Author manuscript, published in "International Conference on Objects, Models, Components, Patterns (TOOLS) (2011)"

inria-00614770, version 1 - 28 Oct 2011

Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems

Accepted to TOOLS 2011

Jean-Rémy Falleri[1], Simon Denier[2], Jannik Laval[2], Philippe Vismara[3], and Stéphane Ducasse[2]

[1] Université de Bordeaux
[2] Rmod - USTL - INRIA Lille Nord Europe
[3] LIRMM, UMR5506 CNRS - Université Montpellier 2
MISTEA, UMR729 Montpellier SupAgro - INRA

Abstract. Many design guidelines state that a software system architecture should avoid cycles between its packages. Yet such cycles appear again and again in many programs. We believe that the existing approaches for cycle detection are too coarse to assist the developers to remove cycles from their programs. In this paper, we describe an efficient algorithm that performs a fine-grained analysis of the cycles among the packages of an application. In addition, we define a metric to rank cycles by their level of *undesirability*, prioritizing the cycles that seems the more undesired by the developers. Our approach is validated on two large and mature software systems in Java and Smalltalk.

1 Introduction

Large object-oriented software projects are usually structured in *packages* (or *modules*). A package is primarily used to group together related classes which define a functionality of the system. Classes belonging to the same package should be built, tested, versioned, and released together. Martin consequently proposed to see the package as the software release unit [**?**]. Design guidelines state that cyclic dependencies between packages should be avoided [**?**,**?**]. Indeed, packages depending cyclically on each other are to be understood, tested, released, or deployed together.

Several tools and approaches have been developed over the years [**?**,**?**,**?**,**?**] to help the developers to detect cycles. Yet, an exhaustive experimental study [**?**] shows that in a lot of programs, classes are involved in huge cyclic dependencies. It seems therefore plausible that the way cycles are detected is not sufficient to help the developer to remove them.

We claim that the existing approaches have two main issues. First, some focus on cycles between classes, when cyclic dependencies at the package level should have the priority. Indeed classes are not deployment units, and a lot of cycles among classes are due to the associations, being thus totally expected. Second, and most important, existing approaches are all based on the same algorithm by

inria-00614770, version 1 - 28 Oct 2011

Tarjan [?]. This algorithm finds the maximum sets of packages depending (directly or indirectly) on each other, called *strongly connected components* (SCC) in graph theory. Within a SCC, a package is in cycle with all other packages, and there can be multiple cycles in one SCC. In our experience, we have seen software systems with a single huge SCC containing dozens of packages. The above algorithm becomes useless in such cases as it does not provide further information to understand and remove the cycles.

A dependent problem which is not well addressed in current approaches is ranking cycles so that the most "undesired" ones are given top priority for removal. Indeed, not all cyclic dependencies have the same importance. In a hierarchical system of packages (as in Java), a package such as *ui.internal* can be in cyclic dependency with *ui* without much consequences, since they both implement the same functionality. On the contrary, a cycle between *ui* and *core* packages should be avoided as it hampers reuse and deployment of the system. We further discuss this issues as well as the prevalence of packages cycles in four Java programs in Sect. 2.

Our approach advocates the decomposition of a SCC in *multiple short cycles* covering all dependencies of the SCC. Computed cycles usually involve two to four packages. They are therefore easy to understand and to remove, if necessary. Developers can iterate over a set of short cycles and assess them one by one rather than dealing with the single large set of packages contained in the SCC. Moreover, our approach is able to rank the extracted cycles, *prioritizing the ones that seems the more undesired*.

In this paper, we present two major contributions to assist developers in understanding and removing cyclic dependencies in software systems:

- First, we present an efficient algorithm that decomposes a SCC. This algorithm retrieves a set of short cycles that covers all dependencies of the SCC. It has a polynomial time and space complexity (Sect. 3.1).
- Second, we introduce a new metric that evaluates the level of undesirability of a cycle. This metric, called *diameter*, is based upon the notion of distance between packages involved in the cycle (Sect. 3.2).

Our approach is validated against two large and mature programs, in Java and Smalltalk (see Sect. 4).

2 Motivation

This section presents a small study showing why the SCCs are not fine-grained enough to assist developers in understanding and removing cycles in large programs (Sect. 2.1). Then, it explains using an example why some cycles among packages of a software system can be desired by their developers (Sect. 2.2).

2.1 Limitation of the main cycle detection algorithm

Most of the approaches perform cycle detection by using an algorithm [?] that is capable of finding the *maximum sets of packages that depend (directly or in-*

inria-00614770, version 1 - 28 Oct 2011

Program	#P	#LSCC	LSCCR
ArgoUML 0.28.1	79	38	48%
JEdit 4.3.1	29	18	62%
Choco 2.1.0	147	38	26%
AntLR 3.2	31	7	23%

Table 1. Measures among packages and package cycles on the Java programs. #**P** is the number of packages, #**LSCC** the size of the largest SCC and **LSCCR** the ratio of packages in the largest SCC.

directly) on each others. Such a set of packages is called, in the graph theory, a *strongly connected component* (SCC). In a SCC, each package is in cycle with all other packages, and cycles exists only among the packages of a same SCC. To remove package cycles, it is therefore necessary to remove several dependencies among the packages of a given SCC. We believe that the SCCs are not fine-grained enough to help the developer to understand and remove the undesired dependencies in their programs. Indeed, they indicate *which* packages are involved in cyclic dependencies, but they can not explain *how*. Whenever a SCC contains only a few packages, it remains possible to visualize the dependencies between them and to remove the cycles. On the other hand, when a SCC contains a lot of packages, it does not help the developer at all. Indeed, if it contains dozens of packages, it becomes hard to understand how packages connect with each other to create the SCC.

To show that mature and large programs can contain huge SCCs, we proceed to a small experiment. We select four mature and medium-sized Java programs: ArgoUML (http://argouml.tigris.org), JEdit (http://www.jedit.org), Choco (http://www.emn.fr/z-info/choco-solver) and AntLR (http://www.antlr.org). On these programs we compute: #**P** the number of packages, #**LSCC** the size of the largest SCC and **LSCCR**: the ratio of packages in the largest SCC.

Tab. 1 shows that the programs we selected contains large SCCs. In ArgoUML the largest SCC contains almost half of the packages (see the **LSSCR** measure). Worse, in JEdit almost two third of the packages are in the largest SCC, whereas the total number of packages is not too large. Apart from AntLR, the size of the largest SCC in the programs of our corpus will make their understanding hard (see the #**LSCC** measure).

2.2 Desired and undesired cycles

In the introduction, we stated that not every cycle should be removed. In fact, we believe that a significant proportion of the cycles among the packages of a program are desired by the developers. To show this, let us take the example of the JFace (http://wiki.eclipse.org/index.php/JFace) main widget library used in the Eclipse development environment. A great deal of attention has been devoted to its design by several software design experts. We therefore assume

inria-00614770, version 1 - 28 Oct 2011

that the cycles present in JFace are not accidental. Package *jface.text* is dedicated to the text widgets. This package provides classes such as *TextViewer*. Package *jface.text.hyperlink* is dedicated to the management of textual hyperlinks. In JFace, there is a cycle between *jface.text* and *jface.text.hyperlink*. The *TextViewer* class is able to display texts containing hyperlinks and therefore *jface.text* depends on *jface.text.hyperlink*. Also, *jface.text.hyperlink* uses a lot of classes and interfaces defined in *jface.text*. For instance an hyperlink is able to trigger text events and therefore depends on the *TextEvent* class, which is defined in the *jface.text* package. Therefore *jface.text.hyperlink* depends on *jface.text*. In this case, the complexity of the hyperlink motivates its isolation in package *jface.text.hyperlink*. Yet it is not necessary to break the cycle with *jface.text* as it would make no sense to release one without the other.

More generally, in several languages such as Java, a package can contain other packages, leading to a package containment tree. It is usual that when a package is too big (i.e. contains two many classes), it is split in several sub-packages. In this case it is very likely that cycles exist between these sub-packages.

3 Our Approach

In this section, we present our two contributions:

- First, we present an efficient algorithm that decomposes a SCC. This algorithm retrieves a set of short cycles that covers all dependencies of the SCC. It has a polynomial time and space complexity (Sect. 3.1).
- Second, we introduce a new metric that evaluates the level of undesirability of a cycle. This metric, called *diameter*, is based upon the notion of distance between packages involved in the cycle (Sect. 3.2).

3.1 A new cycle retrieval algorithm

Intuition of our algorithm To explain better the intuition of our new algorithm, let us first introduce a sample class diagram, shown in Fig. 1. From this class diagram, we extract the directed graph shown in Fig. 1. This graph shows the dependencies between the packages, therefore we call it a *package dependency graph*. On this graph, the SCCs are rounded by dashed circles.

In the previous section, we stated that the algorithm that computes the SCCs is not fine-grained enough to help the developers to understand and remove cycles from their programs. Fortunately, another algorithm from the graph-theory literature is able to perform a fine-grained analysis of cycles in a directed graph [**?**]. It computes the set of *elementary cycles*. A cycle is elementary if no node (here no package) appears *more than once* when enumerating the sequence of nodes in the cycle. For instance, in our sample graph of Fig. 1, this algorithm finds the six elementary cycles shown in Fig. 1. Figuring out if an elementary cycle should be removed or not is straightforward, it only requires to decide if the

inria-00614770, version 1 - 28 Oct 2011

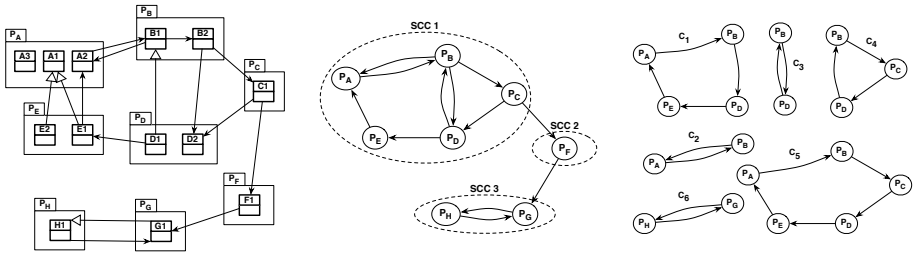

Fig. 1. A sample class diagram (left), the corresponding package dependency graph (midde, the dashed lines round the SCC) and the elementary cycles for this graph(right).

dependencies involved in the cycle are correct. Unfortunately, the number of elementary cycles in a directed graph can be exponential. Therefore, this algorithm does not scale on programs composed of many packages.

We introduce a new algorithm that still computes elementary cycles in a SCC but that retrieves only a polynomial number of them, reducing time and space complexity. Indeed, some elementary cycles can be seen as redundant. In Fig. 1, cycle $\mathcal{C}_5$ is not useful if we consider cycles $\mathcal{C}_1$ and $\mathcal{C}_4$. Indeed, the dependencies covered by $\mathcal{C}_5$ have already been covered by the two other cycles. We reduce the number of cycles by selecting only a subset of the elementary cycles, ensuring that each dependency of the SCC is covered by at least a cycle. Still, to get all dependencies covered in Fig. 1, it is possible to select cycles $\mathcal{C}_2$, $\mathcal{C}_3$, $\mathcal{C}_6$, and either $\mathcal{C}_1$ and $\mathcal{C}_4$, or the longer cycle $\mathcal{C}_5$. We assume that a long cycle is harder to understand than a short one because it requires the analysis of more dependencies. Therefore our final solution is to select for each dependency **one of the shortest cycles going through the dependency**.

Mathematical model A package dependency graph G is a couple (P, D) with P a set of nodes (the packages) and D a set of edges (dependencies between the packages). An edge is a couple $(s, t) \in P^2$ where s is the source and t the target package. There is an edge from a package s to a package t iff a class of s uses a class of t. We define the function $\Gamma^+ : P \to \boldsymbol{\mathcal{P}}(D)$ (with $\boldsymbol{\mathcal{P}}(E)$ the power-set of E) which has the following definition $\Gamma^+(x) = \{(x, y) \in D\}$. This function gives all the dependencies where a given package appears as source. Reversely, $\Gamma^- : P \to \boldsymbol{\mathcal{P}}(D)$ is $\Gamma^-(x) = \{(z, x) \in D\}$. This function gives all the edges where a given package appears as target. We denote a path in G by a sequence of nodes, written this way: (a, b, c), where every node has an edge to its successor. We denote a cycle by such a sequence of nodes : $x \to y \to z$, the last node being implicitly linked to the first one.

Details of our algorithm To understand the algorithm, it is important to notice that cycles exist *only* among the nodes of the same SCC. Also, the set of

inria-00614770, version 1 - 28 Oct 2011

SCCs of a directed graph is a partition of its nodes. Therefore as a preliminary step to our algorithm, we retrieve the SCCs from the directed graph using the algorithm of [**?**], remove the inter-SCCs edges, then run our algorithm on each SCC containing more than two nodes (SCCs of size one cannot contain a cycle). The SCCs of size two contain only one cycle involving the two nodes. Therefore on the graph of Fig. 1, only *SCC 1* is considered by our algorithm, while *SCC 2* is discarded and *SCC 3* directly leads to the creation of the cycle $P_G \rightarrow P_H$. In the following, we therefore focus on what happens in a SCC of size greater than two. To find shortest cycles, we use the well-known *breadth-first search* (BFS) algorithm. This algorithm can be used to find the shortest path between two nodes in a graph where the edges are unweighted. A SCC has the following property: for each possible pair of nodes x, y of the SCC, there is a path from x to y and from y to x. A simple algorithm to find a shortest cycle for every edge of a strongly connected graph is therefore to perform for each edge $(x, y) \in D$ a BFS from target node y going back to source node x. Indeed since there is an edge from x to y, this edge is already the shortest path from x to y. Since we are in a SCC, it is mandatory that at least a path exists from y to x. A shortest path from y to x (found by the BFS) concatenated with the edge (x, y) would therefore be a shortest cycle in which this edge is involved.

The only problem of this simple algorithm is that it requires a BFS for each edge of the graph. Since there are less nodes than edges in a strongly connected graph, it would be better to perform a BFS only for each node of the graph. The idea is therefore to gather the ancestors $A = \{y \in P | (y, x) \in \Gamma_P^-(x)\}$ of a node x, and perform a BFS from x until all its ancestors $y \in A$ are found. This way, only one BFS is performed for each node. The pseudo code of this optimized version is given in Algorithm 1. To avoid the retrieval of identical cycles, we consider that two cycles are equals if the first is a cyclic permutation of the second. For instance $c \rightarrow a \rightarrow b = a \rightarrow b \rightarrow c$. To have a fixed order to represent the cycles and compare them efficiently, we always place the lowest node (using the lexicographic order) at the beginning of the cycle. We call this operation *normalize*. For instance $\mathit{normalize}(c \rightarrow a \rightarrow b) = a \rightarrow b \rightarrow c$.

Let see how this algorithm works on *SCC 1*, shown in Fig. 1. Remember that the edge from P_C to P_F has been deleted because it is a inter-SCCs edge. The set of nodes is $P = \{P_A, P_B, P_C, P_D, P_E\}$. We start with an empty set of cycles: $\mathcal{C} = \{\}$. Here are the steps followed by our algorithm:

1. The first node being picked up is P_A. Therefore, $A = \{P_E\}$. The BFS starting from P_A will find P_E by the following path: (P_A, P_B, P_D, P_E). Since $\mathcal{C}$ is empty, the cycle $\mathcal{C}_1 = P_A \rightarrow P_B \rightarrow P_D \rightarrow P_E$ is added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1\}$.
2. The second node being picked up is P_B. $A = \{P_A, P_D\}$.
   - The BFS started from P_B will find P_A by the following path: (P_B, P_A). This cycle is normalized in $\mathcal{C}_2 = P_A \rightarrow P_B$ and added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2\}$.
   - The BFS started from P_B will find P_D by the following path: (P_B, P_D). The cycle $\mathcal{C}_3 = P_B \rightarrow P_D$ is added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3\}$.
3. The third node being picked up is P_C. $A = \{P_B\}$. The BFS starting from P_C will find P_B by the following path: (P_C, P_D, P_B). After normalization, it becomes $\mathcal{C}_4 = P_B \rightarrow P_C \rightarrow P_D$ and it is added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.

inria-00614770, version 1 - 28 Oct 2011

Algorithm 1: Our cycle retrieval algorithm

```
Data: A strongly connected package dependencies graph G = (P, D)
Result: A set of shortest cycles C
begin
    C ← {} ;                                              // the set of cycles
    for x ∈ P do
        V ← {} ;                                  // the set of the visited nodes
        A ← {z ∈ P|(z, x) ∈ Γ⁻(x)} ;               // the set of the x ancestors
        x.bfs_ancestor ← ∅ ;                      // the path followed by the BFS
        Q ← (x) ;                                  // a queue, initialized with x
        /* BFS from x that stops when every ancestor of x is found  */
        while size(A) > 0 do
            p ← pop(Q);                       // removes the first element of Q
            for (p, y) ∈ Γ⁺(p) do
                /* if y has not been visited or put on the stack yet */
                if y ∉ V ∪ Q then
                    y.bfs_ancestor ← p;
                    push(Q, y) ;                        // adds y at the end of Q
                /* if an ancestor of x is reached                    */
                if y ∈ A then
                    c ← () ;                // the list of the nodes of the cycle
                    i ← y;
                    /* builds the cycle                              */
                    while i ≠ ∅ do
                        add(c, i);
                        i ← i.bfs_ancestor;
                    /* adds the cycle to the set of cycles           */
                    normalize(c);
                    if c ∉ C then C ← C ∪ {c};
                    remove(A, y) ;                          // removes y from A
            V ← V ∪ {p} ;                                   // p is now visited
```

4. The fourth node being picked up is P_D. $A = \{P_B, P_C\}$.
   - The BFS started from P_D will find P_B by the following path: (P_D, P_B). This cycle is normalized in $\mathcal{C}_3$ and therefore is not added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.
   - The BFS started from P_D will find P_C by the following path: (P_D, P_B, P_C). This cycle is normalized in $\mathcal{C}_4$ and therefore is not added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.
5. The fifth and last node picked-up is P_E. $A = \{P_D\}$. The BFS starting from P_E will find P_D by the following path: (P_E, P_A, P_B, P_D). This cycle is normalized in $\mathcal{C}_1$ and therefore is not added to $\mathcal{C}$. $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$.

inria-00614770, version 1 - 28 Oct 2011

Finally, we have $\mathcal{C} = \{\mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3, \mathcal{C}_4\}$. We can notice that in contrast to the enumeration of all elementary cycles (see Fig. 1), the long cycle $P_A \rightarrow P_B \rightarrow P_C \rightarrow P_D \rightarrow P_E$ is not retrieved by our algorithm.

Complexity of Algorithm 1 Let $n = |P|$ be the number of nodes and $m = |D|$ be the number of edges. In the worst case, we pick-up a different cycle for every edge, the maximum number of cycles is therefore m. We split the computation of the worst-case time complexity in three parts: worst time spent in the pre-processing step (finding the SCCs), worst time spent in the BFSes, and worst time spent to add the cycles in the cycle set. Since we work with strongly connected graphs, we have $m \geq n$.

1. The worst case time complexity of the algorithm that computes the SCCs in the pre-processing step in $O(n + m)$ [**?**].
2. The worst case time complexity of a BFS in a graph is $O(n + m)$. Since we perform a BFS for every node of the graph, it leads to a $O(n(m + n))$ complexity for the BFSes.
3. The addition of a cycle in the set of cycles can be done in $O(n \times log(n))$ using appropriate data structures (like a self-balancing binary search tree). In the worst case, we try to add the same cycle involving all packages for each edge. Therefore the worst case time complexity for the additions is $O(m \times n \times log(n))$.

Since $m \geq n$, the overall complexity of our algorithm is $O(m \times n \times log(n))$. Since the number of packages in a program cannot be too large (we consider $1,000$ packages as a fair upper-bound), this complexity is perfectly acceptable to be applied at development-time (for an immediate feedback) as well as maintenance-time (for an in-depth architecture assessment).

3.2 Our distance-based metric to detect undesired cycles

In the previous section, we showed how we efficiently retrieve cycles from a package dependency graph. Unfortunately, there can be many cycles, especially in a large and complex program. A developer is not going to inspect manually all the cycles, because it is a tedious and time-consuming task. Moreover, a significant amount of these cycles is probably *desired*, like we have seen in Sect. 2. To assist in understanding and removing the cycles, it is critical to propose in priority the cycles that seems the most *undesired*. This is the purpose of our *diameter* metric. To define it, we assume that packages are located in a *containment tree*. This is the case in many languages such as Java, C#, Ruby, or PHP. Even when it is not the case as in Smalltalk, such a tree can often be inferred from conventions and names given by the developers to the packages.

To better illustrate the phenomenon described in Sect. 2, let us imagine the sample package containment tree shown in Fig. 2. In this package tree, a cycle between *ui.dialog.wizard* and *ui* seems desired. It is common that a class in a package uses classes of its parent packages. It is also possible that in the

inria-00614770, version 1 - 28 Oct 2011

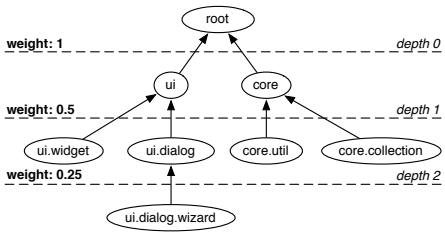

Fig. 2. A sample package containment tree, with the weight associated to the edges.

parent package, several classes depend on the classes of the sub-packages (such as factory classes). In our example, *ui.dialog.wizard* is likely to use several classes defined in *ui*, like a *Widget* class. It is also likely that *ui* furnishes a factory class to create wizards (such as *WizardFactory*), that uses the different wizards defined *ui.dialog.wizard*. In this case this cycle would be totally desired since the developer would neither use nor deploy *ui* without *ui.dialog.wizard*. On the other hand a cycle between *core* and *ui* seems strongly undesired. Although the dependence from *ui* to *core* seems normal, it is unlikely that a package such as *core* requires *ui* to be used or deployed.

We do not want the developers to have the burden of inspecting all the desired cycles. We prefer to show them the cycles that seem the most undesired first. To do so, we use the package containment tree to define a distance between two packages: for instance the number of edges required to go from a package to the other package. We assume that the further away are the packages involved in a cycle, the more undesired the cycle seems. Unfortunately, with this definition of distance, the packages *ui.dialog.wizard* and *ui* are at the same distance from each other as *core* and *ui* (two edges). To deal with this problem we add a second assumption: the farther away the common ancestor between two packages is from the root of the tree, the less the distance between them is significant. For instance, the common ancestor between *ui.dialog.wizard* and *ui* is *ui*, while the common ancestor between *core* and *ui* is *root*.

To deal with the two previously described assumptions, we define a weighting function that assigns a high weight to the edges close to the root and a low weight to the edges far from the root. The weight of an edge depends on its depth. For an edge e at depth d, the weight $w(e) = \frac{1}{2^d}$. Fig. 2 shows a sample package containment tree with weights associated to its edges. The distance between two packages $D_P : P^2 \rightarrow \mathbb{R}^+$ is then equal to the sum of the weights of the edges that lead from the first package to the second one. For instance $D_P(core, ui) = 2$, $D_P(ui.widget, ui.dialog) = 1$ and $D_P(ui.dialog.wizard, ui) = 0.75$.

We can now define our metric that indicates the level of undesirability of a cycle, called *diameter* (denoted by $\mathcal{D}$). It is defined as the worst possible distance between two packages contained in the cycle. More formally, let $\mathcal{C}$ be a cycle, and let $P_\mathcal{C}$ be the set of packages contained in the cycle. $\mathcal{D}(\mathcal{C}) = max(\{D_P(x, y) | \{x, y\} \in P_\mathcal{C}^2, x \neq y\})$. Let us imagine that there is the fol-

inria-00614770, version 1 - 28 Oct 2011

lowing cycle: $ui \rightarrow ui.widget \rightarrow core$. The diameter of this cycle is $\mathcal{D}(ui \rightarrow ui.widget \rightarrow core) = 2.5$ because $D_P(core, ui) = 2$, $D_P(ui, ui.widget) = 0.5$ and $D_P(core, ui.widget) = 2.5$. We also have: $\mathcal{D}(ui \rightarrow ui.widget \rightarrow ui.dialog) = 1$ because $D_P(ui, ui.widget) = 0.5$, $D_P(ui, ui.dialog) = 0.5$ and $D_P(ui.dialog, ui.widget) = 1$. As one can notice, the larger the diameter is, the more undesired it seems to be.

4 Validation

We validate our approach on two large programs with an experiment involving their maintainers. Our approach can be used both at development-time and at maintenance-time. Nevertheless, we believe that it is harder to understand and remove a cycle at maintenance-time, because it is necessary to remember the past design decisions that led to its creation.

To show that our approach is useful we take the use-case where a developer use our tool, called *Popsycle*[4], on his software at maintenance-time. Popsycle uses the algorithm described in Sect. 3.1 to extract the cycles. It ranks them using the metric presented in Sect. 3.2 (cycles with a large diameter being ranked first). If two cycles have an equal diameter, the number of packages contained in the cycle is used to rank the cycles (the less packages it has, the better it is ranked). If two cycles have an equal diameter and number of packages, they are ranked using the lexicographic order. In addition, Popsycle provides a view that ease the understanding of the cycles by showing the underlying dependencies between the classes that create the cycle.

4.1 Preparation of the data

We chose two different programs to perform our experiment.

RESYN-Assistant RESYN-Assistant (http://www.lirmm.fr/~vismara/resyn), is a Java program targeting the domain of organic chemistry. It includes several algorithms for perceiving molecular graphs according to their topological, functional and stereo-chemical features. The development of RESYN-Assistant started in 1996 at the LIRMM institute. It received financial support from the Sanofi-Aventis pharmaceutical company and the french Languedoc-Roussillon region. The development team was composed of four persons: two researchers in computer-science, one PhD student in computer science and one PhD student in chemistry. Because of the turnover within the development team, and because it has mostly been developed by students having different resaerch objectives, its architecture has decayed since the initial version.

The characteristics of the RESYN-Assistant architecture are the following:

- 315 classes, 33 packages, 242 package dependencies
- one SCC (of size > 1) containing 29 packages and 221 dependencies

[4] http://popsycle.googlecode.com

inria-00614770, version 1 - 28 Oct 2011

Pharo Pharo (http://www.pharo-project.org) is an open-source Smalltalk environment. It has been forked from Squeak, a re-implementation of the classic Smalltalk-80 system. Squeak development was started by Alan Kay group in 1996 based on an original Smalltalk-80 implementation. It received financial support from the Apple and Disney companies. There were about 15 active developers and more than 100 committers involved in its development. Squeak contains two graphical frameworks, support for advanced sounds and multimedia presentations, kid authoring system, as well as support for networking and web programming. Lot of experimental code was included in the system without attention to the impact on the global architecture. Pharo forked the code of Squeak in 2008. Its goal is to provide a clean and stable version targeting professional companies as well as researchers. Pharo development team involves about 10 active developers and about 50 committers. The system inherits from more than 15 years of development in a monolithic system context.

The characteristics of the Pharo architecture are the following:

- 1800 classes, 102 packages, package dependencies
- one SCC (of size > 1) containing 61 packages and 790 dependencies

Extraction of the dependencies To extract the dependencies in the Java program, we use the Apache BCEL (http://jakarta.apache.org/bcel) library on the byte-code of the program. With BCEL, we extract most of the dependencies between the classes, but some of them can be missed. In particular when a method is overloaded, the dependency extracted from the byte-code is always the class that defines the method. Also it is possible that some types are erased if they are used only internally in a method. For Smalltalk applications, we use the MOOSE (http://www.moosetechnology.org) reverse-engineering platform. Since Smalltalk is dynamically typed, type information is hard to extract. The Moose environment deals with that situation by providing a type inference mechanism. It is possible to select the level of fuzziness of the type inference. We selected only the dependencies that can be statically resolved: only direct class references are used to identify dependencies. In addition, Smalltalk does not provide a tree structure for the packages. Nevertheless, the developers of Pharo we analyzed use the names of the packages to simulate it (typical package names: *Collections-Stream* or *Collections-Strings*). Therefore, we take advantage of this naming convention to extract a tree from the package names.

4.2 Experiment

When using Popsycle to extract package cycles, one expects that the most undesired cycles will be ranked first and that the desired cycles will be ranked last. He also expects that Popsycle will extract short cycles, which are easier to understand than the long ones. To validate this, we set up the following experiment. For each program, we compute and rank the cycles. First, we compute the distribution of the cycle sizes, to ensure that short cycles are retrieved. We then ask the maintainers of the programs to count how many cycles in the k

inria-00614770, version 1 - 28 Oct 2011

first ranked by *Popsycle* are undesired, and how many of the k last cycles are desired. Using this information, we compute the precision over the k first cycles $FP_k = \frac{|\text{undesired cycles}|}{k}$. In our experiment, maintainers will compute FP_{10}, FP_{20} and FP_{30}. These measures will show if our ranking metric is able to rank high undesired cycles. But it could be the case that there are only undesired cycles in the programs of our experiment. In this case, any ranking algorithm would have a good precision. To ensure the fact that our ranking metric is able to rank low the desired cycles, we will also compute the precision over the k last ranked cycles $LP_k = \frac{|\text{desired cycles}|}{k}$. In our experiment, the maintainers will compute LP_{10}, LP_{20} and LP_{30}. If both FP_k and LP_k are close to 1, it means that our ranking metric is useful. Lastly, maintainers has been asked to provide a short explanation on why the first cycles were undesired and why the last cycles were desired.

4.3 Results

Size of the cycles On RESYN-Assistant, our algorithm finds 171 cycles in 17 milli-seconds (mean time computed over 10 runs on a 2GHz Intel Core 2 Duo). The distribution of the cycle sizes is shown in Fig. 3. The largest cycles are of size 6, which is manageable. The majority of the cycles are of size 2, 3 or 4, which are size totally suited for an easy understanding of the cycles. In comparison with the size of the unique SCC (that contains 27 packages), the size of the cycles found by our algorithm is significantly smaller.

Our algorithm finds 619 cycles in Pharo in 40 milli-seconds (mean time computed over 10 runs on a 2GHz Intel Core 2 Duo). The distribution of the cycle sizes is shown in Fig. 3. The largest cycles are of size 5. Like in the previous experiment, the majority of the cycles are of size 2, 3 or 4, even if the size of the SCC is the double of the one in RESYN-Assistant. These sizes are still significantly smaller than the size of the unique SCC of Pharo.

Precision Tab. 2 shows the precision over the k first and last cycles. Precision is good for the two programs, even with $k = 30$. It means that the first ranked cycles were, as expected, undesired. The last ranked cycles were, as expected, desired.

4.4 Analyze of the cycles

RESYN-Assistant

Program	FP_{10}	FP_{20}	FP_{30}	LP_{10}	LP_{20}	LP_{30}
Pharo	0.9	0.9	0.87	1	1	0.97
RESYN-Assistant 1.0	1	1	1	1	1	1

Table 2. Precision over the k first and last ranked cycles

inria-00614770, version 1 - 28 Oct 2011

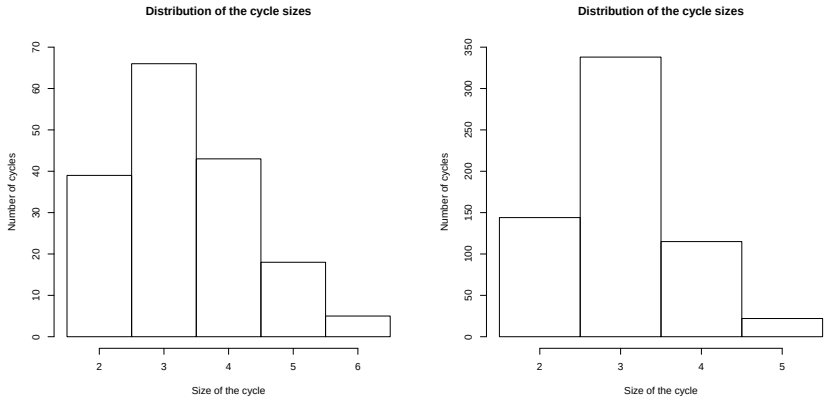

Fig. 3. The distribution of the cycle sizes in RESYN-Assistant (left) and Pharo (right)

First cycles We found several undesired cycles because the lack of a MVC pattern, creating cycles between the GUI and the algorithms. These cycles have not yet been fixed because they require a significant modification of the code. Several cycles were due to the implementation of an unnecessary interface, and have been corrected instantaneously. A cycle was due to a method that in fact was never called, this cycle has therefore been corrected instantaneously.

Last cycles Most of them are cycles between packages of the RESYN-Assistant GUI. These cycles are desired since complex graphical components have been developed in the sub-packages and cycles exists between the main window (located in the parent package) and these components. Several other cycles are related to the implementation of an algorithm that was too complex to be implemented in only one package and was therefore split in three packages. Lastly, several cycles are between several packages defining a graph API.

Pharo

First cycles Most of the first cycles were due to the existence of a multi-purpose package (*System-Support*) that has become huge over the years and contains a lot of misplaced classes. This package creates a lot of undesired dependencies and cycles in the system. We also find several cycles involving the GUI package (*Morphic*), because of the lack of use of an MVC pattern. Several other cycles were due to misplaced methods. Most of the cycles have been corrected in the new version of Pharo.

Last cycles Most of them are cycles between the *Collection* package and its sub-packages. Several cycles are between the *Network* package and its sub-packages. There is also a cycles between the graphical widget package (named *Morphic*) and its sub-packages.

inria-00614770, version 1 - 28 Oct 2011

4.5 Threats to validity

The methods we use to extract the dependencies extract only a subset of them. Therefore it is possible that at runtime several additional dependencies exist, leading to more cycles. Nevertheless, it is unlikely that it would change the precision results. Another threat to validity is that when we compute the precision, we analyze only a subset of the cycles (35% for RESYN-Assistant and 9.7% for Pharo). It is very likely that for greater values of k, the precisions LP_k and FP_k will decrease. Lastly, we selected two softwares that have a lot of architectural problems, leading to a lot of undesired cycles. In cleaner software systems, only a few cycles are undesired. In this case, it is possible that they would be missed by our ranking metric.

5 Related Work

Several tools and approaches have been introduced over the years to deal with the problem of cyclic dependencies among packages and classes in a software system. These approaches can be roughly classified using the following criterion: 1) approaches working at the package level, 2) approaches working at the class level, 3) approaches using graph theory algorithms and 4) approaches based on dependency matrix algorithms.

As a general rule, these approaches are concerned with detecting and reporting cycles using Tarjan SCC algorithm [**?**] or some simpler algorithms. Such approaches do not scale to programs involving large SCCs because they do not provide a deep analysis of how such SCCs arise and how to remove cycles in a SCC. In contrast, we define an algorithm and an approach which computes the information necessary to understand SCCs through subsets of elementary cycles, and that is able to rank cycles by their level of undesirability.

Mudpie [**?**] is a reporting tool to detect cyclic dependencies between packages in Smalltalk. The paper reports on a single case study performed on packages of the Refactoring Browser in Smalltalk. Classycle (http://classycle.sourceforge.net) is a reporting tool which detects SCC both at class level and package level. Classycle proposes some metrics to characterize cycles but no formal definitions are proposed and their goal is unclear. Both tools rely on Tarjan SCC algorithm for detection of cycles, which make them impractical to analyze large SCCs.

PASTA [**?**] 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 SCC as a single package. The other heuristic selectively ignores some dependencies until no more cycle is detected. Thus, PASTA reports on these *undesirable* dependencies which should be removed to break cycles. The paper reports on a case study analyzing the Java core package with effective results. It would be interesting to compare the heuristics for undesirable dependencies with our distance metric for undesired cycles.

JooJ [**?**] is an approach to detect and remove cyclic dependencies between classes. The principle of JooJ is to find statements creating cyclic dependencies

inria-00614770, version 1 - 28 Oct 2011

directly in the code editor, allowing the developer to solve the problem as it appears. It computes the SCC using Tarjan 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 NP-complete problem is called minimum feedback arc set in the literature. It highlights therefore the minimum number of statements that one needs to remove to supress all class cycles. However, no study is made to validate this approach : it is possible that the selected dependencies are in fact not to be removed.

Byecycle (http://byecycle.sourceforge.net) is an eclipse plugin to visualize dependencies at class level. It detects and colors in red dependencies involved in cycles. By construction, set of red edges highlight SCC in the visualization. However, the tool does not provide further help for cycle analysis.

JDepend (http://clarkware.com/software/JDepend.html) is a tool for Java which check Martin's principles [**?**] for package design. In particular, it checks that the package dependency graph is acyclic. Contrary to other approaches, this tool does not detect and retrieve packages in SCCs, but simply reports for each package whether there is a cycle in its transitive dependency graph. For example, with packages A and B in cycle and package C depending upon A, JDepend reports that C depends on a cycle. It can become overwhelming if many packages depends on the same cycle (as each will report separately the cycle) yet is not exhaustive as the tool stops as soon as a cycle is detected (not reporting all cycles in the dependency graph).

Dependency Finder (http://depfind.sourceforge.net) is a set of command line tools to analyze compiled Java code with a focus on dependency graph. One tool detects cycles but at class level only. The algorithm used is not described, although it seems to report elementary cycles.

Dependency structural matrix [**?**] is an approach developed for process analysis. It visualizes dependencies between some elements (tasks, processes, modules) using the adjacency matrix representation. Several algorithms are defined on the dependency matrices. The main step, called *matrix partitioning*, has a similar output to SCC in a directed graph. Dependency matrices rely on visualization to understand cycles. They make direct cycles easy to spot but indirect cycles are hard to understand with this approach. Lattix [**?**] and eDSM [**?**] are two adaptations of dependency matrix to the visualization of package dependencies. They highlight cycles in SCC and can be used as a starting point to understand the architecture of the system. However, due to their limitations in visualizing indirect cycles, they do not benefit from our work which decomposes SCCs in direct and indirect cycles. Instead, we view our work as complementary with DSM as a high level tool and other tools for fine-grained analysis of cycles.

6 Conclusion and Future Work

In this article, we presented two contributions that assist the developers to understand and remove the cycles among packages of a large software system.

inria-00614770, version 1 - 28 Oct 2011

- First, we presented an efficient algorithm that decomposes a SCC. This algorithm retrieves a set of short cycles that covers all dependencies of the SCC. It has a polynomial time and space complexity.
- Second, we introduced a new metric that evaluates the level of undesirability of a cycle. This metric, called *diameter*, is based upon the notion of distance between packages involved in the cycle.

Since our algorithm has a low complexity, it can be applied at maintenance-time as well as at development-time, preventing cycles to appear before it is too late. We validate our approach on several case-studies on mature real-world programs in Java and Smalltalk. It shows that our approach has a practical interest and is easy to adapt to various object languages.

To improve our approach we plan to work on the following problems. First we want to define other metrics on the cycles than our distance-based metric. Second, we want to create a visualization that is not a list, but rather a global view of the cycles of the software, it would allow the developers to have a more global vision of the cycles in their programs. Finally we want to adapt and apply our tool to legacy procedural languages like *C* or *ADA*, because we believe that cycles are frequent in legacy code. An approach able to help the developers to remove some of them would ease the maintenance effort spent on these systems.