Complex Code Querying and Navigation for AspectJ

J.-Hendrik Pfeiffer Andonis Sardos John R. Gurd
Centre for Novel Computing
School of Computer Science
The University of Manchester
Oxford Road, Manchester M13 9PL, UK
{pfeiffer, sardos, gurd}@cs.manchester.ac.uk

ABSTRACT

The ever growing size and complexity of software projects demand good IDE support in order to assist the understanding and navigation of source code during implementation and maintenance. In the case of Aspect-oriented programming, additional supporting IDE tools are needed to make aspect-oriented structures explicit. However, existing tools struggle to provide easy-to-use navigation facilities when the size of the source code increases.

This paper describes *Lost* a query and navigation tool for the AspectJ language, its integration with the *eclipse* IDE, and initial experiences with using the tool. The described tool not only provides features which are novel with respect to current aspect-oriented programming tools but also attempts to overcome deficiencies of existing code querying tools.

Additionally, we briefly discuss the implementation of a framework for code querying tools, which was created in order to maintain high flexibility in implementing the code querying tool presented here.

1. MOTIVATION

Aspect-Oriented Programming (*AOP*) [11] provides the means to encapsulate so called *crosscutting concerns* in separate entities called *aspects*. Aspects contain the implementation of a crosscutting concern. In the terminology of the popular aspect-oriented programming language AspectJ [4], this implementation is called *advice*. Aspects also contain a description (*pointcut*) of which places in the source code (*joinpoints*) are affected (*advised*) by an aspect. The increased modularity introduced through AOP has a number of advantages, but increases the complexity of code understanding in the sense that when reading source code it might not be clear whether or not it is advised by an aspect. Thus additional tools, which provide AOP-specific information to the programmer, are required.

Tools, aiding the comprehension of aspect-oriented programs, exist to address the problems caused by this complexity. However, if during the exploration of an aspect-oriented program one is confronted with a task like the one shown in Figure 1, one would have considerable difficulties solving it with existing tools that support AOP (see Sections 1.1, 1.2). In order to address these difficulties, we developed the *Lost* tool (Section 2).

> *"Find all methods whose name contains* set *or* put*, which are advised by an aspect called* formatChecker *but not by an aspect called* policyEnforcer*."*

Figure 1: An example task.

1.1 Existing IDE support for AspectJ

In this paper we focus on the AspectJ [4] language since it is currently the most popular AOP language, which also has the most mature tools supporting it.

The AspectJ Development Tools (*AJDT*) [2] offer integration of aspect-oriented elements into different views of the eclipse IDE [7]. Since AspectJ enhances Java, the AspectJ enhanced AJDT versions of the regular Java-based eclipse views, such as editor, package explorer and outline, can be used to make aspect-oriented elements visible. However, these largely text-based views struggle with large programs, since the size of these views grows with program size. The advises list, showing which joinpoints an aspect advises, has particular problems with regard to size, since one aspect might advise a large number of joinpoints. Long list and tree structures are difficult to comprehend, and navigating them is tedious, especially, if one searches for specific information, as required in the task in Figure 1.

In addition to this size problem, information is spread across different views, which increases the likelihood that users lose their orientation when they are forced to switch views or combine information shown in different views. This problem is not specific to AOP however, since eclipse lacks a universal browser which is capable of showing all information necessary while a user tries to navigate to a certain part of the code. The *JQuery* tool [8] (see below) and the AspectJ query tool *Lost* (Section 2) attempt to address this problem by providing such a universal browser.

A possible general solution to the problem of overpopulated views is filtering, which is facilitated by the *Mylar* tool [10], for example. Mylar filters information according to a "degree of interest" model, which is created during code navigation, involving visiting or editing parts of the code. The Mylar approach is quite successful in reducing the information overload by connecting the view sizes to user activities.

OOPSLA 2005 Eclipse Technology eXchange (ETX) Workshop Oct. 16-20, 2005, San Diego, USA

But finding interesting parts of the code in the first place remains difficult, since one might still have to navigate through unfiltered views before the tool detects that something is interesting and, therefore, should not be filtered.

In addition to text-based views, visualisations, such as those found in [3], [1], [6], might provide essential information for a certain task, but they either also struggle with size or they might not be fine-grained enough for the level of detail required by the task specified in Figure 1.

1.2 JQuery

An elegant method of combining and providing information in one view is implemented in the JQuery tool [8]. JQuery, which is working on Java source code, is an eclipse plug-in, which creates a database from the source code and provides the means to query this database. After a query has been started it is represented as a node in a hierarchical tree with collapsible nodes, where the sub-nodes of a query node represent the results of the query. This in itself does not solve the above-mentioned problem of overpopulation in a view, since a query might return a large number of results. But JQuery offers the possibility to refine the search iteratively. This is done either by stating a new query, and collapsing or deleting the old query nodes, or by selecting an element in the tree, and performing a query on the basis of this element. For example, one could first find all methods with a certain name and then pick one of the results to see where it is called from. This iterative refinement shortens the list of results and maintains focus on the element on which a query is performed, which, in turn, helps the user to stay focused.

```
package(?P), child(?P,?CU), child(?CU,?M),
child(?M,?T), type(?T), name(?M,?name),
re_match(/^d/,?name)
```

Figure 2: A sample query in JQuery.

For stating queries, the user has a choice of a number of simple predefined queries, but, for everything advanced, the provided logic programming language (TyRuBa) has to be used. The query in Figure 2 is an example of what queries in JQuery look like; it means *Find all packages which have a class which contains a method whose name starts with "d"*. (In fact, the query does a bit more than just this, because it creates some structure in the result tree.)

By formulating queries, and using iterative refinement if necessary, users of JQuery can adapt the query to their needs in certain situations, and gain much more power and flexibility than they would have by using the Java search tool built into eclipse.

Although the above description suggests that JQuery is a useful tool, in practice it has two major disadvantages.

The main problem is the the way queries have to be formulated, which diminishes the usefulness of JQuery. The TyRuBa language, which makes use of predicates (see Figure 2), is fairly unintuitive for Java and AspectJ programmers, especially if they have no prior experience with logic programming languages. The JQuery authors themselves note (in [8]) that:

> "... the logic query language was hard to use for complex queries. This is true even for developers reasonably familiar with the query language."

In addition to the burden of learning a new, rather complex language, which causes users to frequently consult the documentation, the query language also forces users to mentally switch context from one programming paradigm (object-oriented programming) to another (logic programming). This switch is needed when changing focus from the object-oriented source code under examination to the actual formulation of a query. We believe this is not only time-consuming and cumbersome, but also unnecessary. Initial results (Section 3) support this view.

The second disadvantage is that JQuery is not easily extensible, since its design is tightly coupled to Java, and it is therefore hard, if not impossible, to extend for use with AspectJ. Admittedly, this is only relevant to AspectJ programmers, and AOP support was not targeted by the JQuery developers. However, AspectJ programmers would specifically benefit from enhanced tool support, since additional problems arise in the context of AOP. Examples of such problems are: *easy verification that a certain joinpoint really is advised by a certain aspect, without having to navigate around the code*, or *finding all methods which are advised by a certain combination of aspects*.

JQuery lacks features which are likely to enhance its usability. For example, modern user interface features, such as syntax highlighting and instant error feedback, would probably improve the ease of formulating a query, as well as reduce the time spent on the formulation.

1.3 Summary

Retaining detailed information about a problem, such as the example in Figure 1, cannot be done easily with existing IDE tools, especially when the problem involves showing the effect of certain combinations of aspects, or when a certain aspect needs to be excluded from examination. This is either because the tools do not have the means to formulate advanced queries and display their result, or because they do not support AOP. Moreover, the problems posed above, such as the problem of overpopulated views, hamper the professional development of aspect-oriented programs.

2. LOST

The AspectJ query tool *Lost* is an eclipse plug-in, which seeks to enable better code navigation and exploration facilities for AspectJ, by addressing the shortcomings of other IDE tools (Section1) in the context of AOP. Lost aims at providing the beneficial features of JQuery, such as the universal browser view and iterative searches, to the AspectJ programmer. At the same time it also tries to minimise the disadvantages of JQuery. In this respect, Lost tries to simplify the process of query formulation and to reduce the time needed to learn the query language. It also attempts to reduce the amount of initial training needed and the need to refer to its documentation.

Figure 3 shows the Lost view in eclipse. It consists of a query editor (1), where queries are formulated, a query result browser (2), which is used for navigation, and the element information area (3), which shows enhanced information about elements selected in the browser. At the beginning of the navigation process, the user selects a project in the package and builds the Lost model, which can then be queried.

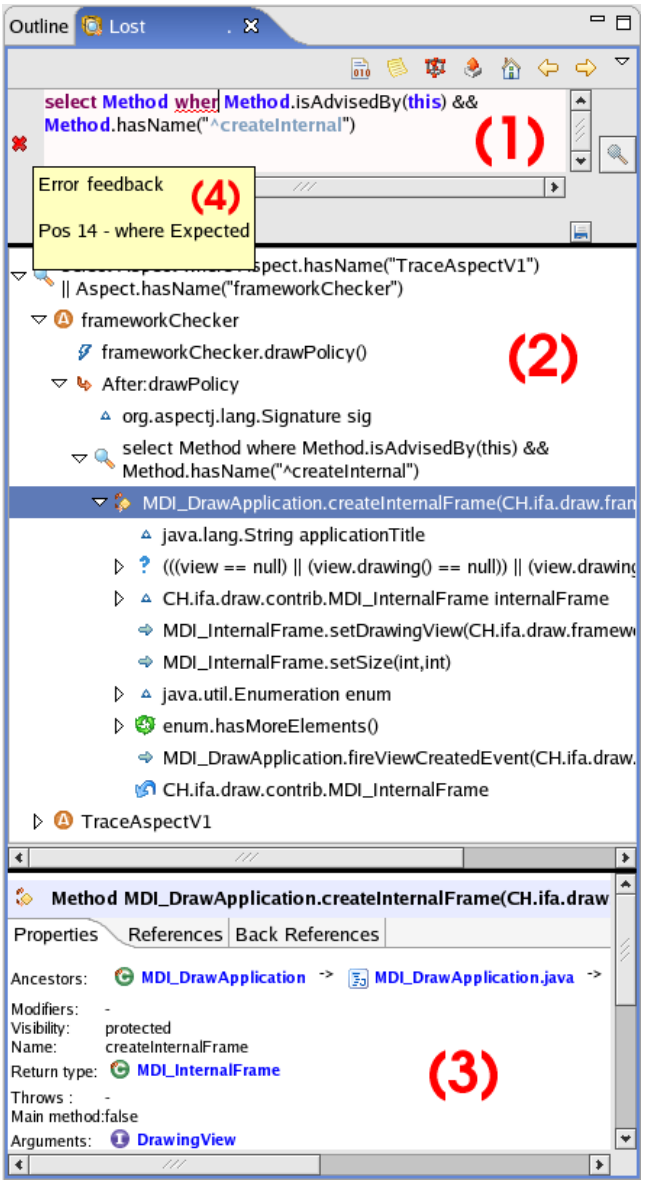

Figure 3: The Lost view, including query edit area (1), query result browser (2), element information area (3) and error feedback (4).

2.1 Queries

The query edit area of Lost is used to formulate the queries. We developed a simple query language for use with Lost, which has some similarity to the Object Query Language OQL [5]. In our view, a query language for AspectJ and Java should be an object query language, in order to avoid the problem of switching programming paradigms (see Section 1.2), which disrupts the user's workflow.

As an example for a query in the Lost language, consider the JQuery example query in Figure 2, which translates to the query in Figure 4 in the Lost query language. The latter is much more familiar to a Java programmer, since it is similar to the Java style of referring to classes and methods and passing arguments. For reasons of limited space we

```
select Package where Package.hasClass(Class)
&& Class.hasMethod(Method)
&& Method.hasName("^d")
```

Figure 4: The Lost version of the query in Figure 2.

do not show the full syntax of the Lost query language, but only a few examples. However, a programmer needs to know only little, if anything, about the language, since an auto-completion feature (see below) provides considerable help in phrasing syntactically correct queries.

Lost also works on plain Java, as the query in Figure 4 shows, and it can be used instead of JQuery. However, our main incentive was to provide complementary tool support for AspectJ. Therefore, the Lost query language can also query aspects and offers aspect-oriented queries, such as `advises` or `isAdvisedBy` for aspects or the respective elements, such as methods and classes. This might be clarified by considering the example given in Figure 1 (Section 1) and the corresponding Lost query in Figure 5. Further aspect-oriented queries, such as searching for pointcuts, all *around* advices, or all aspects declaring a warning are possible as well.

```
select Method
where Method.hasName("set|put")
where Method.isAdvisedBy(Aspect)
where !Method.isAdvisedBy(Aspect[2])
where Aspect.hasName("formatChecker")
&& Aspect[2].hasName("policyEnforcer")
```

Figure 5: The Lost query for the problem in Figure 1.

In order to achieve the target of providing a query tool that is intuitive to use and which requires little documentation, Lost offers IDE features, such as syntax highlighting (e. g. Figure 3), auto-completion, and instant error feedback, which help in creating queries and are natural for eclipse users to use, since they are also supported in standard eclipse views. The error feedback mechanism is demonstrated in Figure 3: the red underlining immediately highlights errors, and, additionally, a tool tip (4) containing an error message can be accessed by hovering the mouse over the small error icon which appears on the left hand side of the query edit area. Furthermore, the user is neither allowed to use erroneous queries (the query icon becomes disabled) nor to store them as predefined queries (which can easily be done for valid queries).

Even more important is the auto-completion feature, that works in the same way as the auto-complete in eclipse editors. Using this feature, almost no knowledge of the syntax of the query language is required by the user. For example, when starting with a blank query, hitting `Ctrl-+Space` would produce the *select* statement, hitting `Ctrl-+Space` again would produce a list of elements which can be queried (e. g. classes, aspects or packages). Auto-completion also shows the allowed queries on a certain element and their allowed arguments. Additionally, in an unfinished query, like `Aspect.hasName("st")`, using auto-completion on `"st"` would produce aspects which start with "st" as possible arguments.

The only concepts of the language which perhaps need to be learned are the way of referring to elements in the query and the use of regular expressions in arguments. Referencing within a query might be needed to distinguish different elements of the same type, e. g. two aspects `Aspect[1]` and `Aspect[2]`, within one query (Figure 5).

Note that, although we chose an object-oriented style for the query language, it is not tightly coupled to Java and AspectJ keywords. Most notably, it can also be used to search the browser for queries made earlier, e. g. before a particular date, or for comments attached to elements (see below).

2.2 Browsing

The browsing and navigation to query results in Lost is done in a collapsible tree structure (see Figure 3), similar to the one introduced by JQuery. This tree is used to further search and examine the results of a query, as well as to navigate to the location of interesting elements in source code (double clicking an element opens it in the associated editor and moves to the corresponding location in the code).

Nodes in the browser can not only be expanded, collapsed and deleted, they can also be integrated into the query in the edit area (using `this`) or have comments added. As in JQuery, queries are saved as nodes which have their results in subordinated nodes. Old queries can, therefore, be re-run or edited again. Furthermore, it is possible to expand all nodes of the tree, or to zoom into a node with the *go into* command, which makes the currently selected node the root of the tree, thereby filtering all nodes which do not have the selected node as an ancestor. This zooming in can be a multi-stage process, but one can always zoom out, back to the previous tree. The zooming described here reduces the size of the visible tree structure, if necessary, which makes navigation easier and also eases the effect of the problem of vertical scrolling described in [8].

The tree can be expanded, if necessary, to a very fine-grained level, where, for example, for-loops, advices or if-statements are visible (see (2) in Figure 3, but note that the elements are not fully expanded there). These fine-grained elements can be queried and used in queries as well; it is, for example, possible to find all *after* advices which contain an if-statement which references a certain variable in its condition. If, on the other hand, the level of detail provided by Lost is too fine-grained, or the user is not interested in certain elements, filters for these elements can be toggled easily.

In addition to the above-described code exploration features of Lost, the information area (see Figure 3) is available. This area shows additional information about the element selected in the browser. This information depends on the type of element, and is organised under three different tabs. The *properties* tab can contain information about the element itself, e. g. return type and arguments for a method, while the *reference* tab would show which other elements are referenced by the selected element, and the *back references* tab shows which other elements reference the selected element. For a method, *Calls* and *Called by* would be included in the associated reference tab. The properties tab always includes a list of ancestors of the selected element. This list is hyperlinked so that the information area of the respective ancestors can be shown (as a separate entity, which allows cascading exploration of the ancestors).

3. EVALUATION

In order to evaluate the usefulness of the query language and the query formulation features, we undertook a simple user study, in which we compared Lost against JQuery. For the study we only evaluated the Java-based features of Lost, because JQuery is not designed to handle aspect-oriented programs. A more extensive user study is in progress (see Section 5).

We asked three test persons (*P1*,*P2*,*P3*) to solve six tasks with both JQuery and Lost. The tasks were ordered according to complexity, ranging from simple (e. g. Task 1: "Find all main methods") to more complex (where Task 6 is the task underlying Figures 2 and 4). All tasks were run on the JHotDraw [9] source code and they were identical or similar to the tasks set up in [8]. In order to minimise the effects of working on the same task twice, they had to be solved in alternating order, starting with Task 1 being processed with Lost followed by JQuery, then Task 2 being processed with JQuery first and then Lost, and so on.

In order to verify our aim of reducing the documentation and training needed, users were not given any documentation of Lost, apart from an introduction to the autocomplete feature and where to put queries. In the case of JQuery all essential functionality was explained, documentation was made available and users were allowed to use predefined queries.

		P1	P2	P3
Task 1	Lost	3:00	1:13	0:40
	JQuery	>10:00	4:56	2:00
Task 2	Lost	4:25	1:30	0:25
	JQuery	>8:55	>5:00	0:40
Task 3	Lost	2:22	2:50	1:20
	JQuery	>6:43	>5:00	3:20
Task 4	Lost	1:30	2:15	1:57
	JQuery	>1:30	—	>13:45
Task 5	Lost	3:33	>5:00	1:55
	JQuery	>3:48	—	3:00
Task 6	Lost	1:46	1:40	0:56
	JQuery	>1:46	>5:00	4:10

Table 1: Times (min:sec) needed by test persons P1, P2, P3 to solve six tasks with Lost and JQuery.

The test persons are all experienced Java developers, who know and use eclipse; P2 has some, and P3 has good, knowledge of logic programming languages, while P1 has no such experience. We measured the time that our test persons needed for formulating a working query for each of the tasks; these times are given in Table 1 (the response times of the tools were not measured but they did not cause significant delays). If a time in Table 1 is not given, the task was not attempted; if a time is marked with ">", the task was not completed with a valid result (the test person either gave up or was stopped).

The results in Table 1 clearly show that Lost is more suitable for querying Java code than JQuery. All test persons spent significantly less time for formulating valid queries with Lost than with JQuery, if they managed to solve the given tasks with JQuery at all. Moreover, the rate of success using JQuery correlates with the knowledge of logic programming languages. This seems to be the reason why,

for the given tasks, only person P3 managed to make satisfactory use of JQuery. The timing results also suggest that the query language plays a significant role in a query tool and needs careful design. Although only three persons were involved in the study, the timings seem to confirm our view that, in the context of Java programming, requiring knowledge of a logic programming language as a basis for successful querying might not be appropriate.

The test persons also complained about the insufficient help provided by JQuery, although all resources were made available, while they had no difficulties using Lost, even though no accompanying documentation was provided and almost no explanation of the tool was given in advance. We are, therefore, confident that our initial target of minimising the need for documentation and building an intuitive-to-use use tool was met, at least with respect to the querying part of Lost.

4. IMPLEMENTATION

4.1 CQT Framework

When we were planning the implementation of Lost (Section 2) we wanted to be as flexible as possible for making future changes. For this reason, the Code Query Tools (*CQT*) framework was created, and Lost is just one implementation of the framework. The CQT framework encapsulates features common to code querying tools and allows the attachment of features which might be specific to a certain tool.

That is why *models* for different programming languages, for example, can be registered in the framework as components. This can even happen dynamically, so that a running tool implementing the framework can be used for multiple programming languages. In addition to the model, *builders*, which create a model of a project, need to be implemented. It is also possible to export these models of a project, and different export managers might be registered with the framework. At the moment, an XML export mechanism is available.

Elements in the model of the framework have *searchers* and *renderers*. The former allows us, for example, to add new possible queries to a certain type of element if needed. In the event that the query language of Lost leads to difficulties, new parsers for other query languages can be added. Finally, the way model elements appear is determined by a renderer and new renderers can be added later.

The described hot spots of the CQT framework allow the flexible creation and change of query tools for different programming languages. At the moment, however, only the necessary elements for Lost, such as models and builders, for AspectJ and Java and the parser for the query language of Lost are implemented.

4.2 Eclipse Integration

The CQT framework (Section 4.1) can be used for the creation of stand alone tools, if the required information providers, builders and renders are implemented. The Lost (Section 2) is, however, integrated with eclipse as a plug-in, which provides its own view. Additionally, it requires the AJDT and AspectJ plug-ins to be installed, since it accesses parts of the model built by them. Lost also opens files in the editors provided by the AJDT in order to facilitate a seamless integration. Furthermore, Lost makes use of the Standard Widget Toolkit in order to provide its user interface (Figure 3).

5. FUTURE WORK

At present, only initial results about Lost's behaviour are available and the implementation of the tool is not complete (e. g. the way that regular expressions are used is likely to be changed). Both deficiencies need to be addressed, but a full comparison against state-of-the-art AOP and IDE tools is work in progress. Depending on the results of this comparison, we might consider the integration of Lost with the Asbro tool [6] in order to speed up navigation further. Another question which needs to be answered in a user study is, in which situations querying is capable of doing better than filtering approaches. Finally, it needs to be verified that query formulation is not too costly in time, since there is a trade-off between navigation of long lists and the time taken to formulate good queries.

6. CONCLUSIONS

As software projects are getting more and more complicated, query tools are likely to be an essential part of any IDE. We have introduced a code querying and navigation tool which not only widens the applicability of code querying tools, by making their features available to aspect-oriented programming, but also aims to overcome deficiencies in today's AOP support tools. Initial results suggest the tool can meet its purpose, but further comparison to other AOP support tools is necessary to fully establish its usefulness.

7. REFERENCES

[1] ActiveAspect. `http://www.cs.ubc.ca/labs/spl/projects/activeaspect/`, 2005.

[2] AspectJ development tools. `http://www.eclipse.org/ajdt/`, 2005.

[3] AspectJ development tools visualiser. `http://www.eclipse.org/ajdt/visualiser/`, 2005.

[4] AspectJ. `http://www.eclipse.org/aspectJ/`, 2005.

[5] R. Cattell, editor. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[6] Asbro. `http://www.cs.manchester.ac.uk/cnc/projects/asbro.php`, 2005.

[7] Eclipse. `http://www.eclipse.org/`, 2005.

[8] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 178–187. ACM Press, 2003.

[9] JHotDraw 5.3. `http://www.jhotdraw.org/`, 2002.

[10] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, New York, NY, USA, 2005. ACM Press.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.