Conceptual Exploration of Software Structure: A Collection of Examples

Richard Cole[1], Thomas Tilley[1] and Jon Ducrou[2]

[1] School of Info. Tech. and Elec. Eng.
University of Queensland
St. Lucia, Australia

[2] School of Comp. Sci. and Info. Tech.
University of Woolongong
Woolongong, Australia

Abstract. Software systems are often highly structured, consisting of artifacts (types, methods, variables, and packages), and relationships between these artifacts. Domain models, meta models, and software design documentation provide additional artifacts such as roles, associations, use cases, and paragraphs of text. This paper describes, and demonstrates the use of a tool for software structure understanding. The tool consists of a knowledge base containing software artifacts, relationships between artifacts, and rules for generating new relationships. The knowledge base is then explored using formal concept analysis (FCA).

Exploration of the software structure via FCA is useful in: (i) understanding the software structure, (ii) the recognition of parts of the software structure violating design principles, (iii) the reorganisation of type and package hierarchies, and (iv) retrieving artifacts of the software development process.

Keywords: formal concept analysis, software engineering

1 Introduction

The design of software systems is fundamentally a human activity and therefore necessarily involves human understanding of software structure. This observation conflicts with the complexity of modern software systems and so there is a need for automated tools which help humans understand software systems. Such tools need to be able to summarise aspects of the software structure so as not to overload the human user with too much information. To do this it seems natural to exploit the explicit generalisation/specialisation in hierarchies and to allow diagrams to range in their level of abstraction from very general to very specific.

The process of software design and implementation often contains many arbitrary decisions, such as the name of methods or variables or indeed sometimes the structure of a class hierarchy. As the design proceeds these decisions need to be reviewed in order to achieve consistent, orthogonal and simple designs. Agile methods, and extreme programming (XP) in particular, advocate regular refactoring activities undertaken to regularise and revise the software structure [8, 2].

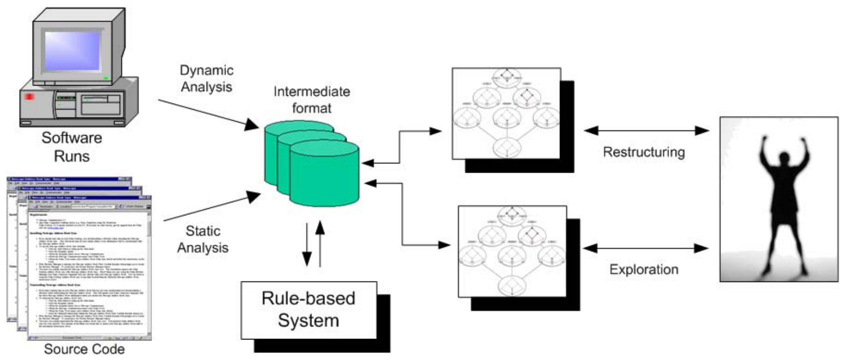

Fig. 1. Architecture to support the Conceptual Analysis of Software Structure (CASS).

Our tool, Conceptual Analysis of Software Structure (CASS), attempts to address these requirements. The architecture of CASS is shown in Figure 1. Source code analysers and profilers produce information which is stored in a knowledge base. A rule based system is then used to extend the knowledge base with new relationships and artifacts. Graph based queries define an aspect of the code to be explored and are used to generate result sets that are visualised using concept lattices. Hypotheses and questions may then be investigated by (i) generating new lattices, perhaps displaying new aspects of the software structure; and (ii) by navigating back to the source artifacts within the software or its documentation. Since each concept lattice is generated from a query graph, a natural refinement ordering allows general views to be elaborated and made more specific. Thus the user is able to progress from a general view to a more specific view, or vice versa. In addition, the theory of formal concept analysis (FCA) allows two or more aspects of the software structure to be combined coherently in nested line diagrams.

This paper presents and discusses examples of concept lattices extracted during the analysis of a piece of software called OntoRama. OntoRama is an open source project of a reasonable size that has been in development for several years, has had several developers, and has gone through a number of significant design changes. Due to these properties we think that OntoRama is a realistic example of software analysis. OntoRama was choosen because although we were not familiar with the software design we have had some access to the software developers who worked on OntoRama. Thus the examples in this paper are constructed from the point of view of a developer who is new to piece of software and seeking to understand it for the purpose of maintaining or extending it. This is a common situation in the software development industry.

The examples have also been choose to illustrate a range of activities made possible within the CASS framework. The first example (Section 4) is an analysis of the call graph between top level packages in OntoRama and gives an indication of how modular the source code is. It shows which packages are dependent on which other packages. The second example shows how an aspect of the call graph can be investigated by unfolding the lattice structure. The third example, still making use of the call graph, focuses attention on two specific packages. The forth example shows how FCA can be used to analyse the naming of packages to gain insight into the software structure. In the fifth example, two different aspects of the software structure — one derived from the call graph information, and a second derived from package naming — may be combined together in a nested line diagram.

Before discussing the examples we first review some related work and then explain how concept lattices are produced in response to graph queries.

2 Related Work

Tilley et al. [21] categorise the use of FCA for software engineering via its application to either early phase or late phase development activities in their survey of FCA approaches to software engineering. Early phase approaches include the identification of candidate classes in use case descriptions [5], reconciling use cases from multiple stake-holders [3], software component retrieval [11, 7], and the visualisation of formal specifications [20].

Late phase approaches focus on the maintenance and reengineering of software. These approaches include; (i) identification of objects [13], class hierarchies or modules in legacy code [10, 15, 12], (ii) restructuring of class hierarchies based on method names [9], method usage [17], associations, or documented properties [14], (iii) analysis of software configurations [16], (iv) proposing a code review order for methods of classes based on call graphs [4], and (v) analysis of execution profile information [1].

There are a number of approaches that store and visualise software structure as graphs. Two well known examples being Rigi [19] and SHriMP[18]. Similar to our approach, Rigi stores software information as triples and thus is compatible and possibly complimentary to our approach.

The novelty of our approach lies in the exploration of program information stored in a knowledge base using FCA. The techniques presented by other authors generally focus on a single aspect of software structure and analyse just that aspect. For example Snelting and Lindig consider preprocessor statements in C programs, while Snelting and Tip consider the static call graph in a C++ program. Our approach provides a mechanism to bring these different aspects of software structure together in a single framework and application. By using a graph based query language we can exploit the implicit structure captured in class hierarchies, package hierarchies, and call graphs to help the user focus on particular aspects of the software structure. Our approach differs from tools such as Rigi and SHriMP in that the data under analysis is organised using concept lattices. These lattices are algebraic structures and convey information about the logic underlying the data. The user must be trained to read concept lattices in the same way that users learn to interpret UML diagrams. Although a full exposition of the mathematics underlying FCA takes several years of study, an understanding of the diagrams is usually acquired in about half an hour [6].

3 From Graph Queries to Concept Lattices

Figure 2 contains an example query graph. Edges in the graph are ternary; meaning they connect three vertices — a subject, a predicate, and an object. The graph has two types of vertex: variables and constants. Variable nodes are distinguished by an underscore, e.g. `_caller`, and `_g`. The graph is matched by finding a morphism from the vertices in the query graph to vertices in the knowledge base that preserves edges and maps constants to themselves.

The graph in Figure 2 will return pairs of methods, a caller and a callee, in which the caller contains a transitive call to the callee. The top level package transitively containing each method is also returned. The pairs of top level packages (one containing a callee, and one containing a caller) are then used for form a binary relation from which a concept lattice is derived.

The concept lattice contains information about which top level packages make calls to which other top level packages. This information is of interest because it shows the modularity of the software package and it also serves as a mechanism to organise the software packages.

In our system a graph query returns a relation with a column for each variable. This relation is then projected to select just the _g and _m columns, thus forming a binary relation. The binary relation is then used as the incidence relation in a formal context. In some instances the objects and attributes of the formal context are derived from the domain and range of the incidence relation, but usually a separate query is used to define the object and attribute sets.

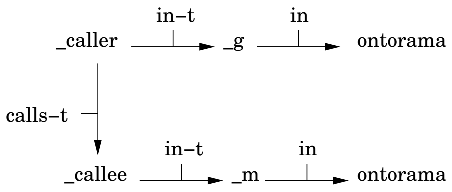

Fig. 2. Query graph extracting the call graph of the top level packages.

In some instances the query is made up of several graphs. In this case a binary relation is returned from the result set of each graph and a union is taken between these binary relations to form the formal context. This allows two aspects of a software structure to be combined within a single diagram.

The graph in Figure 2 can be expressed in a linear form. The linear form is a list of triples separated by commas. Each triple corresponds to an edge in the graph. Two triples can be merged together if the last node of the first triple is the same as the first node of the second triple. The linear form for Figure 2 is:

```
_caller in-t _g in ontorama,
_caller calls-t _callee,
_callee in-t _m in ontorama.
```

The objects of the formal context are retrieved using the following query graph: "`_g in ontorama.`" While the attributes are retrieved using: "`_m in ontorama.`" Having separate queries for the object and attribute sets is important because otherwise objects and attributes that one might expect, for example in a lattice showing the call structure of the top level packages, can otherwise be omitted from the diagram and thus be a cause of confusion. It is also often the case that these otherwise omitted objects and attributes are a point of interest in the diagram. When they are included, as occurs with explicit query graphs for the object and attribute sets, then they are attached to the top and bottom concepts respectively.

For our knowledge base we use a triple store called Kowari[3]. Kowari is designed for storing and retrieving RDF statements and scales to hundreds of millions of triples. Both our graph queries and our rules are translated automatically into iTQL which is a query language of Kowari. iTQL is similar at the syntax level to SQL. Our requirements are somewhat simpler than those of RDF since we: (i) don't make use of namespaces, (ii) don't distinguish between resources and literals, (iii) we don't store anonymous nodes within the database, (iv) are rules have no anonymous nodes in their conclusions and so our rule system is decidable and has a finite closure. As a rough guide it takes about 10 seconds to import OntoRama into Kowari and then another 10 or so seconds to calculate the rule closure. Subseqent queries are usually processed in under a second.

4 Example: Call Graphs

Figure 3 is a concept lattice of the transitive call graph of the top level classes and packages in OntoRama. It shows which top level packages and classes contain calls into which other top level classes and packages.

The lattice was generated from a formal context, itself constructed using the query graphs listed in the Section 3. The incidence relation of the formal context is the transitive closure of the static call graph. The static call graph is calculated from the source code and can be distinguished from the dynamic

[3] See http://www.kowari.org

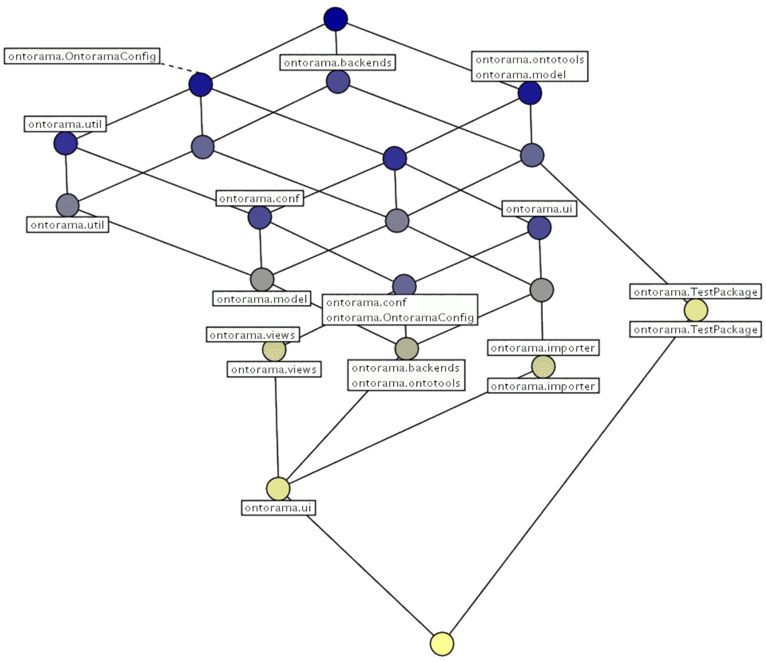

Fig. 3. Lattice showing the transitive call graph between the top level packages and classes of OntoRama. The large intervals between object and attribute concepts for many packages indicates that the software structure is not layered, or at least the package structure does not reflect any layering.

call graph which is generated by recording the actions of a running program. By taking the transitive closure, if method A calls method B which in turn calls method C then we also record a transitive call from A to C.

The reason, in this instance, for taking the transitive closure is to focus on whether or not the top level packages fall neatly into layers with classes from upper layers making calls to classes from lower layers. If such layers exist in the call graph then we expect to see them in the concept lattice. In contrast, if there are cyclic dependencies then we shall expect to see large intervals between the object and attribute concepts for a package.

OntoRama has cyclic dependencies. To understand why these result in large intervals let us consider an example. Consider the interval formed by the object and attribute concepts for `ontorama.ui`. The object concept for `ontorama.ui` is near the bottom of the diagram while the attribute concept for `ontorama.ui` is around the middle. Objects concepts that occur in the interval both make calls into `ontorama.ui` and are themselves called into from within `ontorama.ui`. We know these packages make calls into `ontorama.ui` because they have `onto-rama.ui` as an attribute. We can also assume that `ontorama.ui` calls into these packages because the object concepts for these packages are above the object concept for ontorama.ui, and it is usual for each package to make calls into itself.

The top of the concept lattice is a chain product. With the attributes OntoramaConfig, and util forming one chain, the attribute backends forming another, and the conjunction of ontotools and model forming yet a third. These chains can be thought of as dimensions. The model package uses all four dimensions, while importer leaves out util and TestPackage leaves out both OntoramaConfig and util. The attributes at the top of the diagram can be considered low level packages because they are called into by many other packages.

All most all of these low level packages have large intervals indicating that they both make calls into many packages, and are also called into from within many packages. The exception is util which has a relatively small interval. Even so the util package is still involved in cyclic dependencies with both OntoramaConfig and backends.

The concept for TestPackage sits out to the right being mutually exclusive with many of the attributes in the diagram. It calls very few packages and is itself called by very few packages. The TestPackage class initiates tests stored within the other packages. Since TestPackage only makes calls ontotools and model we can assume that many of the packages don't have testing. Another possibility is that there are other unit tests in the other packages, but they have not been hooked into the TestPackage class.

The lower part of the diagram has to do with ui. The ui package is called by several packages and also makes calls to many packages. We see that the view package is both called by ui and also makes calls into the ui package. We see that view occurs as an attribute fairly low down in the structure indicating that it is a relatively high level package, being called from relatively few other packages, but itself making calls into many other packages. By seeing that the attribute concept for views is on the top plane of the lattice (visually speaking) we can see that it does not make calls into the backends package which is the attribute forming the lower plane of the lattice.

The ui package is somewhat in the middle of the structure both being called by many classes and calling many classes. To see whether the substructure of the ui package reflects this we unfold the ui package. This forms the second example.

5 Example: Unfolding a Package

Figure 4 shows a locally scaled nested line diagram with the contents of the ui package added as attributes to the diagram. In order to do this the query for the objects was modified to be a disjunction:

```
_m in "ontorama". OR _m in "ontorama.ui".
```

The query for the incidence relation was also modified to become a disjunction:

```
_caller in-t _g in ontorama,
_caller calls-t _callee,
_callee in-t _m in ontorama.
OR
_caller in-t _g in ontorama,
_caller calls-t _callee,
_callee in-t _m in "ontorama.ui".
```

The lattice has been drawn as a locally scaled nested line diagram. The attributes of the inner scale are the contents of the `ui` package while attributes of the outer scale are the same in the first Example.

The first interesting aspect is that the inner scale is a very simple lattice being composed of a single chain. By far the majority of the content of `ui` appear as attributes of the bottom concept, which is instantiated for the first time in the object concept for `ui`. This indicates that this part of the `ui` module is only used by the `ui` module itself. The other three parts which are called by other packages are: the class `OntoRamaApp`, the class `ErrorDialog`, and the package `events`. Because `OntoRamaApp` is attached to the top concept of the inner scale we can infer that everything that calls `ui` also calls `OntoRamaApp`. The only package to call `ui` and not call `ErrorDialog` is `ontotools`. While events are accessed (perhaps in the mode of event creation) from the importer and backends packages.

Returning to our initial question of whether or not the organisation of `ui` reflects the usage of `ui` by other packages and classes. We can say that usage of `ui` by other packages is not a primary concern of `ui` since it is not used by many other classes and packages. Therefore it can be said that its package structure is not organised in that way. Constructing the locally scaled nested line diagram however has provided an organisation over the contents of the ui package that is aligned with its usage by other packages.

6 Example: Focused Call Graph

Figure 5 contains a concept lattice that organises the event classes in `onto-rama.ui.events` according to which top level packages they access. The concept lattice was generated from the following query:

```
_caller in-t _g in "ontorama",
_caller calls-t _callee,
_callee in-t _m in "ontorama.ui.events".
```

The concept lattice shows that the `ui` class uses all the events in the package, and also that, the packages `importer`, `views` and `backends`, each have their own events that are not shared with the other packages.

It is curious that of three events, according to their names, are associated with the life cycle of a query, they are each accessed separately from the three different packages. `QueryEndEvent` is access from `importer`, `QueryStartEvent` is accessed from `backends`, and `QueryCancalledEvent` is accessed from `ui`. It

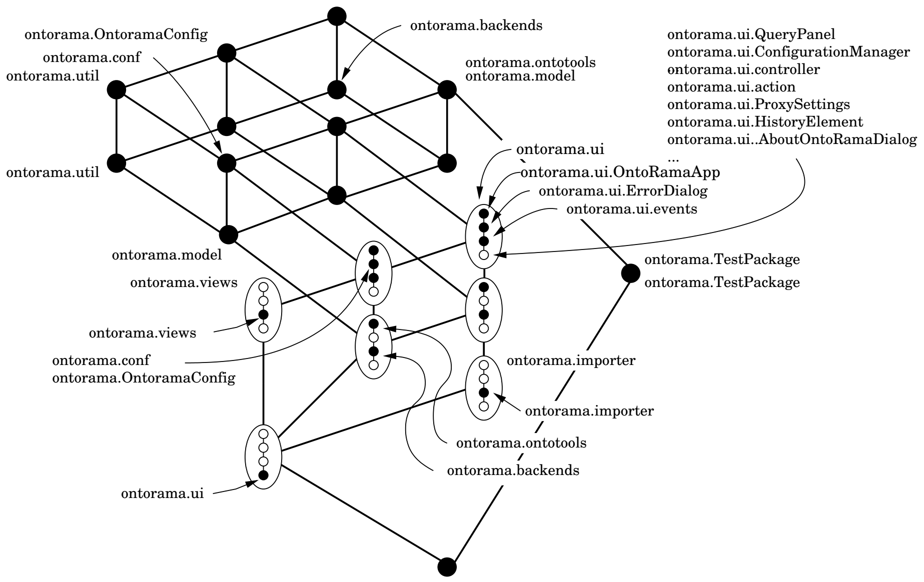

Fig. 4. Locally scaled nested lattice showing the call graph of the top level packages; formal objects call formal attributes.

could be that these events are being generated respectively in each of the three packages. This hypothesis can be verified by submitting the following query[4]:

```
_caller in-t _g in "ontorama",
_caller calls _callee in _m,
_m in-t "ontorama.ui.events",
_callee has-name "<init>"
```

This process of generating concept lattices, forming hypothesis and then generation more concept lattices to verify these hypotheses can continue at length as the programmer becomes slowly more familiar with the structure of the software. In the interests of showing a variety of techniques we leave this thread here and move on to the next example.

7 Example: Package Names

Packages in Java are identified by path names. The path names contain names separated by full stops and usually are used to place the packages within a tree structure. It is interesting however to consider the names in the path name of a package as attributes in a concept lattice. This has been done for the packages contained in the `org.ontorama.model` and the result is shown in Figure 6.

[4] `<init>` is the internal name of Java constructor methods.

The concept lattice exhibits quite a lot of structure. It is the semi-product of an M3 and an M2. The M2 says that the `model` is divided into `graph` and `tree`, while the M3 says that the model is divided into `controller`, `events`, and `text`. The semi-product produces all combinations between these two structures.

One may surmise that the reason such an arrangement of the packages exists within OntoRama is that one of the main developers was familiar with formal concept analysis. Even so there is an irregularity in the diagram. The attribute concept for test has an object in its contingent; the package `ontorama.model.test`.

It is somewhat curious that both `controllers` and `events` could be partioned into `graph` and `tree` packages and that there were no `controllers` or `events` that are shared between `graph` and `tree`. One hypothesis is that such events are also shared perhaps with the `ui` or view packages and thus are to be found in `ontorama.events` or `ontorama.controllers`. Looking back to the lattice in Figure 3 we see that there is indeed neither an `ontorama.events` nor an `ontorama.controller` package and thus we can conclude that all events and controllers associated with the model have been assigned to either the graph or the view.

8 Example: Combining Aspects

Two aspects of the software structure can be combined together within a nested line diagram. In software structure it is often the case that different aspects are correlated and a nested line diagram provides a mechanism to look for, and examine, any such correlations.

Figure 7 combines two different aspects of software structure. It combines the static call graph that we saw in the first three examples with the package names that we saw in the previous example. The inner scale organises sub-packages of the model package according to names in their pathnames, while the outer scale organises the packages according to which sub-package and classes in the view class they contain calls into.

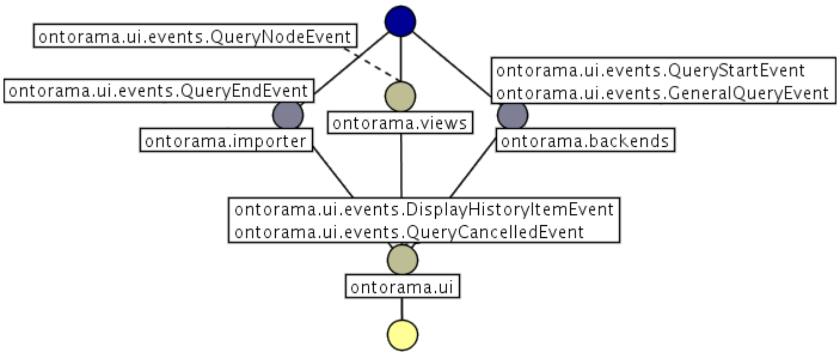
ontorama.ui.events.QueryNodeEvent
ontorama.ui.events.QueryEndEvent
ontorama.ui.events.QueryStartEvent
ontorama.ui.events.GeneralQueryEvent
ontorama.views
ontorama.importer
ontorama.backends
ontorama.ui.events.DisplayHistoryItemEvent
ontorama.ui.events.QueryCancelledEvent
ontorama.ui

Fig. 5. Concept lattice showing which packages access which `ui` events.

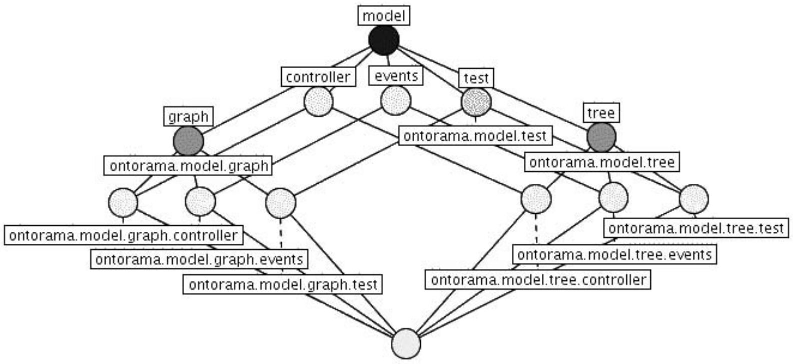
model
controller
events
test
graph
tree
ontorama.model.test
ontorama.model.graph
ontorama.model.tree
ontorama.model.graph.controller
ontorama.model.graph.events
ontorama.model.graph.test
ontorama.model.tree.test
ontorama.model.tree.events
ontorama.model.tree.controller

Fig. 6. Lattice showing the names of packages contained in `ontorama.model`.

Looking inside the top concept of the outer scale we can see that many of the packages don't contain calls into the view package. We can tell this because the object concept contained in the top concept of the outer scale has no out scale attributes. The packages not making calls in the view are: `graph.events` and `graph.test` and `tree.test`. The `graph.controller` contains a call to `views.textDescription`, while `tree.controller` contains calls to `view.tree` and `view.hyper`. This suggests that `view.tree` and `view.hyper` are views of the tree while, `views.textDescription` is a view related to the graph. Interestingly we see that `tree.events` contains calls to both `views.tree` and `views.textDescription`, breaking the separation of the views into tree and graph views. The irregularity is curious and can be further investigated by considering the calls made from `tree.events` into `views.textDescription` and the calls made from `graph.controller` into `views.textDescription`.

9 Conclusion

In summary, this paper has demonstrated via a number of concrete examples how FCA, when used in conjunction with a knowledge base augmented by a rule based system, can be used to explore software structure from different points of view. We began with a very general lattice summarising the static call graph for the whole program and then elaborated the diagram to look inside the package. Having then become interested in a specific package we zoomed in and organised its components based on usage by the top level packages. Changing tack we then explored the names used in the package structure, before combining two aspects of software structure — package naming of components in the model package, and usage by view components. The examples presented here form only a small part of the exploration techniques afforded by our system.

Our approach is primarily aimed at human understanding of software structure with the purpose of refactoring the software to make it simpler and more

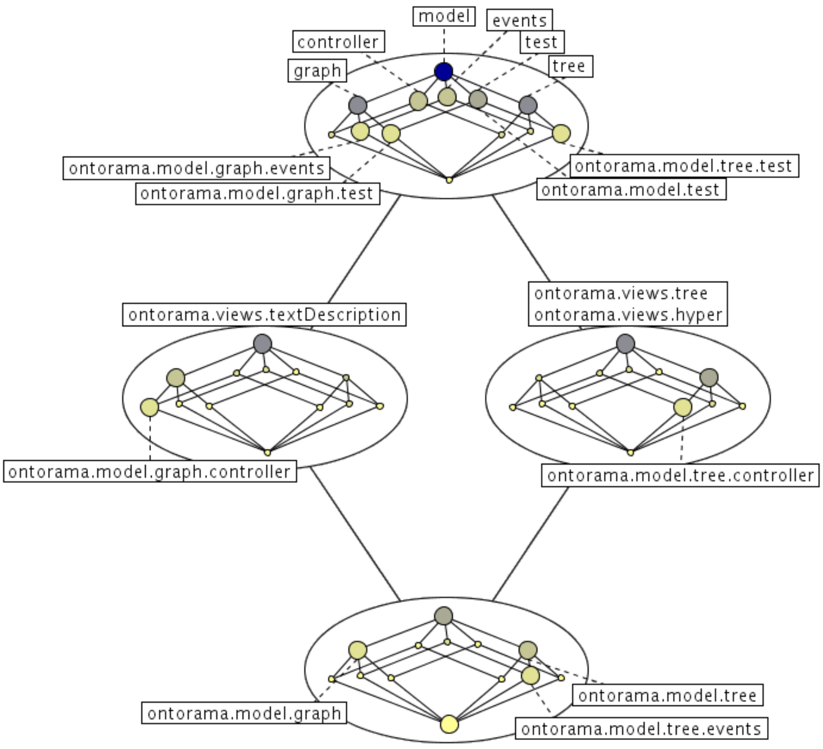
model
events
controller
test
graph
tree
ontorama.model.graph.events
ontorama.model.graph.test
ontorama.model.tree.test
ontorama.model.test
ontorama.views.textDescription
ontorama.views.tree
ontorama.views.hyper
ontorama.model.graph.controller
ontorama.model.tree.controller
ontorama.model.tree
ontorama.model.graph
ontorama.model.tree.events

Fig. 7. Lattice combining the call graph from `ui.model` into `ui.views` and the names of packages in `ui.model`.

uniform. Buy understanding the software structure and identifying places where it is irregular or doesn't meet with expectations the user is able to identify opportunities for refactoring.

Acknowledgements

This research was supported by a Postdoctoral Fellowship granted by the Australian Research Council, an Australian Postgraduate Award, and the Distributed System Technology Centre. We gratefully acknowledge the contribution of Peter Becker, and Nataliya Roberts. ToscanaJ[5] was used for the preparation of concept lattice diagrams.

References

1. T. Ball. The concept of dynamic analysis. In *Proc. of ACM SIGSOFT Symposium on the Foundations of Software Eng*, pages 216–234, September 1999.

[5] See http://toscanaj.sf.net

2. K. Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.
3. K. Böttger, R. Schwitter, D. Richards, O. Aguilera, and D. Mollá. Reconciling use cases via controlled language and graphical models. In *INAP'2001 - Proc. of the 14th Int'l Conf. on Applications of Prolog*, pages 20–22, Japan, October 2001. University of Tokyo.
4. U. Dekel. Applications of concept lattices to code inspection and review. In *The Israeli Workshop on Programming Languages and Development Environments*, chapter 6. IBM Haifa Research Lab, IBM HRL, Haifa University, Israel, July 2002.
5. S. Düwel and W. Hesse. Bridging the gap between use case analysis and class structure design by formal concept analysis. In J. Ebert and U. Frank, editors, *Modelle und Modellierungssprachen in Informatik und Wirtschaftsinformatik. Proc. "Modellierung 2000"*, pages 27–40, Koblenz, 2000. Fölbach-Verlag.
6. Peter Werner Eklund, Jon Ducrou, and Peter Brawn. Information visualization using concept lattices : Can novices read line diagrams? In Peter Eklund, editor, *Proc. of the 2nd Int. Conference on Formal Concept Analysis*, volume 2691 of *LNAI*, pages 57–72. Springer-Verlag, 2004.
7. B. Fischer. Specification-based browsing of software component libraries. In *Automated Software Eng*, pages 74–83, 1998.
8. M. Fowler. *Refactoring, Improving the Design of Existing Code.* Addison Wesley, 1999.
9. R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of class hierarchies based on concept (galois) lattices. *Theory and Application of Object Systems (TAPOS)*, 4(2):117–134, 1998.
10. T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. Technical Report SEN-R0017, Centrum voor Wiskunde en Informatica, July 2000.
11. C. Lindig. Concept-based component retrieval. In J. Köhler, F. Giunchiglia, C. Green, and C. Walther, editors, *Working Notes of the IJCAI-95 Workshop: Formal Approaches to the Reuse of Plans, Proofs, and Programs*, pages 21–25, August 1995.
12. C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. of the Int'l Conf. on Software Eng (ICSE 97)*, pages 349–359, Boston, 1997.
13. H.A. Sahraoui, W. Melo, H. Lounis, and F. Dumont. Applying concept formation methods to object identification in procedural code. In *Proc. of Int'l Conf. on Automated Software Eng (ASE '97)*, pages 210–218. IEEE, November 1997.
14. S. Schupp, M. Krishnamoorthy, M. Zalewski, and J. Kilbride. The "right" level of abstraction - assessing reusable software with formal concept analysis. In G. Angelova, D. Corbett, and U. Priss, editors, *Foundations and Applications of Conceptual Structures - Contributions to ICCS 2002*, pages 74–91. Bulgarian Academy of Sciences, 2002.
15. M. Siff and T. Reps. Identifying modules via concept analysis. In *Proc. of the Int'l Conf. on Software Maintenance*, pages 170–179. IEEE Computer Society Press, 1997.
16. G. Snelting. Reegineering of configurations based on mathematical concept analysis. Technical report, Technische Universität Braunschweig, 1996.
17. G. Snelting and F. Tip. Reengineering class hierarchies using concept analysis. Technical Report RC 21164(94592)24APR97, IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA, 1997.

18. M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM'95)*, 1995.
19. Margaret-Anne D. Storey, Kenny Wong, and Hausi A. Muller. Rigi: A visualization environment for reverse engineering. In *International Conference on Software Engineering*, pages 606–607, 1997.
20. T. Tilley. Towards an fca based tool for visualsing formal specifications. In *Contributions to ICCS 2003*. Springer-Verlag, 2003. To Appear.
21. T. Tilley, P.eklund, R. Cole, and P. Becker. A survey of formal concept analysis support for software eng activities. In *Proc. of the First Int'l Conf. on Formal Concept Analysis, ICFCA03*. Springer-Verlag, 2003. To Appear.