 is information already conveyed by the color of the stripe, and therefore this pop-up is useless, not adding any *interactivity*.

Other Aspect Visualization Tools.

To the best of our knowledge there are only three other tools that provide for a visualization of how aspects crosscut the code, namely Asbro by Pfeiffer and Gurd [19], ActiveAspect by Coelho and Murphy [9], and ITDVisualiser by Zhang et al [29]. The two former tools 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 however 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. The third tool, ITDVisualiser, is dedicated to analyzing the impact of structural modifications made by aspects to the base code of the system. It uses a marker mechanism to indicate source-code entities that are affected and provides a number of dedicated views for assessing how method lookup and shadowing are impacted by the aspects. We discuss the above approaches in some more detail 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 neither 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 tool 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. Advice can execute before, around or after a join point, and we distinguish between after returning and after exception throwing. Aspects may contain various advice, and an execution order may be specified between aspects. The above effectively allows us to visualize a subset of AspectJ [16] and Java code. In this case we ignore inter-type declarations as well as advice that applies to fields. The AspectMaps tool is however not restricted to the AspectJ/Java combination, which is discussed in more detail 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 Zooming.

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 zoom level. This allows the user to skim over a number of elements, getting more information of each in turn without zooming in and zooming out on each element.

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 shows up to 10 aspects simultaneously, each using distinctive colors. The user can for each aspect separately select a specific color and turn visualization on or off (visualizing more than 10 if needed).

Scalability: The Fine-Grained Shadow Point View.

AspectMaps also scales down to a very fine level of granularity. At the most detailed zoom level on a shadow point, it shows a wealth of information at a single glance. The user can see the specification of the type 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.

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 where we visualize the Coordination aspect in green and the EnsureShipIsAlive aspect in red. In section 3.3, we use an additional artificially created example, as Spacewar does not suffice to show all the features of AspectMaps.

In addition to the (annotated) figures we show in this paper, a status line at the bottom of the AspectMaps window details textual information of the element the mouse pointer is hovering over, if any. When we wish to discuss this information in the text, we will typeset it [in this form].

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

3.1 Package Level

When opened, AspectMaps provides an overview of all the top-level packages in the system.

Overview.

At this level AspectMaps shows the names of packages as well as indicating which aspects apply in this package. The former is restricted to the last part of the fully qualified name, the complete name is given in the status line. For the latter AspectMaps colors the background of the package name with the color of the aspect that applies, if it is currently enabled for visualization.

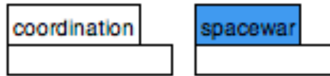

Figure 2: Visualizing packages coordination and spacewar.

If multiple aspects apply in the package this is indicated by using an alternative color (that can also be selected by the user). An example of this is shown in Figure 2, which shows two packages, named coordination and spacewar. Multiple aspects apply in the spacewar package, indicated by the blue color. The contents of packages, be it sub-packages or classes is not shown at this point.

Selective Structural Zooming on Packages.

To view package contents, the user performs a zoom operation on a selected package. In Figure 3 the package spacewar has been zoomed in on, revealing the different classes and aspects that it contains (this package contains no sub-packages).

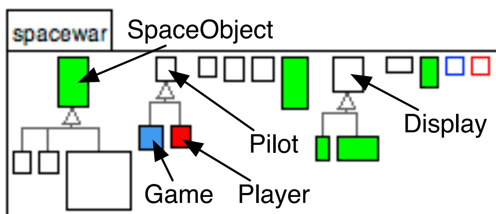

Figure 3: Visualizing the spacewar package, showing its content (annotated with selected class names).

The visualization at class level is a version of the work of Lanza and Ducasse on Polymetric Views [18], which we extended with support for the visualization of aspects. Polymetric Views display entity as boxes and box dimensions reflect entity properties (LOC, number of methods...). A variety of different types of information is shown at this point:

- **Classes:** rectangles with black borders. Inheritance relations are visualized using the standard UML notation. (A conventional *mapping to reality.*)
- **Aspects:** rectangles with colored borders. The border color is the color for the aspect.
- **Where aspects apply:** class rectangles have the color of the aspect that applies, or the multiple aspect color.
- **Class metrics:** the vertical size of the class rectangle is proportional to the number of methods in that class; the horizontal size is proportional to the number of fields in that class. (Increasing *information density.*)

Note that this visualization does not display the names of the classes. This is chosen to avoid clutter, which would increase *complexity*. Class names are instead revealed in the status line when the mouse pointer passes over the respective classes. In Figure 3, we see three class hierarchies with as roots [SpaceObject], [Pilot] and [Display], along with two aspects: [Debug] and [EnsureShipIsAlive]. The Coordinator aspect (in green) is not part of this package but applies in five classes ([SpaceObject], [Game], [Display1], [Display2], [Registry]). In the Pilot hierarchy, multiple aspects apply in the first subclass [Robot], and only the EnsureShipIsAlive aspect (in red) applies in the second subclass [Player].

3.2 Class Level

The visualization at the class level is similar to that at the package level. Here methods are shown as squares with a gray border and are colored according to the aspects that apply (again taking into account the multiple aspect color). Method squares have a gray border to more easily distinguish them from classes, decreasing *complexity.*

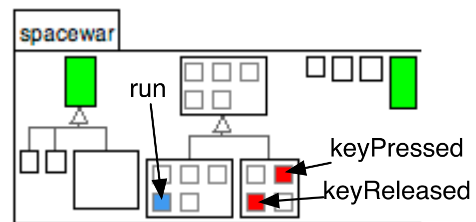

Figure 4: Visualizing selected classes of Spacewar (annotated with selected method names).

In Figure 4 we have zoomed in on all classes in the [Pilot] hierarchy. This reveals that multiple aspects apply on the method [run] of [Game] and that the EnsureShipIsAlive aspects 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 potentially of interest, and hence the visualization is more complex.

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 shadow point needs to separate showing before, after and around advice. Also, at one join point multiple aspects may apply, so the shadow point must be able to show the execution of various advice at that point. Considering the method level, we can have a shadow point for the execution of the method, and within the method body various shadow points for method calls.

Figure 5 shows a template for the visualization of method execution shadow points. On the right, the Robot constructor is displayed using this template: we see that the blue

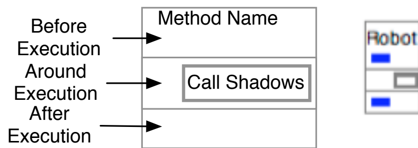

Figure 5: Template for visualization of execution shadow points (left), and an example Robot constructor (right). Figure 9 shows call visualization.

aspect 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 shadow point, AspectMaps draws a small figure in the color of the corresponding aspect. This is done for all aspects that apply, aligning the figures horizontally. Figure 6 shows after execution advice of an example set method.

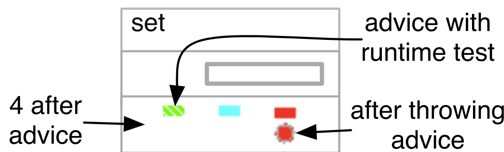

Figure 6: Four after execution advice of a set method.

If multiple advices of one aspect apply, for each of these a figure is drawn and these are stacked vertically.In the example this occurs for the red aspect. AspectMaps currently has two kinds of figures: a circle for after throwing advice and a rectangle for all other advice. This is to emphasize the special nature of after throwing. 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 is hashed. In the example this is evident in the green aspect. This allows easy identification of advice that will always run at this shadow point: these are not hashed. Note that each figure shows three different data points, increasing the *information density*, however without overly increasing the *complexity*.

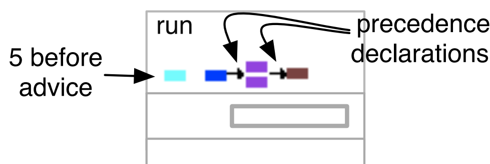

Figure 7: Five before execution advice for a run method, with 2 precedence declarations.

When multiple advices apply at the same shadow point, the order of their application may be specified by the programmer, *e.g.*, using the declare precedence construct in AspectJ [16]. 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[1]. This increases *information density* and maintains a good *mapping to reality*. An example is shown in Figure 7. Here the dark blue code is run before the two purple advice, and then the brown advice code is run. There is no ordering specified between the cyan aspect and any of the other aspects, hence no arrow is drawn. Figure 6 does not show any arrows, indicating no ordering is specified between the green, cyan 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*. Furthermore, this results in AspectMaps not being limited to declaring execution order on aspects, it is able to visualize code where aspect ordering is declared at the advice level.

3.3.2 *Execution Shadow Points*

For execution shadow points the groups of figures detailing advice execution are placed in their corresponding locations as given by the template in Figure 5.

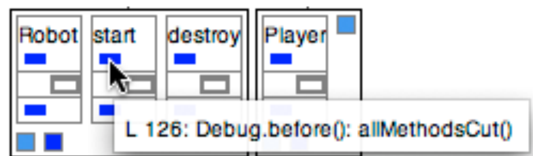

Figure 8: A selection of methods in Spacewar showing before and after execution of the Debug aspect, as well as a popup of one advice execution.

An example of this is shown in Figure 8, which furthermore illustrates the pop-up information given at this level. Recall that up until now, 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. 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 Shadow points*

The body of a method may contain multiple call shadow points, sequentially ordered by the source code of the method. We visualize advice execution in this same order, aligning

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

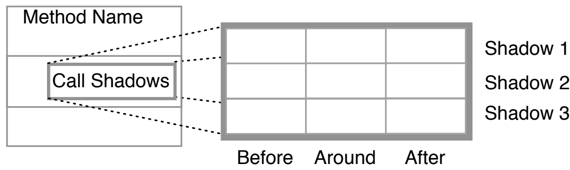

Figure 9: Template for call shadow points

them vertically as a suitable *mapping to reality*. The visualization of call shadow points uses the same visualization as execution shadow points. It however orders the before, around and after divisions horizontally instead of vertically. A template of this is shown in Figure 9. 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 shadow point (decreasing *complexity*).

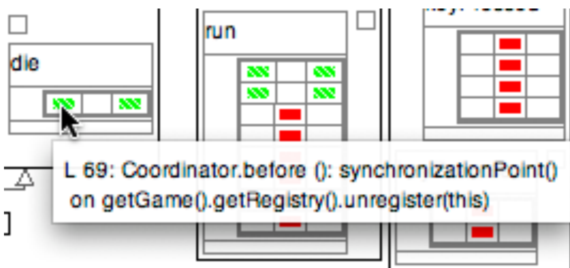

Figure 10: A selection of methods in Spacewar showing call shadow points, as well as a pop-up of one advice execution (in the die method).

In Figure 10 we show a number of methods of the Spacewar example that demonstrate the visualization of call shadow points. This figure also illustrates the pop-up information for each advice execution. It consists of the signature of the advice, including its line number in the aspect source code, as well as the source code for the shadow point (again increasing *interactivity* and *information density*).

3.3.4 Summary

At method level, AspectMaps provides a visualization that shows both call and execution shadow points at that method, concisely visualizing a large amount of data. For each shadow point it shows the execution of before, after and around advice, 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. An example of this is shown in Figure 11, detailing 14 advice executions of six aspects on two shadow points, with three aspect ordering specifications, two run-time tests and one after throwing advice.

3.4 Quick Zoom Options

To aid the developer in quickly visualizing interesting locations in the code, AspectMaps provides a number of predefined zoom operations:

Max Zoom Out. Zooms all elements out to the top level.

Figure 11: A method where many advice apply.

Max Zoom In. Zooms in maximally on all shadow points where an aspect being visualized applies.

Interactions Zoom. Zooms in maximally on all shadow points where multiple aspects being visualized apply.

Query Zoom. Given a query, zooms in maximally on classes or methods of which their names match. (Currently a substring match of the query in the name.)

Each of these zoom operations are performed by clicking on a button in the user interface. The advantage of these zoom levels 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.

3.5 On Language Independence

AspectMaps is intended to be relatively independent from the programming languages used both for the base code as well as for the aspect code. To perform visualization, AspectMaps does not consider the actual source code of the program, but instead uses its own data structures. This data structure consists of a generic object-oriented model enriched with information of shadow points as well as advice ordering. The idea is to have language-specific importers for the model, which obtain the required information from the source code.

AspectMaps currently only has one importer for a base and aspect language combination: Java and AspectJ. The importer uses the Eclipse Java model to extract the OO structure of the source code, and the AspectJ development tools [4] to obtain the shadow point information.

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. In each case we argue why AspectMaps is more effective than the AJDT visualization. As a quick foretaste compare Spacewar in AJDT, shown in Figure 1, with Spacewar in AspectMaps, shown in Figure 12.

Our first example is a case of unintended join point capture causing infinite loops, a typical pitfall of AspectJ [5]. Second we show a case of aspect interactions, deemed an important research challenge of the future of AOSD [7]. Third we perform an analysis of the Spacewar example, similar to code understanding efforts in re-engineering.

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 [5]. Consider the following scenario: a payroll application is being developed for a large

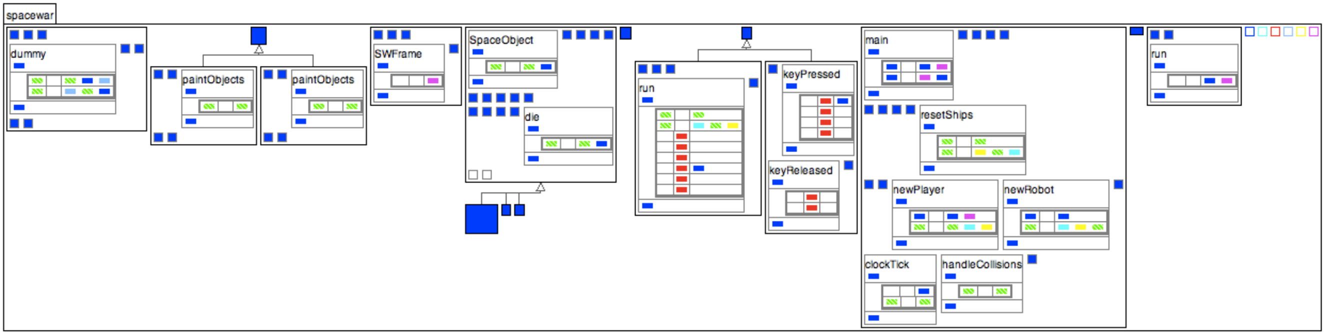

Figure 12: AspectMaps visualization of Spacewar, fully zoomed in on all shadow points, except for Debug.

organization that has multiple physical office sites. Each site is responsible for the payroll of the employees at that site. The payroll database is maintained centrally, replicated over multiple redundant data warehouses.

To resume, database objects are accessed over the network and modifications to these must be broadcast to all data warehouses. A networking package is therefore developed, which implements 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, which intercepts all getter and setter accesses to database objects, and hands these to the network package. For this, it uses the execution(public * Persistent+.get*(..)); pointcut.

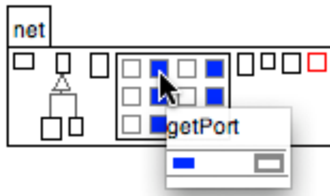

Figure 13: The cause of an infinite loop.

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 pop-ups of its methods, an example of which is shown in Figure 13. This reveals that it contains the configuration settings that are used to establish a call over the network. Therefore making a network call is intercepted by the aspect, which leads to a network call being made, which leads to 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 popup. The user needs to scrub over all bars, wait for the popup to appear, and read the text of the popup. 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.[2]

4.2 Case 2: Aspect Interactions

The second case of the use of AspectMaps considers interactions between aspects. More specifically, we focus on the execution of multiple advice at one shadow point. Dependencies and interactions with aspects is a large and complex area, and we do not claim that AspectMaps is a silver bullet. Instead we show that in some cases the use of AspectMaps allows one to quickly understand interactions at a given shadow point.

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 ought to be determined: Logging should be performed at the end of the call, and Timing should include the work of RepStats. If such a precedence order is omitted, the system might behave in an erratic way.

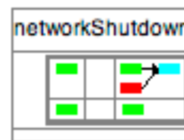

Figure 14: A precedence declaration is missing!

Figure 15: No precedence is shown.

Using the interaction zoom button (see Section 3.4) of AspectMaps to visualize shadow points where all three aspects apply, such an omission is immediately clear. We show this in Figure 14. It shows one method with two relevant shadow points, where Timing is in green, Logging is in cyan, and RepStats in red. At the second shadow point only Timing

[2]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.

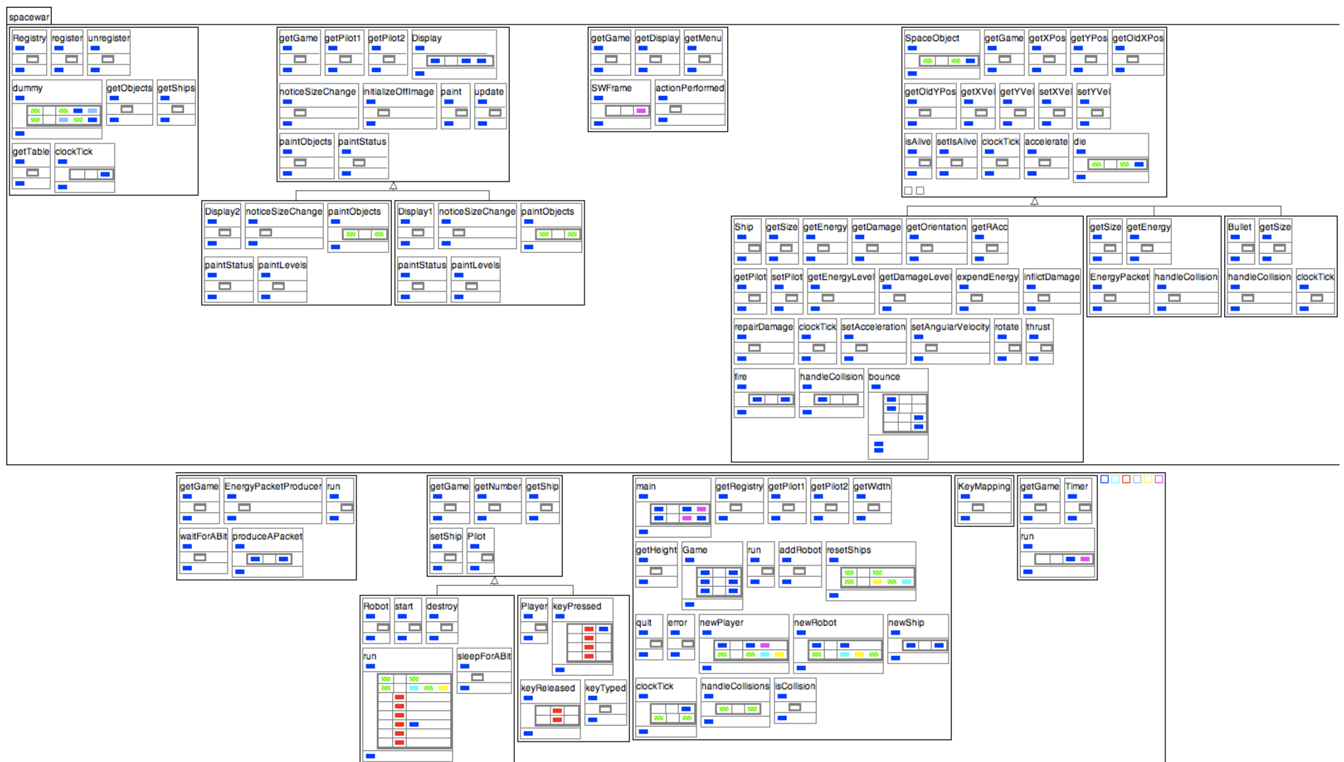

Figure 16: AspectMaps visualization of Spacewar, showing all 329 advice executions. For a fair comparison with the AJDT visualization, the scale is as for Figure 1, and the figure is split in two parts to fit the page.

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 required precedence declaration for RepStats is missing.

Performing the same analysis with the AJDT visualization tool is simply impossible, as revealed in Figure 15. The only information that the tool gives us is that the Timing, Logging and RepStats apply at one shadow point. It segments a stripe in a seemingly random number of green, red, and blue parts, in a random order. We need to navigate to all the different aspects and examine all their source code to be able to build a mental map of the 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 one 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 any information. This requires a whole-source analysis by the developer to obtain this information, which is an unlikely scenario in large applications.

4.3 Case 3: Understanding Existing Code

As we have indicated in Section 2, software visualization is a well-established means for program comprehension, for example in reverse engineering or re-engineering efforts. 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. To contrast the abilities of AspectMaps and the AJDT visualization, compare Figure 1 and Figure 16. Both show the same code, zoomed in to show maximum detail, revealing 329 advice executions. It is clear that Figure 16 (of AspectMaps) provides us with much more information than Figure 1 (of AJDT). For example, we can see that the Debug aspect (in dark blue), applies at the beginning and at the end of each method and constructor declaration. This is impossible to ascertain when looking at Figure 1. Note that, while we show much more information, achieving a higher *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 Figure 16, four interesting elements are further revealed by AspectMaps:

1. No precedence declarations have been declared. We

find this remarkable as there is interaction between theCoordinator aspect (in green), and other aspects. In other words, advice of other aspects executes at a number of coordination points. However there is no specification of whether this behavior should be also be coordinated or not.

2. 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, where the line number given is specific for each method.

3. The EnsureShipIsAlive aspect (in red) applies only in the [Pilot] hierarchy. Only one piece of 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.)

4. At every shadow point where SpaceObjectPainting1 applies (in yellow), SpaceObjectPainting2 (in cyan) also applies.

Note that none of the first three observations can be made using only the AJDT visualization tool, *i.e.*,the user will need to inspect the source code. In contrast, all the above information has been obtained solely through AspectMaps, *i.e.*,without looking at the source code.

Considering *scalability*, Figure 16 uses more than the double of screen real estate, making it impossible to visualize on one screen. We however envision AspectMaps to be used differently: using a different zoom level for different elements, depending on the focus of the developer. For example, we can ignore the details of the Debug aspect and solely focus on the remaining aspects in the system, only zooming in where these apply. This can be done using the max zoom in button and results in Figure 12. This figure is smaller in size than Figure 1, and easily fits on a laptop screen. Nonetheless it still yields the same four observations as Figure 16. This attests to the scalability of the AspectMaps visualization.

5. DISCUSSION AND FUTURE WORK

One striking property of the 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: every box has a specific meaning. The syndrome is due to the deep nested structure we are showing: the most inner boxes are call shadow points, 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 shadow points are shown in a vertical order, while call shadow points are shown horizontally.

Performing a full-fledged validation of a development tool requires a user study demonstrating that development tasks benefit from the use of the tool. Unfortunately we currently do not have enough resources at our disposal to perform such a validation thoroughly. Instead, in this paper, we have continuously evaluated AspectMaps with regard to established guidelines for visualization tools, showing how AspectMaps follows these guidelines. Also, we have provided a number of case studies that illustrate the effectiveness of AspectMaps for different usage scenarios. Lastly, for these case studies we have demonstrated that AspectMaps is a sizable improvement over existing visualizations for aspect-oriented code. Nonetheless, an avenue for future work is performing a user study, to give us deeper insights in how the tool is used and how it can be improved.

One contribution of a user study could be about the displaying of metrics at the class level view. Now, in the package level view the visualization of a class encodes two metrics: the height reveals the number of methods, and the width the number of fields of the class. It is possible to do the same for methods, at the class level visualization. For example we could have the method height reflect the length of the method and the width the number of shadow points. We found however that this renders the visualization too confusing, increasing *complexity* too much without significantly increasing *information density.* We therefore made a conservative choice to not show these metrics, to reduce confusion. A user study could reveal if this choice is correct.

We were surprised to discover that 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 manually investigate all the source code of all the aspects in the system. Using AspectMaps this question is instead resolved in a single glance, which is a testament to the power of a good visualization.

Further 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 shadow points. This weakness is visualization of interactions at method call and method execution shadow points. Consider for example the pointcuts call(* * AClass.aMethod()) and execution(* * AClass.aMethod()). The shadow points 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 however not yet encountered a suitable visualization for this, and consider this as future work.

A last limitation of AspectMaps we discuss here is the limits of the Java and AspectJ importer. Due to its reliance on the Eclipse model of the Java code, it does not provide information on structural modifications made by the aspects, also known as static cross-cuts or inter-type declarations. As a result, the diagram does not show inter-type declarations, nor the aspects that apply there. Secondly, the importer does not provide information on the internal structure of aspects, *e.g.*,the methods that they contain. Consequently, there is no class level or method level visualization of aspects, nor any visualization of whether aspects apply within other aspects. As this feature is orthogonal to the core visualization concepts of AspectMaps we have decided to not yet implement 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 [4]. 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 [19] propose a visualization tool that is based on the concept of Treemaps. A Treemap maps the nodes of a tree to rectangles in a plane, using a space-filling layout. In contrast to graph-bases layouts of tree nodes, this does not waste any screen space. The tool proposed in [19] 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 to a fine granularity: it does not reveal aspect application at finer levels than types. Furthermore, it does not provide any information of aspect interaction at a given shadow point. 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 [9]. 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 UML extended 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 has full control over 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 shadow point.

Zhang et al. [29] present an analysis toolkit for assessing the impact of structural modifications through AspectJ inter-type declarations on the behaviour of the system. Due to the inherent obliviousness of such declarations, it can become increasingly difficult for a developer to understand how a program will behave. They propose analyses to assess how the declarations impact the method lookup of the base program, and to identify how particular base-code entities are shadowed by inter-type declarations. To present the results of the analyses to a developer, an integration with Eclipse is offered by means of visual clues (markers) and dedicated views that represent the lookup impact and shadowing impact. This approach is complementary to ours: AspectMaps focusses on the visualization of shadow points while ITDVisualizer aids in comprehending inter-type declarations.

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 [12]. 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 cross-cutting properties like code owners, commits, symbolic information. As a result it can 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 [4, 19, 9]. However all of these are victims of common visualization pitfalls, 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 three example case studies. Most noteworthy here is the demonstration that AspectMaps does scale from a fine-grained level up to a large code base.

Additional Information

More information, including some screencasts, is available on http://pleiad.cl/research/software/aspectmaps

Acknowledgments

We wish to thank Éric Tanter, Jacques Noyé, Alexandre Bergel, Awais Rashid, Thomas Cleenewerck, Kris De Schutter, Kim Mens, Simon Denier and Andrew Eisenberg for their invaluable feedback when discussing early versions of AspectMaps. Thanks also to Andrew Eisenberg for helping us understand the AJDT crosscutting model. We are

grateful to Theo D'Hondt for supporting this research. This research is supported by the IAP Programme of the Belgian State and the SticAmSud project CoReA.

8. 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] AJDT: Aspectj development tools. http://www.eclipse.org/ajdt.

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

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

[7] 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.

[8] R. Coelho, A. Rashid, A. Garcia, F. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 207–234, 2008.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Eduardo Figueiredo, Nelio Cacho, Claudio Sant'Anna, Mario Monteiro, Uira Kulesza, Alessandro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Castor Filho, and Francisco Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 261–270, New York, NY, USA, 2008. ACM.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] Sérgio Soares, Paulo Borba, and Eduardo Laureano. Distribution and persistence as aspects. *Softw., Pract. Exper.*, 36(7):711–759, 2006.

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

[23] 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.

[24] 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.

[25] 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.

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

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

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

[29] 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.