Navigating through Java Programs with Concept Lattices

Kim Mens Diego Ordóñez Camacho Mathieu Syben

December 20, 2006

Abstract

In this article we explore the use of formal concept analysis to search Java programs for methods that implement a certain functionality. The approach relies on the navigation of a concept lattice where the concepts represent groups of methods that share keywords in their name. In two experiments carried out on Java programs, the results of searching with a navigation-based search tool are compared to those obtained with a traditional Java search tool. Even though the results do not provide conclusive evidence that a lattice-based search performs better than a lexical search, the insights gained provide useful input for future improvements of the proposed approach and tool.

1 Introduction

Jonathan Fallon [FAL 04] showed that formal concept analysis [GAN 99] can be applied successfully to the domain of information retrieval, to efficiently look for documents of interest in a large repository of documents indexed by keywords. One of the benefits of using concept lattices for that purpose is that the information contained in the concepts can be exploited for a directed navigation through the search space. In contrast to more traditional search techniques, such an approach can guide users in their search by proposing relevant keywords to refine their search (or to remove irrelevant keywords). In the experiments reported on in this paper we investigate whether a similar technique can be applied to search Java programs for methods of interest.

Especially for very large programs, advanced search engines that allow a developer to find quickly a relevant program entity are valuable. For example, they can help in locating a method which implements a certain functionality that needs to be extended or adapted. Most contemporary source-code search tools rely on text-based searches or regular expressions over a flat text representation of the program code. In spite of their simplicity they have the advantage of being relatively efficient and requiring little input from the user. However, they tend to be rather noisy for large code bases and are less useful when a programmer doesn't know exactly what he is looking for (i.e., when the exact combination of keywords needed to locate the desired program entity is unknown).

An approach based on formal concept analysis partly overcomes this problem by working over a more structured representation of the code (a concept lattice) and by associating with each relevant program entity a set of attributes (keywords) that the programmer may use to conduct or refine his search. More specifically, the special characteristics of concept lattices offer to a programmer who wants to find a particular program entity the ability to conduct his search by navigating through the lattice guided by the attributes of the entities he or she encounters during his search. In this paper we will restrict the search to program entities which are methods, using as keywords only the words occurring in the names of those methods.

A prototype of a tool that implements our approach was implemented as a plug-in for the Eclipse development platform (`http://www.eclipse.org/`). Using this plug-in, two case studies were conducted, where the results of our tool were compared to those obtained by the Java search engine that comes with Eclipse. The goal of these case studies was to try and identify

relevant methods that implement a certain functionality, based on the words used in the names of such methods.

The remainder of this paper is structured as follows. We sketch related work on lexical, keyword-based and navigation-based search tools in Section 2. Section 3 details Fallon's work [FAL 04] where formal concept analysis was used in the context of information retrieval. Taking inspiration from that approach, Section 4 explains our own variant of this approach and tool for navigating through Java programs. Section 5 explains the set-up and results of the experiments we conducted to validate our tool. Based on our interpretation of the obtained results in Section 6, we propose some avenues for future research, before concluding the paper in Section 7.

2 Related work

Tools that search for information in program code often rely on lexical or text-based search. A tool like *grep*, for example, provides a generic text-based search based on regular expression pattern matching. Tools like the *Java search* integrated in the Eclipse development platform, and *sgrep* [JAA 96] perform lexical search on specific types of code entities (like Java methods or fields). A disadvantage of search tools like the above is that they do not allow to refine the results of a previous search. To conduct a refined search one has to provide a new more complex search query and re-launch the query from the start.

In the domain of information retrieval, two well-known techniques to look for information in large document repositories are *keyword-based search* and *navigation-based search*. A classic keyword-based search method relies on the *Vector Space Model* [BER 99], that can refined with *Latent Semantic Indexing* [BER 99]. Several keyword-based source code search tools have been developed (e.g., *JSearch* [SIN 06] and *IRiSS* [XIE 06]). An important advantage of keyword-based search techniques is their scalability and ability to efficiently perform very specific searches, but the user needs to have some knowledge of the names of the code entities he is looking for.

In contrast to keyword-based search, navigation-based search starts from the premise that it is much easier for a human being to recognise something than to describe it. Tools like LaSSIE [DEV 90], RiGi [WON 98] or CodeCrawler [LAN 03] allow to visualise or navigate through the source-code structure, by making a higher-level model of the relevant source-code entities and their interrelations. Because they work on a model of the source code, however, they may miss some lower-level details.

3 Information Retrieval with Formal Concept Analysis

Search based on concept lattices is a search method that tries to find the middle ground between keyword-based and navigation-based search. Similar to a keyword-based search method, a user starts his search by providing one or more keywords and receives a set of entities that match those keywords. At the same time, he gets access to a lattice structure that allows him to refine his search, as he would do with a navigation-based search. The lattice structure provides detailed information on what keywords could be added to make the user's search query more precise, or what keywords could be removed to make it more general. In the remainder of this section, we briefly explain the idea of formal concept analysis as well as a search method based on that theory. In the remainder of this paper we then explore the utility of this search method to navigate through Java programs in search for relevant methods.

Formal Concept Analysis.

Starting from a Boolean matrix of entities and their properties, the technique of formal concept analysis (FCA) groups all entities in the matrix that share common properties and, inversely, all properties that are shared by common entities. This gives rise to a double hierarchy of sets (a

Galois lattice). For a good theoretical introduction to FCA we refer to [GAN 99]. Nevertheless, we think that the example below is sufficiently intuitive to be understood by readers less knowledgeable in FCA.

Suppose that we have indexed all Tintin comics, by the Belgian artist Hergé, by keywords representing the main characters in the collection (Captain Haddock, Cuthbert Calculus, Thomson and Thompson, Snowy) or by objects and themes appearing in them (the moon, Marlinspike Hall, a boat). Taking as entities the different comic books (for example, bd_1 = 'Red Rackham's Treasure') and as properties the keywords, we obtain a Boolean matrix like the one depicted in Table 1.

Table 1: Tintin comics indexed by keywords.

	Haddock	Moon	Thomson	Marlinspike	Boat	Calculus	Snowy
bd_1	1	0	1	1	1	1	1
bd_2	1	0	0	0	1	0	1
bd_3	0	0	1	0	0	0	1
bd_4	0	0	1	0	1	0	1
bd_5	1	1	1	1	0	1	1
bd_6	1	1	1	0	0	1	1
bd_7	0	0	1	0	1	0	1

Taking this matrix as input, an FCA algorithm determines all possible *concepts*, each of which consist of a set of entities and a set of properties, such that each entity of the concept shares all properties of the concept and every property of the concept holds for all of its entities. In addition, the sets in each concept are supposed to be *maximal* in the sense that no entity outside the concept has those same properties, and no other property outside the concept holds for all entities in the concept. The set of concepts thus obtained can be structured in a Galois lattice structure. For example, the concept lattice for the matrix shown in Table 1 is depicted in Figure 1.

Navigating the lattice.

We now illustrate with a simple scenario how such a lattice structure can help us to locate efficiently the right information in a large repository of documents. Suppose that a user has read most albums in the Tintin collection and that he is currently looking for the book 'Red Rackham's Treasure' (corresponding to comic bd_1 in the table). Although he forgot the exact title of the book he remembers that it was about a boat.

To conduct his search the user starts with the keyword 'Boat'. After having filled in this keyword, the search takes the user to the most general concept in the lattice that has this keyword (i.e., the unique highest concept in the lattice that contains the keyword 'Boat' in its property set). As can be seen in Figure 1, this is the concept ({bd4, bd1, bd7, bd2},{Snowy, Boat}). The user is confirmed in his search because all books in this concept do not only share the keyword 'Boat', but also the keyword 'Snowy', and he remembers that the dog named 'Snowy' indeed appears in the story he is looking for. Furthermore, the search can be refined by either adding the keyword 'Haddock' or the keyword 'Thomson', which will take him to more specific concepts (i.e., with less comic books sharing all those keywords). This corresponds to navigating downwards in the lattice. The user can also navigate upwards, i.e. to generalise his search, by removing the keyword 'Boat' from his search.

An important point is that, apart from the initial keyword used to start the search, the user did not have to "imagine" any keyword. It sufficed to recognise the keywords proposed along the way and to navigate in the right direction based on those keywords. In this particular scenario, the user recognised that captain Haddock played a central role in the story and thus further refined his

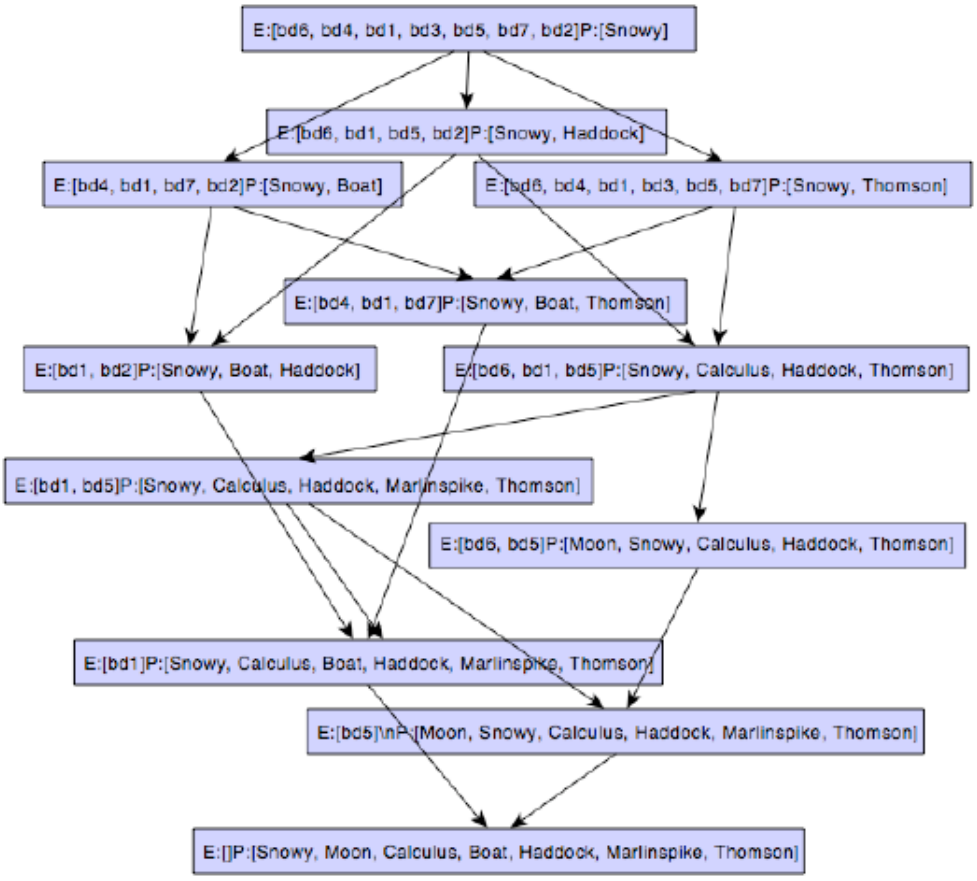

Figure 1: A concept lattice of Tintin comics and their keywords.

search with that keyword to further restrict the search results. At this point the set of results only contained two comic books and the user recognised the one he was looking for and selected it.

4 Source-code Navigation with FCA

Inspired by Fallon's information retrieval tool based on FCA [FAL 04], we conceived a similar tool to search Java programs for methods implementing a certain functionality. We use Java methods as entities in our concept analysis and as properties we take the keywords appearing in the names of those methods, because method names tend to reveal a lot about the intention of a method. More specifically, to obtain the keywords from a method's name, we split the name in different fragments depending on where the uppercase characters appear in the name. For example, with a method named `getConfigFile` the keywords 'get', 'config' and 'file' would be associated.

In addition to applying an FCA algorithm on the Boolean matrix consisting of those methods and their corresponding keywords, we apply some extra pre-processing, post-processing and filtering to reduce the noise and improve the quality of the results. We *pre-process* the set of all entities (Java methods) to remove irrelevant methods, like test methods. We also pre-process the set of all keywords by excluding those that belong to a previously established *blacklist*. This blacklist contains very generic keywords (like 'name', 'result', 'value'), auxiliary verbs (like 'must',

'will'), prepositions ('for', 'with'), conjunctions ('and', 'or'), and so on. To bring keywords with the same root to the same form, we also apply Porter's *stemming* algorithm [POR 80] on the keywords, before putting them in our Boolean matrix. For example, the words 'connection' and 'connected' are brought to the same stem 'connect'. In our experiments, no *post-processing* was performed, but the tool can be configured, amongst others, to remove concepts containing too many or too few entities or keywords.

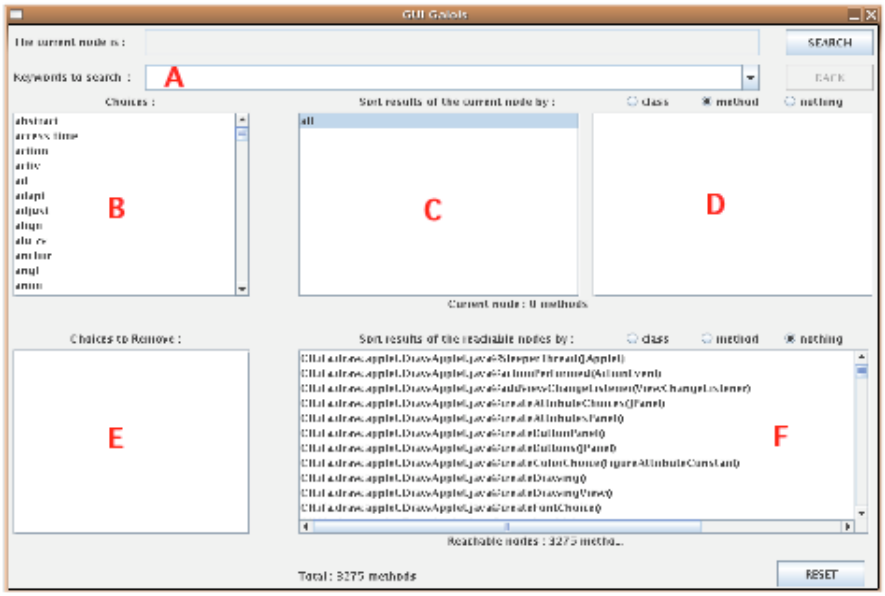

Figure 2: Our Eclipse source-code navigation plug-in based on FCA.

Once the lattice has been created using the approach outlined above, our dedicated lattice-based search tool shown in Figure 2 allows us to navigate the lattice, based on the methods' keywords. Suppose we are looking for methods implementing the functionality of "selecting a graphic component". The first step is to imagine what are the possible keywords identifying those methods. Intuitively, we decide to start with the keyword 'select'. We can either type it in field 'A', or choose it from list 'B' where we find all keywords in the current node (the top node in this case). After selecting this keyword, the context changes and we start seeing results displayed in lists 'C' and 'D', in this case all methods indexed by 'select' and their implementing classes. List 'B' is also updated and now shows the keyword 'figure', which seems interesting enough to try it. Indeed, it brings us to a concept containing the method `figureSelectionChanged`, whose code indeed implements the functionality we were looking for. Throughout our search, list 'F' always shows all methods that can be reached from the current node, which is useful information to help us decide whether it is worthwhile to further refine our search. When we notice that the current branch does not lead to interesting results, we can use the list 'E' to go back upwards in the lattice, by removing certain keywords from the search query, and from there either visit another branch or move further upwards.

5 Experiments

Experimental set-up

To validate our tool and approach, we carried out an experiment on two different case studies. Both of our experiments consisted of eight basic steps:

1. Select a representative set of use cases.
2. For each use case define a specific functionality to look for, and describe it with a group of keywords.
3. To look for a specific functionality, start the search from one of the keywords describing the functionality. Trying the more specific keywords first has the advantage of quickly reducing the search space.
4. If too many results are produced, refine the search with one of the other keywords, to navigate to a more specific concept.
5. If no results are found, remove one of the keywords to choose another likely keyword, or to move back up to a less specific concept.
6. When likely results are found, validate them by using, for instance, any (or a combination) of the following techniques:
   (a) Look at the source code of the proposed methods
   (b) Ask a developer who knows the code for confirmation
   (c) Check whether the discovered method is indeed executed when running the corresponding use case
7. Search for the same functionality with the Eclipse Java search tool.
8. Compare the results obtained with both tools.

Case studies

We applied this experiment on two different cases: JHotDraw and our own source-code navigation plug-in based on FCA (called FCAPlugin).

JHotDraw is a graphics framework to build structured drawing editors. It is a medium-sized case composed of 467 classes and is a fairly well-designed object-oriented application. One of our main reasons for selecting this case was the availability of a set of uses cases for that application, as well as the execution traces for each of those use cases. This allowed us to validate the results by comparing them to the execution traces (step 6c).

FcaPlugin is an Eclipse plug-in which builds a concept lattice out of Java source code and is the basis of our navigation tool. It is a small-sized prototype application. Our motivation for selecting this case was the availability of a developer to confirm the validity of the results found by our tool (step 6b).

In the remainder of this section, we detail the results of the experiment we conducted on the JHotDraw case, and categorise the discovered results by comparing them to the results produced by a standard Java search. Due to space limitations, we will not detail the results for our FcaPlugin case, but we will summarise and discuss its results in Section 6.

Experiment on JHotDraw

In the JHotDraw case study we considered 24 use cases of which we wanted to locate the methods implementing the main functionality. For each of the discovered methods, we inspected their implementation to verify if they indeed implemented the expected functionality (step 6a). In addition, we compared the discovered results with the execution trace for the corresponding use case (step 6c). If a discovered method was actually executed for that particular use case, this was an extra indication that the right method was probably found. Rather than showing the details of each of our 24 searches, below we give a representative example for each category of results only. Table 2 summarises all use cases and what kind of results were obtained when searching for those use cases.

Table 2: Results of JHotDraw experiment.

	Use case	Result
1	Maximise window	Method implementing use case not found
2	Select graphical element	Lattice useful to suggest interesting subconcepts Less noise than Java search
3	Move figure	Less noise than Java search
4	Resize figure	Method implementing use case not found
5	Activate tracing	Method implementing use case not found
6	Create new text field	Essentially same results as with Java search
7	Add text to figure	Essentially same results as with Java search
8	Add URL to figure	Essentially same results as with Java search
9	Draw rectangle	Essentially same results as with Java search
10	Draw round rectangle	Essentially same results as with Java search
11	Connect figures	Lattice useful to suggest interesting subconcepts Less noise than Java search
12	Create elbow connector	Essentially same results as with Java search
13	Add border to figures	Essentially same results as with Java search
14	Use 'copy' command	Essentially same results as with Java search
15	Use 'paste' command	Essentially same results as with Java search
16	Use 'cut' command	Less noise than Java search
17	Use 'duplicate' command	Essentially same results as with Java search
18	Delete figure	Essentially same results as with Java search
19	Group figure	Less noise than Java search
20	Ungroup figures	Essentially same results as with Java search
21	Select group	Essentially same results as with Java search
22	Bring figure to front	Essentially same results as with Java search
23	Fill figure with colour	Essentially same results as with Java search
24	Save drawing	Essentially same results as with Java search

Method implementing use case not found. In some cases (1, 4 and 5), neither a lattice search nor a text-based search lead to a relevant result, typically when the person conducting the research was not capable of finding the right keyword to start the search. This illustrates that the quality of the search obviously depends on the accuracy of the keywords. For example, our search for the 'Maximise window' use case (1) was negative, because the obvious keywords of 'maximise' and 'window' did not lead to any interesting search results for either search technique. The reason for this is that maximising a window is actually implemented by a *listener* on a *frame*, which calls a method *componentResized*. So the use of the keyword 'resize' might have lead to a better result. Luckily we found only a few negative results of this kind in this experiment, thanks to the relatively good naming conventions adopted in the JHotDraw code.

Lattice useful to suggest interesting subconcepts. In 2 of our searches (2 and 11), even though the lattice search and the Java search eventually lead to the same result, the lattice search was slightly more advantageous because it suggested the right keywords to refine our search, so that the user did not have to come up with these keywords himself. (Since we conducted the Java search after the lattice search, the user was already biassed when conducting the Java search. Therefore, we cannot confirm whether the user would actually have found the right combination of keywords if he would have conducted the Java search directly.)

A typical example of this was encountered for the 'Select graphical element' use case (2). Using the lattice search tool we started our search from the keyword 'select'. Given that this

keyword was quite generic and thus shared by many methods in the system, this search brought us to a node in the lattice which had no less than 17 sub-nodes, each identified by different additional keywords to reach such a sub-node. After briefly inspecting each of those keywords, we discovered a keyword 'figure' which we decided to navigate to. The corresponding node again had three sub-nodes of which one contained the keyword 'change'. Following our intuition that this combination of keywords might lead us to a method to change the selected figure, we decide to navigate further in that direction, which lead us to discover several methods named `figureSelectionChanged(DrawingView)` which were clearly related to the functionality we were looking for. To get more confidence in the validity of the discovered methods we verified if they actually appeared in the execution trace for the 'Select graphical element' use case, which was indeed the case. In summary, in this particular example the lattice search tool clearly helped us to navigate in the right direction, by allowing us to inspect and select the keywords that seemed most related to our search.

Less noise than Java search. We also encountered 5 cases (2, 3, 11, 16, 19) where both the Java search and our lattice-based search eventually lead to the desired result, but the Java search contained slightly more noise in the results produced, thus requiring a bigger effort from a user interpreting the results to filter out the noise from the relevant results. The use case 'Group figures' (19) provides a typical example of this category. We started our search by selecting the keyword 'group'. One of the sub-nodes of the corresponding concept had the keyword 'figure' so we navigated to that one and discovered a method `groupFigures()`. Although this method seemed to be relevant for the use case, it did not appear in the execution trace, which made us belief that (in this particular execution trace) another method must have been responsible for the grouping of figures. So, we went back to our lattice navigation tool and noticed that one of the other sub-nodes of the concept with keyword 'group' added the keyword 'command'. Like before, we then followed our intuition that this might lead to a method implementing a command to group figures and, indeed, when following this alternative path we discovered a method `groupCommand(String,DrawEditor)` which seemed to implement that functionality. Next, we verified whether the Java search tool lead to the same results. We used the search pattern '*group*' to conduct our search and did discover the two methods mentioned above. However, the result set contained much more noise because we also found lots of methods with 'ungroup' in their name (which matched the search pattern '*group*'), which was not the case for the lattice search.

Essentially the same results. In the 16 remaining cases (6–10, 12–15, 17–18, 20–24) there was no noticeable difference between the Java search and the lattice search, merely because both almost immediately lead to the desired solution after filling in a single keyword. For example, for the 'Add URL to figure' use case (8) we performed a lattice search on the very specific keyword 'URL' and immediately discovered the two methods `getURL()` and `setURL(Figure, String)` of which the latter clearly corresponded to the functionality we were looking for. When doing a Java search using the string pattern '*URL*' we immediately discovered these two methods as well.

Partial results. One disadvantage of both techniques is that we have no guarantee to have found the complete set of methods implementing a given use case. Let us illustrate this by means of the 'Move figure' use case (3). Without entering in all details, after having performed a search using the keywords 'move' and 'figure', we discovered a method named `moveAffectedFigures(Point,Point)` which seemed relevant but did not appear in the execution trace for that use case. Hence, we tried a search on the keywords 'move' and 'select', which lead to the discovery of another interesting method `moveSelection(int,int)` which still did not appear in the execution trace, however. Therefore, even though we were convinced that the methods we discovered were implementing at least a part of the 'move figure' functionality, we concluded that there must be other

ones that we failed to discover.

6 Discussion of the results

Table 3[1] summarises the results of the experiments we conducted on both JHotDraw and on FCAPlugIn. Perhaps the most important conclusion we can draw from the experiments is that they do not seem to provide any conclusive evidence that our navigation-based search tool performs better than a traditional text-based source code search tool. Indeed, for the majority of use cases there was no essential difference between using one tool or the other. Only for a limited percentage of use cases our lattice-based search seemed to offer some advantages (producing less noise or suggesting interesting keywords to refine the search).

Table 3: Summary of results of our JHotDraw and FCAPlugIn experiment.

Result	JHotDraw	FCAPlugIn
Lattice useful to suggest interesting subconcepts	2 (8%)	0 (0%)
Less noise than Java search	5 (21%)	1 (7%)
Method implementing use case not found	3 (13%)	5 (36%)
Essentially same results as with Java search	16 (67%)	8 (57%)

After the positive results that were achieved when using FCA for information retrieval [FAL 04], we were somewhat disappointed by the results of our concept lattice based source-code navigation tool. Overall, it seems as if a traditional text-based Java search tool (or an extension thereof) does not perform much worse than our proposed tool. We see two possible causes for this.

First of all, although Java methods could be regarded as a very specific kind of 'documents' and the identifiers appearing in their names as 'keywords', the number of such keywords associated with a Java method is typically much more limited than the number of keywords that would be associated with a typical document. Indeed, since it is an accepted coding convention in object-oriented programming that methods should have short and clear names, the average number of keywords per method is small and thus the maximum navigation path to reach a given method remains small. In other words, the corresponding concept lattice will be wide but flat, and little navigation is needed for finding methods with a given set of keywords.

Secondly, we limit our search to the identifiers appearing in the methods' names only. Instead, we could take the entire method into account and consider the names and types of parameters in its signature, words appearing in the method's comments, the calls it makes to other methods, or the names and types of variables and objects it uses. Not only would this increase the number of keywords associated to a method, thus allowing us to find more relevant information, it would also make the lattice increase in size, making the need for navigation through the lattice more relevant. How exactly to exploit these alternative attributes in a navigation-based search tool remains a topic of future investigation. A potential risk, however, is that having all this extra information will lead to more noise in the lattice and thus clutter our search results.

Another possibility is to apply the technique to program entities of coarser granularity, like classes, files or packages, to which many more keywords can be associated (for example the names of methods, classes or variables appearing in those entities).

[1]The sum of the values in the first column yields more than 100% because two use cases were an example of both "Lattice useful to suggest interesting subconcepts" and "Less noise than Java search."

7 Conclusion

The objective of this work was to build and assess the usefulness of a source-code search tool based on the theory of formal concept analysis. The idea was to represent a Java program as a concept lattice and to search the program for relevant information by navigating through that lattice. Given that a similar approach applied in the context of information retrieval lead to positive results, our goal was to verify whether this would also be the case for searching Java programs. We build a dedicated lattice-based tool to search Java programs for methods implementing a certain functionality (based on the names of those methods) and compared this tool to a lexical Java search tool. Although the developed tool did not perform worse than the more traditional Java search tool, we could not conclude that it performed significantly better either. This disappointing result seemed to be caused mainly by the fact that we only take the names of the methods into account, thus leading to a very flat lattice where navigation is of limited use. As a topic of future research we propose to take more advanced properties of methods into account (like types, variables, calls, ...) which may lead to a larger concept lattice where the use of navigation might prove more useful.

8 Acknowledgments

We are grateful to Mariano Ceccato for having provided us with some use cases and corresponding execution traces for the JHotDraw case study. We thank Andy Kellens for proofreading and commenting on a final draft of this paper.

References

[BER 99] BERRY M. W.BROWNE M., *Understanding Search Engines: Mathematical Modeling and Text Retrieval*, SIAM, 1999.

[DEV 90] DEVANBU P. T., BRACHMAN R. J., SELFRIDGE P. G.BALLARD B. W., LaSSIE: a Knowledge-based Software Information System, *International Conference on Software Engineering*, 1990, 249-261.

[FAL 04] FALLON J., Application des treillis de Galois à la recherche d'informations, Master's thesis, Université catholique de Louvain, Département d'Ingénierie Informatique, 2004.

[GAN 99] GANTER B.WILLE R., *Formal Concept Analysis: Mathematical Foundations*, Spring-Verlag, 1999.

[JAA 96] JAAKKOLA J.KILPELAINEN P., Using sgrep for querying structured text files, 1996.

[LAN 03] LANZA M., CodeCrawler: Lessons Learned in Building a Software Visualization Tool, *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003)*, IEEE Computer Society, 2003, 409-418.

[POR 80] PORTER M., An algorithm for suffix stripping, *Program*, 14, 3, 1980, 130–137.

[SIN 06] SINDHGATTA R., Using an information retrieval system to retrieve source code samples., *ICSE*, 2006, 905-908.

[WON 98] WONG K., Rigs User's Manual, 1998.

[XIE 06] XIE X., POSHYVANYK D.MARCUS A., 3D Visualization for Concept Location in Source Code., *ICSE*, 2006, 839-842.