Reverse Engineering with Logical Coupling

Marco D'Ambros
Faculty of Informatics
University of Lugano

Michele Lanza
Faculty of Informatics
University of Lugano

Abstract

Evolutionary information about software systems has proven to be a good resource to complement existing reverse engineering approaches, because it helps in giving a historical perspective of the system to be reverse engineered. Moreover, it provides additional types of information that are not present when only one version of a system is considered. Logical coupling, the implicit dependency between artifacts which changed together, is one example of such information. However, the recurrent problem is that such information comes in large amounts and must be processed to be useful for the reverse engineering of a system.

In this paper we propose an approach to use logical coupling information at different levels of abstraction to detect areas in the system which may lead to maintenance problems. They represent a good starting point to decrease the coupling in the system. Our approach uses an interactive visualization technique called the Evolution Radar*, which can effectively break down the amount and complexity of the logical coupling information. We present our technique in detail and apply it on a large open-source software system.*

1 Introduction

The analysis of the evolution of software [15], has two main goals, namely to infer causes of its current problems, and to predict its future development. Many approaches based on evolutionary information demonstrated that not only can such information be used to predict the future evolution [16, 23], but it can also provide good starting points on where to start reengineering activities [11].

The history of a software system also holds information about *logical coupling*. These are implicit and evolutionary dependencies between the artifacts of a system which, although potentially not structurally related, evolve together and are therefore linked to each other from an evolutionary point of view. In short, logically coupled entities have changed together in the past and are thus likely to change in the future. Logical coupling information reveals potentially misplaced artifacts in a software system, because entities that evolve together should be placed close to each other for cognitive reasons: A developer who modifies a file in a system could forget to modify related files because they are placed in other subsystems or packages.

Logical coupling information can be used to find good starting points for a reengineering process, because logically coupled artifacts lead to maintenance problems: Decreasing the logical coupling in a system leads to an improved system structure.

The problem is that although seemingly lightweight (the analysis of logical coupling does not imply parsing entire versions of the system, but only the simple log-files of versioning systems), extracting the logical couplings leads to vast quantities of information that must be processed and understood. The current approaches based on logical coupling work at two distinct levels of abstraction: (i) the module or package level [9] or (ii) class, file and finer-grained levels [10, 18, 27]. In focusing only on one of those levels either we lose the details or we lose the big picture.

In this paper we propose an integrated technique to inspect logical coupling relationships, which combines information both at a module-level (which subsystems are coupled with each other) and at a file-level (which files are responsible for the logical couplings). Our technique is based on a specific visualization called *Evolution Radar* [6,8]. We decided to use visualization [20, 24] because it provides effective ways to break down the complexity of information, and because it has proven to be a successful means to study the evolution of software systems [1, 5, 12, 13, 17, 22, 23]. By rendering logical coupling information in an intuitive and simple way, we enable a developer to study and inspect these hidden dependencies and guide him to the responsible files.

Structure of the paper. In Section 2 logical coupling is described in detail. In Section 3 we introduce our approach based on the *Evolution Radar* to render logical coupling information. We validate our technique on a large software system in Section 4. Benefits and shortcomings of the approach are discussed in Section 5. In Section 6 we look at

related work and we conclude by summarizing our contributions in Section 7 .

2 Logical Coupling

Logical coupling is the implicit dependency between two or more software artifacts that have been observed to frequently change together during the evolution of a system. This co-change information can either be present in the versioning system, or must be inferred by analysis.

The analysis of logical coupling is useful for reverse engineering because of two reasons:

1. It can reveal dependencies that are not structural, and therefore are not present in the code or in the documentation. These dependencies are the most troublesome and tend to be sources of bugs in software projects.

2. It is more lightweight than structural analysis, as we need to analyze a smaller amount of data, *i.e.,* only the data provided by the CVS log files. Moreover, as it works at text level, it can analyze systems written in different languages without the trouble of parsing and analyzing the data.

The concept was first introduced by Gall *et al.* [9] to detect implicit relationships between modules, by using information extracted from the CVS versioning control system. They used logical coupling to analyze the dependencies between the different modules of a large telecommunications software system and showed that the approach could be used to derive useful insights on the architecture of the system.

Later the same authors revisited the technique to work at a lower abstraction level. They detected logical couplings at class level [10] and validated it on 28 releases of an industrial software system. The authors showed through a case study that architectural weaknesses such as poorly designed interfaces and inheritance hierarchies could be detected based on logical coupling information.

Ratzinger *et al.* [18] used the same technique for analyzing the logical coupling at the class level with the aim of learning about, and improving the quality of the system. To accomplish this, they defined *code smells* based on the logical coupling between classes of the system.

Working at a finer granularity level, Zimmermann *et al.* [27] used the information about changes that are occurring together to predict entities that are likely to be modified when one is being modified.

The main problem with the mentioned approaches is that they either work at the architecture level, *i.e.,* without knowing which finer-grained entities cause the logical coupling, or they work at the file (or even finer) granularity level, *i.e.,* losing the global view of the system.

In this paper we propose an approach to overcome this shortcoming by means of a visualization technique called the *Evolution Radar*, presented next.

3 The Evolution Radar

The Evolution Radar is a visualization technique which renders dependencies between groups of entities. It is implemented as an extension of the BugCrawler tool [7]. In this paper we use it to visualize the logical coupling between files and groups of files, *i.e.,* modules or subsystems.

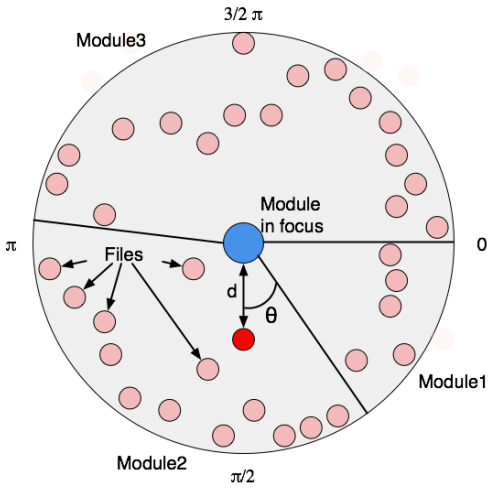


Figure 1. The Evolution Radar structure.

In Figure 1 we see the principles of the Evolution Radar: The module in focus is visualized as a circle and placed in the center of a pie chart. All the other system modules are represented as sectors. The size of the sectors is proportional to the number of files contained in the corresponding module. The sectors are sorted according to this size metric, *i.e.,* the smallest is placed at 0 radian and then all the others clockwise (see Figure 1). Within each sector files are represented as colored circles and positioned using polar coordinates where the angle and the radius are computed according to the following rules:

- *Radius* d (or distance from the center). It is inversely proportional to the logical coupling the file has with the module in focus, *i.e.,* the more they are coupled, the closer the circle (representing the file) is to the center circle (representing the module in focus).

- *Angle* θ. The files of each module are alphabetically sorted considering the entire directory path, and the circles representing them are then uniformly distributed in the sectors with respect to the angle coordinates.

We can map arbitrary metrics on the color and the size of the circle figures. For example for the color a color-temperature mapping is used where pure blue represents the lowest value and pure red the highest.

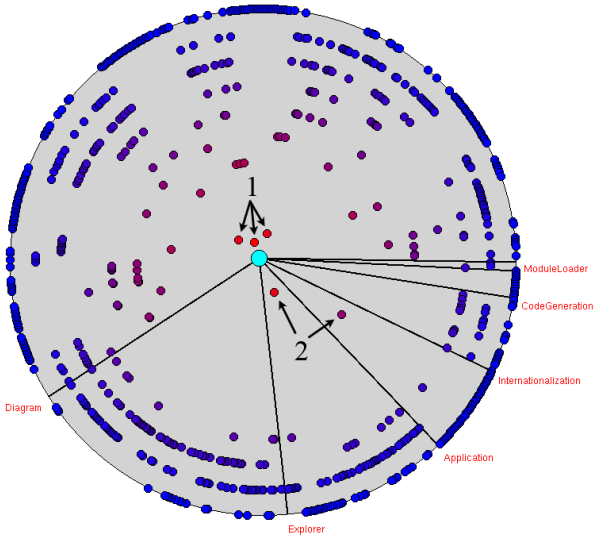


Figure 2. An example Evolution Radar applied on the *Model* module of ArgoUML.

Example. Figure 2 shows an example Evolution Radar visualizing the coupling between the *Model* module (represented as the cyan circle in the center) and all the other modules of ArgoUML[1] (represented as the sectors). The size of the figures is fixed and the color metric is the same as the distance, *i.e.,* the logical coupling. We see that the *Diagram* module is the largest and most coupled module. The three files marked as 1 in the figure are the ones with the strongest coupling. They should be further analyzed to understand which is the most appropriate module to contain them: *Model* or *Diagram*. For the remaining modules the coupling is not as strong as for *Diagram* but we see the presence of some outliers (files for which the coupling is much higher with respect to their context). The two files marked as 2, belonging to the *Application* and *Internationalization* modules, have a very strong coupling with respect to the other files belonging to the same modules. They should also be analyzed and moved in case they belong to the wrong module.

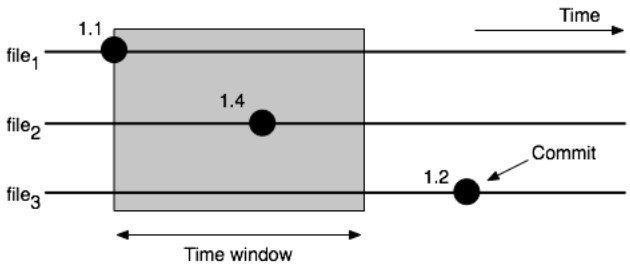
3.1 Logical Coupling Measure

In the Evolution Radar files are placed according to the logical coupling they have with the module in focus. To compute this metric value we use the following formula:

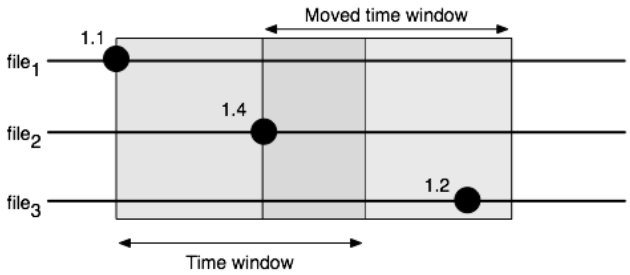
$$LC(M, f) = \max_{f_i \in M} LC(f_i, f) \quad (1)$$

$LC(M, f)$ is the logical coupling between the module in focus M and a given file f and $LC(f_i, f)$ is the coupling between the files f_i and f. It is also possible to use other group operators instead of the maximum like the average or the median. We use the maximum because it points us to the files with the strongest coupling, *i.e.,* the main responsible for the module dependencies.

The value of the coupling between two files is equal to the number of transactions which include both files. Since transactions are not recorded by CVS we reconstruct them using the sliding time window approach proposed by Zimmermann and Weißgerber in [26], which is an improvement of the simpler fixed time window approach.



(a) Fixed time window

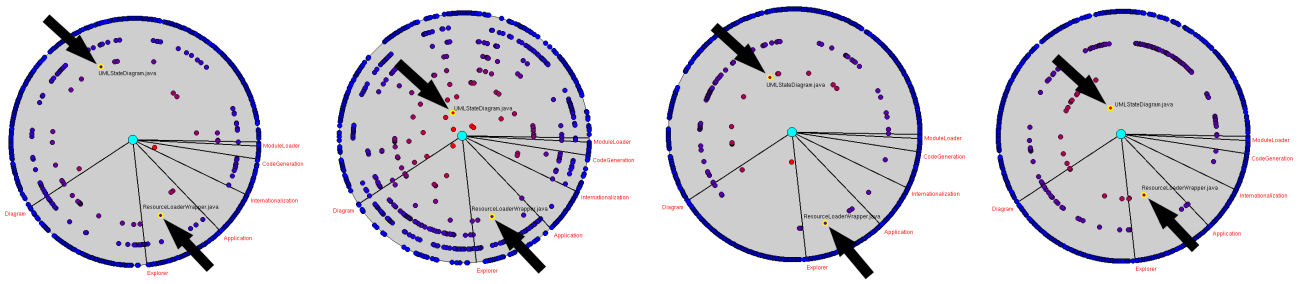


(b) Sliding time window

Figure 3. Fixed and sliding time window.

Figure 3(a) shows an example of both techniques. In the fixed time window approach the beginning of the time window is fixed to the first commit (file$_1$, version 1.1). Then all the other commits with a timestamp included in the window are considered to be in the same transaction (only file$_2$ version 1.4). With the sliding window approach the beginning of the time window is moved to the most recent commit recognized to be in the transaction. By doing this, file$_3$ version 1.2 is also included in the transaction. The transactions reconstructed using the sliding time window include commits which take longer than the size of the time window. As in [26] we use a time window of 200 seconds.

[1] ArgoUML is an UML modeling tool written in Java. It is available at: http://argouml.tigris.org.



(a) January - June 2004 (b) June - December 2004 (c) January - June 2005 (d) June - December 2005

Figure 4. The logical coupling evolution of the *Model* module of ArgoUML. Moving through time, the Evolution Radar can keep track of certain files (yellow border).

3.2 Interaction

The Evolution Radar is implemented as an interactive visualization. This is not just a feature, but a constraint to exploit its full potential. It is possible to inspect all the entities visualized, *i.e.,* files and modules, to see commit-related information like author, timestamp lines added and removed *etc.* Moreover, it is also possible to see the source code of selected files. Three important features for performing analyses with the Evolution Radar are (1) moving through time, (2) tracking and (3) spawning.

Moving through Time. The logical coupling measure is relative to a period of time. We compute it either considering the entire history of files or with respect to a given time window, *i.e.,* the Evolution Radar is time dependent. When creating the radar the user can divide the lifetime of the system into time intervals. For each of them a different radar will be created, and the logical coupling is computed with respect to the given time interval. The radius coordinate has the same scale in all the radars, *i.e.,* the same distance in different radars represents the same value of the coupling. This makes it possible to compare radars and to analyze the evolution of the coupling over time. In our tool implementation the user "moves through time" by using a slider, which causes the corresponding radar to be displayed.

By considering the entire history, the obtained value of the logical coupling takes into account all the changes. Since we want to understand design problems in the current version of the system we are more interested in recent changes and coupling. Moreover, we are also interested in understanding the sources of these current problems, which can be far in the past. A good solution consists in dividing the lifetime of the system into time intervals (*e.g.,* six to three months) and then create the corresponding Evolution Radars. By inspecting the last one we can detect recent couplings and then, by moving the time, we can see whether these couplings were strong or not in the past, *i.e.,* in the previous time intervals.

Tracking. This feature allows the user to keep track of files over time. When a file is selected for tracking in a visualization related to a particular time interval, it is highlighted in all the radars (with respect to all the other time intervals) in which the file exists. Figure 4 shows an example of tracking through four radars, related to four consecutive time intervals, from January 2004 to December 2005. The highlighting consists in using a yellow border for the tracked files and in showing a text label with the name of the file (indicated with arrows in Figure 4). Like this it is possible to detect files with a strong logical coupling with respect to the last period of time and then move the time and analyze the coupling in the past. This allows the distinction between persistent and recent logical coupling.

Spawning. The spawn feature is aimed at inspecting the logical coupling details. Outliers indicate that the corresponding files have a strong coupling with certain files of the module in focus, but we ignore which ones. To uncover this dependency between files we spawn a secondary Evolution Radar as follows (see Figure 5):

- *Group.* The outliers are grouped to form a temporary module M_t represented by a circle figure.
- *Expand.* The module in focus (M) is expanded, *i.e.,* a circle figure is created for each file composing it.
- *Display.* A new Evolution Radar is created. The temporary module M_t is placed in the center of the new radar. The files belonging to the module previously in focus (M) are placed around the center. The radius coordinate, *i.e.,* the distance from the center, is inversely proportional to the logical coupling they have with the module in the center M_t. For the angle coordinate alphabetical sorting is used. Since all the files belong to the same module there is only one sector.

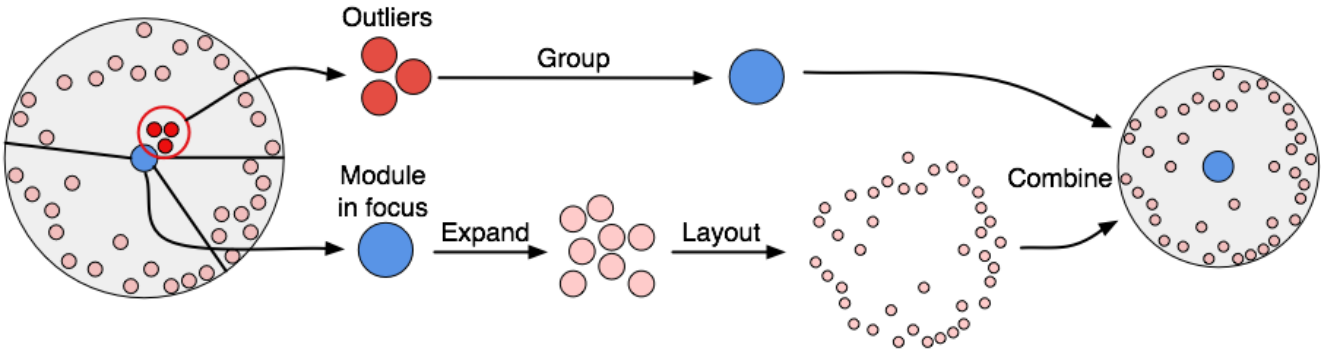


Figure 5. Spawning Evolution Radars.

3.3 Discussion

The Evolution Radar has several visual advantages: It occupies a settable amount of screen space, *i.e.,* it is always possible to visualize the whole radar on screen, independent of its resolution. It is rotation invariant like Chuah's time wheels [4]. It does not visualize the coupling relationships as edges and therefore does not suffer from overplotting: The radar always remains intelligible, *i.e.,* it is easy to make out the heavily coupled modules which are displayed as "spikes" pointing to the center. It is also easy to make out single files responsible for the coupling which are placed close to the center.

This visualization technique can be enriched by adding more structural information. A sector can be further divided in sub-sectors, using both the radius and the angle coordinates, for visualizing sub-groups, *e.g.,* sub-modules, directories *etc.* This visual decomposition is proposed in [21].

The Evolution Radar is a general visualization technique, *i.e.,* it is applicable to any kind of entities. The only requirement is to define a group criterion and a distance metric. Possible examples are:

- Visualization of classes for architecture recovery using the package as group criterion and the number of invocation for the distance.

- Visualization of developers for accessing the team structure. The group criterion is the team and the distance is computed according to the number of files two authors share, *i.e.,* both of them committed the files at least once.

The main drawback, as with all visualizations, it that it requires a trained eye to interpret the visualization. In our experience this is not a major problem, and most people to whom we presented the radar had few problems understanding and using it.

4 Validation

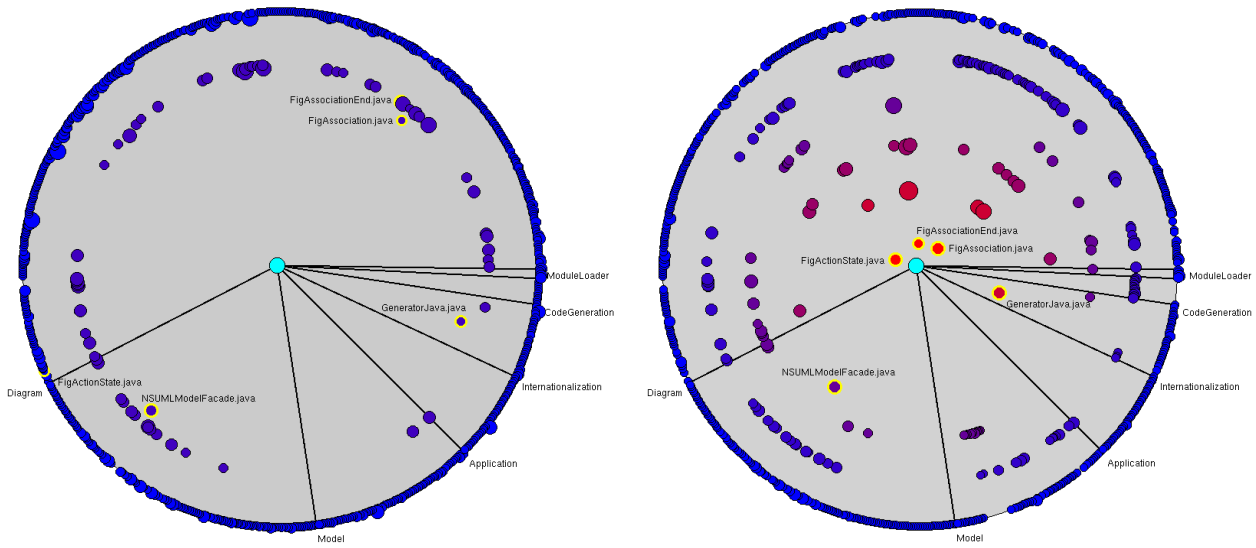
To validate our approach we applied it on ArgoUML, an open-source UML modeling tool, consisting of more than 200,000 lines of code.

From its documentation on the web site we know the system decomposition in modules. We did not consider some modules for which the documentation says "They are all insignificant enough not to be mentioned when listing dependencies". We focused our analysis on the three largest modules: *Model*, *Explorer* and *Diagram*. From the documentation we know that *Model* is the central module that all the others rely and depend on. *Explorer* and *Diagram* do not depend on each other.

For the initial analysis we created a radar for every six months of the system's history. We started the study from the most recent one, since we are interested in problems in the current version of the system. Using a relatively short time interval (six months) ensures that the coupling is due to recent changes and is not "polluted" by commits far in the past. As metrics we used the logical coupling for both the position and the color of the figures. The size (the area) is proportional to the total number of lines modified in all the commits performed during the considered time interval.

Figure 6(b) shows the Evolution Radar for the last six months of history of the *Explorer* module. From the visualization we see that the coupling with *Diagram* is much stronger than the one with *Model*, although the documentation states that the dependency is with *Model* and not with *Diagram*. The most coupled files in *Diagram* are *FigActionState.java*, *FigAssociationEnd.java*, *FigAssociation.java*. Using the tracking feature, we found out that these files have only been recently coupled with the *Explorer* module. In the other radar (Figure 6(a), showing the previous six months) they are not close to the center. This implies that the dependency is due to recent changes only.

To inspect the logical coupling details, we used the



(a) From January to June 2005.

(b) From June to December 2005.

Figure 6. Evolution Radars applied to the *Explorer* module for the year 2005.

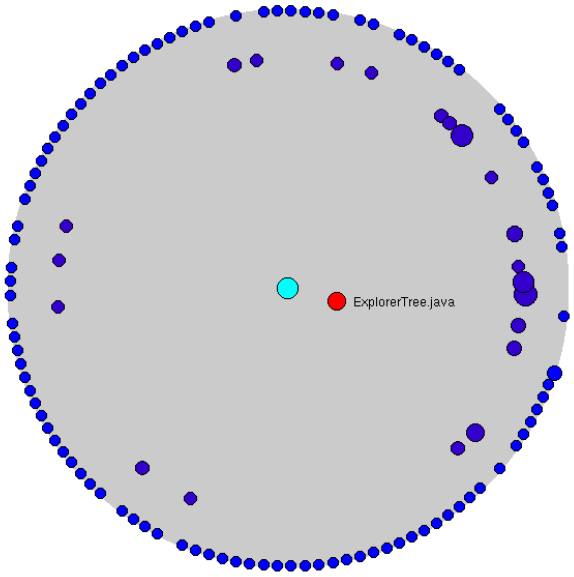


Figure 7. Details of the logical coupling between the *Explorer* module and the files *FigActionState.java*, *FigAssociationEnd.java* and *FigAssociation.java*.

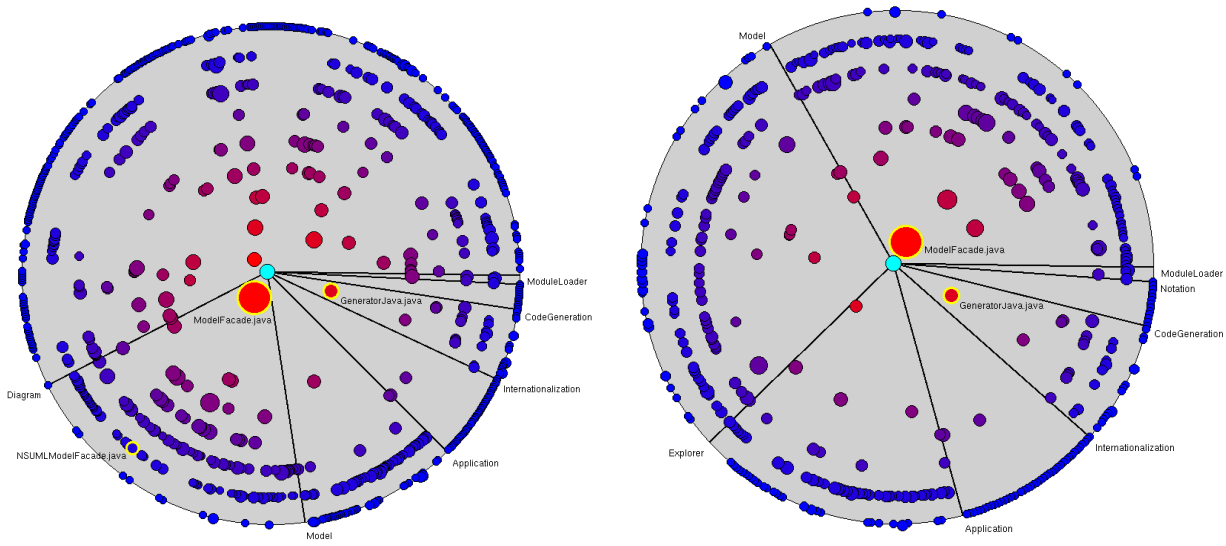
spawning feature: We grouped the three files and we generated another radar, shown in Figure 7 having this group as the center. We now see that the dependency is mainly due to *ExplorerTree.java*. The high-level dependency between two modules is thus reduced to a dependency between four files. These four files represent a problem in the system, because modifying one of them may break the others. The fact that they belong to different modules makes it easier to forget this hidden dependency.

The visualization in Figure 6(b) shows that the file *GeneratorJava.java* is an outlier, since its coupling is much stronger with respect to all the other files in the same module (*CodeGeneration*). By spawning the group composed of *GeneratorJava.java* we obtained a visualization very similar to Figure 7, in which the main responsible for the dependency is again *ExplorerTree.java*. Reading the code revealed that the *ExplorerTree* class is responsible for managing mouse listeners and generating names for figures. This explains the dependencies with *FigActionState*, *FigAssociationEnd* and *FigAssociation* in the *Diagram* module, but does not explain the dependency with *GeneratorJava*.

The past (see Figure 6(a) and Figure 8(a)) reveals that *GeneratorJava.java* is an outlier since January 2003. This long-lasting dependency indicates design problems.

A further inspection is required for the *ExplorerTree.java* file in the *Explorer* module, since it is the main responsible for the coupling with the modules *Diagram* and *CodeGeneration*.

The radars in Figure 6(b) and Figure 6(a) show that during 2005 the file *NSUMLModelFacade.java* in the *Model*



(a) Explorer module.

(b) Diagram module.

Figure 8. Evolution Radars of the *Explorer* and *Diagram* modules from June to December 2004.

module had the strongest coupling with *Explorer* (module in the center). Going six months back in time, from June to December 2004 (see Figure 8(a)), we see that the coupling with *NSUMLModelFacade.java* was weak, while there was a very strong dependency with *ModelFacade.java*. This file was also heavily modified during that time interval, given its dimension with respect to the other figures (the area is proportional to the total number of lines modified). *ModelFacade.java* was also strongly coupled with the *Diagram* module (see Figure 8(b)). By looking at its source code we found out that this was a God class [19] with thousands of lines of codes, 444 public and 9 private methods, all static. The *ModelFacade* class is not present in the other radars (Figure 6(b) and Figure 6(a)) because it was removed from the system the 30th of January 2005. By reading the source code of the most coupled file in these two radars, *i.e., NSUMLModelFacade.java*, we discovered that it is also a very large class with 317 public methods. Moreover, we found out that 292 of these methods have the same signature of methods in the *ModelFacade* class[2], with more that 75% of the code duplicated. *ModelFacade* represented a problem in the system and thus was removed. Since many methods were copied to *NSUMLModelFacade*, the problem has just been relocated!

This example shows how historical information can reveal problems, which are difficult to detect with only one version of the system. Knowing the evolution of *ModelFacade* helped us in understanding the role of *NSUMLModelFacade* in the current version of the system.

As a final scenario, we analyzed the evolution of the logical couplings of the *Explorer* module with all the others. From Figure 8(a), we see that from June to December 2004 the couplings were very strong. Then, from January 2005 to June 2005 (Figure 6(a)), they decreased a lot. This suggests that in the previous period the module was restructured and its quality was improved, since in the next time interval the couplings with the other modules were weak. The effort spent for the restructuring can be seen from the size of the figures, representing the total number of changed lines. In the radar relative to June - December 2004 (Figure 8(a)) the figures are bigger than in the radar relative to January - June 2005 (Figure 6(a)). At the end of the restructuring phase, the class *ModelFacade* was removed. From June to December 2005 (see Figure 6(b)) the coupling increased again. This can be related to a new restructuring phase.

For lack of space we cannot present an in-depth analysis of ArgoUML. We instead showed examples of how to use the Evolution Radar to detect problematic parts of the system, which represent good candidates for reengineering. The main results we found in the discussed example scenarios are:

- The *Diagram* and *Explorer* modules are the most coupled. Since this dependency is not mentioned in the module relationships page in the documentation, either the modules should be restructured to decrease the coupling or the documentation should be updated. We identified the four files mainly responsible for this hid-

[2]With the difference that in *NSUMLModelFacade* the methods are not static and that it contains only two attributes, while *ModelFacade* has 114 attributes.

den dependency.

- The files *GeneratorJava.java* in the *CodeGeneration* module and *ExplorerTree.java* in the *Explorer* module should be further analyzed and, in case, refactored. *GeneratorJava.jav*a has a persistent coupling with the *Explorer* module, while *ExplorerTree.java* is coupled with both *CodeGeneration* and *Diagram*.

- Two problematic classes were detected: *ModelFacade* and *NSUMLModelFacade*. Most of the methods of the first class were copied to the second one, and then *ModelFacade* was removed from the system.

- The evolution of the coupling between *Explorer* and all the other modules was studied. Different phases were identified, where two of them are likely to be restructuring phases.

5 Discussion

The main benefits of the Evolution Radar consists in its simplicity and scalability. It is a lightweight approach which breaks down huge amounts of complex data, visualizing at the same time information at different levels of abstraction, *i.e.,* information about modules and individual files. Other important advantages come from the interaction capabilities. Each file in the visualization can be inspected and its code can be read on-the-fly. This code proximity allows the user to immediately figure out design issues as God classes or code duplication, as we did with *ModelFacade* and *NSUMLModelFacade* in the previous section. In these cases the reasons which generated the logical coupling were detected without any additional analysis. In other cases our approach guides us in the detection of the dependencies. To uncover the design issues behind the coupling a further analysis is required. However this analysis will be a lot simplified with respect to the dimension of the system and the modules, since using the Evolution Radar the coupling between modules is reduced to small set of files by means of the spawning feature.

A last benefit of our approach is the control of time. Instead of summarizing the coupling computed for the entire history of the system in a single value (per each pair of files), the Evolution Radar shows how the coupling evolved over time. This is helpful to discriminate between durable dependencies and coupling due to recent changes only (using the tracking feature). It is also possible to identify phases in the system or in the modules by seeing if the logical coupling is ameliorating or degrading.

Concerning shortcomings, an authority system decomposition (in terms of modules or subsystems) is required to apply the Evolution Radar. This documentation can be hard to find, incomplete or even absent. In such a case using the package structure is a good trade-off. Once the decomposition is found, an initial visualization showing all the modules and their relationships (such as the one provided in [17]) is a good starting point.

The radar visualization may suffer from the outliers problem, *i.e.,* files having a logical coupling much higher with respect to the average value. This files may deform the visualization by pushing all the other files to the boundary of the radar[3]. In such cases the solution consists in computing the distance from the center according to the square root or, if the gap is large, according to the logarithm of the coupling value.

The Evolution Radar tool has been improved using the experience acquired in previous work: It was applied on Mozilla [8] and PostgreSQL [6]. The wisdom gained highlighted the importance of interaction, bringing us to the implementation of new features like tracking and code proximity.

6 Related Work

Since Section 2 already introduced related work on logical coupling, this section presents work related to software evolution visualization.

A similar approach to visualize logical coupling has been presented by Pinzger et al. [17] with Kiviat Diagrams. As a difference they do not visualize file-level information but use surfaces to depict complete releases, while in our visualization we depict all evolving files in one diagram. Another difference is that they represent the coupling as edges between the visible modules.

The graph based representation in which entities involved in logical coupling were nodes in a graph and coupling was represented as edges between them was used since the first publications related to logical coupling [9,10]. However, the problem with this representation is that it either represents only modules, and then it is too coarse grained, or it represents modules and files, but then it does not scale to large systems.

A visual data-mining tool to represent both binary association rules and n-ary association rules is EPOsee [3]. The tool adapts standard visualization techniques for association rules to also display hierarchical information.

Chuah and Eick present a way to visualize project information through glyphs called infobugs. Glyphs are graphical objects representing data through visual parameters. Their infobug glyph's parts represent data about software [4]. The difference with respect to our work is that they use glyphs to view project management data, while our work focuses on describing how a module is logically coupled to the others. One common advantage is that both approaches are rotation invariant.

[3] We did not have this problem with ArgoUML.

Lanza's Evolution Matrix [14] visualizes the system's history in a matrix in which each row is the history of a class. A cell in the Evolution Matrix represents a class and the dimensions of the cell are given by evolutionary measurements computed on subsequent versions. The evolution matrix does not represent any relationship between the evolving entities.

Beyer [2] computes a co-change graph and proposes a layout which reveals clusters of frequently co-changed artifacts. Jazayeri *et al.* [13] visualizes software release histories using colors and the third dimension. They do no visualize any coupling relationships between modules.

Girba *et al.* used the notion of history to analyze how changes appear in the software systems [11] and succeeded in visualizing the histories of evolving class hierarchies [12].

Taylor and Munro [22] visualized CVS data with a technique called *revision towers*. Ball and Eick [1] developed visualizations for showing changes that appear in the source code.

Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of systems [23].

Wu *et al.* described an Evolution Spectrograph [25] that visualizes historical sequences of software releases.

7 Conclusion

In this paper we have presented a novel visual approach for reverse engineering based on logical coupling information. The two main benefits of the technique are:

- *Integration*. The Evolution Radar shows logical coupling information at different levels of abstraction, *i.e.,* files and modules, in a single visualization. This makes it possible to understand the dependency between modules and to detect the main responsible for the coupling in terms of files, *i.e.,* the files with the strongest coupling. Using the spawning feature of our tool a dependency between modules can be reduced to a coupling between a small set of files.
- *Customizable time.* Considering the history of logical coupling is helpful to uncover hidden dependencies between software artifacts. However, summarizing the information about the entire history in a single visualization may lead to imprecise results. Two artifacts which were strongly coupled in the past but not recently may appear as coupled. The Evolution Radar solves this problem by dividing the system lifetime in settable time intervals and by rendering one radar per each interval. A slider is used to "move through time". A tracking feature is provided to keep track of the same files in different visualizations.

We have validated our approach on a large open source software system: ArgoUML. We have provided example scenarios of how to use the Evolution Radar to understand module dependencies and to detect candidates for reverse engineering. We have found design issues and dependencies between modules not mentioned in the documentation. We have also reduced these dependencies to coupling between small sets of files. These files should be reengineered in order to decrease the coupling at the module level.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science foundation for the projects "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997), and the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976).

References

[1] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] D. Beyer and A. Noack. Clustering software artifacts based on frequent common changes. In *Proceedings of the 13th IEEE International Workshop on Program Comprehension (IWPC 2005)*. IEEE Computer Society Press, Los Alamitos (CA), 2005.

[3] M. Burch, S. Diehl, and P. Weissgerber. Visual data mining in software archives. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 37–46, New York, NY, USA, 2005. ACM Press.

[4] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[5] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 77–86, New York NY, 2003. ACM Press.

[6] M. D'Ambros and M. Lanza. Applying the evolution radar to postgresql. In *Proceedings of MSR 2006 (3rd International Workshop on Mining Software Repositories)*, pages 177 – 178, 2006.

[7] M. D'Ambros and M. Lanza. Software bugs and evolution: A visual approach to uncover their relationship. In *Proceedings of CSMR 2006 (10th IEEE*

European Conference on Software Maintenance and Reengineering), pages 227 – 236. IEEE Computer Society Press, 2006.

[8] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of MSR 2006 (3rd International Workshop on Mining Software Repositories)*, pages 26 – 32, 2006.

[9] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.

[10] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

[11] T. Gîrba, S. Ducasse, and M. Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, 2004. IEEE Computer Society Press.

[12] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[13] M. Jazayeri, H. Gall, and C. Riva. Visualizing Software Release Histories: The Use of Color and Third Dimension. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 99–108. IEEE Computer Society Press, 1999.

[14] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution)*, pages 37–42, 2001.

[15] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[16] T. Mens and S. Demeyer. Future trends in software evolution metrics. In *Proceedings IWPSE2001 (4th International Workshop on Principles of Software Evolution)*, pages 83–86, 2001.

[17] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[18] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[19] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

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

[21] J. T. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *INFOVIS*, pages 57–, 2000.

[22] C. Taylor and M. Munro. Revision towers. In *Proceedings 1st International Workshop on Visualizing Software for Understanding and Analysis*, pages 43–50, Los Alamitos CA, 2002. IEEE Computer Society.

[23] F. Van Rysselberghe and S. Demeyer. Studying software evolution information by visualizing the change history. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM '04)*, pages 328–337, Los Alamitos CA, Sept. 2004. IEEE Computer Society Press.

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

[25] J. Wu, R. Holt, and A. Hassan. Exploring software evolution using spectrographs. In *Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 80–89, Los Alamitos CA, Nov. 2004. IEEE Computer Society Press.

[26] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In *Proceedings 1st International Workshop on Mining Software Repositories (MSR 2004)*, pages 2–6, Los Alamitos CA, 2004. IEEE Computer Society Press.

[27] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572, Los Alamitos CA, 2004. IEEE Computer Society Press.