oZone: Package Layered Structure Identification in presence of Cycles

Jannik Laval[a], Nicolas Anquetil[a], Stéphane Ducasse[a]

[a]*RMoD Team, INRIA Lille - Nord Europe, France, http://rmod.lille.inria.fr*

Abstract

Packages are complex entities and it can be difficult to understand them as they play different roles (*e.g.,* core package, UI class container, tests package ...). In particular, package interdependencies make difficult their management (substitution, evolutions, deprecation). Understanding whether a package has high or low impact on the system is another valuable piece of information. In addition, package organization represents the backbone of large software system. It is is usually agreed that packages should form layered structures. However, identifying such layered structure is difficult since packages are often in cycles. Several approaches propose to recover software structure or to visualize classes or files organization. Only few approaches provide layered organization and in particular take cycle into account. In this paper, we propose an approach which provides (i) a strategy to highlight dependencies which break Acyclic Dependency Principle and (ii) an organization of package (even in presence of cycles) in multiple layers. While our approach can be run automatically, it also supports human inputs and constraints. We validate our approach with a preliminary study on the structure of the Moose software analysis platform: it shows promising results.

Keywords: remodularization; layered organization; package cycle; package dependency

1. Introduction

Package are complex entities, they contain classes communicating inside and outside their boundaries. Understanding them and their role (*e.g.,* core package, UI class container, tests package ...) is important for reengineering because making changes to a package may impact the entire system depending on its role. Modifying a core package can have a large impact, whereas modifying a peripheral package should have a low impact on other packages. Maintainers need to know the place of a package before modifying it.

Previous work provides information on packages and their relationships, with visualizations, metrics [DPS+07, DG07]. In these approaches, it is not easy to understand the place of a package in the system, particularly when this system grows. Some other approaches propose to recover software structure or to visualize the organization of classes or files [MMCG99, MM06]. To understand the complexity of large object software systems and particularly the package structure, there are some visualization tools ([DGK06, DL05, BDL05, LSP05]). Package BluePrint ([DPS+07]) shows the communications between packages; eDSM ([LDDB09]) and CycleTable ([LDD09]) highlights the cycle problems in a system.

Moreover, package organization represents the backbone of large software systems. It is largely acknowledged that packages should form layered structures [BBC+00, DDN02] Having a layered organization allows for a simpler system maintenance because side effects are limited to layers. However, identifying such layered structure is difficult since package dependencies often form cycles. Several approaches propose to recover software structure or to visualize classes or files organizations [Vai04]. Only few approaches provide layered organization and in particular take cycles into account.

We know two main approaches extract layered structure from package dependencies and are exemplified in the NDepend (http://www.ndepend.com) and Lattix (http://www.lattix.com [SJSJ05]) tools. These two approaches do not really solve the problem of cyclic dependencies between package.

In this paper, we propose an approach, oZone, which provides (i) a strategy to highlight dependencies which break the Acyclic Dependency Principle; (ii) an organization of packages (even in presence of cycles) in multiple layers; and (iii) a simple visualization to allow expert inputs. While our approach can be run automatically, it also

supports human inputs and constraints. We validate our approach with a preliminary study on the structure of the Moose software analysis platform: it shows promising results.

The paper is organized as follows. Section 2 details the problem of layered organization provided by current tools, Section 3 explains our intuition and Section 4 explains our approach. Section 5 proposes a UI interface to interact with the layered organization. Section 6 is a preliminary study to validate the approach. Related work is analysed in Section 7. Section 8 discusses and concludes the paper.

2. Layer Problem Identification

Using a layered view of the software architecture is a simple and common view to understand it [DP09]. In addition, when the structure of a large system is layered, this simplifies its evolution since changes are limited to layers. To build a layered organization, there are two important principles at the package level: the Layered architecture that we want to build and the Acyclic Dependency Principle which is a pre-condition of a layer architecture.

Layered architecture. For Bachmann *et al.* [BBC+00], there are two properties in a layered architecture: (i) a layer B is below a layer A if elements (*i.e.,* packages) in layer A can use elements in layer B and (ii) layer A can use only packages below it (in Figure 1, layer B). In Figure 1 the dotted dependency from *Kernel* to *pA* should not exist. Szyperski [Szy98] and Bachmann [BBC+00] make a distinction between *Closed* layering and *Open* layering. In closed layering, the implementation of one layer should only depend on the layer directly below. In this case, in Figure 1 the dependency from *pE* to *Kernel* should not exist. In open layering, any lower layer may be used.

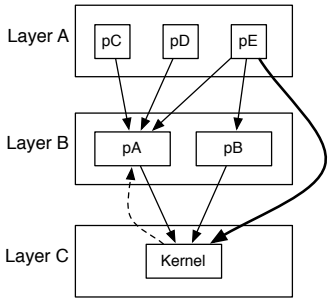


Figure 1: Layer Description - dashed arrow represents an *unwanted* dependency, thick arrow represents a dependency allowed in *Opened* layering

Acyclic Dependency Principle (ADP). A layered system offers good properties of modifiability and portability [BBC+00]. It means that there are no cycles between packages and that dependencies do not cross non-contiguous layers. Martin defines the Acyclic Dependencies Principle (ADP) [Mar00], which proposes that the graph of dependencies between packages should be a directed acyclic graph: there should not be any cyclic dependency between packages. Package structures with cycles are in general more difficult to understand, maintain and deploy than those that conform to the ADP. Legacy and large systems often present structures which do not respect this principle. Like an organic system and following the entropy principle, the more a software grows, the harder it is to keep this property.

Identifying layers in a package system is not trivial. In particular, when some packages are in cycle, layers cannot be computed without considering cycles as a special artifact. Different strategies may be used: for example, a cycle may be considered outside the layered architecture as NDepend does (Section 2.1) or it may be integrated in a single layer as Lattix or MudPie [Vai04] do (Section 2.2). The same strategy is used at the class level [Lut01, MMCG99, MM06], all classes in a cycle are in the same layer.

But this way to compute layers is not well suited for packages. First, because packages should not be in cycles, as the Acyclic dependency Principle (ADP) states. Second, because in a large software system with numerous packages in cycle, the granularity of layer would become so large that the layered structure would be useless and totally artificial.

For example, working on the remodularization of Pharo[1], an open-source Smalltalk environment, we found 70 packages in cycles. These packages are from the core (package Kernel), from the UI (package Morphic), from protocol (package Network) and a couple of other subsystems. In this case, grouping cyclic packages would lead to giant layers.

To better understand the different approaches to build package layered view in presence of cycles, we illustrate common approaches used in the two commercial tools already cited: NDepend and Lattix. We now present these two applications.

2.1. NDepend

NDepend [2] is a tool to help engineers to maintain software with the help of visualization and metrics. It provides a UI to manage large software maintenance. The website presents the tool as: "NDepend is a Visual Studio tool to manage complex .NET code and achieve high Code Quality. With NDepend, software quality can be measured using Code Metrics, visualized using Graphs and Treemaps, and enforced using standard and custom Rules."

NDepend considers a Layered Structure as an acyclic structure. When there are cycles, they are considered outside of the layer representation and structure. Based on the contents of the website information (http://www.ndepend.com/Features.aspx#DependencyCycle and http://www.theserverside.net/tt/articles/showarticle.tss?id=ControllingDependencies), we build the algorithm that computes layers:

```
1:   Model::computeLayers(): void {
2:       for( Package package: allPackages() ) {
3:           if (package.isInCycle() or package.useAPackageInCycle())
4:               package.layer := notAttributed
5:           elseif (package.providers() = nil)
6:               package.layer := 0
7:           elseif (package.providers().layer() = 0)
8:               package.layer := 1
9:           else
10:              package.layer := 1 + Max(package.providers().layer())
11:      }
12:  }
```

So, NDepend considers a layer organization only if the architecture respects the Acyclic Dependency Principle. In the previous pseudo code, the layered structure is computed without cycles and without the packages which use directly or indirectly a package in a cycle. Line 3 and 4 show that the algorithm puts out of the layered structure packages implied in cycles (a "not attributed" layer number is assigned to them). Line 5 to 8 initialize the two first layers of the structure: the layer 0 which represents the "Core" packages and layer 1 which use only layer 0. Finally the layer of other packages are computed considering the maximum layer they use.

Figure 3(a) shows an example of layers computed with the previous algorithm. "Not attributed" represents the packages in cycle and packages using packages in cycle. They are not included in the layer organization but they access a package in Layer 0.

The problems with this approach are multiple: (i) it is not possible to differentiate multiple cycles, (ii) if there is a cycle between two "Core" packages, most of the packages in the system will end up outside the layered organization.

2.2. Lattix LDM

Lattix LDM is a tool which allows engineers to create a dependency model of a software and manage dependencies with help of visualizations based on Dependency Structural Matrix (DSM). The tool provides features to make propositions for reengineering cyclic dependencies and provides a what-if approach to work on the structure.

[1] http://www.pharo-project.org
[2] http://www.ndepend.com

From the website (http://www.lattix.com/), the description of the tool is: "Lattix has pioneered an award-winning approach using system interdependencies to create an accurate blueprint of software applications, databases and systems. Architects and developers can analyze their systems in detail, edit the structure to create what-if scenarios, and specify design rules, allowing them to formalize and communicate the architecture to the entire organization. The result is higher quality, improved reliability, and much easier maintenance."

Lattix is the concrete implementation of the work of Sangal *et al.* [SJSJ05]. In the paper, Sangal *et al.* propose to organise the Dependency Structural Matrix (DSM) based on used and using packages, which provides a layer organization. This work considers a cycle as a feature, not as a modularization problem. It proposes to group each cycle in a container, named "module". A module contains a group of packages in cycle. When computing layered organization, modules are considered as a layer, non-cyclic packages are computed with the same kind of algorithm as NDepend (Section 2.1). This behavior makes a cycle considered as a layer.

Figure 3(b) shows an example of the conceptual architecture diagram made with Lattix. The three boxes represent modules of conceptual groups defined by software engineers. These boxes are organized in layers. Inside these boxes, packages are also organized in layers when it makes sense (e.g middle module).

Problems in this approach are multiple: (i) if cycles exist between packages that should be in different layers, they will be grouped in the same layer; (ii) it is not possible to differentiate a layer built from a cycle and one built from dependencies to a lower layer; (iii) when there are dependencies creating cycles between multiple packages, it is not possible to identify them because the layered view does not provide this kind of information (does not show package dependencies).

2.3. Other Tools

There are other tools like JDepend[3] or Classycles[4] which allow software engineers to see package dependencies and cycles. But these tools do not provide a layered organization of the package structure. JDepend is a tool which computes design metrics on Java class directories to generate a quality view of packages. Classycle is a tool to see cycles between classes and shows package dependencies and cycles based on class information. It uses the same algorithm as JDepend.

Another tool, named Structure101[5] proposes to build layer organization and to analyze dependencies which break the layered organization. This tool is well integrated with the source-code environment and it provides services to modify directly source code. However, this particular feature of the tool starts off empty, and stays empty until the developer defines some architecture diagrams. Then it helps the developer understands how the view of the architecture differs from the source code.

3. The intuition behind OZONE: selecting Shared Dependencies

In previous section, we showed that a layered organization can be provided using different strategies: considering cycles as layers or considering that package in cycles and all their dependent packages are outside the layered organization. In these two approaches, the fundamental problem of cycles is set aside and no effort is done to break these cyclic dependencies to respect the ADP.

As a general principle cycles between packages should not exist. We consider cycles as design issues, architecture violations or programming errors. A dependency which creates a cycle should be highlighted when one analyses a system so that the software engineers has an opportunity to focus his/her attention on it and try to remove it. Package dependencies stem from dependencies at the level of classes inside a package: package A depends on a package B if a class within A depends on a class within B. The intuition behind OZONE is that, in a cycle, all package dependencies don't have the same strength depending on how many class dependencies they are made of. We hypothesize that an *unwanted* dependency has a strong impact on the system structure. Particularly if we decompose a cycle in small ones, some dependencies appear in multiple small cycles. These dependencies seems to be removed as they have a strong and bad impact on the system. These dependencies should be the ones that should not exist and that they are

[3] http://www.clarkware.com/software/JDepend.html
[4] http://classycle.sourceforge.net/
[5] http://www.headwaysoftware.com/products/structure101/

the result of a programming error. In this sense, we consider that it is legitimate to ignore ("remove logically") these shared dependencies when assigning packages to layers. Dependencies creating cycles with high sharing result are removed from the computation of the layers and such removal supports the identification of layers not showing the problems mentioned in previous sections.

In Figure 2, we propose a simple graph with 5 nodes (*i.e.,* packages). The three grey packages (PackA, PackB, PackC) are in cycle, the two other packages are the Kernel and an UI package. All edges in the figure are labeled with the number of dependencies between classes, the *weight* of the package dependency.

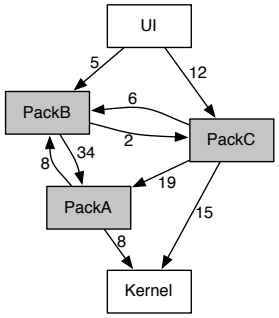
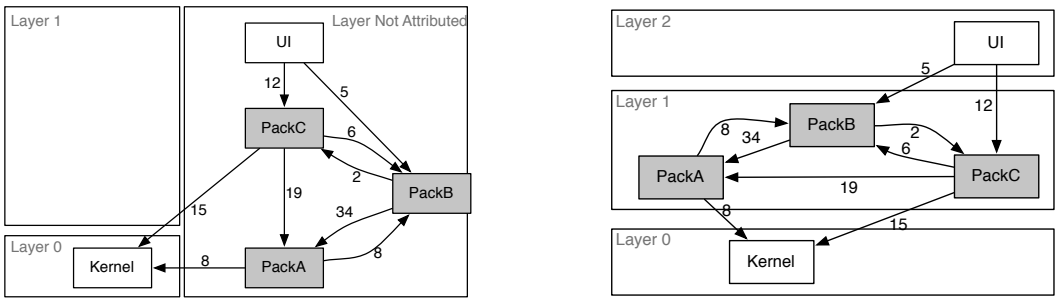


Figure 2: An example of cycle between packages

Figure 3 presents the layered organizations obtained by the algorithms used in NDepend and Lattix LDM. Note that our representation shows the dependencies between packages while the tools would not do it. These organizations are not adapted to cycles between packages. NDepend (on the left) has only one layer because only one package is not concerned by the cycles. Lattix LDM (on the rigth) puts the three concerned packages in a single layer.



(a) Result of applying NDepend's Strategy

(b) Result of applying Lattix's Strategy

Figure 3: Layered organization obtained by NDepend algorithm (left) and Lattix algorithm (right) when applied to the system described in Figure 2

Figure 4 presents the layered organization that we would like to see. We would like to have a decomposition of the structure in multiple layers while highlighting the dependencies which break the ADP. In Figure 4 the two weaker dependencies, that break the ADP, are dotted. Applying the intuition presented above: the dependencies with the higher sharing result in the cycles are not taken into account to compute the layered architecture. Here, the cycle between the three grey packages (PackA, PackB, PackC) can be decompose in three cycles: (PackA -> PackB -> PackA), (PackB -> PackC -> PackB) and (PackA -> PackB -> PackC -> PackA). We see that dependencies (PackA -> PackB) and (PackB -> PackC) appear two times while the others appear only one time.

These dependencies introduce cycles that clutter the previous algorithms' results. Their existence hampers the creation of two layers (layers 1 and 3 in Figure 4).

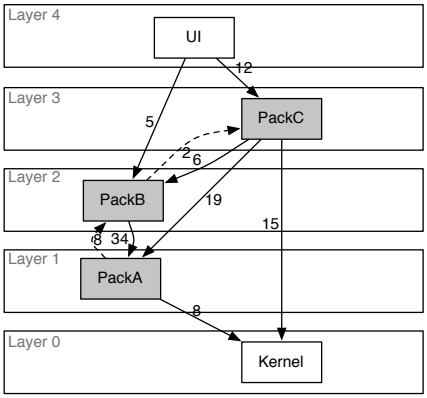


Figure 4: The layered organization from Figure 2 we expected to see

4. Our solution: Detecting dependencies hampering layer creation

In this section, we propose a strategy to build a layered organization of system packages potentially having cycles. The approach considers all cycles and finds the "adequate" dependencies to remove to create distinct layers. We named them *dependencies breaking or hampering layer creation*. These dependencies are defined based on the intuition explained in previous section. It focuses only on dependencies in cycles. The strategy is not concerned with other dependencies because the first goal is to break cycles to build layered organization.

As explained in Section 2, there are two kinds of layering [BBC+00]: (i) the *Closed* one which considers that a layer can only call the layer immediately below it, and the *Open* one which considers that a layer can call all layers below it. Our layered organization is based on the Open layering because the Closed one is not adapted to software architecture. The Closed layering is useful for a clean and well encapsulated system, which is not adapted to our purpose because we do not want to identify a well-encapsulated system, but to identify dependencies which break the layered organization.

Next we give some definitions used in our approach. Second we explain how *dependencies breaking layers* are found. Then we explain how to compute a layered organization. Finally, we explain a feature to define constraints on dependencies manually, for adapting layered organization to the reengineer vision.

4.1. Definitions

Definition 1. A Strongly Connected Component (SCC) in a graph is the maximal set of nodes (here, packages) that depend on each others. In Figure 5 (left), all nodes are in a single SCC. We use the Tarjan SCC algorithm [Tar72] for the implementation.

Definition 2. A cycle is a circular dependency between two or more packages. We distinguish an SCC and a cycle. An SCC is a collection of node, a cycle is a path which comes back to its origin. We distinguish two kinds of cycles:

- *direct cycle*. It represents a cycle between two packages. In Figure 5, C and D are in direct cycle because there is one dependency from C to D, and one dependency from D to C.
- *indirect cycle*. It represents a cycle between more than two packages. In Figure 5, A, B and C are in indirect cycle. A, B and E are also in indirect cycle.

Definition 3. A *minimal* cycle is a cycle with no node appearing more than once. In graph theory, it is named a simple cycle. In Figure 5, A-B-E and A-B-C are two different minimal cycles, but A-B-C-D-C is not because C is present twice. A-B-C-D-C can be reconstructed with the two minimal cycle A-B-C and C-D. To retrieve minimal cycles, some algorithms exist as [Tie70, Wei72]. We use the algorithm proposed by Falleri et al. [FDL+10], which is the most recent and the most efficient.

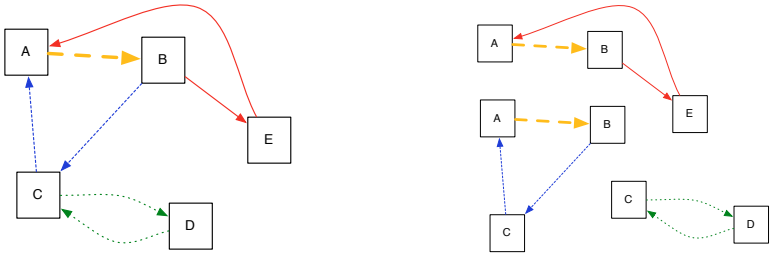


Figure 5: (left) Sample graph representing a SCC. (right) Sample graph decomposed in three minimal cycles

Definition 4. A *shared* dependency is a dependency presents in at least two minimal cycles. In Figure 5 (right), the edge between A and B is shared by the two minimal cycles A-B-E and A-B-C.

Definition 5. A *dependency breaking layers* is a dependency which (i) breaks the Acyclic Dependency Principle and (ii) seems to be the best dependency to remove to be able to build a "adequate" layered organization.

4.2. Highlighting dependencies breaking layers

As we showed in previous section, a problem to build a layered architecture is how to take into account cycles between packages. We propose a strategy to highlight dependencies which could break cycles to make a layered system.

This heuristics is based on observations and experiments from our previous works [LDDB09, LDD09]. We compute the strategy on a graph where each node represents a package and each edge represents a dependency between two packages. A dependency between packages is complex because it depends on relations between classes and methods inside these packages. Edges are weighted with the number of dependencies between elements in the package (class inheritance, class extension, class reference, method invocation, variable access).

The algorithm runs in two steps and removes unwanted edges from a copy of the model of the system:

1. one of the two dependencies of each direct cycles is cut. To remove a direct cycle the algorithm considers the weight of each dependency in the cycle and select the lightest.

2. when all direct cycles are removed, the algorithm considers the new version of the graph and computes SCC and minimal cycles to retrieve shared dependencies. When there are shared dependencies, it removes the one most shared, because this is the one with the highest impact on minimal cycles. We repeat this action as long as there are cycles. If two dependencies are shared by the same number of cycles, the algorithm selects the lightest of them.

The algorithm computes first direct cycles. There are two reasons to begin with them: (i) when a cycle is broken, it could have an impact on other larger cycles that include it[6], (ii) breaking a direct cycle is somehow simpler because there are only two solutions: breaking one or the other of the two package dependencies.

Based on the definitions and strategy previously explained, we propose an algorithm to remove edges in a package dependencies graph. In this algorithm, the term "remove" is used to (i) remove an edge from the result graph and (ii) to store the dependency to highlight it with the goal to later understand it in the source code.

```
1:   Model::getRemovedEdges(Graph graph): Collection {
2:       for (Cycle cycle: computeDirectCycles()) {
3:           if (cycle.edgeOne.weight() > cycle.edgeTwo.weight() * 3)
4:               graph.remove(cycle.edgeTwo)
```

[6]For example in Figure 2 breaking the direct cycle PackA-PackB, can also break the cycle PackA-PackB-PackC.

```
5:          elseif (cycle.edgeTwo.weight() > cycle.edgeOne.weight() * 3)
6:              graph.remove(cycle.edgeOne)
7:          else
8:              graph.removeTheMostSharedEdge(cycle.edgeOne, cycle.edgeTwo)
9:              or (graph.removeTheLightestEdge(cycle.edgeOne, cycle.edgeTwo))
10:             or (graph.remove(cycle.edgeOne))
11:     }
12:     while (computeSCC().notEmpty()) {
13:         graph.computeMinimalCycles()
14:         graph.removeTheMostSharedEdge()
15:     }
16:     return graph.removedEdges
17: }
```

The presented algorithm works as follows: from line 2 to line 11, direct cycles are removed from the graph. It checks for large differences between the two edges of the direct cycle (it uses a ratio of 1/3) and removes the lightest (l.3 to l.6). If the difference is not important enough, the algorithm checks shared dependencies and removes the most shared (l. 8). If the two edges have the same number of shared dependencies, it removes the lightest edge (l.9). If none of these conditions are satisfied, the algorithm removes the first edge (l.10). This last line is necessary to remove all cycles to make a layered architecture. It is akin to removing a random dependency in the cycle, but the engineer can specify constraints to guide the tool by explicitly marking a dependency as valid or not (explained in Section 4.4). Then from line 12 to 15, the algorithm removes other cycles if there are still SCC. It computes minimal cycles (l.13) and removes the most shared (l.14). If there are multiple dependencies with the same shared number, it selects the less weighted dependency. The algorithm returns a collection of *dependency breaking layers*.

4.3. Building layers

When the previous algorithm returns *dependencies breaking layers*, we can build an acyclic graph of the package dependencies. With this acyclic graph, we can easily build a layered organization. This second algorithm does not take into account the dependencies removed by the first one. These specific dependencies are added after the creation of the layers to highlight them for the software engineer.

The algorithm is simple. It is the same one used in NDepend (Section 2.1). The first layer is built with the packages which do not use any other packages. Then each layer is built with packages which use only packages on lower layers.

```
1:  Model::buildLayers(Graph aCyclicGraph): Collection {
2:      L := Collection
3:      N := aCyclicGraph.nodes
4:      while (N.notEmpty()) {
5:          currentL = L.addNewLayer()
6:          concernedNodes = N.selectNodesWithoutOutgoing(N)
7:          currentL.add(concernedNodes)
8:          N remove(concernedNodes)
9:      }
10:     return L
11: }
```

The previous algorithm builds the layered architecture. The lines 2 and 3 initialize variables: L is a collection of layers. Each layer is a collection of packages. N is the collection of all packages of the system. Lines 4 to 9 build the layered architecture. Each layer (l.5) is filled by putting into it packages without outgoing dependencies to packages contained in N (l.6 and 7). Finally, it removes the selected nodes from N (l.8).

4.4. Defining constraints manually

The algorithm provides a result which may not match totally the vision of the reengineer. Because of his/her knowledge of the system, s-he can add constraints on dependencies and ask the algorithm to recompute the organization. The automatic computation based on the strategy provides a first step to understand. Then the engineer can add constraints. He can evaluate each dependency by giving it a value (see below) which is taken into account when the algorithm is run another time.

We design four level of evaluation: (i) *flaggedByAlgo* for dependencies detected by the algorithm as unwanted (the *dependencies breaking layers*) (ii) *unwanted* (flagged -1) for dependencies that the software engineer would like to remove, (iii) *notFlagged* (with the value 0) for the dependencies for which the software engineer does not know whether they are expected or not, (iv) *expected* (flagged 1) for the dependencies which should not be removed in the software engineer opinion. These constraints add a new dimension to the algorithm:

- By default the algorithm flags with *flaggedByAlgo* dependencies which it considers breaking layers. Then the engineer should confirm with the flag *unwanted* or invalidate with the flag *expected*.
- When a dependency is flagged *expected*, it is not removed by the algorithm, and it checks a new dependency to remove the cycle. If all dependencies of a cycle are flagged as *expected*, the cycle cannot be removed. In this case, all involved packages are in the same layer.
- When a dependency is flagged *unwanted*, it is automatically removed from the graph. So the algorithm does not take into account the dependency. When it is a part of a cycle, it is not computed in SCC and minimal cycle search.
- When a dependency is *not flagged*, the algorithm considers it as removable it if necessary.

5. An interactive browser to build layers

We built a UI to deal with features provided by the approach: (i) highlighting dependency strategy, (ii) user defined constraints, (iii) layers architecture building. This interface is a prototype, it has been created for our study and should probably be improved.

It has been implemented on top of the Moose reengineering environment [DGLD05] and it is based on the FAMIX language independent source code metamodel [DTD01]. It can work on Java, C#, and C++ as well. Therefore while implemented in Smalltalk, it can be applied to mainstream object-oriented languages.

It is composed of three main panels: a layers visualization with polymetric view [LD03] (on the left of the UI), a list of unwanted dependencies (on the top center of the UI), and a list of all dependencies of the system (on the top right of the UI)[7]. The list of unwanted dependencies contains dependencies selected by the algorithm and dependencies removed manually by the engineer.

5.1. The layer visualization

As the UI, the layer visualization is an experimental tool. We build it to show our main concerns: relations between packages, packages concerned by SCC, and some convenient metrics about the size of packages to help reengineer to have a first view of the system.

Layer. It is a rectangle with boxes representing packages inside. Bottom packages are in Layer 0 (*i.e.,* core layer). Each new rectangle represents a new Layer (Figure 6, left part).

[7] The lower right part of the figure is for more advanced analysis and should be ignored here.

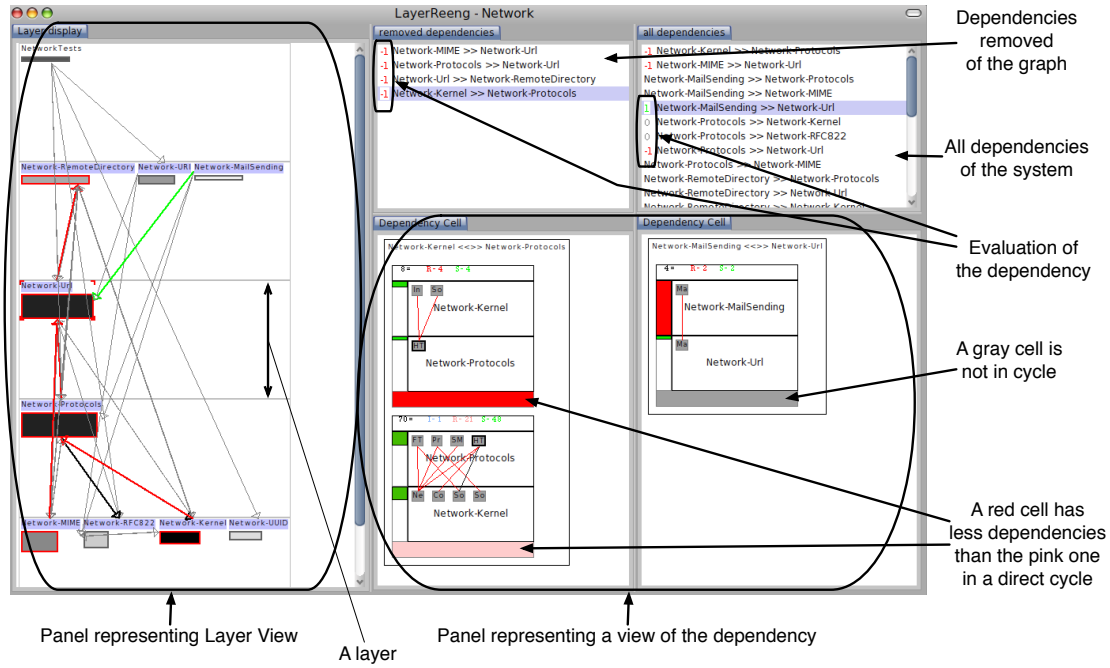


Figure 6: UI to build layered view based on constraints

Package. It is a box with polymetric views [LD03, DDL99, GLD05]. Each package is represented by its name and a box. The height, the width, the fill color and the border color have a meaning (Figure 7):

- height: it represents the number of client packages. It means that tall packages should be located at lower layers.
- width: it represents the number of provider packages. It means that a package in a low layer should be narrow.
- fill color: it represents the number of classes in the package. The darker a package is, the more classes it contains.
- border color: it represents an SCC. If the color is gray, the package is not in an SCC. If it is another color, the package is in an SCC with all other packages with the same border color.

This kind of information allows the engineer to understand at a glance the implication of a package in the system.

Dependency. It is represented by an arrow. It can have three colors. The color red represents a dependency flagged *unwanted*, a dependency that the engineer wants to remove. The color green is a dependency flagged *expected*, a normal dependency. The color black is a dependency *not flagged*. Finally the color gray represents dependencies not yet evaluated.

This system allows the reengineer to understand which dependencies should be removed. In a layered view, all dependencies should go from a higher layer to a lower layer.

5.2. Interactions

Interactions on lists. On the two lists on the top right of the UI (one as unwanted dependencies and one with all dependencies of the system), reengineers can define constraints explained in Section 4.4. With a simple right click, a popup menu appears and proposes to select one of the three available value: 1 (for *expected*), 0 (for *notFlagged*) or -1 (for *unwanted*).

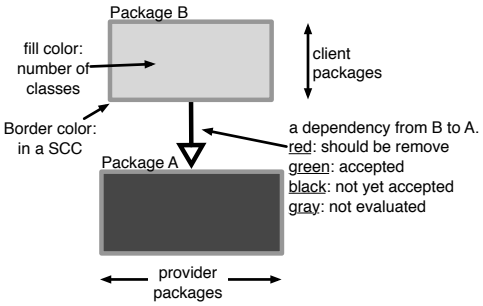


Figure 7: The polymetric view used in Layer View

When a new value is given to a dependency, the algorithm recomputes the layered organization and the UI is updated. One can see the result automatically. To help the reengineer to flag a dependency, we integrate a view of dependency (lower right part of the UI). It is an eCell, which has already been used in eDSM [LDDB09]. We do not provide information of this feature because it is not the topic of the paper.

Interactions on layer view. On the layered view, it is possible to select a package and put it in another layer (for now, as the interaction system is not complete, the reengineer uses a right click and selects the layer where he wants to put the package).

This behavior supports the computation of layers with constraints based on package layer, not on dependencies. A package (fixed on a layer) influences the algorithm to consider as a *dependencies breaking layers* all dependencies which do not respect the Layered architecture definition. Then the reengineer should confirm that all these dependencies are unwanted by flagging them with *unwanted*.

6. Validation

We performed a study to validate our approach. The goal of this preliminary study is to validate two important features of the approach: (i) Are *dependencies breaking layers* relevant ? (ii) Does the layer organization reveal the system organization ?

6.1. Protocol

The case study was realized on the beta version 4 of Moose 1.0. It proves than it is not easy, it contains 33 packages and 106 dependencies between these packages. This version is particularly interesting because it contains a lot of cyclic dependencies, it has not been well modularize. A developer from the Moose team (not the authors) evaluated all package dependencies of the system. The possible values that the developer can give are: 1 (*expected*) if the dependency is expected in the system architecture, 0 (*notFlagged*) if he can not answer precisely, -1 (*unwanted*) if the dependency should be removed.

Then, we performed our algorithms on the system and compared *dependencies breaking layers* found by the algorithm and *unwanted* dependencies given by the engineer.

6.2. Results and discussion

The algorithm proposed to remove 15 dependencies (Table 1) that we controlled and compared with the values given by the Moose engineer. Table 1 shows that 10 out of 15 *dependencies breaking layers* (66%) are considered *unwanted* by the engineer. Without manual changes, results validate that the approach has a good strategy. But the 5 false-positive dependencies are due to two kinds of issue in the algorithm.

- First, the two dependencies *Moose-Finder to Moose-Wizard* and *Moose-SmalltalkImporter to Famix-Implementation* are in direct cycle. It means that there are also two dependencies *Moose-Wizard to Moose-Finder* and *Famix-Implementation to Moose-SmalltalkImporter*.

  Our algorithm constrains all cycles to be broken. In these two cases, both dependencies in the two direct cycles have similar weights. Without shared dependencies available, the algorithm had no clue which dependency to break and had to choose "randomly" the first of the two. We already alluded to this problem in Section 4.2.

  This point should be improved in future work with the possibility for the maintainer to choose the *unwanted* dependencies avoiding this kind of false-positive.

- Second, the three false-positive results going to Moose-Core (*Famix-Core to Moose-Core*, *Moose-SmalltalkImporter to MooseCore* and *Moose-GenericImporter to Moose-Core*) are related to a choice of the algorithm to remove the dependency *Famix-Core to Moose-Core* instead of *Moose-Core to Famix-Core*.

  In this case, the problem comes from two dependencies from a direct cycle which have the same weight but *Famix-Core to Moose-Core* is shared one more time than *Moose-Core to Famix-Core*.

  As in the previous case, we should improve the algorithm to be more flexible and propose to maintainer to have the choice before the false-positive results. If we constrain to remove *Moose-Core to Famix-Core*, these three false-positive results disappear.

Dependency breaking layers	Value given by engineer
Famix-Extensions » Moose-Finder	-1
Moose-Core » Famix-Implementation	-1
Fame » Moose-Core	-1
Famix-Extensions » DSMCore	-1
Famix-Core » Famix-Implementation	-1
DSMCore » DSMCycleTable	-1
Glamour-Helpers » Glamour-Core	-1
Glamour-Browsers » Glamour-Scripting	-1
Moose-Core » Famix-Extensions	-1
Famix-Smalltalk » Famix-Extensions	-1
Moose-SmalltalkImporter » MooseCore	1
Moose-Finder » Moose-Wizard	1
Moose-GenericImporter » Moose-Core	1
Famix-Core » Moose-Core	1
Moose-SmalltalkImporter » Famix-Implementation	1

Table 1: *Dependencies breaking layers* returned by the algorithm

After a first pass to validate the 10 *dependency breaking layers* as *unwanted* and giving the value *expected* to the 5 false-positive results, the algorithm retrieved 4 new *dependency breaking layers*, shown in Table 2. All of them are considered *unwanted* dependencies by the engineer. In total, the algorithm proposes 14 dependencies considered *unwanted*.

Dependency breaking layers	Value given by engineer
MooseCore » Moose-SmalltalkImporter	-1
Moose-Core » Famix-Core	-1
Moose-Wizard » Moose-Finder	-1
Moose-Core » Moose-GenericImporter	-1

Table 2: New *dependencies breaking layers* after manual constraints

The first version of the layer organization proposes 11 layers. Figure 8 shows a simple view of layers organization. It shows particularly the main problem revealed in the first result of the algorithm: Moose-Core is too high in the layer organization, due to a false-positive result (*Famix-Core to Moose-Core*).

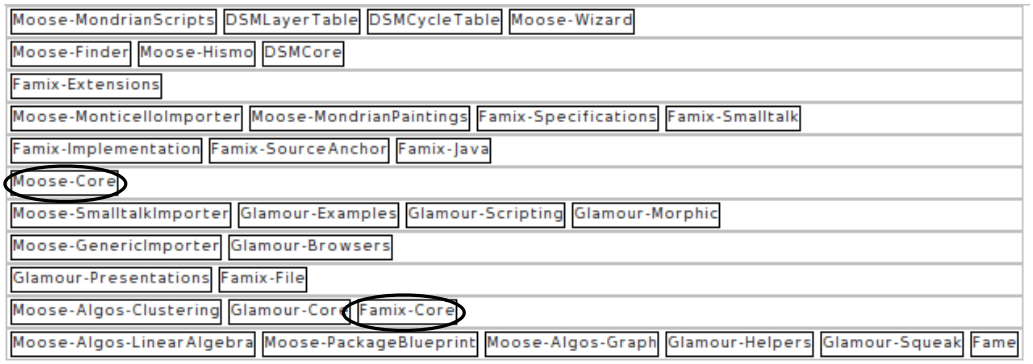


Figure 8: Layer organization proposed by the algorithm.

After weighting manually false-positive results, the algorithm recomputes the layers organization and provides a 8 layers organization (Figure 9). In this view, we see that the problem of Moose-Core is resolved because Moose-Core appears on the second layer, which is the correct place for this package.

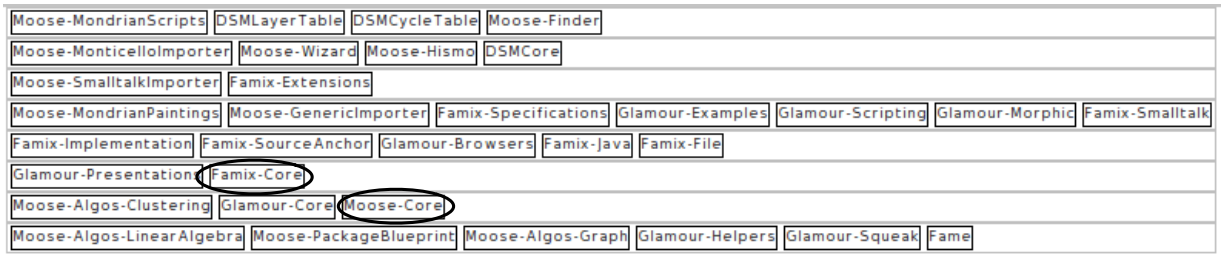


Figure 9: Layer organization proposed after manually constraints.

7. Related Work

Much researche work have been done on software reengineering. In this section, we select some works related to our work.

PASTA [Hau02] is a tool for analyzing the dependency graph of Java packages. It focuses on detecting layers in the graph and consequently provides two heuristics to deal with cycles. One views packages in the same strongly connected component as a single package. The other heuristic selectively ignores some *undesirable* dependencies until no more cycle is detected. Thus, PASTA reports on these *undesirable* dependencies which should be removed to break cycles. The *undesirable* dependencies are selected by computing these weight and selecting the minimal ones. Our approach takes one more parameter into account: we consider also the impact of the dependency on the system with the *shared dependencies*.

JooJ [MT07b] is an eclipse plugin (not released) to detect and remove as early as possible cyclic dependencies between classes. The principle of JooJ is to highlight statements creating cyclic dependencies directly in the code editor. It computes the strongly connected components to detect cycles among classes. It also computes an approximation of the minimal set of edges to remove in order to make the dependency graph totally acyclic. This problem

is called minimum feedback arc set in the graph literature, but is NP-complete. It highlights the minimum number of statements that one needs to remove all cycles among classes. However, no study is made to validate this approach for cycle removal. It is possible that the selected dependencies are not to be removed because they are valid in the domain of the program.

Melton et al. [MT07a] proposes an empirical study of cycles among classes. They compute metrics to define better candidate dependencies to remove. They particularly indicates that it is crutial to take into account the semantic of the software architecture to not break dependencies that should not be broken. In our work, we include a part of user validation to take into account the semantics.

Lutz [Lut01] proposes a hierarchical decomposition of a software system. It uses a genetic algorithm to find the best way to group components of the system into coarse-grained components. Mudpie [Vai04] is a tool to help the maintenance of software system by bringing out SCCs and focusing on dependencies in SCC. Multiple works [ADSA09] exist to decompose a system by using genetic heuristics. Our work is not in this domain. Our goal is to recover dependencies which break the system, in particular, the layered organization of the system.

Dong and Godfrey [DG07] propose an approach to study dependencies between packages and to give a new meaning to packages with (1) characterization of external properties of components, (ii) usage of resource and (iii) connectors. It helps the maintainers to understand the nature of package dependencies. This kind of tool is useful to understand a global system. It could be used in the view of a dependency to replace eCell. We can also replace eCell by node-link visualization which does not need learning time.

Lungu et al.[LLG06] propose a collection of package patterns to help reengineers to understand large software system. They propose to recover architecture based on package information and an automatic process to recover defined patterns. Then they propose an UI to interact with the package structure. This approach is useful to understand the behavior of a package in the system. It can provide information about the position of a package in a layered organization. This kind of patterns could be used to add more informations on a package and to propose more information about the breaking of a dependency, for example knowing that a package is autonomous is a valuable information.

Bunch [MMCG99, MM06] is a tool which remodularizes automatically a software. It proposes to decompose and to show an architectural-view of the system structure based on classes and function calls. It helps maintainer to understand relations between classes. This tool breaks the package concerns and does not provide the information we need to make a layered organization of a package system. Our work is based on package architecture.

The Kleinberg algorithm [Kle99] defines authority and hub values for each class in a system. A high authority means the class is used by a big part of the system, and the hub value means the class uses multiple other classes in the system. Scanniello et al. [SDDD10] propose an approach to build layers of classes based on this algorithm. They identify relations between classes and use the Kleinberg algorithm to group them into layers. They propose a semi-automatic approach which allows the maintainer to manipulate the architecture and add its proper meaning of the system.

In graph theory, a feedback arcset is a collection of edges which should be removed to obtain an acyclic graph. The *minimum* feedback arcset is the minimal collection of edges to remove to obtain an acyclic graph. This theoretical approach cannot be used for three particular reasons: (i) It is a NP-complete problem (optimized by Kann [Kan92] to become APX-hard). Some approaches proposes heuristic to compute the Feedback Arc Set Problem in reasonable time[ELS93]; (ii) It does not take into account the semantic of the software structure. Optimizing a graph is not equivalent to a possible solution at the software level; (iii) The goal of breaking cycles in software applications is not to break a minimal set of links, but the more pertinent ones.

Software clustering is another domain in relation to our work. The goal of clustering is to order elements into modules based on some criteria defined by the engineer [ADSM05, PHY10]. This kind of approaches can be useful to manipulate fine-grained information. In our work we manipulate packages and we consider that a package has a meaning for engineers, as it should not be broken automatically.

8. Conclusion

In this paper we show that building a package layer organization is not trivial in presence of cycles. Two existing approaches are presented where they use a simple strategy to take into account cycles. Based on our experience, we

propose a strategy to organize a system containing cycles between packages in layers. We consider *dependencies breaking layers* defined in the strategy and provide a user interface to add manual constraints. As the weight is not enough to break cycles, our approach select *unwanted* dependencies based on two characteristics: *shared* dependencies and *light* dependencies.

The preliminary study shows that the strategy provides good results but the strategy should be improved to be more flexible. First, we can see that a manual verification is needed to validate or not *dependencies breaking layers*. By extension, the user interface and the visualization provided in the Section 5 should be improved to be more usable and to highlight some feature of the system. The idea is that the algorithm should not remove absolutely all cycles, but ask to engineers its decision when it is is difficult to decide.

Finally, the strategy is based on *shared dependencies*, it depends on the analysis of the complete system to have all shared dependencies. In the case of computing the algorithm on only a part of a system, shared dependencies are lower and the algorithm should return more false-positive values. Here again, the algorithm should be more flexible and ask to engineer when it cannot decide.

In future work, we plan to (i) improve the strategy to detect *dependencies breaking layers*, (ii) have the possibility to simulate some layered organization and select the better for the reengineer vision and (iii) improve the informations provided by the list to understand why a dependency has been removed by the algorithm. Finally, we plan to analyze some algorithms as the Kleinberg algorithm [Kle99] which seems to be a good perspective for evolving our algorithm.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

References

[ADSA09] Hani Abdeen, Stéphane Ducasse, Houari A. Sahraoui, and Ilham Alloui. Automatic package coupling and cycle minimization. In *International Working Conference on Reverse Engineering (WCRE)*, pages 103–112, Washington, DC, USA, 2009. IEEE Computer Society Press.

[ADSM05] Olena Andriyevska, Natalia Dragan, Bonita Simoes, and Jonathan I. Maletic. Evaluating UML class diagram layout based on architectural importance. *VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 0:9, 2005.

[BBC+00] Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord, Reed Little, Norton L. Compton, and Lt Col. Software architecture documentation in practice: Documenting architectural layers, 2000.

[BDL05] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172, New York, NY, USA, 2005. ACM.

[DDL99] Serge Demeyer, Stéphane Ducasse, and Michele Lanza. A hybrid reverse engineering platform combining metrics and program visualization. In Francoise Balmas, Mike Blaha, and Spencer Rugaber, editors, *Proceedings of 6th Working Conference on Reverse Engineering (WCRE '99)*. IEEE Computer Society, October 1999.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DG07] Xinyi Dong and M.W. Godfrey. System-level usage dependency analysis of object-oriented systems. In *ICSM 2007*. IEEE Comp. Society, 2007.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[DGLD05] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[DL05] Stéphane Ducasse and Michele Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.

[DP09] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[DPS+07] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47:319–323, 1993.

[FDL+10] Jean Rémi Falleri, Simon Denier, Jannik Laval, Philipe Vismara, and Stéphane Ducasse. Efficient retrieval and ranking of undesired package cycles in large software systems. Technical report, INRIA, November 2010. Computer Science Technical Report.

[GLD05] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[Hau02] Edwin Hautus. Improving Java software through package structure analysis. In *IASTED, International Conference on Software Engineering and Applications*, 2002.

[Kan92] Viggo Kann. *On the Approximability of NP-complete Optimization Problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992.

[Kle99] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *JOURNAL OF THE ACM*, 46(5):604–632, 1999.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[LDD09] Jannik Laval, Simon Denier, and Stéphane Ducasse. Identifying cycle causes with cycletable. In *FAMOOSr 2009: 3rd Workshop on FAMIX and MOOSE in Software Reengineering*, Brest, France, 2009.

[LDDB09] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[LLG06] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[LSP05] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.

[Lut01] Rudi Lutz. Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47(7):613–634, 2001.

[Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[MM06] Brian S. Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MMCG99] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[MT07a] Hayden Melton and Ewan Tempero. An empirical study of cycles among classes in java. *Empirical Software Engineering*, 12(4):389–415, 2007.

[MT07b] Hayden Melton and Ewan D. Tempero. Jooj: Real-time support for avoiding cyclic dependencies. In *APSEC 2007 - 14th Asia-Pacific Software Engineering Conference*, pages 87–95. IEEE Computer Society, 2007.

[PHY10] Kata Praditwong, Mark Harman, and Xin Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 99(PrePrints), 2010.

[SDDD10] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico, and Teodora D'Amico. Architectural layer recovery for software system understanding and evolution. *Softw. Pract. Exper.*, 40(10):897–916, 2010.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[Szy98] Clemens A. Szyperski. *Component Software*. Addison Wesley, 1998.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Tie70] James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM*, 13:722–726, December 1970.

[Vai04] Daniel Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.

[Wei72] Herbert Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *J. ACM*, 19:43–56, January 1972.