Design Decisions in AspectMaps

Johan Fabry, Alexandre Bergel
PLEIAD Laboratory, Computer Science Department (DCC)
University of Chile, Santiago, Chile
http://pleiad.cl

***Abstract*—AspectMaps is a visualization that shows the structure of aspectual source code. In its design and implementation we made a number of design decisions that we present and discuss in this text. This in the light of more than two years of using, extending and maintaining the AspectMaps visualization and tool. The purpose of this paper is to share our experience with other visualization designers and implementers, as an aid in the making of their design decisions.**

Keywords—Aspect-Oriented Programming, Visualization Design Decisions

I. Introduction

To tackle the modularity issues of cross-cutting concerns, Aspect-Oriented Programming [1] proposes a new kind of module: an aspect. Aspects are structural entities, like classes, with the difference that they also specify the invocation conditions for their behavior. Software developed using aspects automatically calls the aspect behavior whenever the invocation conditions are satisfied. Unfortunately, traditional text-based code editors are suboptimal with regard to programs with aspects, as a simple inspection of the source code only reveals a small portion of the program behavior and intent. This is because the invocations to the behavior of the aspect are not explicit and straightforwardly visible.

A number of aspect-aware software visualizations have been developed previously (which we discussed in [2]). However, these suffer from multiple drawbacks, most importantly is a lack of scalability and a limited way to navigate within a possibly large amount of source code. To address the issues with these visualizations, we have developed AspectMaps [2], [3]: a source-code visualization and tool for aspectual source code. AspectMaps scales up to relatively large pieces of software as well as down to fine-grained detail in the source code, revealing the detailed interactions that take place at that level.

In this text we reflect on design decisions of salient points of the visualization and its implementation. This based on the experience we gained of its use, extension and maintenance since its inception over two years ago. This is in contrast to previous work on AspectMaps, which focused first on the presentation of the visualization itself [2], and then showing the use of the tool and outlining its construction [3]. The contribution of this text lies in the documentation of five design decisions and their discussion in the light of our experience.

This paper is structured as follows: We now first give a brief overview of the AspectMaps visualization, before treating design decisions of the visualization in Section III and concluding in Section IV.

Johan Fabry is partially funded by FONDECYT project number 1130253

II. An Overview of AspectMaps

This section briefly describes the AspectMaps visualization, a full treatment is outside of the scope of this paper, for this we refer to the first publication on AspectMaps [2].

AspectMaps is a visualization that reveals and offers options to navigate the implicit invocations to behavior of aspects that are present in aspectual source code, as well as showing the structure of the aspects. Aspects are like classes, having state and behavior, but in addition the invocation conditions of the behavior are also present. Pieces of behavior in aspects are called *advice*, and their invocation conditions are called *pointcuts*. Places in the source code where a pointcut can cause an advice[1] to execute are called *join point shadows*. The join point shadows are hence the locations in the code where aspectual behavior can be called and these possible calls should be visualized.

AspectMaps shows the structure of aspectual source code from package level all the way down to the level of methods. Yet it is a scalable visualization of join point shadows [2], thanks to the use of a selective structural zoom, also known as a semantic zoom. This zoom allows for each element of the visualization (packages, classes, aspects and methods), to be shown in a compact or extended view. The closed/compact view does not reveal any internal structures. The open/extended view does show nested structures, each of these again being shown in closed or open view. Initially all packages of the software under study are shown in the closed view. The user selectively opens (and closes) packages, classes, aspects and methods as the software is being explored, in effect selectively increasing (and decreasing) the zoom level, showing more (or less) detail. Figure 1 shows part of an example exploration of AJHotDraw, a well-known case study of aspectual software [4]. Four packages are shown, of which one is closed: org.jhotdraw.samples, and the remaining three are open. The first two open packages show open and closed aspects, *e.g.,* DeleteCommandUndo is an open aspect, revealing two advice and one pointcut. In the org.jhotdraw.standard package we see a number of closed classes, the open class AbstractCommand, and inheritance relations between classes. In AbstractCommand, the constructor has been opened, as well as an execute method.

In AspectMaps, a color, changeable by the user, is assigned to each aspect. That color serves to visualize join point

[1]The community on aspect-oriented software development uses "advice" instead of "pieces of advice", and the plural of "advice" is "advice".

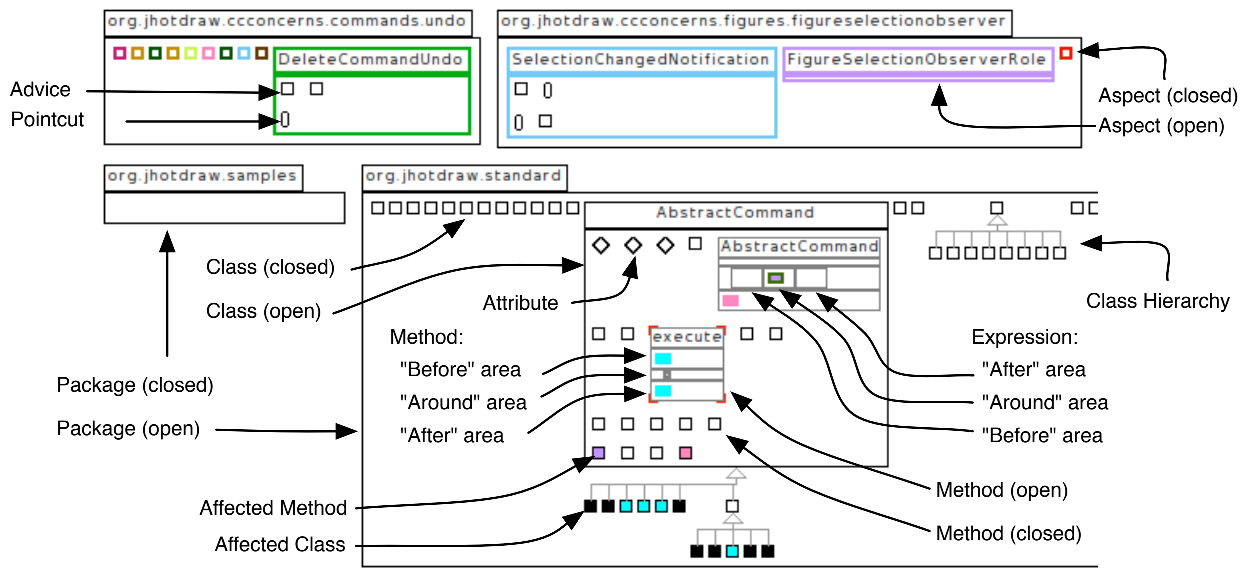

Fig. 1. AspectMaps visualization of part of AJHotdraw, with descriptive annotations. Packages, classes, aspects and methods shown at different zoom levels.

shadows where the aspect is called, as well as the aspect itself. Considering the latter, the color allows classes to be distinguished from aspects when they are shown in the closed visualization. This is because rectangles for classes have black borders, while rectangles for aspects have colored borders: the color of the aspect. Considering the former, to visualize the presence of join point shadows in an entity that is shown as closed, the inside of the figure is colored in the color of the aspect. The color black is used when multiple aspects have join point shadows there. In the figure, this is shown, *e.g.,* in the bottommost row of classes, as well as in two methods in the AbstractCommand class.

Methods that are shown open reveal the most fine-grained amount of detail where aspects apply. In them, AspectMaps shows two types of join point shadows: those applying to the entire execution of a method, and those applying to a method call that is being performed inside the method. The figure illustrates how the former join point shadows are visualized in the execute method[2], and how the latter join point shadows are visualized in the AbstractCommand constructor (which also contains an execution join point shadow). For both types of shadows the advice of the aspect can be invoked before, after or instead of (a.k.a. “around”) the original code. AspectMaps indicates this by drawing a rectangle in the color of the aspect in the “before”, “after”, or “around” area of the visualization of the join point shadow. For example, in the execute() method, the execution of its method body is preceded and followed by an invocation of some behavior of the cyan aspect. In the AbstractCommand constructor, a specific method call is being replaced by a call to some behavior of the violet aspect. Not shown in the figure is that hovering the mouse pointer over the different visualizations of the invocations reveals the name of the aspect as well as the identification of the advice that is executed. In general, AspectMaps provides more detailed information of each element being shown, as a tool tip window, when the mouse pointer hovers over this element. Furthermore, each entity has a context-sensitive menu that allows for relevant navigation actions, *e.g.,* for a pointcut to reveal all the advices that use it, and for these advices what their join point shadows are, or vice-versa [2].

As an illustration of how AspectMaps eases program understanding, we consider the pointcut and advice of this last join point shadow: a self call to getDrawingEditor() in the AbstractCommand constructor. The violet aspect is UndoableCommand, which handles undo functionalities. Below we excerpt the relevant pointcut and advice, *i.e.,* the code responsible for inserting the implicit call to the behavior of the aspect, and then briefly discuss it.

```
1  pointcut undoableCommands() :
2   (
3    (target(AlignCommand) &&
4     !within(AlignCommand) &&
5     !within(AlignCommandUndo))
6    ||
7    (target(BringToFrontCommand)&&
8     !within(BringToFrontCommand) &&
9     !within(BringToFrontCommandUndo))
10   ||
11   [... 12 more omitted ...]
12  )
13  && !within(UndoableCommand);

14 pointcut callCommandGetDrawingEditor():
15  call(DrawingEditor Command+.getDrawingEditor())
16   && undoableCommands();

17 DrawingEditor around() :
18   callCommandGetDrawingEditor() {
19   [... actual aspect behavior ...] }
```

In the code above, lines 1 through 13 declares a pointcut undoableCommands that basically declares which code calls an undoable command yet does not belong to this command. For example, lines 3 to 5 identify calls to AlignCommand that are not performed from within an AlignCommand or AlignCommandUndo instance. Then, lines 14 through 16

[2]The correspondance of the name ‘execute’ is a coincidence.

defines a pointcut that identifies calls to the method getDrawingEditor(), made within undoable commands. This is then used in the advice in lines 17 to 20 that replaces the call to these commands with its own implementation. This implementation will call the original behavior at some point, but we removed this code as it is not key to this example.

From the code above it is clearly nontrivial to determine what the join point shadows in the code are that will lead to lines 17 through 19 to be executed. In the end, there are only two such shadows in the entire code base, the one shown in Figure 1, and one more in the view() method of the same class. In AspectMaps it takes less than a second to reveal all these shadows, *e.g.,* using the context-sensitive menu on the join point shadow shown in the figure.

III. Design Decisions of the Visualization

A. The Use of Colors for Identifying Aspects

The design decision that has generated the most discussions, in presentations of the visualization and reviewer comments, is the choice of associating one color to one aspect. The human eye is only able to distinguish unambiguously between a limited amount of colors, especially if the colored areas are small [5]. This limits the ability to discern at a glance which aspect applies where. For example, in Figure 1, the color of the light blue aspect: SelectionChangedNotification, is slightly different from the cyan aspect that applies within the AbstractCommand class and some of its subclasses.

While this decision follows the use of colors to identify aspects in other visualizations, *e.g.,* the AJDT visualization [6], it was not made lightly. Typical scenarios for the use of AspectMaps are to determine where in the source code an aspect applies and, given a set of join point shadows, assess which aspects apply there and in what way. Hence the aim was for the user to be able to identify where aspects apply quickly, in a scalable way. In other words, we need to allow precognitive identification of a wide spectrum of values, spread throughout the visualization. The only visualization option we found that allows this is the aforementioned use of colors, which we combined with the ability to choose a color for the aspect and to turn off visualization of join point shadows of selected aspects. In our experience, this use of colors in AspectMaps does reach the goal: It allows immediate identification of where in the software the aspects apply, and very fast detailed inspection when focussing on a subset of aspects (by turning off the visualization of unimportant aspects).

Outside of the obvious colorblindness issues, we have identified two downsides to the use of colors for wich we have not yet found a suitable solution: multiple aspects applying in one entity and the limits to the use of colors.

When multiple aspects apply in one package, class or method, and this entity is shown in the closed representation, the figure for the entity is colored black to indicate this. It would however be better if the entity could indicate which aspects apply as well, increasing the amount of information shown to the user. We have not yet found an acceptable way in which to visualize this information. Experiments with showing patterns and textures of multiple colors did not convince. This is because the different colors tend to blend and the textures also cause confusing optical effects (also noted by Bertin [5]).

AspectMaps allows the user to pick a color for aspects by selecting from a list of sixteen colors, picked for their discernability. Furthermore, visualization of aspects can be 'switched off': the join point shadows of switched off aspects are not visualized. This allows for unambiguous visualization of up to sixteen aspects. However we have encountered various case studies with a larger number of aspects. For example, AJHotDraw has 31 aspects and another well-known case study: HealthWatcher [7], has 21. For these case studies it is impossible to visualize all aspects without any confusion between different aspects, as some colors will identify two aspects. In practice this has shown to be a minor problem, yet any improvement would of course be welcome.

B. Following Conventions and Resulting Complexity

In order to lower the barrier of entry to the visualization, the choice of how to show the structure of the code was to respect conventional representations of entities as much as possible. We chose UML class diagrams as the convention, as this arguably is the best-known graphical representation of code structure. This directly led to the visualization of packages as UML packages, and inheritance hierarchies as trees with lines that have an empty arrowhead pointing to the parent. The extended visualization of classes and aspects is a variant of the visualization of classes: a rectangle where we removed the separation between attributes and methods to save space. The visualization of attributes and methods is new, UML does not provide for a shape that can be adapted to our needs.

Following these conventions however leads to a diagram that can be quite complex when zoomed in maximally, *e.g.,* the AbstractCommand constructor in Figure 1. This was expected from the onset and confirmed by the user study [2]: some users were confused about whether the rectangle they were seeing is a method or a class. The visualization attempts to mitigate this by showing the borders of opened methods as gray (instead of black), yet the user studies revealed that this does not eliminate confusion totally.

Apart from issues of distinction between methods and classes, some users also reported having difficulties with understanding the visualization of methods themselves. This visualization shows a wealth of information and can become quite complex. Users have asked for some kind of legend for the representation of methods to be present. This such that, at least in the beginning, this part of the visualization is clarified. In the version of AspectMaps that is currently under development, we include such a legend that appears whenever the mouse hovers over a method. It however still remains to be shown whether this will successfully address the issue of comprehension of the visualization of methods.

C. Absence of Source Code view and IDE Integration

AspectMaps was designed and implemented as a standalone visualization and tool, with integration to an IDE being limited to the extraction of data required for the visualization. The idea was to have AspectMaps as a separate window, to be used next to the IDE. As such, the tool almost shows none of the source code of the program being visualized. The only exception is on mouse hover over the join point shadow of a call expression: there the source code of the expression is shown. In all other points of interest, sufficient information is shown such that the user can navigate to the corresponding source code relatively easily, if required.

User studies have shown that this was not the correct decision to make. One consistent request from users was to have a source code view integrated in the visualization tool [2]. This view should reveal the source code for entities that are currently being studied by the user. As a result of this request, in the version of AspectMaps that is currently under development, such a source code view is included. When clicking on an entity, it shows the source code for the entity, in read-only mode. An immediate downside of this is that the tool now has less space for the visualization. A derived effect is that users may wish to start editing the code they are seeing, to make changes when the visualization reveals to them that such changes are required. It is clear that allowing the tool to also edit code entails extra complexity for the tool and this might be better avoided. Instead, we are considering providing a better integration with existing IDEs. Ideally, the AspectMaps tool should instruct the IDE to automatically navigate to source code entities, *e.g.,* when these are clicked in the tool. The implementation and validation of this feature is future work.

D. Restrictions Driven by the Implementation

As previously reported [3], AspectMaps is built on top of the Moose reverse engineering platform [8], utilizing the Mondrian visualization framework [9]. Using Mondrian entails declaratively specifying a graph of nodes, what shapes to use for those nodes, selecting a layout algorithm and stating what edges need to be drawn between nodes (if any). For each shape, Mondrian also allows, *e.g.,* mouse-over actions to be specified.

The use of Mondrian greatly helped in the implementation of the visualization, as we solely needed to focus on what needs to be shown, and not how to show it. Our experience showed that AspectMaps pushes the boundaries of what is possible in Mondrian in various ways. Firstly, there originally was no support for replacing nodes in the graph and selectively updating part of the visualization. This is however what is required for the zoom feature: a closed entity that is opened entails replacing the corresponding node in the graph with the specification for the opened visualization (and vice-versa). To allow this, a new feature had to be added to Mondrian [3]. Mondrian has to be extended to offer interactive option to modify the visualization while being rendered, implying all the caches to enable scalability have to be adjusted accordingly. Secondly, the size and complexity of the diagrams when zoomed in are almost beyond the performance limits of Mondrian. Updates on such diagrams, *i.e.,* scrolling and zooming, are unacceptably slow (as reported by some subjects in the case studies).

There are also less immediate downsides of using Mondrian, which is that the features of the visualization are inherently limited to what Mondrian allows. While we encountered no restrictions that impeded fundamental parts of the visualizations, it is clear that a greater feature set of the visualization framework will allow for a more fully-featured AspectMaps. For example, on mouse hover Mondrian allows for a drawing to be displayed as a tooltip. Consider that instead we would be able to draw multiple drawings and place these as a second layer over the existing visualization, positioned respective to the element being hovered over. This would allow for more context-specific information to be shown on mouse hover, for example a legend explaining the various parts of the diagram being hovered over. This could address some of the complexity issues we discussed in Section III-B.

To address the current scalability issues of AspectMaps, and to allow enriching the visualization with new usability features, we are currently porting AspectMaps in Roassal [10], the successor to Mondrian. Roassal is an agile visualization engine that natively offers a large range of interaction options, *e.g.,* semantic zooming, smooth scrolling, animation.

IV. Conclusion

In the design and implementation of AspectMaps we were confronted with a number of design decisions that had notable impact in the resulting visualization and tool. These decisions and their motivation have not yet been completely and explicitly documented. In this text we provide such documentation, also in the light of its use, extension and maintenance in the last two years. It is our hope that this documentation serves to other visualization designers and implementers to aid in the making of their design decisions.

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

Acknowledgments

This work was partially sponsored by the European Smalltalk Users Group: http://esug.org.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 97)*, ser. Lecture Notes in Computer Science, M. Akşit and S. Matsuoka, Eds., vol. 1241. Jyväskylä, Finland: Springer-Verlag, Jun. 1997, pp. 220–242.

[2] J. Fabry, A. Kellens, and S. Ducasse, "Aspectmaps: A scalable visualization of join point shadows," in *Proceedings of 19th IEEE International Conference on Program Comprehension (ICPC2011).* IEEE Computer Society Press, Jul 2011, pp. 121–130.

[3] J. Fabry, A. Kellens, S. Denier, and S. Ducasse, "AspectMaps: Extending Moose to visualize AOP software," *Science of Computer Programming*, 2013, to Appear.

[4] A. V. Deursen, "AJHotDraw: A showcase for refactoring to aspects," in *In: Workshop on Linking Aspect Technology and Evolution, International Conference on Aspect-Oriented Software Development.*, 2005.

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

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

[7] S. Soares, P. Borba, and E. Laureano, "Distribution and persistence as aspects," *Software: Practice and Experience*, vol. 36, no. 7, 2006.

[8] S. Ducasse, T. Gîrba, A. Kuhn, and L. Renggli, "Meta-environment and executable meta-language using Smalltalk: an experience report," *Journal of Software and Systems Modeling*, vol. 8, no. 1, Feb. 2009.

[9] M. Meyer, T. Gîrba, and M. Lungu, "Mondrian: An agile visualization framework," in *ACM Symposium on Software Visualization (SoftVis'06).* New York, NY, USA: ACM Press, 2006, pp. 135–144.

[10] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep into Pharo.* Square Bracket Associates, 2013. [Online]. Available: http://rmod.lille.inria.fr/deepIntoPharo/