Visualizing Co-Change Information with the Evolution Radar

Marco D'Ambros Michele Lanza Mircea Lungu
REVEAL @ Faculty of Informatics - University of Lugano, Switzerland

***Abstract*—Software evolution analysis provides a valuable source of information that can be used both to understand a system's design and to predict its future development. While for many program comprehension purposes it is sufficient to model a single version of a system, there are types of information that can only be recovered when the history of a system is taken into account. Logical coupling, the implicit dependency between software artifacts that have been changed together, is an example of such information. Previous research has dealt with low-level couplings between files, leading to an explosion of the data to be analyzed, or has abstracted the logical couplings to the level of modules, leading to a loss of detailed information.**

In this article we present a visualization-based approach that integrates logical coupling information at different levels of abstraction. This facilitates an in-depth analysis of the logical couplings, and at the same time leads to a characterization of a system's modules in terms of their logical coupling.

The presented approach supports the retrospective analysis of a software system and maintenance activities such as restructuring and re-documentation. We illustrate retrospective analysis on two large open source software systems.

***Index Terms*—Software Evolution, Software Visualization, Change Coupling**

I. Introduction

Versioning systems, also known as software configuration management systems, saw their advent in the 1970s, with tools such as SCCS [26] and RCS [33]. They allow developers to record the history of a project in a software repository. Since the advent of open source software, CVS and recently Subversion emerged as de facto standards. Software repositories contain large amounts of valuable historical data, and their widespread usage has revamped research fields like software evolution [19] and also created new communities like the one dedicated to mining software repositories [12].

Researchers have demonstrated that versioning information can not only help to predict future evolutionary trends [21] [34], but can also provide starting points for reengineering activities [15].

The history of a software system also holds information about the logical coupling of software artifacts. Logical couplings are implicit and evolutionary dependencies between artifacts of a system that evolve together, although they are not necessarily structurally related (for example by means of inheritance, subsystem membership, usage, etc.). They are therefore linked to each other from a development process point of view: logically coupled entities have changed together in the past and are likely to change together in the future. Logical coupling information reveals potentially "misplaced" artifacts in a software system: To prevent a developer modifying a file in a system from forgetting to modify logically related files only because they are placed in other subsystems or packages, software artifacts that evolve together should be placed close (e.g., in the same subsystem) to each other.

In this article we present a technique to inspect logical coupling relationships, which integrates 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 dedicated interactive visualization that we named the *Evolution Radar* [11] [10].

We use visualization [7], [28] because it provides effective ways to break down the complexity of information, and because it has shown to be a successful means to study the evolution of software systems [2], [9], [16], [17], [22], [31], [34].

With our approach we tackle the following problems:

- Presenting very large amounts of evolutionary information in a scalable way.
- Identifying outliers among logical coupling relationships.
- Enabling developers and analysts to study and inspect these relationships and to guide them to the files that are responsible for the logical couplings.

The results and the examples presented in the paper have been obtained by applying our technique on the evolution data of ArgoUML and Azureus, two large and long-lived open source Java systems, and CodeCity, a 3D visualization tool implemented in Smalltalk.

Structure of the paper: In Section II we define logical coupling in detail. In Section III we introduce our approach based on the *Evolution Radar*, a visualization technique that renders logical coupling information, and we discuss its benefits and shortcomings. We illustrate the use of the radar for retrospective analysis in Section IV, and show how it can also be used for maintenance tasks in Section V. In Section VI we discuss related work and conclude by summarizing our contributions in Section VII.

II. 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 [13]. This co-change information can either be present in the versioning system, or must be inferred by analysis. For example Subversion marks co-changing files at commit time as belonging to the same *change set* while in CVS the logically coupled files must be inferred from the modification time of each file.

The concept of co-change in versioning system was first introduced by Ball and Eick [1]. They used this information to visualize a graph of co-changed classes and detect clusters of classes that often changed together during the evolution of the system. The authors discovered that classes belonging to the same cluster were semantically related.

Gall et al. revised the concept of co-change to detect implicit relationships between modules [13], and named it logical coupling. They analyzed the dependencies between modules of a large telecommunications system and showed that the concept helps to derive useful insights on the system architecture. Later the same authors revisited the technique to work at a lower abstraction level. They detected logical couplings at the class level [14] 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.

Other work has been performed at finer granularity levels. Zimmermann et al. [38] used the co-change information to predict entities (classes, methods, fields etc.) that are likely to be modified when one is being modified. Breu and Zimmermann [5] applied data mining techniques on co-changed entities to identify and rank crosscutting concerns in software systems. Bouktif et al. [4] improved precision and recall of co-chancing files detection with respect to previous approaches. They introduced the concept of change-patterns, in particular the "Synchrony" pattern for co-changing files and proposed an approach to detect such change-patterns in CVS repositories using dynamic time warping.

The analysis of logical coupling has two major benefits:

1) It is more lightweight than structural analysis, as only the data provided by the CVS log files is needed, i.e., it is not necessary to parse and model the whole system. Moreover, as it works at the text level, it can analyze systems written in multiple languages.
2) It can reveal hidden dependencies that are not present in the code or in the documentation.

The main problem with existing approaches is that they work either at the architecture level or at the file (or even lower) level. Working at the architecture level provides high-level insights about the system's structure, but low-level information about finer-grained entities is lost, and it is difficult to say which specific artifact is causing the coupling. Working at the file level makes one lose the global view of the system and it becomes difficult to establish which higher-level consequences the coupling of a specific file has.

Our approach makes up for this dichotomy by integrating both levels of information by means of a visualization technique called *Evolution Radar*, presented next.

III. The Evolution Radar

The Evolution Radar is a visualization technique to render file-level and module-level logical coupling information in an integrated and interactive way. It is interactive, and allows the user to navigate and query the visualized information.

The Evolution Radar shows the dependencies between a module in focus and all the other modules of a system. The module in focus is represented as a circle and placed in the center of a circular surface (see Figure 1). All the other modules are visualized as sectors, whose size is proportional to the number of files contained in the corresponding module. The sectors are sorted according to this size metric, and placed in clockwise order. Within each module sector, files belonging to that module are represented as colored circles and positioned using polar coordinates where the angle and the distance to the center are computed according to the following rules:

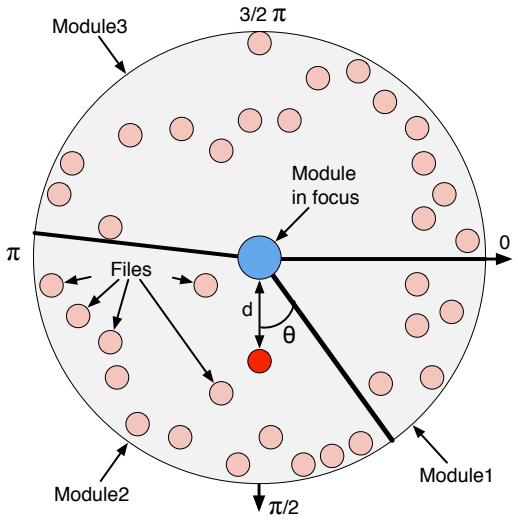


Fig. 1. Principles of the Evolution Radar.

- Distance d to the center is a linear function of 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 placed to the center of the circular surface. The exact definition of the distance is:

$$d = \frac{R}{lc_{max}} \times (lc_{max} - lc) \quad (1)$$

  where R is the radius of the circular surface, lc_{max} the maximum value of the logical coupling and lc the value of the logical coupling.
- 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. Like this, files belonging to the same directory, or classes belonging to the same package in Java, are close to each other. Although this is not the only type of sorting possible, it was particularly useful in our experiments because it maintained the modules decomposition (in directories or packages) in the visualization. Other types of sorting might expose different insights into the system.

Algorithm 1 shows the pseudo code of the layout algorithm. The Evolution Radar can map arbitrary metrics on the color and the size of the circle figures representing files.

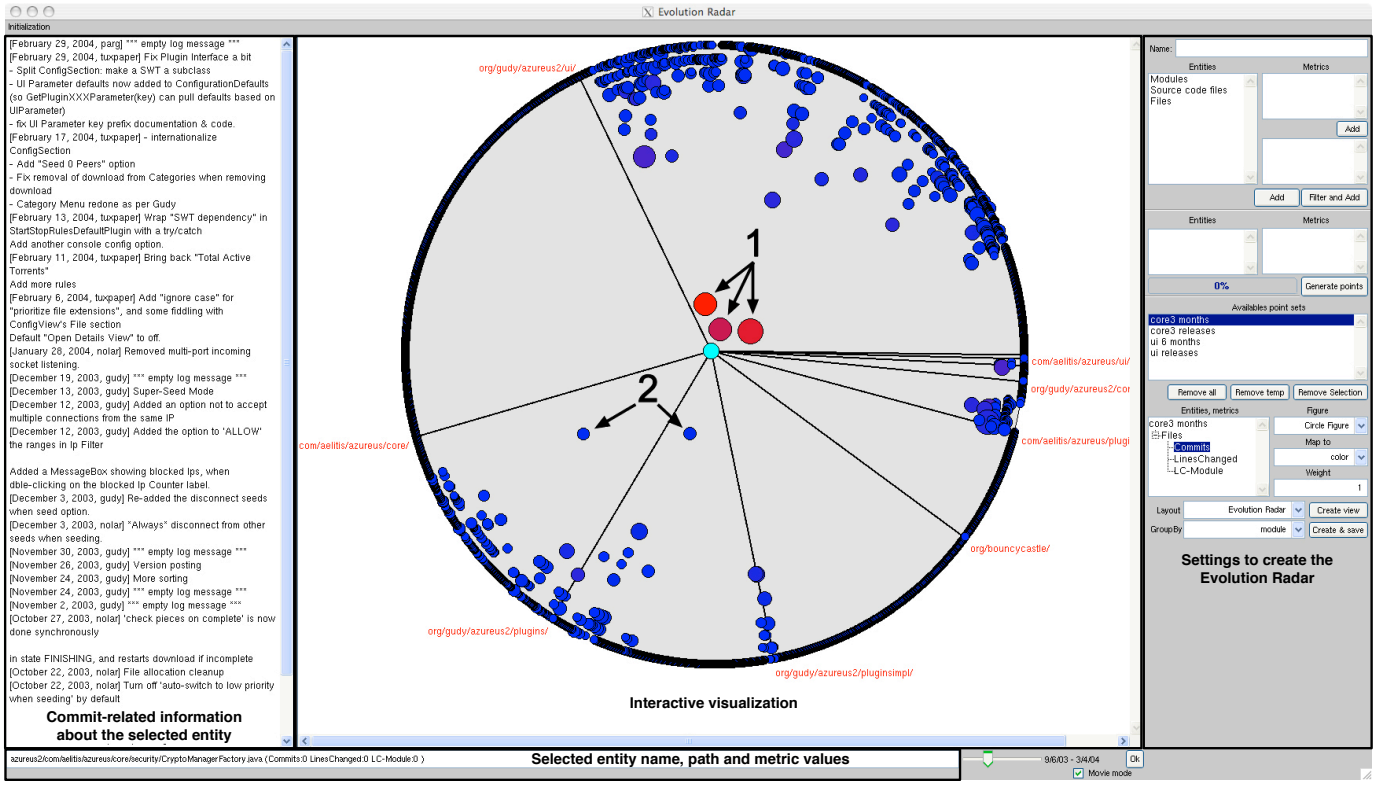


Fig. 2. An example Evolution Radar applied on the *core3* module of Azureus.

Algorithm 1: The Evolution Radar layout algorithm.

```
; // Let M be the module in focus, R the
radar's radius and LC(M,f) the logical
coupling between a module M and a file f
S := {All files f|f ∉ M}
lc-max := max_{f∈S} LC(M, f)
theta := 0
angle-step := 2π / |S|
forall m ∈ {All modules m̄|m̄ ≠ M}, sorted by size do
    draw module sector initial boundary at theta
    forall f ∈ m, sorted by path do
        θ(f) := theta
        r(f) := R / lc-max × (lc-max − LC(M, f))
        theta := theta + angle-step
        draw f in polar coordinates (θ, r)
    end
    draw module sector final boundary at theta
end
```

A. Example

Figure 2 shows an example Evolution Radar visualizing the coupling between Azureus' *core3* module (represented as the cyan circle in the center) and all the other modules (represented as the sectors). The size of the figures is proportional to the number of lines changed while their colors map the number of commits, using a heat-map from blue (lowest value) to red (highest value). We see that the *ui* module (on the top-right part of the radar) 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 identify the most appropriate module to contain them: *core3* or *ui*. Other modules do not have such a strong coupling, but we see the presence of some outliers, i.e., files for which the coupling measure is much higher with respect to the context. The two files marked as 2, belonging to the *plugins* and *pluginsimpl* modules, are such outliers and should also be analyzed and moved in case they belong to the wrong module.

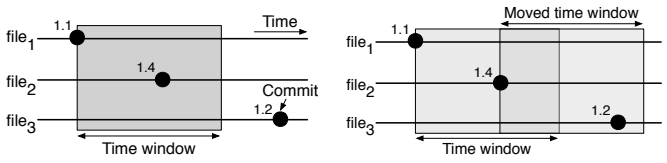
Figure 2 also shows the structure of the Evolution Radar tool. In the center the Evolution Radar has the interactive visualization which is set up using the panel on the right, selecting the entities (e.g., source code files only or all files), the module in the center and the metrics. When an entity is selected in the visualization, it is possible to show the commit-related information about it (author, timestamp, comments etc.) in the left panel. The tool allows the user to either consider all the commits or just the ones involved in the coupling. On the bottom part it displays information about the selected entity, such as its metric values.

B. Logical Coupling Measure

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

$$LC(M, f) = \max_{f_i \in M} LC(f_i, f) \tag{2}$$

where $LC(M, f)$ is the logical coupling between the module in focus M and a given file f and $LC(f_i, f)$ is the coupling



(a) Fixed time window (b) Sliding time window

Fig. 3. Fixed and sliding time window.

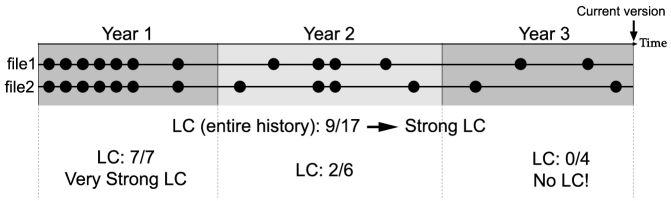


Fig. 4. An example of misleading results when considering the entire history of artifacts to compute the logical coupling value: file1 and file2 are not coupled during the last year.

between the files f_i and f. It is also possible to use other group operators such as the average or the median. We use the maximum because it points us to the files with the strongest coupling, i.e., the main causes for the module dependencies. The value of the coupling between two files is equal to the number of transactions including both files. Since transactions are not recorded by CVS we need to reconstruct them. To do so, the following commit data has to be taken into account: Username, comment and time. Given two or more commits, a necessary (but not sufficient) condition for them to be considered in the same transaction is that the usernames and the comments coincide. We consider them to be in the same transaction if also the time condition holds. For that, two possible techniques are the "fixed time window" and the "sliding time window" approach (depicted in Figure 3):

1) In the fixed time window approach, the beginning of the time window is fixed to the first commit (file_1, version 1.1). All other commits with a timestamp included in the window are considered to be in the same transaction (only file_2 version 1.4).
2) In Zimmermann's sliding window approach [37], 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 this approach include commits taking longer than the size of the time window. We use a time window of 200 seconds as in [37].

C. Interaction

The Evolution Radar is implemented as an interactive visualization. It is possible to inspect all the visualized entities, i.e., files and modules, to see commit-related information such as author, timestamp, comments, lines added and removed, etc. Moreover, it is possible to see the source code of selected files. Three important features to perform analyses with the Evolution Radar are (a) moving through time, (b) tracking and (c) spawning.

a) Moving through time: The logical coupling measure is time-dependent. If we compute it considering the whole history of the system we can obtain misleading results.

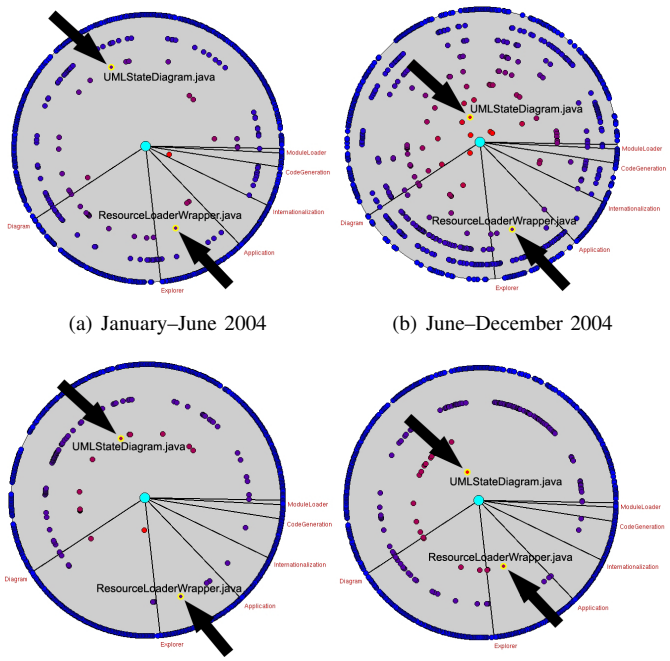
Figure 4 shows an example of such a situation. It depicts the history, in terms of commits, of two files: *file1* and *file2*. The time is on the horizontal axis from left to right and commits are represented as circles. If we compute the coupling measure according to the entire history we obtain 9 shared commits out of a total of 17, a high value because the files changed together more than fifty percent of the time. Although this result is correct, it is misleading since we could conclude that *file1* and *file2* are strongly coupled. Actually *file1* and *file2* were strongly coupled in the past but they are not coupled at all during the last year of the system. Since we analyze logical coupling information to detect architectural decay and design issues in the current version of the system, recent dependencies are more important than old ones. In other words, if two files were strongly coupled at the beginning of a system, but are not anymore in recent times (perhaps due to a reengineering phase), we do not consider them as a potential problem.

For this reason the Evolution Radar is time-dependent, i.e., it can be computed either considering the entire history of files or a given time window. When creating the radar, the user can divide the lifetime of the system into time intervals. For each interval a different radar is created, and the logical coupling is computed with respect to the given time interval.

In each visualization all the files are displayed, even those inserted in the repository after the considered time interval or removed before. Like this, the theta coordinate of a file does not change in different radars, and the position of the figures, with respect to the angle is stable over time. This does not alter the semantic of the visualization, since these files are always at the boundary of the radar, their logical coupling being always 0. 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. However, having several radars raises the issue of tracking the same entity across different visualizations, discussed next.

b) Tracking: 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 5 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 5). It is thus possible to detect files with a strong logical coupling with respect to the last period of time and then analyze the coupling in the past allowing us to distinguish between persistent and recent logical coupling.



(a) January–June 2004 (b) June–December 2004

(c) January–June 2005 (d) June–December 2005

Fig. 5. Evolution of the logical coupling with the *Model* module of ArgoUML. The Evolution Radar keeps track of selected files along time (yellow border).

c) Spawning: The spawning 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.

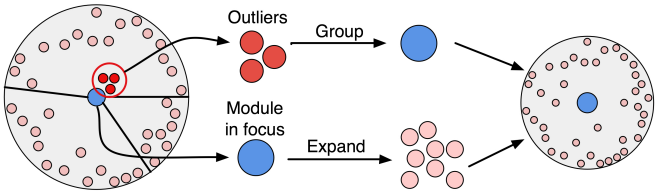


Fig. 6. Spawning an auxiliary Evolution Radar.

To uncover this dependency between files we spawn an auxiliary Evolution Radar as shown in Figure 6: The outliers are grouped to form a temporary module M_t represented by a circle figure. The module in focus (M) is expanded, i.e., a circle figure is created for each file composing it. A new Evolution Radar is then 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 distance from the center is a linear function of 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.

D. Discussion

One of the advantages of the Evolution Radar is that 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 spot the heavily coupled modules (they are displayed as "spikes" pointing to the center). It is also easy to spot single files responsible for the coupling (they are placed close to the center).

The Evolution Radar is a general visualization technique, i.e., it is applicable to any kind of entity. The only requirement is to define a grouping criterion and a distance metric. The radar can also be enriched by adding more structural information. A sector can be divided in sub-sectors, using both radius and angle coordinates, to visualize sub-groups, e.g., sub-modules, as proposed by Stasko and Zhang [29].

The main drawback, common to many visualizations, is that it requires a trained eye to interpret the displayed information.

IV. Using the Evolution Radar for Retrospective Analysis

We implemented two versions of the Evolution Radar: One is a stand alone tool to analyze systems developed using CVS or SubVersion, while the second version is integrated in an IDE environment to develop Smalltalk code. In this section, we apply the stand alone Evolution Radar tool to perform retrospective analysis on two open source software systems: ArgoUML and Azureus. ArgoUML is a UML modeling tool written in Java, consisting of 1,565 classes and more than 200,000 lines of code. Azureus is a BitTorrent client written in Java, with 4,222 classes and more than 300,000 lines of code. In the following we present a summary of the analyses of the systems, giving several examples of our approach.

During the analyses, we use the term *strong coupling* (or strong dependency) between a file f and a group of files M when the following two conditions[1] hold:

1) $LC(M, f) \geq 10$ i.e., there is at least one file f_M of M having a number of shared commits with f greater or equal than 10.
2) The number of commits shared by f and f_M divided by the total number of commits of f is greater than 0.35, i.e., f is modified more than 35% of the times together with f_M.

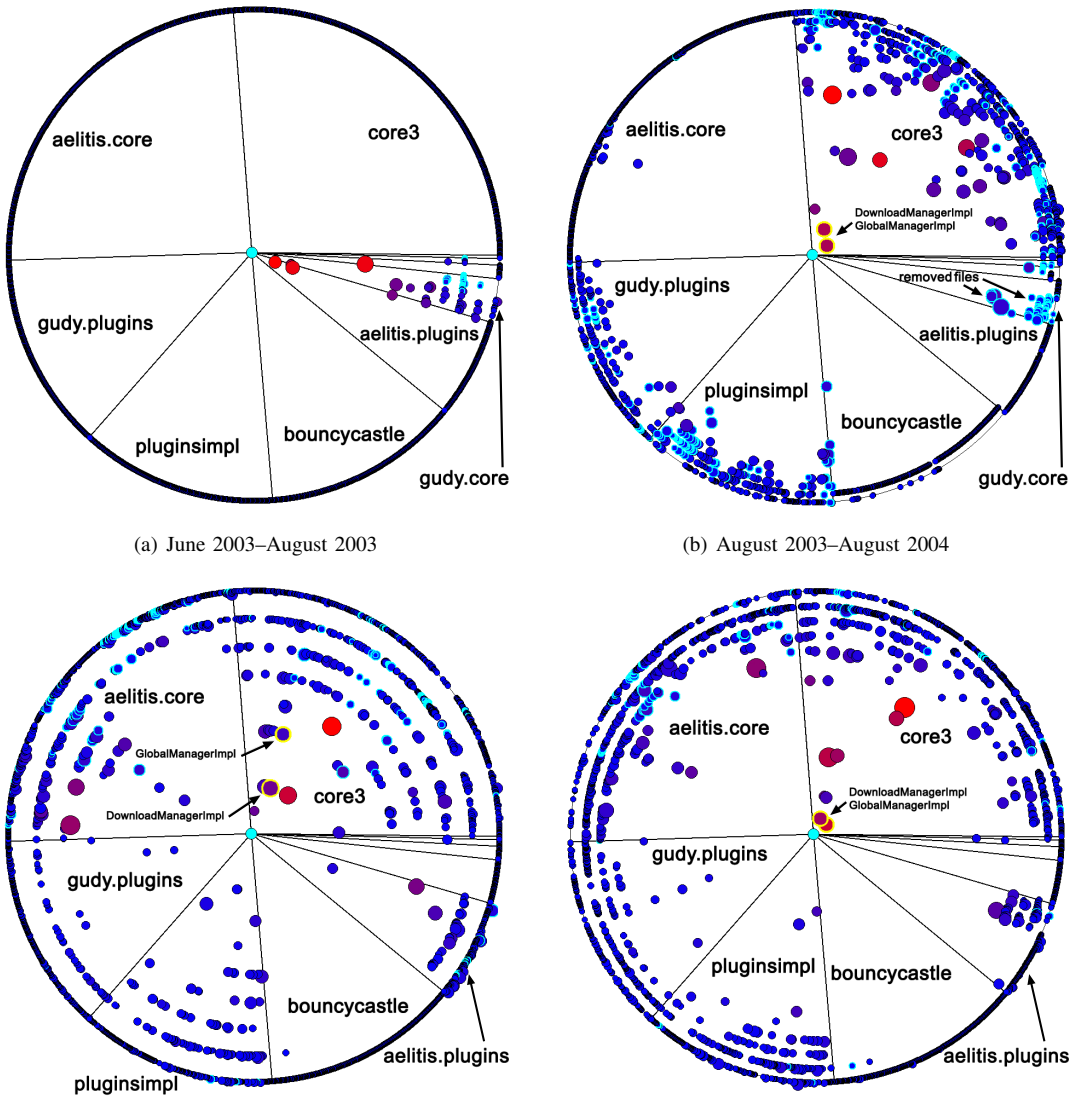
A. Azureus

Since Azureus does not have any documentation about its architecture, i.e., an explicit decomposition of the system in modules, we used the Java package structure to decompose the system. Throughout the analysis, we use the following metric mappings: the size (area) of the figures representing files maps the number of lines changed and the color heat-map represents the number of commits. Both metrics are computed relative to the considered time interval.

Getting an overview: The first goal of our analysis is to get a first understanding of the history of the system packages, i.e., when they were introduced or removed from the system.

We apply the Evolution Radar with the package *org.gudy.azureus2.ui* in the center, since it is one of the packages existing from the first to the last version of the

[1]This definition is valid in the context of the analyzed software systems and its goal is to avoid mentioning the thresholds all the times.



(a) June 2003–August 2003

(b) August 2003–August 2004

(c) August 2004–August 2005

(d) August 2005–August 2006

Fig. 7. Evolution Radars of the *org.gudy.azureus2.ui* package of Azureus.

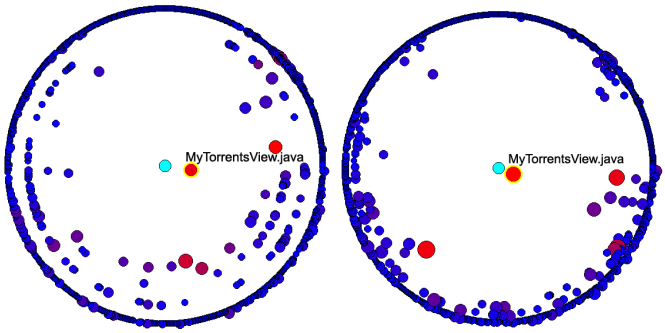
system. Figure 7 shows the result with a time interval of one year[2]. Figures with cyan borders represent files that were removed during the considered time interval. During the first three months (Figure 7(a)) *org.gudy.azureus2.ui* was coupled with *org.gudy.azureus2.core* (*gudy.core* from now on), while the other packages did not exist. In the following year (Figure 7(b)) *gudy.core* was removed, since all the figures have a cyan border and in the following radars there is no activity in this package. The core of the system became *org.gudy.azureus2.core3* coupled to the *ui* (because of the figures close to the center) and *com.aelitis.azureus.core* (*aelitis.core* from now on) with very low activities (few figures). The plugins were also introduced with a clear separation from the *ui*, since the coupling was weak (no figures close to the center). From August 2004 to August 2005 (Figure 7(c)) the architecture decayed, since most of the packages were strongly coupled with the *ui*. Finally, during the last year (Figure 7(d)) the couplings decreased with all the packages, but in *core3* there were still files with a strong dependency with the *ui* (figures close to the center).

Detailing the dependency between core3 *and* ui*:* The two files indicated with the arrows and highlighted with the tracking feature (yellow border) are *GlobalManagerImpl.java* and *DownloadManagerImpl.java*. GlobalManagerImpl is a singleton responsible of keeping track of the DownloadManager objects that represent downloads in progress. They are the most coupled from 2005 to 2006 and from 2003 to 2004, and they have a strong dependency from 2004 to 2005. This strong and long-lived coupling indicates a design problem (a code smell), conforming to Martin's Common Closure Principle [20]): "classes that change together belong together". The coupling indicates that the classes are misplaced or there are other design issues. We spawn two other radars having these files as the center to see which parts of the *ui* package they have dependencies with.

[2]The first radar refers to a time interval of only 3 months because the time intervals are computed backward starting from the latest version of the system.

Figure 8 shows the radars corresponding to August 2004–2005 and August 2005–2006. The dependency between *GlobalManagerImpl* and *DownloadManagerImpl* and the *ui* package is mainly due to the class *MyTorrentsView*, a God class (defined by Riel as a class that tends to centralize the intelligence of the system [24]).

Such information is useful for (i) an analyst, because it points to design shortcomings and for (ii) a developer, because when modifying *GlobalManagerImpl* or *DownloadManagerImpl* in *core3* she knows that *MyTorrentsView* is likely to be modified as well.



(a) August 2004–August 2005 (b) August 2005–August 2006

Fig. 8. Evolution Radars for *GlobalManagerImpl* and *DownloadManagerImpl*.

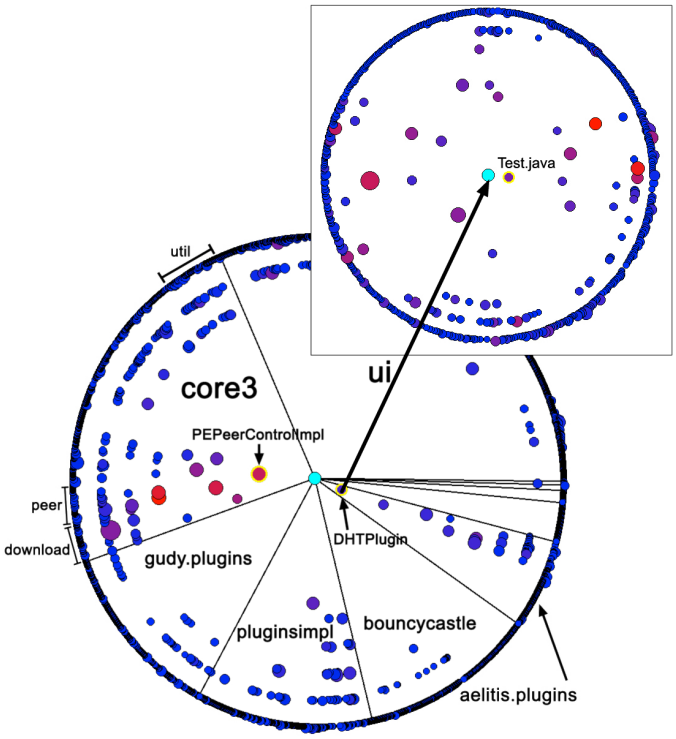


Fig. 9. Evolution Radar for Azureus's *aelitis.core* package and details of the coupling between this package and the class *DHTPlugin*, Feb–Aug 2006.

Understanding the roles of the modules: With this first series of radars we obtained an overall idea of the dependencies and evolution of the packages. We still do not have any idea of why there are two core packages and which roles they have. To obtain this information we select the files with the most intense activities (big and red) and we display their commit comments, as shown on the left of Figure 2. We find out that *core3* has more to do with managing files, download, upload and priorities while *aelitis.core* is more related to the network layer (distributed hash tables, TCP and UDP protocols). In all the radars from Figure 7 the *org.bouncycastle* module has very low activity and coupling. A closer inspection reveals that the mentioned package was imported in the system at a given moment and never modified later. It implements an external cryptographic library (available at http://www.bouncycastle.org) used by the Azureus developers.

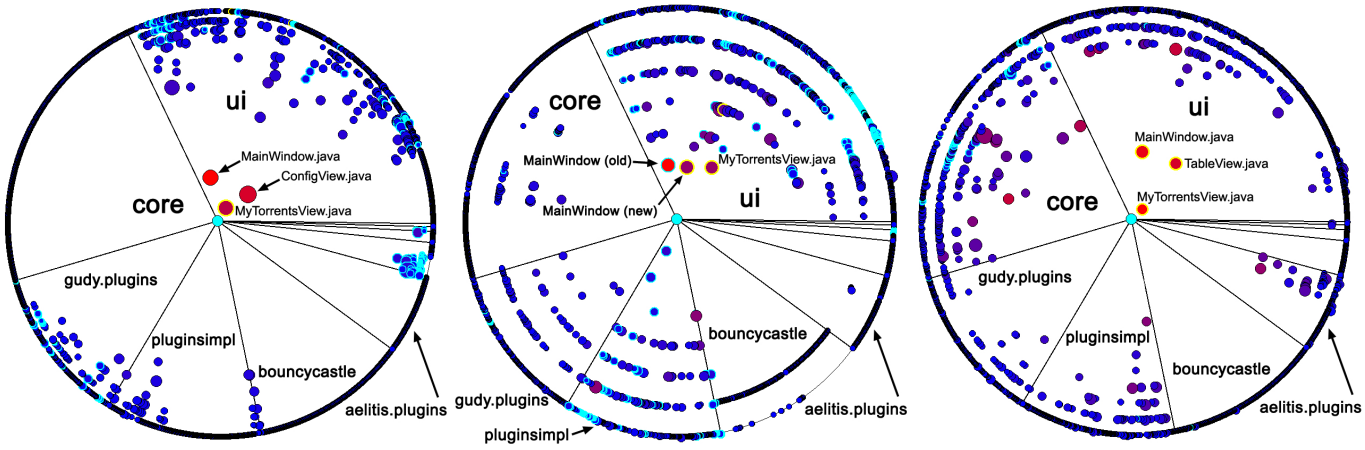
Analyzing the core *package:* As a second step, we want to understand the dependencies of *aelitis.core* with the rest of the system and we want to detect which are the files responsible for these dependencies. We create a radar for every six months of the system's history. We start the study from the 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 biased by commits far in the past.

Figure 9 shows the radar of *aelitis.core* from February to August 2006. Two packages are strongly coulped with *aelitis.core*: *core3* and *com.aelitis.azureus.plugins*. Comparing the radar with the static dependencies of *aelitis.core* extracted from the source code, we see that:

- Looking at the static dependency data (e.g., invocation and inheritance) we observe that the module with the strongest static dependency is *core3*. In fact, the strength of the dependency between packages, measured by the total number of dependencies between the contained classes, is one order of magnitude stronger for *core3* than for any other module. However, when we look at the radar we are surprised to see that the module with the strongest coupling is a different one: *com.aelitis.azureus.plugins*.
- Most of the static dependency with *core3* comes from the sub-package *util*, responsible for 45 % of the dependencies between the two packages. However, the radar shows that *util* is less coupled than its sibling sub-packages *peer* and *download*. This holds also in the past: in the radars for previous time intervals *util* is always less coupled than *download* and *peer*.

These observations demonstrate that logical coupling is an implicit dependency. It complements static analysis, but it can only be inferred from the evolution of the system.

The two classes with the strongest dependencies are *DHTPlugin* in the *plugin* package and *PEPeerControlImpl* in the *core3* package. Using the tracking feature we found out that, in the previous year, these two classes were outliers, i.e., had a coupling much stronger than the other classes in the package. To see the details of the dependency for *DHTPlugin* we spawn a new radar having the class as the center. The situation is different from the one shown in Figure 8 for *DownloadManagerImpl* and *GlobalManagerImpl*. For these two classes the coupling was mainly due to one single class (*MyTorrentsView*), while here it is scattered among many files. This means that the file *DHTPlugin.java* was often changed together with many files in the *aelitis.core* package. This is a



(a) August 2003–February 2004 (b) February–August 2004 (c) February–August 2006

Fig. 10. Evolution Radars for the *core3* package of Azureus. They help to detect the transition from the old *MainWindow.java* to the new one.

symptom of a misplaced file. Looking at the source code of the ten most coupled files, we discovered that all of them use *DHTPlugin* or its subclasses, meaning that core classes use plugin classes. Moreover, the class with the strongest coupling is a test case using *DHTPlugin*. Therefore, a modification in the plugin package can break a test in the *core* package. This scenario repeats itself for the most coupled file in *core3*: the file *PEPeerControlImpl.java* is coupled with several files in *aelitis.core*. For these reasons, *DHTPlugin*, *PEPeerControlImpl*, and their subclasses should be moved to *aelitis.core*.

Detecting renaming: The Evolution Radar can keep track of files, even if they are renamed or their code is moved to another place. This is possible because files are positioned according not only to their names, but also to their relationship with the center package. When a file f_{old} is renamed to f_{new}, the relationship f_{old} used to have with the package in the center, will hold for f_{new} (note that CVS records it as a removal of f_{old} and an addition of f_{new}). Looking at similar couplings and removed files, we can detect such situations.

Figure 10 shows Evolution Radars having the *core3* package in the center. From February to August 2006 *core3* had dependencies with the plugins packages and *aelitis.core*, and a strong coupling with the *ui*. In the *ui* there are three outliers: *MyTorrentsView.java* (already detected and discussed), *TableView.java* (the superclass of *MyTorrentView*, a God class as well) and *MainWindow.java*. Figure 10(a) shows the radar corresponding to August 2003–February 2004. In the *ui* there are again three outliers: *MyTorrentsView.java*, *ConfigView.java* and *MainWindow.java*. This *MainWindow.java* is a different file from the one in Figure 10(c), and in fact it is not highlighted by the tracking feature: the two *MainWindow* classes belong to different sub-packages. However, since they have the same type of coupling with *core3*, they can represent the same logical entity. Figure 10(b) shows the transition between the old and the new *MainWindow.java*. They both have a strong coupling with *core3* and the old *MainWindow.java* is removed, since it has a cyan border.

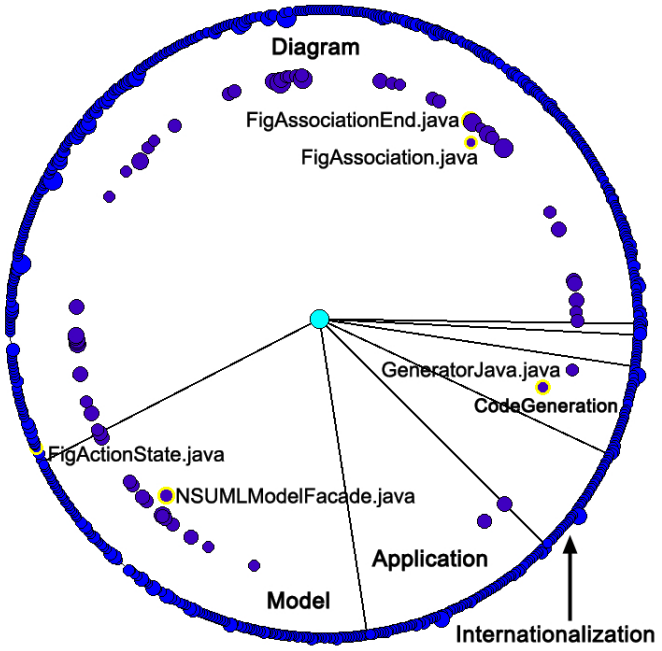
B. ArgoUML

We inferred ArgoUML's system decomposition into modules from its web site. We omitted the modules for which the documentation says "They are all insignificant enough not to be mentioned when listing dependencies" and focus 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.

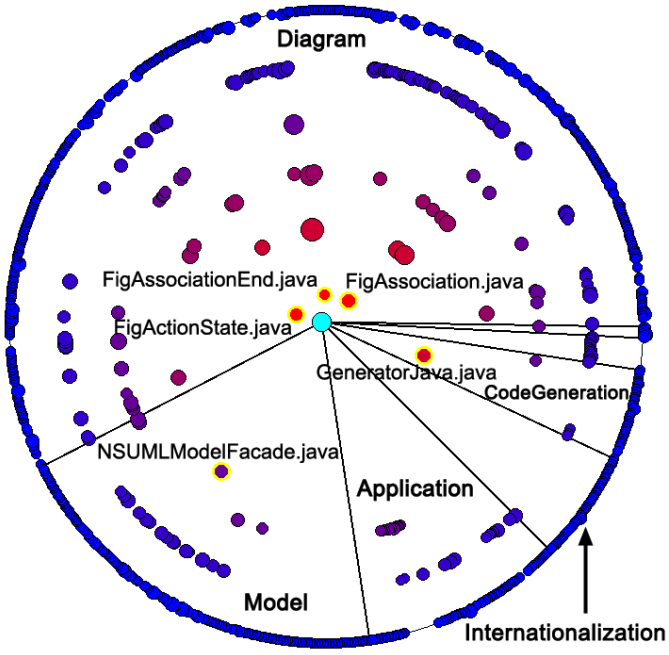
We use the same analysis approach as for Azureus: we create a radar for every six months of the system's history. As metric we use the logical coupling for both the position and the color of the figures. The size is proportional to the total number of lines modified in the considered time interval.

The Explorer *module:* Figure 11(b) shows the Evolution Radar for the last six months of history of the *Explorer* module. This module is much more coupled with *Diagram* than 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 discover that the coupling with these files is recent: in the radar for the previous six months (Figure 11(a)) 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 spawn an auxiliary radar: we group the three files and generate another radar centered on them, shown in Figure 12. 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, and since they belong to different modules, it is easy to forget this hidden dependency.

The visualization in Figure 11(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 a group composed of *GeneratorJava.java* we obtain a visualization very similar to Figure 12, in which the main responsible for the dependency



(a) January–June 2005



(b) July–December 2005

Fig. 11. Evolution Radars applied to the *Explorer* module of ArgoUML for the year 2005.

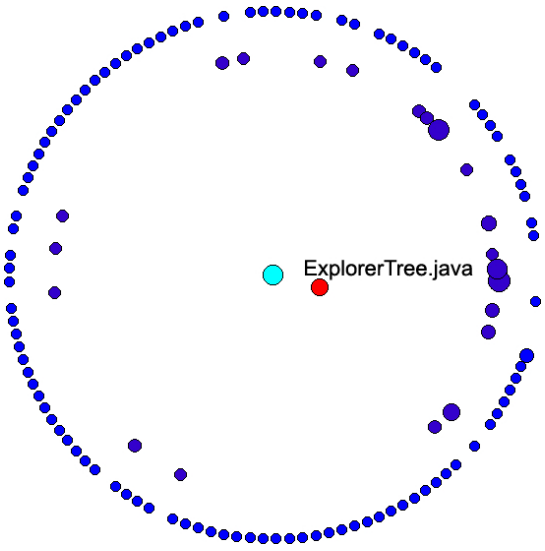


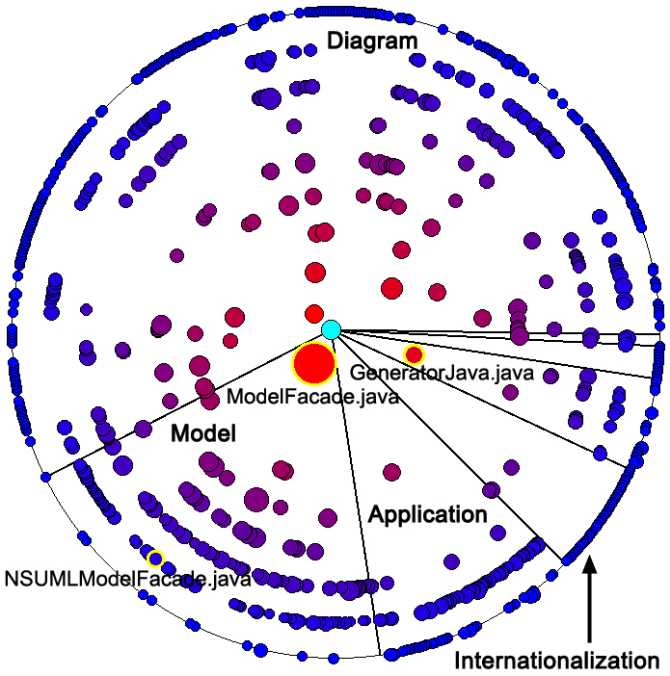
Fig. 12. Details of the logical coupling between ArgoUML's *Explorer* module and the classes *FigActionState*, *FigAssociationEnd* and *FigAssociation*.

is again *ExplorerTree.java*. Reading the code reveals 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 11(a) and Figure 13(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*.

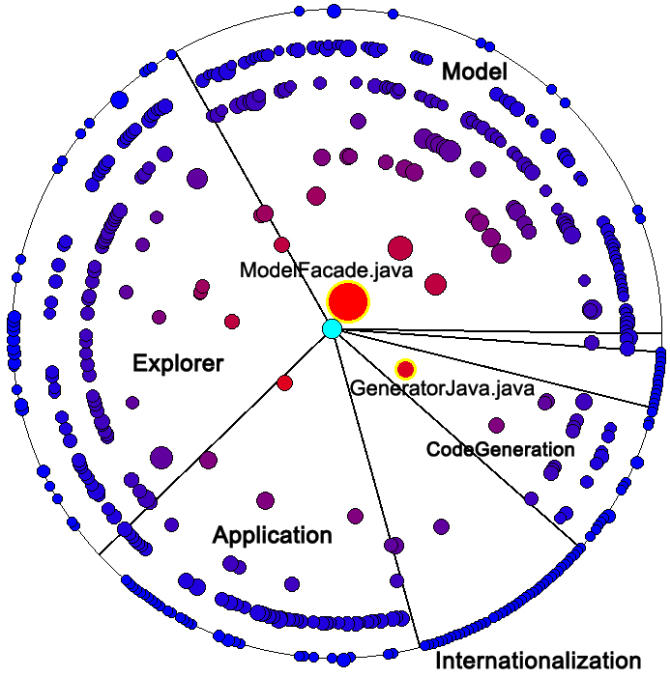
Detecting a move operation: The radars in Figure 11(b) and Figure 11(a) show that during 2005 the file *NSUMLModelFacade.java* had the strongest coupling, in the *Model* module, with *Explorer* (module in the center). Going six months back in time, from June to December 2004 (see Figure 13(a)), we see that the coupling with *NSUMLModelFacade.java* was weak, while there was a 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 13(b)). By looking at its source code we find that it was a "God class" [24] 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 11(b) and Figure 11(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 discover that it is also a very large class with 317 public methods. Moreover, we find out that 292 of these methods have the same signature of methods in the *ModelFacade* class. The only difference is that *NSUMLModelFacade*'s methods are not static. Also, it contains only two attributes, while *ModelFacade* has 114 attributes. 75% of the code is 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 reveals problems, that are difficult to detect looking at one version of the system only. Knowing the evolution of *ModelFacade* helped us to understand the role of *NSUMLModelFacade* in the current version of the system.

Identifying system phases: As a final scenario, we analyze the evolution of the logical couplings of the *Explorer* module with all the others.



(a) Explorer module



(b) Diagram module

Fig. 13. Evolution Radars of the *Explorer* and *Diagram* modules of ArgoUML from June to December 2004.

From Figure 13(a), we see that from June to December 2004 the couplings were very strong. Then, from January 2005 to June 2005 (Figure 11(a)), they heavily decreased. This suggests that in the previous period the module was restructured and its quality was improved, since in the next time interval the coupling with the other modules was 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 13(a)) the figures are bigger than in the radar relative to January–June 2005 (Figure 11(a)). At the end of the restructuring phase, the class *ModelFacade* was removed. From June to December 2005 (see Figure 11(b)) the coupling increased again. This can be related to a new restructuring phase.

C. Discussion

We applied the Evolution Radar on two open source software systems, showing that it helps in answering questions about the evolution of a system that are useful to developers, analysts, and project managers. The Evolution Radar offers a visual way to assess the files that might change in the future based on the prediction offered by logical coupling. Due to the fine-grained level of the visualization, files can be inspected individually. For example in Azureus we discovered that a change in the class *GlobalManagerImpl* or *DownloadManagerImpl* in the *core3* package, is likely to require a change in the class *MyTorrentsView* in the *ui* package.

The Evolution Radar can be used to (i) understand the overall structure of the system in terms of module dependencies, (ii) examine the structure of these dependencies at the file granularity level, and (iii) get an insight of the impact of changes on a module over other modules. This knowledge will help them in the following activities:

- Locating design issues such as God classes or files with a strong and long-lived dependency with a module. Examples of design issues detected in ArgoUML include the classes *GeneratorJava* in *CodeGeneration* and *ExplorerTree* in *Explorer*. *GeneratorJava* has a persistent coupling with the *Explorer* module, while *ExplorerTree* is coupled with both the *CodeGeneration* and *Diagram* modules. In Azureus we detected God classes (*MyTorrentsView* and its superclass *TableView*) having strong dependencies with files belonging to different packages.
- Deciding whether certain files should be moved to other modules. In Azureus's case we discussed why the class *DHTPlugin* should be moved to the *aelitis.core* package.
- Understanding the evolution of the logical coupling among modules. This activity can reveal phases in the history of the system, such as restructuring phases. In ArgoUML different phases were identified, with respect to the module *Explorer*, where two of them are likely to be restructuring phases. The evolution of the dependencies, together with the information about the removed files, is also helpful to see when modules were introduced or removed from the system. We used this information to get an overall idea of Azureus' structure in terms of packages.
- Detecting when artifacts have been renamed or moved. For example, discovered that in Azureus the class *MainWindow* was moved between packages. In ArgoUML we found out that most of the code of the *ModelFacade* class was moved to *NSUMLModelFacade*.

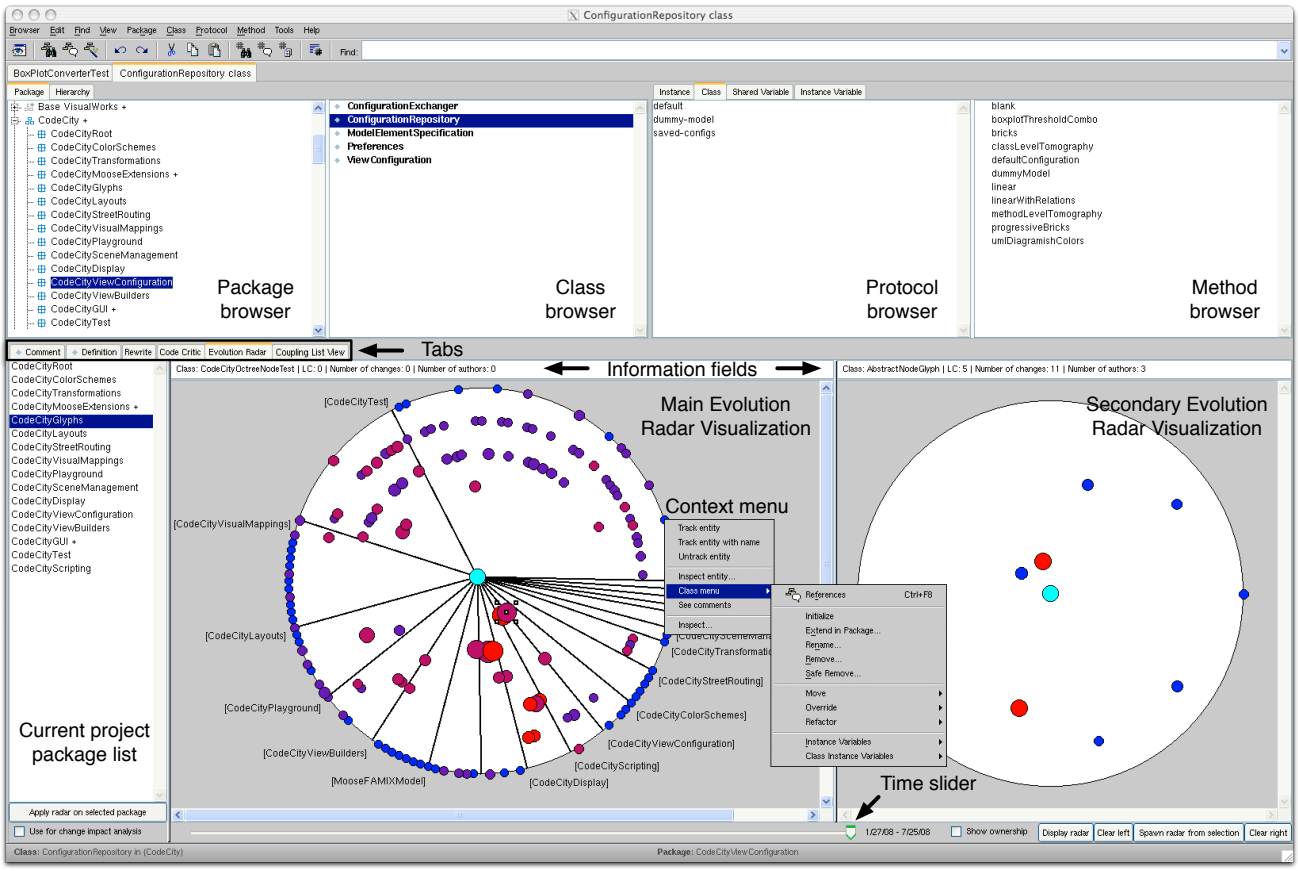


Fig. 14. The Evolution Radar integrated in the System Browser IDE. It shows the logical coupling between the CodeCityGlyphs module and the rest of CodeCity's modules.

V. Supporting Maintenance with the Radar

As a second application of the Evolution Radar, we integrated it in an IDE, to support maintenance activities. In this case, the radar visualization is part of the code browser, to allow a developer to directly go from the visualization to the code and back. The IDE we enriched with the Evolution Radar is the System Browser [25] of the Cincom Smalltalk Visualworks distribution (http://www.cincomsmalltalk.com/).

The first thing we needed to do in order to integrate the radar with the Smalltalk IDE was to adapt our model and tool, since Cincom Smalltalk uses its own versioning system called Store [30]. In Cincom Smalltalk the code is organized in bundles: a bundle can contain packages and bundles, and packages contain classes. As opposed to CVS and SubVersion, Store is not file-based but "entity based", i.e., every entity (from bundle to method) is versioned. To render the Evolution Radar, we consider the system decomposition in packages, "flattening" the bundles hierarchy. We consider packages instead of bundles because the latter cannot contain classes, but only packages. To compute the logical coupling between two classes versioned with Store, we consider all the versions of the classes corresponding to the considered time interval, and we count how many times they changed together, i.e., how often they were committed in the same transaction. This number of co-changes is the logical coupling between the two classes in the considered time interval.

A. Integration in the IDE

Figure 14 depicts the enriched System Browser IDE. In the top part there are four panels used to browse the code. They present respectively packages, classes, protocols [3] and methods. In the bottom part there are multiple tabbed views that present details or allow the editing of the element that is selected on top. They include a code editor, a code analysis tool, a comment editor, etc. We extended the IDE with a new tabbed view for the Evolution Radar.

The Evolution Radar view is composed of three panels (see Figure 14): The list of available packages in the current project, the main radar visualization and the second visualization in which a second radar can be spawned from a selection in the main one. The Evolution Radar panels include the information fields that show information about the entity under the mouse pointer and the time slider which supports the visual navigation through time.

The integrated version of the Evolution Radar adds the following interaction modes:

- When clicking on a figure, the browser in the top part displays the corresponding class or package, allowing the developer to immediately see and, in case, modify the source code. Moreover, through the context menu (see

[3] Protocols are a Smalltalk-specific way of grouping methods in a class.

Figure 14), it is possible to directly move, modify and apply refactorings on the corresponding class.
- When selecting a figure or a group of figures, it is possible to: (1) spawn a radar in the secondary view, (2) spawn a second window with a code browser, (3) track the entity over time, and (4) read the commit comments corresponding to all the commits or just to the ones involved in the coupling.

To assess the usefulness of the visualization against the same coupling information presented in a list, we added another tabbed view to the IDE, called "Coupling List View". In this view instead of the main and secondary evolution radar visualizations, there is a list of the classes coupled with the package selected in the package list (bottom left part of Figure 14), sorted according to the logical coupling value.

This implementation of the Evolution Radar is designed to support Smalltalk developers in three types of activity: System restructuring, system re-documentation and change impact estimation.

Restructuring: Having the Evolution Radar integrated into the IDE makes it easy to inspect the classes that are coupled and if they are at the wrong place to restructure the system, i.e., move the classes to the appropriate package. This operation can be performed directly from the context menu in the Evolution Radar visualization (see Figure 14).

Re-documentation: The developer uses the radar to analyze the coupling of a package with the rest of the system, and by looking at the commits comment and the source code he can discover the reasons for the coupling. Then, the developer can annotate this information directly on the involved classes and/or packages by writing a comment into the "Comments" tab (see in the list of tabs in Figure 14). These comments are part of the system code and get versioned as every other entity.

Change impact estimation: When two classes are logically coupled, they are likely to change together in the future [27]. This information can be useful to a developer who is about to make a change to a class in the system because it can support him in estimating the impact of the change.

The developer can select the class (or classes) he needs to modify and see which are the classes that are coupled with it. If there is no coupling (no figures close to the center), the developer can go on with the modification. If the class is coupled with few other classes, he can get more insight by looking at their source code and reading the comments written in the re-documentation phase. If the class is coupled with many other classes in the system, the developer has to find out whether these couplings are due to large commits or whether the class is affected by design issues. To do so, he can exploit the information gained in the re-documentation phase or can look at the commit comments accessible directly in the radar.

The developer can access this functionality using the radio button in the bottom left corner of the tool (see Figure 14). Whenever he selects a package, a class or multiple classes in the code browser (in the upper part), an Evolution Radar having the selection in the center is generated and rendered on the fly.

B. Experimental Evaluation

To experiment the IDE version of the Evolution Radar, we asked a developer to use it, and report on his experience. The developer is Richard Wettel, a PhD student at our University. The system to which he applied the Evolution Radar is called CodeCity[4], a software analysis tool that visualizes software systems as interactive, navigable 3D cities. At the time of the experiment, CodeCity was composed of 478 classes and had about 15,000 lines of code. It has been developed since April 2006 mostly by a single developer, and occasionally by two other developers, in the context of pair-programming sessions. The following, in italic, is a slightly adapted extract of his experience report, organized according to the performed maintenance activities.

Re-documentation phase: *We started by analyzing logically coupled modules (i.e., packages in Smalltalk), looking for coupling to the core packages, in our case the ones dealing with the glyphs, layouts, and mappings. Figure 14 shows the coupling to the glyphs module, represented by the circle in the middle of the left visualization panel, which seems to be distributed in five levels (i.e., five concentrical layers besides the external one which shows the uncoupled classes). Whenever we needed to see which are the entities inside the module in the middle logically coupled with a particular class, we spawned a radar from the selection (the right visualization panel in Figure 14) and observed these in isolation. Selecting a class circle triggers its selection also in the code editor. Integrating the information about logical coupling with the code and with versioning logs allowed us to reason about the system's evolution.*

Overall, we were able to determine the cause of the unexpected logical couplings with the help of the context menu option which allows looking at the log entries of the system versions which produced the coupling. The causes were either larger commits incorporating several unrelated changes to the system or in one case the system was massively restructured. While we did not find any coupling that needed refactoring, all of them were accurately detected. We finally added all information we found to the comments of each analyzed class.

Change impact estimation phase: *The second task we performed was assessing the impact of change of the* AbstractLayout *class (the root of the layout hierarchy), since we needed to reengineer the way the layouts communicate with the glyphs and mappings. To do so, we enabled the "Change Impact" functionality through the radio button, and then selected the* AbstractLayout *class in the code browser, which automatically generated the corresponding radars. In Figure 15, we see the evolution of the logical coupling to the* AbstractLayout *class, which provides an accurate representation of the way the entire system has evolved. In a first time period (top left) classes are being changed (i.e., large circles) in many packages being developed in parallel and logically coupled among each other. The following period is very unfocused, but with smaller changes (i.e., small fixes). The third period is one of coupling with classes from almost all the packages, while the last period*

[4]See http://www.inf.unisi.ch/phd/wettel/codecity.html

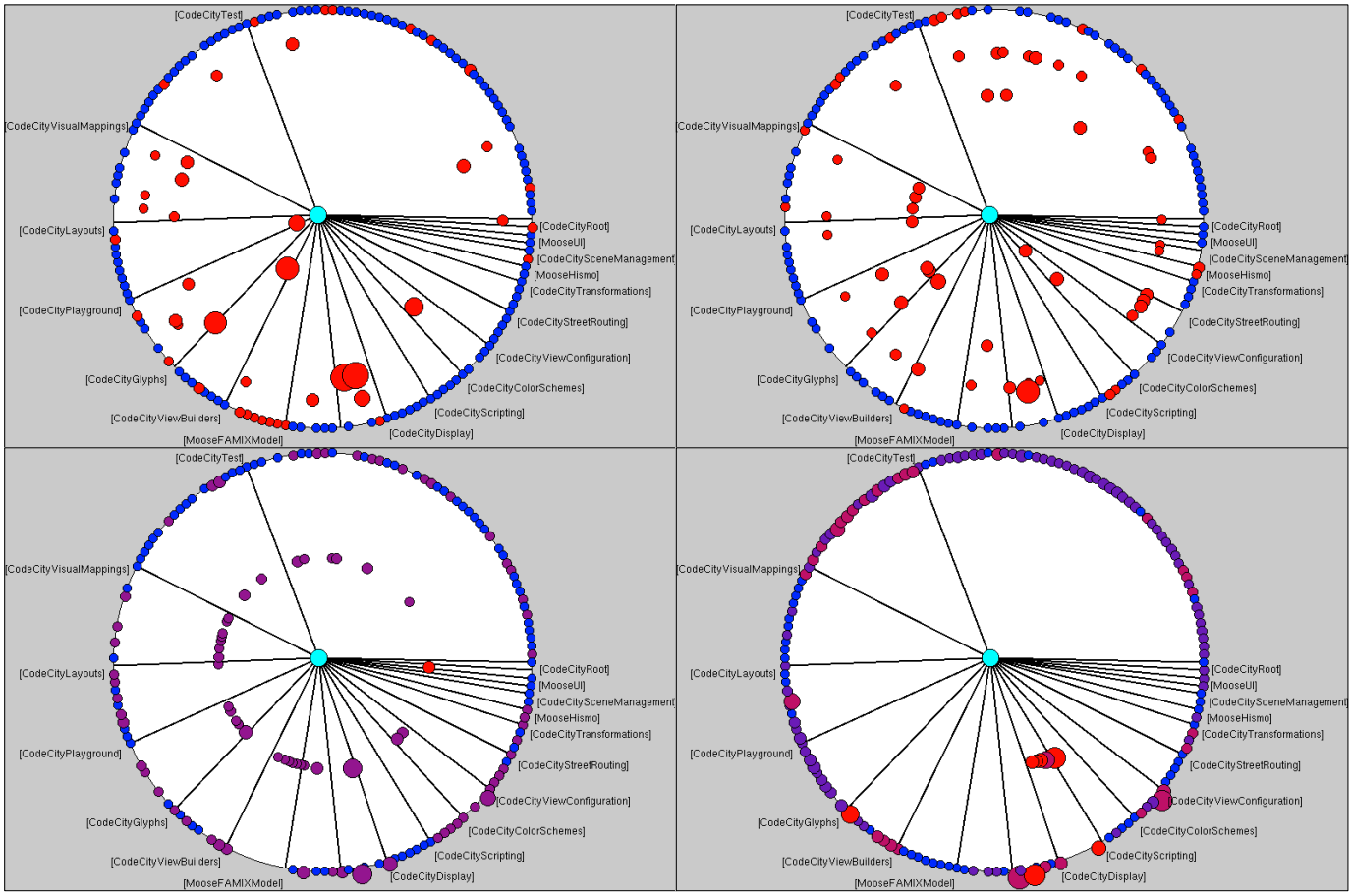


Fig. 15. The logical coupling evolution of the class AbstractGlyph.

shows coupling only to few new classes all defined in the CodeCityScripting *package. Reading the comments written in the re-documentation phase, we knew that these classes of* CodeCityScripting *(which implement a basic scripting language for 3D visualizations) are coupled with many core classes (including* AbstractLayout*) and the coupling is justified: The core classes changed together with the scripting classes in order to make the system compatible with the new scripting language. Since the coupling was justified, we knew that we could proceed with reengineering the layouts, but being careful to comply with the new scripting language, not to break the dependency with the* CodeCityScripting *package.*

Comments: *We were asked to try the list view and compare it with the radar view. In our experience, the list view was useful to easily spot the most coupled class with a given package. However, with the list we had difficulties in understanding the coupling at the system level and to see how it was distributed among packages and classes. We found it useful to be able to navigate in time using the radar, while keeping track of certain classes. With the list view, we did not succeed in navigating in time, because it was difficult to keep track and compare the values of the logical coupling for a class over different time intervals.*

C. Lessons Learned

Although we do not consider this experiment a full-fledged user study, it points to the fact that the Evolution Radar can be successfully used to support maintenance activities. The visualization itself, without the interactive features (moving through time, spawning, tracking, inspecting), is not sufficient to perform the tasks, as the developer continuously used them during the maintenance activities. In particular, the possibility to read the commit comments seems crucial.

Overall, while the first implementation of the radar as a stand alone tool is helpful to perform retrospective analysis, only through the integration in a development environment the radar exploits its full potential, as an IDE enhancement.

VI. Related Work

A. Logical Coupling Visualization

Ball et al. were the first to visualize information about logical coupling. In [1] they proposed a graph visualization technique where nodes represent classes and two classes are connected with an edge if there is a modification report (a commit) in which both the classes changed. The nodes are positioned according to the number of times the corresponding classes changed together, in such a way that coupled classes are close to each other. The color and shape of a node represents the

module the class belongs to. The visualization is static and the approach does not support moving through time and spawning.

The most similar approach to the Evolution Radar is the Evolution Storyboards [3], by Beyer and Hassan. A storyboard is a sequence of animated panels that shows the files composing a CVS repository, with an energy based layout, where the distance of two files is computed according to their logical coupling, similarly to the approach of Ball et al. [1]. The visualization allows the user to easily spot clusters of related files and to compare this cluster with the system decomposition in module, by rendering the module information on the color of the files (files belonging to the same module are rendered with the same color). Each panel is computed according to a particular time period and the animation in the panel shows how the files move according to how their logical coupling changed over the considered time. The visualization is scalable and the authors were able to apply it on large software systems. A benefit of the Storyboards, with respect to the Evolution Radar, is that the Storyboards shows the coupling of all the files at the same time, while in the Radar the files belonging to the module in focus (in the center) are not rendered. Therefore, to get the "big picture" of the evolution of the coupling in the system, the Evolution Storyboard is better then the Radar. On the other hand, having all the logical coupling information in the same view, makes it difficult to see the details about individual files, because they are surrounded by hundreds of other files and sometimes hidden behind, when they overlap. Analyzing the details of certain files or modules is also difficult because spawning is not provided. With the Storyboards it is possible to move through time, but it is not possible to track files over different time periods, since there is no tracking facility. Thus, the Evolution Radar is better for analyzing the details about the logical coupling and for analyzing them over time. Another difference is that the Storyboard does not show the dependencies between modules, but only among files, while the radar provides both at the same time.

Ratzinger et al. in [23] proposed a visualization technique to render the logical coupling between java classes, visualizing also module (package) information. Classes are rendered as small ellipse and grouped in larger ellipses representing the packages they belong to. The visualization shows the logical coupling among classes through edges connecting the ellipses whereas the thickness of the edges describes the "strength" of the visualized couplings. Similarly to the Evolution Radar, Ratzinger et al. visualization shows file level and module level coupling information at the same time, since the coupling among modules can be inferred from the ones among the included classes. However, the technique does not scale on large system, since visualizing coupling as edges suffers from overplotting. Moreover, the visualization is static, i.e., it does not allow the user to navigate through time.

Another approach similar to ours has been presented by Pinzger et al. [22] with Kiviat Diagrams. 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. Their work is not the first to represent the coupling as graphs. In fact, the nodes and edges representation was used since the first publications related to logical coupling [13], [14]. The drawback of this representation is that it either represents only modules, thus being very coarse-grained, or it represents modules and files, but then incurs scalability and overplotting problems. Our approach scales well to large systems and also can present detailed, file-level information.

B. Software Evolution Visualization

We already introduced related work on logical coupling in Section II and presented the most similar approaches to ours in the previous Section. Here we complete the related work by presenting approaches for software evolution visualization.

One of the first approaches to visualize historical information in software was proposed by Ball and Eick [2]. They used a very simple line-based visualization where each line in a file was color-coded to present information about the frequency of changes on it. However, no explicit coupling information could be inferred by using their tool.

Jazayeri et al. [17] visualize software release histories using colors and 3D. They do no visualize any coupling relationships between modules. A visual data-mining tool to represent both binary association rules and n-ary association rules is EPOsee [6]. 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 [8], graphical objects representing data through visual parameters. Their infobug glyph's parts represent data about software. The main difference between their and our work is the type of data that is visualized: they use glyphs to view project management data, while our work focuses on describing how a module is logically coupled to the others. Lanza's Evolution Matrix [18] visualizes the system's history in a matrix in which each row is the history of a class, and each column is a version of the system. A cell in the Evolution Matrix represents a version of 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.

Taylor and Munro [31] used a technique called *revision towers* to map various properties, such as size of a file, or the developers who work on it, and their evolution in time. Their technique works at file level but not at module level. Van Rysselberghe and Demeyer used a simple visualization based on information in version control systems to provide an overview of the evolution of a system [34]. Wu et al. described an Evolution Spectrograph [36] that visualizes historical sequences of software releases. Girba et al. used the notion of history to analyze how changes appear in software systems [15] and succeeded in visualizing the histories of evolving class hierarchies [16]. The difference between the mentioned approaches and ours is that the range of analysis is wider for our tool, as we can integrate both module-level and file-level information in our analysis.

Voinea and Telea [35] proposed CVSgrab, a tool to support querying, analysis and visualization of CVS based software repositories. Among others, the tool supports a visualization of

files' evolution, where time is mapped on the horizontal axes and the evolution of each file is rendered as a line from left to right. Several coloring scheme can be used in the visualization to render author information, file size, file type and file contents. In the visualization lines can be sorted and clustered according to the logical coupling of the corresponding files. The authors proved that such clustering can reveal pattern in the evolution of the system. The CVSgrab tool is quite different from our tool, as the former is a general visualization tool for CVS repositories where logical coupling can be used for sorting files and revealing patterns, whereas the Evolution Radar address specifically the problem of understanding logical coupling at the file and module levels and track them over time. Telea et al. in [32] introduced a visualization of source code evolution called Code Flows. The technique renders several versions of a given file, showing the evolution of the source code. It highlights unchanged code as well as important events such as code drift, splits, merges, insertions and deletions. Telea et al. approach does not provide logical coupling information.

VII. Conclusion

In this paper we have presented the Evolution Radar, a novel approach to integrate and visualize module-level and file-level logical coupling information. The Evolution Radar is useful to answer questions about the evolution of the system, the impact of changes at different levels of abstraction and the need for system restructuring. The main benefits of the technique are:

1) *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, using the spawning feature of our tool, to reduce it to a small set of strongly coupled files responsible for the dependency.
2) *Control of 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 that 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 illustrated our approach on two large and long-lived open source software systems: ArgoUML and Azureus. We have provided example scenarios of how to use the Evolution Radar to understand module dependencies and impact of changes at both file and module levels. We have found design issues such as God classes, misplaced files and module dependencies not mentioned in the documentation. We have also reduced these dependencies to coupling between small sets of files. These files should be considered for reengineering in order to decrease the coupling at the module level. The control of time has allowed us to understand the overall evolution of the systems (when modules were introduced/removed) and to identify phases in their histories.

We have shown how the tight integration of the Evolution Radar with an Integrated Development Environment can support maintenance activities like restructuring, re-documentation and change impact estimation. We have described how this support works by presenting an experience report of a real developer using the Evolution Radar inside an IDE, the Smalltalk System Browser.

Our future work is concerned with integrating other types of evolutionary information, such as bug-related information. The long-term goal is to find the missing link between logical coupling and the negative consequences it has on a system.

Acknowledgments: We gratefully acknowledge the financial support of the Swiss National Science foundation for the project "DiCoSA - Distributed Collaborative Software Analysis" (SNF Project No. 118063). We thank Richard Wettel for his availability in experimenting our tool and writing an experience report. We thank Damien Pollet for his feedback on drafts of this paper.

References

[1] T. Ball, J.-M. K. Adam, A. P. Harvey, and P. Siy. If your version control system could talk. In *Proc. ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.
[2] T. Ball and S. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.
[3] D. Beyer and A. E. Hassan. Animated visualization of software history using evolution storyboards. In *Proc. 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 199–210, Washington, DC, USA, 2006. IEEE Computer Society.
[4] S. Bouktif, Y.-G. Gueheneuc, and G. Antoniol. Extracting change-patterns from CVS repositories. In *Proc. 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 221–230, 2006.
[5] S. Breu and T. Zimmermann. Mining aspects from version history. In *Proc. 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society.
[6] M. Burch, S. Diehl, and P. Weissgerber. Visual data mining in software archives. In *Proc. 2nd ACM symposium on Software visualization (SoftVis 2005)*, pages 37–46, New York, NY, USA, 2005. ACM Press.
[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman, editors. *Readings in Information Visualization — Using Vision to Think*. Morgan Kaufmann, 1999.
[8] 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.
[9] C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *Proc. ACM Symposium on Software Visualization*, pages 77–86, New York NY, 2003. ACM Press.
[10] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proc. 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 189 – 198, 2006.
[11] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proc. 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 26 – 32, 2006.
[12] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of software engineering research on the practice of software configuration management. *ACM Transactions on Software Engineering and Methodology*, 14(4):383–430, Oct. 2005.
[13] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proc. International Conference on Software Maintenance (ICSM '98)*, pages 190–198, Los Alamitos CA, 1998. IEEE Computer Society Press.
[14] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, pages 13–23, Los Alamitos CA, 2003. IEEE Computer Society Press.

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

[16] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proc. 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

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

[18] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proc. 4th International Workshop on Principles of Software Evolution (IWPSE 2001)*, pages 37–42, 2001.

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

[20] R. C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

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

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

[23] J. Ratzinger, M. Fischer, and H. Gall. Improving evolvability through refactoring. In *Proc. 2nd International Workshop on Mining Software Repositories (MSR 2005)*, pages 1–5, New York, NY, USA, 2005. ACM Press.

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

[25] D. Roberts, J. Brant, and R. E. Johnson. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems (TAPOS)*, 3(4):253–263, 1997.

[26] M. J. Rochkind. The Source Code Control System. *Transactions on Software Engineering*, 1(4):364–370, 1975.

[27] M. Sherriff and L. Williams. Empirical software change impact analysis using singular value decomposition. In *Proc. International Conference on Software Testing, Verification, and Validation (ICST 2008)*, pages 268–277. IEEE Computer Society, 2008.

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

[29] J. T. Stasko and E. Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *Proc. IEEE Symposium on Information Visualization*, pages 57–. IEEE Computer Society, 2000.

[30] Team Development with VisualWorks. Cincom Techincal White Paper. Cincom Technical Whitepaper.

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

[32] A. Telea and D. Auber. Code flows: Visualizing structural evolution of source code. In *Proc. 10th Eurographics/IEEE Symposium on Data Visualization (EuroVis 2008)*, volume 27, pages 831–838. Eurographics, 2008.

[33] W. F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985.

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

[35] L. Voinea and A. Telea. An open framework for cvs repository querying, analysis and visualization. In *Proc. 3rd International Workshop on Mining Software Repositories (MSR 2006)*, pages 33–39. ACM, 2006.

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

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

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