oZone: Layer Identification in the presence of Cyclic Dependencies

Jannik Laval[a], Nicolas Anquetil[b], Usman Bhatti[b], Stéphane Ducasse[b]

[a] *Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France, http://sphere.labri.fr/*
[b] *RMoD Team, INRIA Lille - Nord Europe - USTL, France, http://rmod.lille.inria.fr*

Abstract

A layered software architecture helps understanding the role of software entities (*e.g.,* packages or classes) in a system and hence, the impact of changes on these entities. However, the computation of an optimal layered organization in the presence of cyclic dependencies is difficult. In this paper, we present an approach that (i) provides a strategy supporting the automated detection of cyclic dependencies, (ii) proposes heuristics to break cyclic dependencies, and (iii) computes an organization of software entities in multiple layers even in presence of cyclic dependencies. Our approach performs better than the other existing approaches in terms of accuracy and interactivity, it supports human inputs and constraints. In this paper, we present this approach and compare it to existing solutions. We applied our approach on two large software systems to identify package layers and the results are manually validated by software engineers of the two systems.

Keywords: Remodularization; layered organization; cycle; package dependency

1. Introduction

Classes and packages are important software entities that form building blocks of object-oriented software. Understanding such entities and their dependencies is important for reengineering because making changes to an entity may impact the entire system depending on its dependencies. Modifying a core entity can have a large impact, whereas modifying a peripheral entity should have a low impact on the other entities. A good organization of these entities eases the understanding, maintenance, test and evolution of software systems [Mar00].

It is considered a good practice to arrange software entities in layered structures [BBC+00, DDN02]. It allows for simpler system maintenance because side effects are limited to the layers above the change. Therefore, layered organization of software entities helps controlling change impact in a software system. Such layered organization is formed by placing client entities on top of their provider entities [BBC+00]. According to this principle, creating a layered organization from an existing program can look as simple as: (i) compute dependencies between software entities; and (ii) put the entities in layers such that no lower layer accesses a layer above it.

However, in practice, it might be difficult to ascertain providers and clients because entities have cyclic dependencies among each other: Cyclic dependencies are common in large software applications [FDL+11]. For example, the largest cycle of ArgoUML contains almost half of the system's packages, and for JEdit, almost two third of the packages are in the largest cycle [FDL+11]. Hence, cyclic dependencies should be dealt with before trying to create a layered organization.

Existing approaches such as the ones based on dependency structural matrix [SJSJ05] or minimal feedback arc set [ELS93] do not adequately handle the problem of cyclic dependencies. These approaches may place unrelated packages, *e.g.,* Tests and Core, in a single layer, which does not make much sense. Other approaches do not take into account program semantics and can try to break dependencies without considering their relevance. Therefore, an approach is needed that is able to take into account the semantics of the dependencies (*e.g.,* by asking the developers) to propose a layered organization of a system.

In this paper, we present an approach, oZone, which proposes an organization in layers of a set of software entities, even in the presence of cyclic dependencies. This is done by ignoring some dependencies for breaking cycles. In addition, while our approach can be run automatically, it also supports human inputs and constraints so that system knowledge can be imparted in the creation of layers. Our contributions are: (i) an algorithm to identify dependencies that break the layered structure; (ii) a method to organize entities (even in presence of cycles) in multiple layers. Such approach takes also constraints set by the developer.

The benefits of our approach are two-fold: First, it helps removing undesired cyclic dependencies. Second, it helps grouping packages into layers. We validate our approach with a study of two large open-source software systems: the Moose and Pharo projects. Engineers of these two projects manually validated the results reported by our approach. Compared to other approaches, ours contains less false-positive and false-negative results when evaluated by experts, and it supports human inputs and constraints.

Although computing a layer organization or handling cyclic dependencies may work at different abstraction levels (*e.g.,* packages, classes), in this paper we focus on packages. Thus, for explanation and experiment purposes, in this paper the term "entities" denotes packages.

The paper is organized as follows. Section 2 details the importance of layered architecture and the problem of layered organization computation in three existing approaches. Section 3 defines used terms and explains our approach and Section 4 proposes a user interface to interact with the tool and fine tune the layered organization. Section 5 presents the validation of the approach. Related work is presented in Section 6. Section 7 concludes the paper and presents the future work.

2. Layer Identification

2.1. Context

When a large system is well structured, its structure simplifies its evolution. One of the structures that ease evolution is the layered architecture. A layered architecture is an architecture where entities in a layer may access only entities in the layers below it. Figure 1 describes an example of such an architecture (the reader should ignore the dotted dependency from *Kernel* to *pack A* at this point as it would break a proper layered architecture). A layered architecture eases software evolution because changes can be limited to layers above, it also offers good properties of modifiability and portability [BBC+00]. Moreover, using a layered view on a software system is a common approach to understand it [DP09].

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 it. In Figure 1, if one omits the bold dependency from *pack E* to *Kernel* (and the dotted dependency), one gets a closed layering. In open layering, any lower layer may be used, as *pack E* using *Kernel* in the figure.

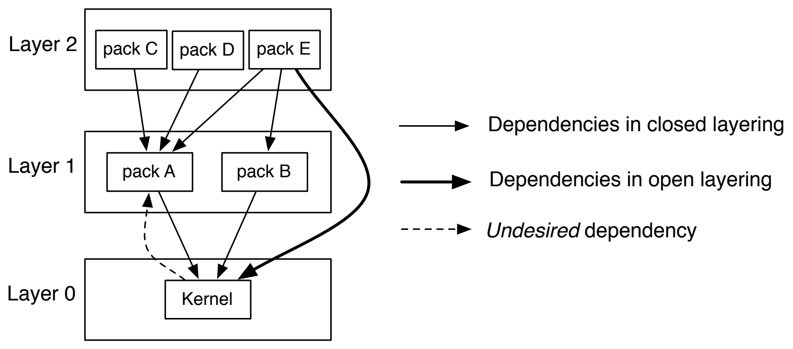


Figure 1: An example of a layered architecture. The dashed arrow represents an *undesired* dependency that would break the layer structure, the thick arrow represents a dependency allowed in *open* layering, but not in *closed* layering.

A basic algorithm to compute a layered architecture consists in putting in the lower layer all entities that do not depend on anything (this algorithm is presented more formally in Section 3.4). From this first layer, the architecture is built recursively: Entities depending on at least one entity in layer n, and possibly on entities in layers $\leq n$, are placed in layer $n + 1$. Note that, typically, in software systems not explicitly conceived with a layered architecture in mind, entities from layer $n + 1$ will also depend on entities from layers below n, which means the result will be an open layering. Closed layering is useful for a clean and well encapsulated system, but can only result of an explicit effort to organize the system according to that principle.

This basic algorithm can usually not be applied because of problems related to dependencies between entities. As illustrated by the dashed arrow in Figure 1, cyclic dependencies between the entities being layered have the potential to either break the layer structure, or to impose to merge layers (in the figure: *Layer 0* and *Layer 1*). In extreme cases, a cycle can include more than half the packages of a system (*e.g.,* ArgoUML or JEdit).

Cyclic dependencies are a known architectural problem. For example, it is specifically targeted by the Acyclic Dependencies Principle, defined by Martin [Mar00], stating that the dependency graph between packages should be a directed acyclic graph: there should not be any cyclic dependency between packages. Unfortunately, legacy and/or large systems often present structures that do not respect this principle [LDDB09]. Consequently, identifying layers in a large system is not trivial. Some approaches support layer creation in the presence of cyclic dependencies. We will now review these approaches and their limitations.

2.2. *Limitation of existing approaches*

Three main approaches try to extract a layered structure from package dependencies. They are incarnated in the tools: Lattix [SJSJ05], NDepend[1] and Structure101[2] [ELS93]. We now present these approaches and their limitations for creating a layered architecture.

We use the graph from Figure 2 to illustrate how these approaches work and what their limitations are. In the figure, the entities to gather in layer (*e.g.,* packages) are represented by nodes. In this example, we consider that the nodes are packages, but other entities (*e.g.,* classes) would lead to the same conclusions. Dependencies between entities are represented by edges. Edges may be weighted, for example in the case of packages, the weight of a dependency from one package to the other could be the number of classes in the first package that depend on classes in the second package.

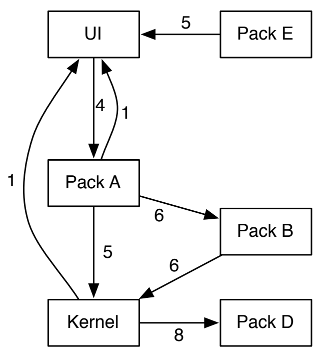


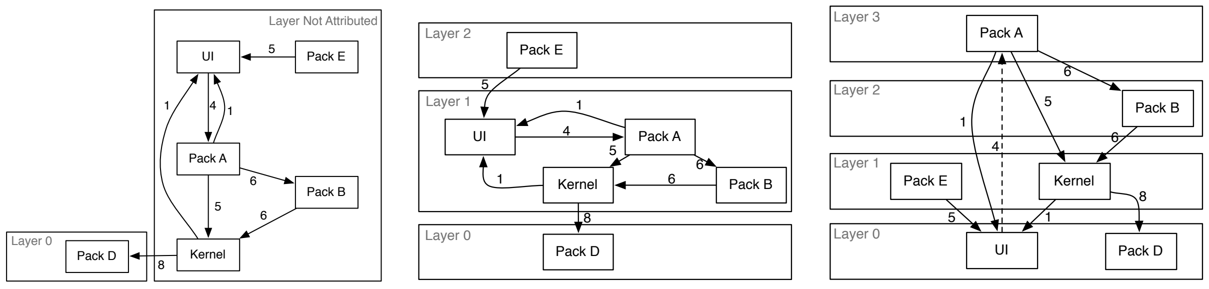
Figure 2: An example of cycles between packages.

NDepend. NDepend is a tool that uses software metrics, dependency structural matrix and simple visualizations to support software engineers. It considers a layered structure as an acyclic structure. When there are cyclic dependencies between entities, these entities are excluded from the layer representation and structure. Entities that depend, directly or indirectly, on entities in a cycle are also excluded from the layered structure. For the excluded entities, a special container, labeled “not attributed”, is created as illustrated in Figure 3(a).

The main problem of this approach is that, if a cycle involves a core entity (*e.g., Kernel* in the figure), most of the other entities will end up outside the layered organization.

[1] http://www.ndepend.com

[2] http://www.headwaysoftware.com/products/structure101/



(a) NDepend's Algorithm: Cycles out (b) Lattix's Algorithm: Cycles in one layer (c) MFAS Algorithm

Figure 3: Layered organization obtained by different strategies when applied to the system described in Figure 2.

Lattix. The layer extraction approach used in Lattix is the concrete implementation of the work of Sangal *et al.* [SJSJ05]. This work considers a cycle as a feature, not as a modularization problem. It proposes to group each cycle in a separate layer. Packages not involved in cyclical dependencies are placed in other layers either above or below the layer for each cycle, depending on whether they provide or require services to/from them. Figure 3(b) shows the layers discovered with Lattix considering the example in Figure 2. As the tool creates a single layer for all the entities in the same cycle, most of them are placed in the same layer.

Problems in this approach are twofold: (i) if cycles exist between entities that should be in different layers, they will be grouped in the same layer; and (ii) it is difficult to differentiate a layer built from a cycle and one built from dependencies to a lower layer.

Minimum Feedback Arc Set. The first two approaches deal with cycles in their entirety, either to put them aside or to put them in their own layer. We now see an approach that tries to remove the cycles to be able to create layers.

In graph theory, a feedback arc set is a set of edges which, when removed from the graph, leaves an acyclic graph. The *Minimum Feedback Arc Set* (MFAS) is the smallest such set. For the problem of extracting a layered architecture from a graph of dependencies, finding and eliminating the Minimum Feedback Arc Set is a possible heuristic. In our example, the MFAS is the singleton containing the dependency from *UI* to *Pack A*. Selecting this edge for removal leaves us with an acyclic graph that produces the layered structure in Figure 3(c) when layers are built following the process explained before. All cases are not that easy, there are cases where multiple dependencies of the same size exist. When this happens tools such as Structure101 rely on dependencies weights, choosing the MFAS with the minimum weight.

This approach can produce good results because it guarantees to make minimal modifications to the software structure to break cycles. However, it does not take into account the semantic of the software structure: what the entities mean, why they have these dependencies. Optimizing a graph is not equivalent to identifying the layered architecture of an existing software system.

In our example, the resulting organization, see Figure 3(c), places user interface (*UI* package) at the bottom of the layer organization, and the *Kernel* package in a higher layer, which does not fit common understanding in software engineering. Cycles should be broken such that the resulting organization would capture the software engineers' understanding of the system. This implies interacting with the software engineers, as the tool cannot have a sufficiently deep understanding of the system by itself.

3. Our solution: Detecting dependencies hampering layer creation

In this section, we present an algorithm to build a layered organization of a system in the presence of cyclic dependencies. As opposed to the existing solutions presented in the preceding section, we want a solution that can (i) propose to break cyclic dependencies so as to be able to place the different entities in a cycle in different layers; (ii) do so considering more information than just the weight of the dependencies; and finally (iii) allow the user to judge the proposed modifications and possibly give instructions on whether they are well-founded or not. The approach, called oZone, tries to find the best dependencies to remove so as to break cycles. The resulting graph of dependencies may then be organized in layers, following the simple algorithm proposed in Section 3.4.

This section is organized as follows: First, we present a few definitions necessary to explain our solution; then we present the oZone approach to find dependencies to break; finally we discuss how user input can be introduced in the process.

3.1. Definitions

Package. A package is a unit of reuse and deployment, it gathers together classes in a coherent unit. A package is built, tested, and released as a whole as soon as one of its classes is changed, or used elsewhere [Mar00]. A good organization of classes into identifiable and collaborating packages eases the understanding, maintenance, test and evolution of software systems [DK76, Mye78, You79, Pre94, PN06]. It is well accepted that packages should form layered structures [BBC+00, DDN02].

Concretely, in Smalltalk, packages group class and method definitions. They serve as units of loading. In Java, packages match directories that group class files. They can include other packages, this matches to the directory inclusion structure. Our importer considers packages as explicitly defined by developers using the resources of the language.

Package dependency. Package dependencies stem from references at the level of classes: package A depends on package B if a class within A refers to a class within B. A dependency from package A to package B is the union of all static dependencies from classes in package A to classes in package B. Consequently, a package dependency can be weighted with the number of its class relationships. The following kinds of dependencies are considered: method invocation, class access or reference, class inheritance, and class extension.

- *Method invocation.* There is a method invocation from class A to class B if there is at least one method in class A invoking one method of class B. In dynamic typed languages (*e.g.,* Smalltalk), one often cannot statically determine the class of the invoked method. Our strategy consists in resolving candidate classes, *i.e.,* every class within which there is a method that has the invoked method signature. In real cases, invocations commonly have only one candidate, *i.e.,* many method signatures are unique in a system.

- *Class access and reference.* There is a class access going from class A to class B if class B is explicitly used in class A code as an instance type, a class variable, or a variable/parameter type.

- *Class inheritance.* There is a class inheritance dependency between class A and class B if class A is a subclass of class B.

- *Class extension.* A class extension is a method defined in a package, for which the class is defined in a different package [BDN05]. Class extensions exist in Smalltalk, CLOS, Ruby, Python, Objective-C and C#. They offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. They support the layering of applications by grouping with a package its extensions to classes in other packages. AspectJ inter-type declarations offer a similar mechanism.

The definition of a package dependency has multiple implications that we want to highlight. First, global variables and exceptions are like a class reference because a global variable has a type and an exception is a class. Second, when a dependency points to an external library, our system does not consider it. By construction, if the library is external to the analyzed system, it is outside the scope of the current system and does not impact the layer structure of the system. Finally, the parameter passing is considered as a class reference for statically typed languages like Java and C++. For dynamically typed languages, the type cannot be found and the dependency cannot be built in our system.

Cycle. A cycle is a chain of dependencies between two or more packages such that there is a directed path which comes back to its origin (each package in the cycle depends on itself). We distinguish two kinds of cycles:

- *Direct cycle*. It represents a cycle between two packages. In Figure 4, *UI* and *Pack A* form a direct cycle.

- *Indirect cycle*. It represents a cycle between more than two packages. In Figure 4, *UI*, *Pack A*, and *Kernel* form an indirect cycle.

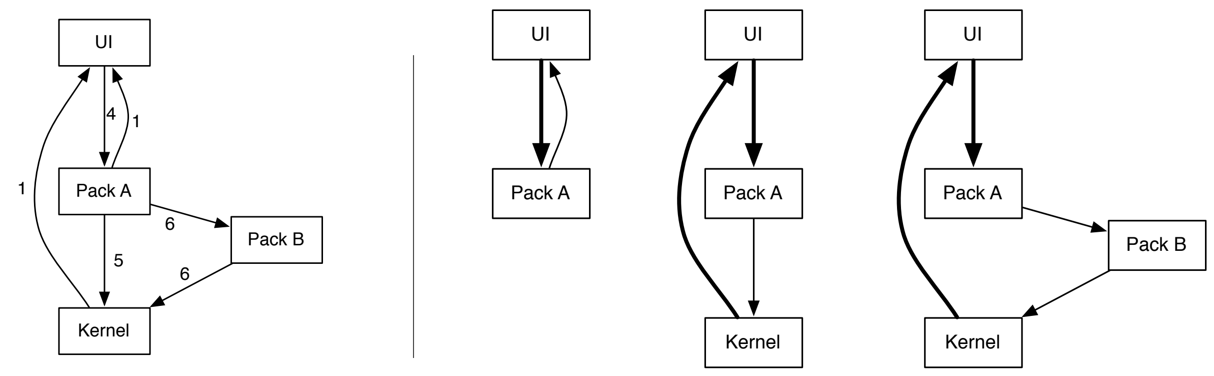


Figure 4: (left) Sample graph representing a SCC. (right) Sample graph decomposed in three minimal cycles. Bold edges are the shared dependencies.

Minimal cycle. A minimal cycle is a cycle with no node appearing more than once. In graph theory, it is also named a simple cycle. It is the minimal path from a node to return to it. In Figure 4, *UI→PackA→UI*, *Kernel→UI→PackA→Kernel* and *PackB→Kernel→UI→PackA→PackB* are three different minimal cycles. *PackA→UI→PackA→Kernel→PackA* is not a *minimal* cycle because *PackA* is present inside of it as well as at both ends. It can be reconstructed with the two minimal cycles *UI→PackA→UI* and *Kernel→UI→PackA→Kernel*. To retrieve minimal cycles, some algorithms exist as [Tie70, Wei72]. We use the algorithm proposed by Falleri et al. [FDL$^+$11], which is the most recent and the most efficient.

Shared dependency. A dependency is shared if it is present in at least two minimal cycles. In Figure 4 (right), the edge between *UI* and *PackA* is shared by three minimal cycles. The edge between *Kernel* and *UI* is shared by two minimal cycles.

Strongly Connected Components. A SCC in a graph is the maximal set of nodes that depend on each other. It means for each node of a SCC, there is a path within the SCC that comes back to this node. A SCC contains one or more cycles, specifically, if there is a shared dependency, the SCC will include all the nodes of the cycles that share this dependency. In Figure 4 (left), all nodes are in a single SCC. We use the Tarjan SCC algorithm [Tar72] to detect SCCs in a graph of dependencies.

Undesired dependency. We assume that in legacy software some dependencies between packages fit the intended architecture of the system whereas others don't. The later are the undesired dependencies. They may occur from an incomplete understanding of the intended architecture, from coding error, or from the impossibility of doing otherwise without refactoring the source code to create a new architecture.

Layer-breaking dependency. A dependency which (i) is undesired, (ii) belongs to a cycle, and (iii) because of this cycle impedes to find a layered organization of the system.

3.2. *Intuition to find* layer-breaking *dependencies*

As we showed, the problem in building a layered architecture has to do with how to take into account cycles between packages. Figure 5 presents the layered organization that we would like to see for the package structure of our example (see Figure 2). We propose heuristics that are based on observations and experiments from our previous work [LDDB09, LDD09]. The cornerstone of our approach is to find *layer-breaking* dependencies. Ignoring them allows one to build a layered architecture more easily, and removing them can be beneficial to the general organization of the system. Our approach is based on two intuitions for finding dependencies that, once removed (we call them *removable*), will allow to generating a layered structure. We defined heuristics that try to ensure that the *removable* dependencies proposed by our approach are truly *layer-breaking* dependencies.

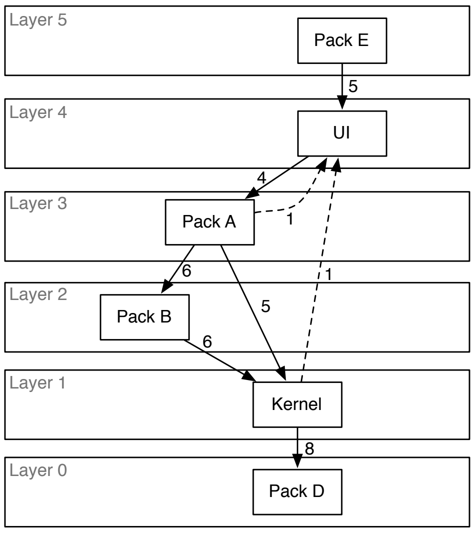


Figure 5: The expected layered organization.

Heuristic 1: Direct cycles. In a cycle between two packages (a *direct cycle*), we hypothesize that the lightest dependency might be inadvertently introduced in the system and might be a design defect or a programming error. Moreover, we hypothesize that lightest dependencies require the least amount of work to be eliminated.

These hypotheses might not always hold true, but software engineers with knowledge on the system are given the opportunity to overrule wrong decisions made by the tool (see Section 3.5).

Heuristic 2: Shared Dependencies. Our second intuition is based on the occurrence of *shared dependencies* in *minimal cycles* involving three or more packages (indirect cycles): A shared dependency is a suitable candidate to be removed to break cyclic dependencies between packages.

We hypothesize that shared dependencies have a strong impact on the system structure because removing them may potentially break several cycles at the same time. Again, if this heuristic proves wrong, the user may overrule the decision afterwards.

Therefore, the general idea of our approach is to decompose package cycles into *minimal* cycles. Among these cycles, first we select the direct cycles and in each direct cycle we mark the weakest dependency as *removable*, that is to say a dependency to be ignored by the layering algorithm. Then, among the rest of the cycles (*i.e.,* indirect cycles), we mark *shared* dependencies as *removable* dependencies. Then we build layers ignoring the removable dependencies: as mentioned previously we start with packages not depending on any others, group them in layer 0, and successfully build the others.

There are two reasons to begin with direct cycles: (i) when a direct cycle is addressed, it could have an impact on other indirect cycles that include it; (ii) addressing a direct cycle is somehow simpler because there are only two solutions: cutting one or the other of the two package dependencies.

For the packages of Figure 5, *removable* dependencies are marked as follows:

1. We decompose cycles among packages into minimal cycles. The cycles among packages *PackA*, *PackB*, *Kernel*, *UI* are decomposed into three minimal cycles: *UI*→*PackA*→*UI*, *UI*→*PackA*→*Kernel*→*UI*, and *UI*→*PackA*→*PackB*→*Kernel*→*UI*, already shown in Figure 4 (right).

2. To eliminate a direct cycle, the algorithm considers the weight of each dependency in the cycle and marks the lightest as *removable*. The cycle *UI*→*PackA*→*UI* is the only direct cycle so we start with it. Applying our first heuristic, we mark as removable the lightest dependency *PackA*→*UI*.

3. When all direct cycles are removed, the algorithm considers shared dependencies, it marks as *removable* the one shared the most, because this is the one with the highest impact on minimal cycles. If multiple dependencies are shared by the same number of cycles, the algorithm selects the lightest of them in terms of dependency weight. We repeat this action as long as there are cycles.

   In Figure 5, there are two *minimal* cycles where the dependencies *Kernel→UI* and *UI→Pack A* appear. Therefore, we select *Kernel→UI* (*e.g.,* the lightest one) to break the cycle among the packages. Once these *removable* dependencies are marked (dotted in Figure 5), all the package cycles are addressed and we can compute the package organization illustrated in Figure 5. Again, the software engineer can then decide that an undesirable dependency is not correct and rerun the algorithm.

3.3. The algorithm of OZONE

Based on the definitions and heuristics previously explained, we present an algorithm to identify *removable* edges in a package dependency graph. In this algorithm, the term *removable* is used (i) to remove an edge from the graph to make it acyclic and (ii) to tag the dependency so that its status can be understood by the engineer evaluating the layers and dependencies.

```
1:  Model::getRemovedEdges(Graph graph): Collection {
2:      graph.computeSharedEdges()
3:      for (Cycle cycle: computeDirectCycles()) {
4:          if (cycle.edgeOne.weight() > cycle.edgeTwo.weight() * 3)
5:              graph.remove(cycle.edgeTwo)
6:          elseif (cycle.edgeTwo.weight() > cycle.edgeOne.weight() * 3)
7:              graph.remove(cycle.edgeOne)
8:          else
9:              graph.remove(max(cycle.edgeOne.sharedEdges(), cycle.edgeTwo.sharedEdges()))
10              or (graph.remove(min(cycle.edgeOne.weight(), cycle.edgeTwo.weight())))
11:             or (graph.remove(cycle.edgeOne))
12:     }
13:     while (computeSCC().notEmpty()) {
14:         graph.computeSharedEdges()
15:         graph.remove(max(cycle.edges.sharedEdges()))
16:     }
17:     return graph.removedEdges
18: }
```

The algorithm works as follows: first the *minimal* cycles are created and shared dependencies are computed on line 2. Then, from line 3 to line 12, direct cycles are removed from the graph. It checks for large differences between the two edges of the direct cycle (using a ratio of 1/3) and removes the lightest (lines 4 to 7). If the difference is not important enough, the algorithm checks shared dependencies and removes the most shared (line 9). If the two edges have the same number of shared dependencies, it removes the lightest edge (line 10). If none of these conditions is satisfied, the algorithm removes the first edge (line 11). 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 expected or not (see Section 3.5). Then from lines 13 to 16, the algorithm removes other cycles by removing shared dependencies. It computes minimal cycles (line 14) and removes the most shared dependency (line 15). If there are multiple dependencies with the same shared number, it selects the one with the lower weight. The algorithm returns a collection of *removable* dependencies.

The computation of *minimal* cycles is based on an algorithm presented in [FDL+11], and we do not detail it here. This algorithm returns a collection of minimal cycles in a graph, from which we compute the *shared* dependencies as follows:

```
1:  Graph::computeSharedEdges(): Collection {
2:      graph.computeMinimalCycles()
3:     for (Cycle cycle: graph.computeMinimalCycles()) {
4:         for (Edge edge: cycle.edges()) {
5:             edge.setShared(edges.getShared() + 1)
6:         }
7:     }
8: }
```

Line 2 calls the minimal cycles algorithm. Then for each dependency in each minimal cycle (line 3 to 7), we increment a counter associated to the dependency.

3.4. Building layers from an acyclic graph

When the previous algorithm returns *removable* dependencies, we can ignore these dependencies to obtain an acyclic graph. With this graph, we can build a layered organization of the packages. These specific dependencies are however, later presented to the software engineer for further analysis. The layering algorithm is the same as intuitively presented at the end of Section 2.1:

```
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: }
```

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 (line 5) is filled by putting into it packages without any outgoing dependencies to packages contained in N (lines 6 and 7). Finally, it removes the selected nodes from N (line 8), and loops until N is empty.

3.5. Manually defining constraints

The algorithm may provide results that do not completely match the expectations of engineers. Automatic computation based on our heuristics provides a first shape of the system. From this, an engineer should be able to impart his knowledge and impose constraints on cyclic dependencies removal. He can evaluate each dependency in the system and introduce constraints. Then the tool recomputes the layer organization according to the provided constraints.

For this purpose, we design four possible marks for a dependency: (i) *expected* for dependencies that fit into the software engineer's understanding of the system architecture; (ii) *undesired* for dependencies that do not fit the software engineer's understanding of the system architecture; (iii) *removable* for dependencies that the algorithm propose to remove; and, (iv) *not flagged* for dependencies that the algorithm would leave untouched.

These constraints add a new dimension to the algorithm. Figure 6 shows the possible states of a dependency after a first run of the algorithm; then an evaluation-and-correction phase by the software engineer; and finally the possible impact of these corrections after a second run of the algorithm. First, the algorithm marks as *removable* the dependencies it proposes to break. By default, all others are marked *not flagged* (i.e. valid). The user may change the mark of any dependency to *expected* or *undesired* to clarify his understanding. He may also leave the algorithm's mark untouched. In a new run, the algorithm will not change the tagging by the user and will only consider the dependencies that are *removable* or *not flagged*. It is a new fresh run and these dependencies are reevaluated independently of their previous mark. For example, if the user marked some dependency in a cycle as *undesired*, this might end up breaking the cycle and other dependencies in the cycle that were initially marked *removable* by the algorithm could now be considered *not flagged*. On the contrary, if the user puts an *expected* mark on a dependency, this could force the

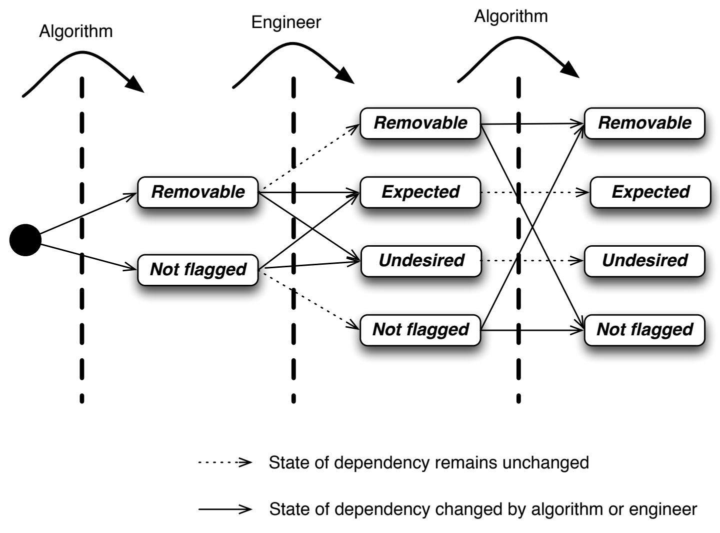


Figure 6: Possible evolution of package dependencies states.

algorithm to mark as *removable* another dependency in the cycle that was initially *not flagged*. If all dependencies in a cycle are marked as *expected*, this cycle cannot be removed. In this case, all involved packages are put together in a single layer. We only show one iteration in the figure, but after the second run of the algorithm, the user could again change some dependencies, until he obtains a satisfying layer structure.

4. An interactive browser to build layers

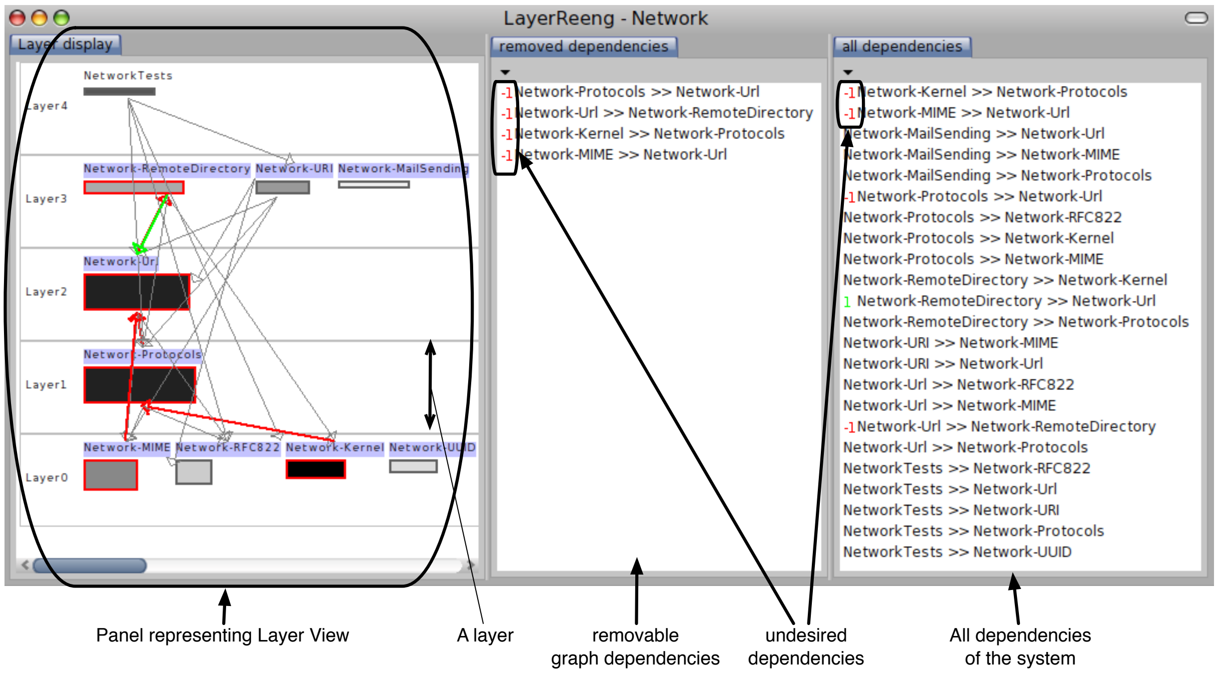
We implemented a prototype tool to visualize and modify the results of the algorithm. This prototype allows users to understand the status of the dependencies, change it and recompute the layers. This prototype is implemented on top of the Moose software analysis platform [DGLD05] and it is based on the FAMIX language independent source code metamodel [DTD01]. Moose provides importers for Java, C#, C++ and Smalltalk. We experimented with two Smalltalk applications.

The user interface of the tool (Figure 7) is composed of three main panels: a visualization of layers with polymetric views [LD03] (left), a list of layer-breaking dependencies (center), and a list of all dependencies (right). The layer-breaking dependencies are those marked as *removable*[3] or *undesired*. They are ignored in the computation of layers.

We build the layer visualization to show our main concerns: relations between packages, packages forming *Strongly Connected Components* (SCC), and some convenient metrics about the size of packages to help the software engineer develop a basic understanding of the system.

- In Figure 7 (left part), a Layer is a row with colored boxes inside. The bottom box is Layer 0.
- A Package is represented 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 of the box have meanings (see also Figure 8):
  - The height represents the number of clients of the package (*i.e.,* those depending on it). Taller packages have more clients, so one would expect to find them in lower layers.
  - The width represents the number of providers of the package (*i.e.,* those it depends on). Wider packages have more dependencies, one would expect to find them in higher layers.

[3]In the prototype, flags are represented by numbers: -1=*undesired*; 1=*expected*; 0=*removable*; and no number=*not flagged*



removed dependencies

```
-1 Network-Protocols >> Network-Url
-1 Network-Url >> Network-RemoteDirectory
-1 Network-Kernel >> Network-Protocols
-1 Network-MIME >> Network-Url
```

all dependencies

```
-1 Network-Kernel >> Network-Protocols
-1 Network-MIME >> Network-Url
Network-MailSending >> Network-Url
Network-MailSending >> Network-MIME
Network-MailSending >> Network-Protocols
-1 Network-Protocols >> Network-Url
Network-Protocols >> Network-RFC822
Network-Protocols >> Network-Kernel
Network-Protocols >> Network-MIME
Network-RemoteDirectory >> Network-Kernel
1 Network-RemoteDirectory >> Network-Url
Network-RemoteDirectory >> Network-Protocols
Network-URI >> Network-MIME
Network-URI >> Network-Url
Network-Url >> Network-RFC822
Network-Url >> Network-MIME
-1 Network-Url >> Network-RemoteDirectory
Network-Url >> Network-Protocols
NetworkTests >> Network-RFC822
NetworkTests >> Network-Url
NetworkTests >> Network-URI
NetworkTests >> Network-Protocols
NetworkTests >> Network-UUID
```

Figure 7: UI to build layered view based on constraints.

  - The fill color represents the number of classes in the package, the darker a package, the more classes it contains.
  - The border color represents *Strongly Connected Components* (SCCs). If the color is gray, the package is not part of a SCC. If it is another color, the package is in a SCC with all other packages with the same border color. In the figure, there is only one SCC, which is marked by red border (showing as dark gray).

- A dependency is represented by an arrow from depending package (client) to depended one (provider). A dependency can take one of four colors (see Figure 8): red (showing as dark gray) for an *undesired* dependency (that the user wants to exclude), green for an *expected* dependency (that the user wants to keep), black for a *removable* dependency (that the tool suggests to exclude), and light gray for a *not flagged* dependency (not flagged by the algorithm nor the user).

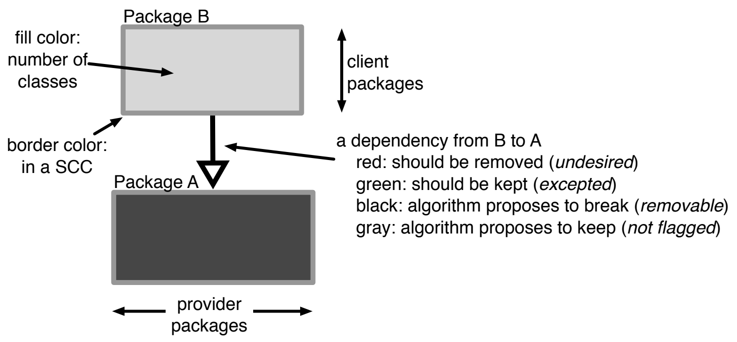

Package B
fill color: number of classes
client packages
border color: in a SCC
a dependency from B to A
red: should be removed (*undesired*)
green: should be kept (*excepted*)
black: algorithm proposes to break (*removable*)
gray: algorithm proposes to keep (*not flagged*)
Package A
provider packages

Figure 8: The polymetric view used in Layer View

This kind of information allows the engineer to understand the dependencies of the packages in the system. It is also possible to interact with the tool to add manual constraints on a dependency.

- On the two lists of the UI (center and right), users can flag dependencies as *expected* or *undesired*. When a dependency is flagged manually, the algorithm recomputes the layered architecture and the UI (left part) is updated.
- On the layered view (left), users may select a package and force it to be placed in any layer. In response, the tool flags as *undesired* all dependencies that would contradict this decision (*i.e.,* dependencies coming from lower layers), and recomputes the layered architecture.

The user interface was used for the first part of the validation of our approach.

5. Validation

We performed three distinct experiments and validation studies. The first study confirms that many of the dependencies selected by our heuristics are *layer-breaking dependencies* and, as a result, the tool can build a layer organization corresponding to the intended structure of the software. The second study shows that our algorithm can make use of user input to improve its results. The third study compares our algorithm with the *Minimum Feedback Arc Set* (MFAS) algorithm.

We performed the first experiment on two open-source systems: the Moose software analysis platform [NDG05] and the Pharo Smalltalk development environment [BDN+09]. The second experiment considers only Moose and the third one Pharo. We selected these two projects because they are large, contain realistic package dependencies, and are open-source and available to other researchers. In addition, both systems are more than 10 years old. Most importantly, we selected them because we could get feedback on our results from members of each project community.

In the three experiments, we compare the results of our algorithm (and MFAS) to a manual evaluation of all dependencies by software engineers of both projects. In both case, software engineers, independent of the authors of this article, manually checked and qualified all the package dependencies of the systems. The engineers were not confronted to the results of our tools before the end of this evaluation. Characterization of the two systems is proposed in Table 1

Table 1: Characteristics of Moose 4.0 beta4 and Pharo 1.1.

	Moose	Pharo
Number of classes	594	1558
Number of packages	33	115
Number of packages in cycles	14	68
Number of SCC	1	1
Number of direct cycles	10	98
Number of package dependencies	106	1328
Number of package dependencies involved in SCC	56	922
Number of undesired dependencies in SCC	14	222

From this, we could compute recall and precision for our algorithm and MFAS. The precision and the recall are computed from the number of true and false positives proposed by the algorithms. A true positive is a dependency marked *removable* (by the algorithm evaluated) and *undesired* (by the human expert), a false positive is a dependency marked *removable* (by the algorithm evaluated) and *expected* (by the human expert), and a false negative in SCC is a dependency member of a Strongly Connected Component marked *not flagged* (by the algorithm evaluated) and *undesired* (by the human expert). tp, fp, and fn_{SCC} are the total number of (respectively) true positives, false positives, and false negatives in SCC. The precision of an algorithm is computed as:

$$Precision = \frac{tp}{tp + fp}$$

and indicates the proportion of dependencies *removable* that are undesirable for the developer. The closer to 1, the better. Recall is defined as:

$$Recall = \frac{tp}{tp + fn_{SCC}}$$

and indicates the proportion of layer-breaking dependencies (undesirable for the developer) found by the algorithm. Best is closer to 1.

5.1. Study one: Finding the dependencies to remove

We performed a first study to validate the relevance of the approach without any human input. The question is: Does the algorithm propose to break the same dependencies as a software engineer that knows the system well?

5.1.1. Protocol

The case study was realized on beta 4 of Moose 4.0. Moose contains 33 packages and 106 dependencies among these packages. This version was chosen because it is known to be poorly modularized, and it contains many cyclic dependencies. We provide some metrics for Moose in Table 1. It contains one Strongly Connected Component (SCC) with 14 packages and 56 dependencies. In this SCC, there are 10 direct cycles involving 14 packages. Note that the 10 direct cycles represent 20 dependencies among the 56 of the SCC (a direct cycle is composed of packages with 2 dependencies).

A developer from the Moose team (independent from this research) evaluated all the 106 package dependencies of the system giving them two possible values: *expected* and *undesired*. 14 dependencies were considered *undesired*. We then ran our algorithms on the system and compared the dependencies that the tool proposes to remove (*i.e., removable*) to the one the engineer considered inadequate. Results are provided in Table 3.

Table 2: Dependencies to remove as proposed by our tool on the Moose system. Discrepancies with the user's evaluation (last five) are discussed in text

Removable dependencies	User evaluation
DSMCore→DSMCycleTable	undesired
Fame→Moose-Core	undesired
Famix-Core→Famix-Implementation	undesired
Famix-Extensions→Moose-Finder	undesired
Famix-Extensions→DSMCore	undesired
Famix-Smalltalk→Famix-Extensions	undesired
Glamour-Browsers→Glamour-Scripting	undesired
Glamour-Helpers→Glamour-Core	undesired
Moose-Core→Famix-Extensions	undesired
Moose-Core→Famix-Implementation	undesired
Moose-Finder→Moose-Wizard	expected
Moose-SmalltalkImporter→Famix-Implementation	expected
Moose-SmalltalkImporter→MooseCore	expected
Moose-GenericImporter→Moose-Core	expected
Famix-Core→Moose-Core	expected

5.1.2. Results and discussion

For Moose, 15 dependencies were marked *removable* in a first run of the algorithm (see Table 3). Ten out of these 15 are considered *undesired* by the engineer. The precision of our tool, on this example, is 66% and the recall is 71%. For Pharo, the precision is similar (64%), but the recall is lower (44%). Results for Pharo are discussed in Section 5.3.

We now give a closer look to the 5 false-positive dependencies for Moose. They are due to two issues:

- First, the two dependencies *Moose-Finder→Moose-Wizard* and *Moose-SmalltalkImporter→Famix-Implementation* belong to direct cycles for which both dependencies have similar weights. Selection of which dependency to break could not be based on the weight difference. Furthermore, none of the dependencies in these two direct cycles is shared. Therefore, the algorithm has no reason for choosing one dependency over the other, and had to choose "randomly" (see Section 3.3). To correct this point, the tool would need more information about the system. This is achieved with the possibility for the maintainer to indicate the *undesired* dependencies.

- Second, the last three false-positive results going to Moose-Core (*Famix-Core→Moose-Core*, *Moose-Smalltalk-Importer→MooseCore* and *Moose-GenericImporter→Moose-Core*) are related to a choice of the algorithm to remove the dependency *Famix-Core→Moose-Core* instead of *Moose-Core→Famix-Core*. This is another case of a difficult choice between the two dependencies of a direct cycle in absence of sufficient weight difference. In this case, *Famix-Core→Moose-Core* is shared one more time than *Moose-Core→Famix-Core* and therefore was wrongly chosen. This is a case where a heuristic based on information from the graph of dependencies shows its limit and where knowledge of the system is required. Again, this knowledge can be provided by the user.

5.2. Study two: User input influence

The second study validates how well the approach can consider user input. The question is: Does the algorithm find better solution with user constraints?

5.2.1. Protocol

This experiment can be seen as a follow-up on the previous one with only the Moose project. In the previous experiment, the tool proposes a layer organization with 11 layers shown in Figure 9 (the rectangle are the packages, each row is a layer). It highlights particularly the main problem revealed by previous results: Moose-Core is too high in the layer organization, due to an incorrect decision to break dependency *Famix-Core→Moose-Core*.

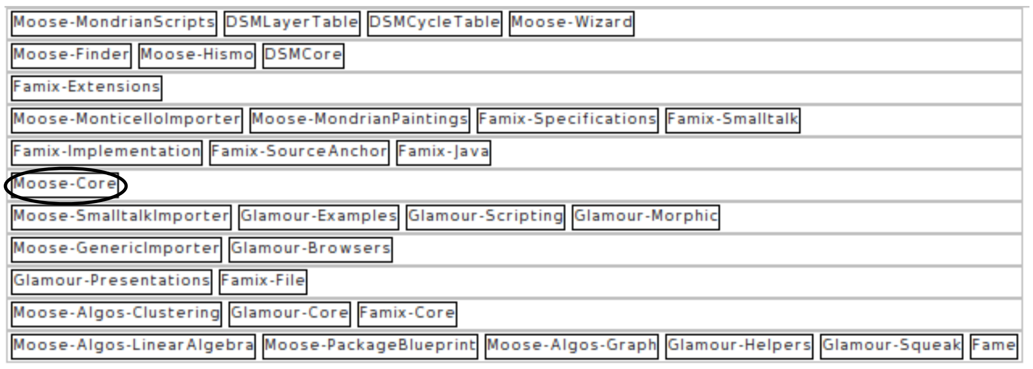


Figure 9: Moose packages layer organization proposed by out tool in a first round. Moose-Core should be in a lower layer.

Based on the manual evaluation of dependencies by the Moose engineer, we manually set as *undesired* the 10 correct propositions of our algorithm and set as *expected* the five false-positive propositions. This corresponds to what a real user might do when using our tool on a system he knows. Note that this means that the four remaining dependencies considered *undesired* by the independent expert and not found by the algorithm, were not explicitly marked as such. After this input from the user, we ran the tool again and compared the new result obtained to the previous one. We expect that the results will be better now that we gave additional information on the system to the tool. Specifically, we expect it to find the four layer breaking dependencies that we left *not flagged*.

Table 3: Automatically corrected *removable dependencies* after adding manual constraints on the Moose system.

Removable dependencies	User evaluation
Moose-Core→Famix-Core	undesired
Moose-Core→Moose-GenericImporter	undesired
Moose-Core→Moose-SmalltalkImporter	undesired
Moose-Wizard→Moose-Finder	undesired

5.2.2. *Results and discussion*

After the second run, the algorithm correctly marked as *removable* the four missing dependencies, shown in Table 4. So it achieves perfect precision and recall.

In this new result, the layer organization has 8 layers (Figure 10). We can see that Moose-Core is now on a lower layer that in Figure 9. It is considered correctly placed now.

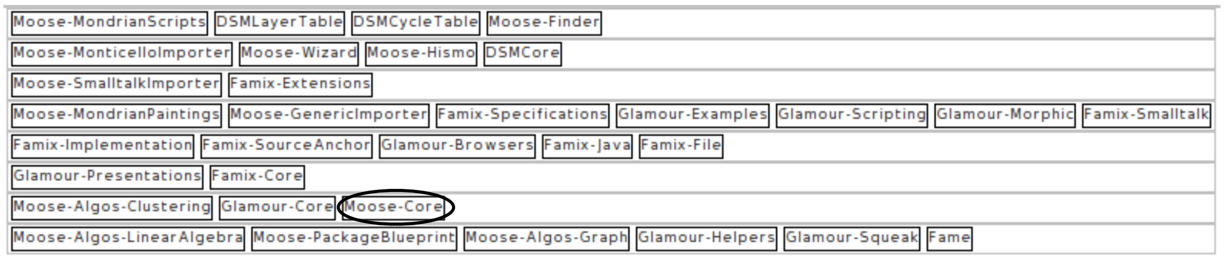


Figure 10: Moose packages layer organization proposed after providing manual constraints.

This study shows that our algorithm was able to improve its results after receiving some input from the user: It achieved a better precision and proposed a layered organization that better fits the expectation of the expert.

5.3. *Study three: Comparative Study with* Minimum Feedback Arc Set *(MFAS)*

In this section, we compare the results of our algorithm to those of the MFAS algorithm proposed in [ELS93].

5.3.1. *Protocol*

We perform this comparison on Pharo 1.1. looking at the precision and recall of both algorithms. Pharo is an open-source Smalltalk-inspired environment. Table 1 summarizes some characteristics of Pharo. It has 1558 classes in 115 packages in the version studied, 1.1. Pharo contains one Strongly Connected Component (SCC) composed of 68 packages and 922 dependencies. In this SCC, there are 98 direct cycles involving 68 packages (one package may be involved in several direct cycles). The 98 direct cycles have 196 dependencies among the 922 involved in the SCC.

We asked engineers from the Pharo community to evaluate all the 1328 dependencies. To help them in this long and tedious task, we developed a little tool to support their evaluation of the package dependencies. Table 5, middle column, gives the proportion of dependencies evaluated by the Pharo experts. 20% of all the 1328 dependencies were considered *undesired*. Since both approaches evaluated (MFAS and our tool) propose to remove dependencies that belong to SCCs, we also give the percentage of undesired dependencies among the 922 SCC dependencies (right column). Only 25% of these dependencies are marked *undesired* by the developers. It implies that in a typical SCC, only one out of four dependencies is irrelevant.

5.3.2. *Results and discussion*

There are differences between oZone and MFAS results because MFAS analyzes the whole graph and removes the minimal number of dependencies, whereas oZone first removes the lightest in direct cycles, and second the shared dependencies in the rest of the cycles. We provide in Table 7 the number of dependencies that each algorithm proposes to remove. We also provide the union and the intersection of dependencies identified by each technique. Each of them

Table 4: Manual evaluation of dependencies in Pharo 1.1.

Expert evaluation	% of all dependencies	% of SCC dependencies
undesired	20%	25%
expected	78%	73%
unknown	2%	2%

found about 150 dependencies. There is no clear difference on the number of dependencies found. However, when computing the intersection we see that only 90 dependencies are common. It means that 60 of the 150 dependencies are unique to MFAS and 61 of the 151 dependencies are unique to oZone. We can conclude from these results that the two algorithms are proposing a significant set of non-overlapping dependencies as removable in the system. It suggests that heuristics from the two algorithms might be merged to try to produce better results.

Table 5: Number of dependencies *removable* with oZone and/or MFAS

Technique	Number of dependencies
MFAS	150
oZone	151
oZone ∩ MFAS	90
oZone ∪ MFAS	211

Table 8 presents the results of oZone and MFAS precision and recall. Whereas, 61% of the dependencies removed by MFAS were *undesired*, precision is 64% for our approach (better precision). In addition, we could identify 44% of the *undesired* dependencies in SCC, whereas MFAS finds only 41% of those (better recall). Hence, our approach performs better than MFAS on this example.

An added advantage is that our approach allows the reengineer to review the dependencies marked by the algorithm and give his feedback (as shown in Section 5.2).

Table 6: Comparison of the Accuracy: oZone and MFAS

Technique	Precision	Recall
MFAS	61%	41%
oZone	64%	44%
oZone ∩ MFAS	77%	31%
oZone ∪ MFAS	56%	53%

We finally compute the precision and the recall of the union and intersection of the two algorithms. The union of the techniques augments the number of *removable* dependencies and logically improves recall (53%) at the expense of precision (56%). On the contrary, taking only the dependencies identified by both techniques (intersection) reduces the size of the result (decrease in recall to 31%), choosing mostly interesting ones (improved precision to 77%).

From these results, we can make two conclusions: (i) The approaches are complementary. They select different dependencies, which used together provide a good precision. Using both heuristics could provide better results. (ii) Even combined, the two approaches do not select all the undesired dependencies. As explained, the goal of the two algorithms is to deal only with dependencies in cycles, dependencies judged uninteresting by the engineers, but not member of a cycle were not treated.

Although our approach produced better results, we believe it can be improved by adding semantic to the dependencies. In our approach, after removing direct cycles and shared dependencies, the algorithm ignores light dependencies. We could consider the nature of dependencies to improve this behavior. For example, we could treat differently dependencies arising from inheritance or method call, or consider where the dependency occurs, *e.g.,* type of a class attribute, type of a method argument, type of a local variable. In addition, the approach could be extended to take into account design patterns such as Model-View-Controller (MVC) that could provide semantic information. In this case, if a cycle is discovered between the view and the model, we should remove the dependency from model to view.

Another point to be taken into account for the improvement is the notion of sub-package (a sub-package is a package inside another package. This feature is available for example in Java). For now, they have no different treatment but we could consider cycles between sub-packages less important than a cycle between multiple different packages.

5.4. Threats to validity

We detail in this section the threats to validity related to the three previous studies.

- Construct validity: Each dependency was evaluated by only one expert. He may have made mistakes. In Moose, the probability of a wrong evaluation is low, because we performed a "counter-evaluation" when we manually analyzed part of the dependencies to discuss the results of the first study. In Pharo, there are more than 1300 dependencies evaluated, which required several hours of concentration. This threat is difficult to reduce because of the costs involved in manually evaluating the dependencies.

  Another threat is related to the size of the experiment. The number of studied software applications is too small to provide statistically significant results. Again, dealing with this threat involves costs that we could not sustain in this research.

  Manually evaluating every dependency in a given software system is a very long and costly task.

- Internal validity: We do not have causal relations examined in this study. So, there is no internal validity threat.

- External validity: The study should work on object-oriented language. However, in this study, only Smalltalk software applications were analyzed. Our analysis is based on the FAMIX language independent meta-model: the source code model extracted from Smalltalk code is similar to the one of Java and the extracted dependencies between packages are the same: class references, inheritances and accesses. To avoid this threat, we must study other object-oriented systems.

- Reliability: We identify two reliability threats: the tool construction and the dependencies assessment.

  The threat related to the tools can be due to the dependencies extractor and to the dependency validator. The threats for the extractor are small because of the large testbed and the use of this extractor by an active community since 1996. The threats to the validator are small too because it is a simple list that can be checked with values (undesired, expected, unknown).

  The threats related to the assessment is due to the validity of the human evaluation. We give to developers the sole instruction that an undesired dependency is a dependency that should not exist in the system. This definition has a semantic impact that can be interpreted differently by developers. This threat can be avoided by asking another developer to analyze the dependencies and to confront the results of each developer. It is difficult to reduce this threat as it takes several hours to analyze all dependencies.

6. Related Work

In this section, we present other approaches on the problem of package cycle identification and removal.

6.1. Heuristics

Some research work has been done on package dependencies and package layer computation. 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. The first heuristic is to consider packages in the same strongly connected component as a single package. The other heuristic selectively ignores some *undesirable* dependencies until no more cycles are detected. Thus, PASTA reports the *undesirable* dependencies which should be removed to break cycles. The *undesirable* dependencies are selected by computing their weights and selecting the minimal ones. Our approach considers one more parameter: we introduce *shared dependencies* and use it as a heuristic to remove package cycles.

In graph theory, a feedback arc set is a collection of edges which should be removed to obtain an acyclic graph. The minimum feedback arc set is the minimal collection of edges to remove to obtain an acyclic graph. This theoretical approach cannot be used for three main reasons: (i) It is a NP-complete problem (optimized by Kann [Kan92] to become APX-hard). Some approaches propose heuristics 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.

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 feedback arc set. However, no study was conducted to validate this approach for cycle removal. The dependencies it selects for removal could be *expected*, *i.e.,* valid given the system's architecture. In another study, Melton et al. [MT07a] propose an empirical study of cycles among classes. They use minimum feedback arc set to resolve cycles present in the software system. They indicate that it is crucial to take into account the semantic of the software architecture. For this purpose, the authors propose to add constraints to the minimum feedback arc set to keep inheritance relationship while breaking cycles. We also consider user input an essential feature to remove cycles. In our work, we include user validation to take into account the semantics of the program.

Mudpie [Vai04] is a tool to help the maintenance of software system by bringing out Strongly Connected Components and focusing on the dependencies in Strongly Connected Components. However, no algorithm is presented to break the cycles present and the tool relies on the developer's intuition to remove cycles.

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

Bunch [MM06] is a tool that uses hill-climbing clustering algorithms for automatic remodularization of software. It proposes to decompose and to show an architectural-view of the system structure based on classes and function calls. It helps maintainers understanding relations between classes. This tool ignores the package structure and does not provide the information we need to make a layered organization of a 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 where the classes with a high authority value are placed on the bottom layers and the classes with a high hub value are placed on the top layers. They propose a semi-automatic approach which allows the maintainer to manipulate the architecture and add its proper meaning of the system. This kind of algorithm could be applied at package level and compared with our algorithm.

Multiple works [ADSA09] exist to decompose a system by using genetic heuristics. Lutz [Lut01] proposes a hierarchical decomposition of a software system. It uses a genetic algorithm to find the best way to group classes of the system into coarse-grained packages. Our work is not in this domain, as our goal is to discover dependencies which break the system architecture, in particular, the layered organization of the system.

6.2. Visualization

Several approaches propose to recover software structure, visualize classes and files organizations [Vai04]. Only a few provide layered organization for packages and in particular consider cycles. Some help determine information on packages and their relationships, with visualizations and metrics [DPS$^+$07]. In these approaches, it is not easy to understand the place of a package in a system, particularly when large systems are analyzed. Others propose to recover software structure and visualize the organization of classes and files [MM06]. To understand the complexity of large object-oriented software systems and particularly the package structure, there are some visualization tools ([DGK06, DL05, BDL05, LSP05]). Package Blueprint ([DPS$^+$07]) shows the communications between packages; eDSM ([LDDB09]) and CycleTable ([LDD09]) highlight the cycle problems in a system. However, these approaches do not identify layers for packages present in a software system.

Dong and Godfrey [DG07] propose an approach to study dependencies between packages and to give a new meaning to packages with (i) 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.

Lungu et al. [LLG06] propose a collection of package patterns to help reengineers understand large software systems. They propose to recover architecture based on package information and an automatic process to recover defined patterns. Then they propose a user interface 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 information on a package and to propose more information about the breaking of a dependency, for example knowing that a package is autonomous is valuable information.

6.3. Tools

There are other tools like JDepend[4] or Classycles[5] which allow software engineers to see package dependencies and cycles. However, 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. NDepend 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. However, the tool does not support the removal of package cycles.

Deissenboeck et al. [DHHJ10] propose the tool named ConQAT that allows developers to define manually a policy for conformance assessment of the architecture. This tools highlights undesired dependencies between dependencies after the policy is defined. Our approach can be complementary to ConQAT in the policy defining step by proposing undesired dependencies between components.

Murphy et al. [MNS95, MNS01] propose a reflexion model that provides a better integration of the design in the implementation. The reflexion model approach supports the checking of architectural dependencies by comparing a high-level model (*i.e.,* static architecture) with information extracted from the source code. During the comparison between the source code and the architectural model, a violation is when a high-level dependency is not found in the code, called an absence, or when the source code produces an unexpected dependency in the high-level model, called a divergence. This work is complementary to ours: it needs a predefined high-level model for the comparison. It would be possible to implement our approach on top of a reflexion model to produce the architectural model.

7. Conclusion

Building a layer organization for software entities in the presence of cyclic dependencies is not trivial. Existing approaches either do not treat the problem of cycles or do not allow user constraints to input semantics of program. In this paper, we propose an algorithm to organize software entities having cyclic dependencies in layers. We compute *layer-breaking dependencies* with our approach and provide a user interface to add manual constraints to correct the results of the algorithm. Instead of limiting the approach to weakest dependencies, we also consider shared dependencies among cycles. Hence, our approach selects *removable* dependencies based on three characteristics: *shared* dependencies, *light* dependencies, and possible input by the user. We study our approach on package cycles: we identify a layered organization for software packages. The study shows that the strategy provides better results than the previously existing approaches. It performs better in terms of accuracy and interactivity: it contains less false-positive and false-negative results, and it supports human inputs and constraints.

Our approach cannot act as oracle to compute layered organization *i.e.,* it tries to improve existing structure only, and cannot be guaranteed to create a good structure from scratch. In addition, the algorithm 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 less and the algorithm could give more false positives.

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

The algorithm can be improved by introducing more flexibility. First, for manual verification, the user interface and the visualization provided in Section 4 should be improved to be more usable. Second, architectural semantics can be introduced to improve the results.

The future work concerns the improvement of the algorithm to detect *layer-breaking dependencies*. The results of MFAS algorithm and Ozone are improvable in term of precision and recall. New heuristics have to be proposed. Another future work is to analyze other algorithms as the Kleinberg algorithm [Kle99] which seems to be a good perspective for evolving heuristics.

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.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, November 2005.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[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.

[DHHJ10] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Elmar Juergens. Flexible architecture conformance assessment with conqat. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 247–250, New York, NY, USA, 2010. ACM.

[DK76] Frank DeRemer and Hans H. Kron. Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering*, 2(2):80–86, June 1976.

[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+11] Jean Rémi Falleri, Simon Denier, Jannik Laval, Philipe Vismara, and Stéphane Ducasse. Efficient retrieval and ranking of undesired package cycles in large software systems. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS-Europe'11)*, Zurich, Switzerland, June 2011.

[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.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *IEEE Transactions on Software Engineering*, pages 18–28, 1995.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Trans. Softw. Eng.*, 27:364–380, April 2001.

[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.

[Mye78] G. J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold, 1978.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[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.

[PN06] Laura Ponisio and Oscar Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[Pre94] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

[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.

[You79] Edward Yourdon. *Classics in Software Engineering*. Yourdon Press, 1979.