Identifying cycle causes with Enriched Dependency Structural Matrix

Jannik Laval, Simon Denier, Stéphane Ducasse, Alexandre Bergel

Identifying cycle causes with Enriched Dependency Structural Matrix

Accepted to WCRE 2009

Jannik Laval, Simon Denier, Stéphane Ducasse, Alexandre Bergel
RMoD Team
INRIA - Lille Nord Europe - USTL - CNRS UMR 8022
Lille, France
{*jannik.laval, simon.denier, stephane.ducasse, alexandre.bergel*}*@inria.fr*

Note for the reader: this paper makes heavy use of colors in the figures. Please obtain and read an online (colored) version of this paper to better understand the ideas presented in this paper.

***Abstract*—Dependency Structure Matrix (DSM) has been successfully applied to identify software dependencies among packages and subsystems. A number of algorithms were proposed to compute the matrix so that it highlights patterns and problematic dependencies between subsystems. However, existing DSM implementations often miss important information to fully support reengineering effort. For example, they do not clearly qualify and quantify problematic relationships, information which is crucial to support remediation tasks.**

In this paper we present enriched DSM (eDSM) where cells are enriched with contextual information about (i) the type of dependencies (inheritance, class reference ...), (ii) the proportion of referencing entities, (iii) the proportion of referenced entities. We distinguish independent cycles and stress potentially simple fixes for cycles using coloring information. This work is language independent and has been implemented on top of the Moose reengineering environment. It has been applied to non-trivial case studies among which ArgoUML, and Morphic the UI framework available in two open-source Smalltalks, Squeak and Pharo. Solution to problems identified by eDSM have been performed and retrofitted in Pharo main distribution.

***Keywords*-software visualization; reengineering; dependency structural matrix; package; dependency**

I. Introduction

Understanding the package organization of an application is a challenging and critical task since it reflects the application structure. Many approaches have flourished to provide information on packages and their relationships, by visualizing software artefacts [22], metrics, their structure and their evolution. Distribution Map [5] shows how properties are spread over an application. Lanza et al. [8] propose to recover high-level views by visualizing relationships. Package Surface Blueprint [10] reveals the package internal structure and relationships among other packages – surface represents relations between the analyzed package and its provider packages. Dong and Godfrey [4] propose high-level object dependency graphs to represent and understand the system package structure.

Dependency Structure Matrix (DSM) is a well-know technique to identify cycles [16]. Originally it has been developed for process optimization to identify dependencies between tasks. This method has been applied with success to identify software component dependencies [17], [18]. MacCormack et. al. [14] have applied DSM to analyze modularity of the architecture of Mozilla and Linux.

While DSM is a robust solution to reveal software structure, DSMs have weaknesses too. DSM current implementations allow one to perform high-level inventory of a situation, but they are limited for fine-grained understanding—tools just offer drop-down lists to show classes and methods creating dependencies between packages.

For example, current DSM implementations do not provide detailed information about interpackage dependencies. Cycles, which constitute a special target for dependency resolution, are commonly identified using the adjacency matrix power method [23]. Unfortunately, this algorithm inaccurately identifies cycles since independent cycles are merged.

Our contribution is two-fold: first, we identify *weaknesses* of current DSM; second, we address these weaknesses. We propose, eDSM, a DSM with enriched cells[1]. eDSM cells contain contextual information which shows (i) the *nature* of dependencies (inheritance, class reference, invocation, and class extension), (ii) the *referencing* entities, (iii) the *referenced* entities, (iv) the *spread* of the dependency. We distinguish independent cycles and differentiate cycles using colors. We applied eDSM on several large systems, *ArgoUML*, *Morphic* an UI framework and Seaside a dynamic web framework. The results of the analysis of *Morphic* were proposed to Pharo implementors (including two of the contributors) and integrated in Pharo [2], a new open-source version of Squeak [3]. Squeak is an open-source implementation of Smalltalk programming language and environment. Our approach is language independent since it is based on the language independent FAMIX metamodel [3], [6].

The paper is organized as follows: Section II introduces DSM and its limitations in existing implementations. Sections III and IV present eDSM specifications and its usage, from overview of an application to detailed view of interpackage dependencies. Section V reports an experiment on *Morphic* and identifies some patterns and solutions found in the study. Section VI discusses about our solution and

[1] a DSM cell represents the intersection of two packages
[2] http://www.pharo-project.org
[3] http://www.squeak.org

future work.

II. DSM Limitations

DSMs are effective for detecting cycles between software components. The use of DSMs gives pertinent results for the verification of the independence of software components [16]. However, in their current form, DSMs must be coupled with other tools to offer fine-grained information and support corrective actions.

Figure 1(a) shows a sample dependency graph and its corresponding binary DSM. A binary DSM shows the existence/lack of a dependency (or reference) by a mark or "1/0". The rule for reading the matrix is: element in column header references element in row header if there is a mark. In our context, A, B, C, and D are packages. The element in column header is also called the source and the one in row header the target. In Figure 1(a), A references B and C, B references A, C references D and D references C.

We applied DSM on a couple large case studies and we identified two limitations with current DSM implementations: inaccurate merging of independent cycles by the adjacency matrix power method (Section II-A), lack of fine grained-overview of the dependency cause (Section II-B).

A. Problem with the adjacency matrix power method

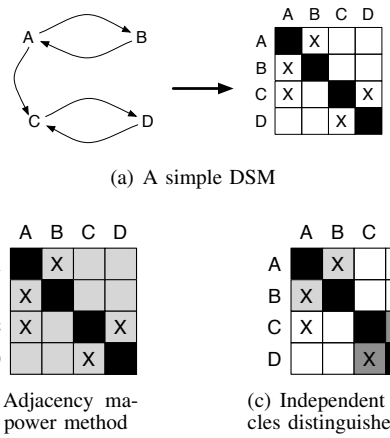
A simple way to identify cycles in DSM is to use the adjacency matrix power method. The principle of this approach is to raise the binary DSM to its n^{th} power to find elements which link back to themselves in n edges, thus making a cycle [23]. A non-zero mark in the diagonal of the power matrix points to elements involved in a cycle of length n.

Figure 1(a) shows two distinct direct cycles (a cycle between two entities): one between A and B and one between C and D. However, the adjacency matrix power does not separately identify these different cycles, resulting in a single and inaccurate merged cycle. Figure 1(b) shows the partitioned matrix with a unique wrong cycle—a single gray zone representing the cycle. On the contrary, Figure 1(c) shows a correctly partitioned matrix with two distinct cycles.

The adjacency matrix power method produces inaccurate results when used to identify independent cycles because it computes the number of edges to come back to an element without considering the cycling path [11]. Instead, path searching algorithms have also been used to detect cycles in DSM and should be systematically preferred when the problem of identifying independent cycles is important.

B. Lack of fine-grained information

A traditional DSM offers an easily readable general overview but does not provide details about the situation it describes. We identify two weaknesses: lack of information on dependency causes and lack of information on dependency distribution.

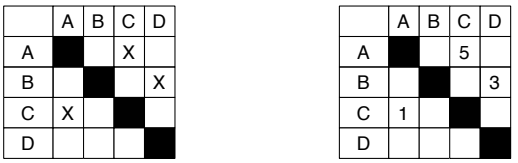


(a) A simple DSM

(b) Adjacency matrix power method

(c) Independent cycles distinguished

Figure 1. Limitation of the adjacency matrix power method. Cycles are shown in gray.



(a) DSM with marks

(b) DSM with numbers

Figure 2. Examples of references in a DSM.

Dependency causes.: Fixing a cycle often means changing some dependencies involved in the cycle. However, the cost of fixing a cycle may vary with the cause of dependency *e.g.,* changing a reference to a class is often easier than changing an inheritance relationship. Dependencies are of different natures (class reference, method invocation, inheritance relationship, and class extension) and a binary matrix (Figure 2(a)) or a matrix providing the number of dependencies in each cell (Figure 2(b)) do not provide such information.

Annotating a DSM with the types of dependencies can give more fine-grained information and it supports a better understanding of the situation. However, a challenge with this solution is that the matrix should remain readable and should not be overloaded [1].

Dependency distribution.: Knowing that a package has 78 dependencies to another one (package *Morphic-Widgets* on *Morphic-Basic* in Figure 6) is a valuable but insufficient information.

The ratio of concerned classes in a package is important since it allows one to quantify the effort to fix a cycle. The intuition is that it is easier to target few classes with some dependencies rather than a lot of classes with few dependencies. For example in Figure 6, 16 classes of package *Morphic-Widgets* reference 10 classes of package *Morphic-Basic*, while only two classes of *Morphic-Basic* reference one class of *Morphic-Widgets*. Consequently, it should be faster to target the dependencies from *Morphic-Basic* to

Morphic-Widgets rather than the ones in the opposite direction.

III. Enriched DSM (eDSM)

To address the problems previously mentioned, we enhance DSM with functionalities that are not present in current DSM implementation such as Lattix [16]. Our solution (i) isolates independent cycles using colors (Section III-A), (ii) enriches contextual cell information (Section III-B) and (iii) supports class extensions (Section III-B).

In particular, enriched cells act as *small multiples* [20] where similar looking side by side little visualizations provide a differentiating effect (see Figure 5). An important design feature is the use of color to focus on packages where it is easier to resolve a cyclic dependency. Therefore we use brighter colors for places having fewer dependencies. The tool is implemented on top of the Moose open-source reengineering environment. Since it is based on the FAMIX meta-model [3], our eDSM works for mainstream object-oriented programming languages [6].

A. Enhanced cycle detection

Our approach enhances the traditional matrix by providing a number of new features: cycle distinctions, direct and indirect cycle identification, and hints for fixing cycles.

Cycle distinctions.: eDSM distinguishes independent cycles using a path searching algorithm [11]. With this method, two independent cycles are detected separately and remain isolated from each other in the DSM (Figure 1(c)).

Direct and indirect cycles.: We use color in DSM cells to identify cycles. DSM cells involved in a cycle have a yellow or red color (Figure 3). The red color means that the two concerned packages reference each other and thus create a direct cycle. Two packages in a direct cycle have two red cells symmetric against the diagonal. The yellow color means that the dependencies from one package to the other participate in an indirect cycle (a cycle with more than two elements). The pale blue background color frame all cells involved in a indirect cycle (visible in Figure 5). Its area is a visual indication of the number of packages in the cycle. On the contrary, rows and columns with white or gray colors indicate packages not involved in any cycle. The diagonal of the matrix, where a package may reference itself, is colored in gray to highlight the symmetry axis but is not used in the current version.

Color hint for targeting cycle.: We define a special rule to highlight cells of primary focus when resolving cyclic dependencies. The intuition is that it will be easier to fix a cycle by focusing on the side with fewer dependencies. A cell with much fewer dependencies is displayed with a bright red color whereas its symmetric cell is displayed with a light red/pink color (Figure 3). The ratio we currently use is 1 to 3. This rule only applies to direct cycles as it is easier to compare two packages side by side than an arbitrary number of packages involved in an indirect cycle.

Figure 3 illustrates the rules for cycle colors in cells. It shows two direct cycles with the red color, one between A and B and one between C and D. The bright red color in the C-D enables one to quickly focus on the dependencies from C to D, since there are only two of them instead of 25 in the opposite direction. Finally, E, F, and G are involved in the same indirect cycle highlighted by the yellow and pale blue cells. The cycle color is in fact one among other pieces of information displayed in a cell, as we explain in the following section.

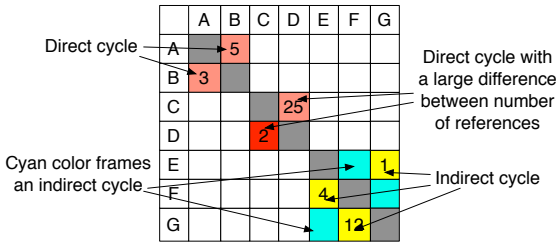


Figure 3. Cell color definition.

B. Enriched contextual cell information

eDSM enriches cell contents to give a detailed overview of dependencies from a source package to a target package. Thus each cell is the intersection between a source and a target. The objective is to create *small multiples* [20] as shown in Figure 5.

The principle of small multiples is that "once viewers decode and comprehend the design for one slice of data, they have familiar access to data in all the other slices" [20]. In eDSM, each cell represents a small context, which enforces comparison with others. Each cell represents a small dashboard with indicators about the situation between the source and the target. This section focuses on the cell contents.

It should be noted that an earlier version of this work did not use small multiples and only showed a summary of the cell contents [1]. The results were limited: It was difficult to see where to begin for resolving issues. To circumvent these limits, we designed enriched cells, which provide visual patterns.

Overall structure of an enriched cell.: An enriched cell is composed of four parts (see Figure 4). The top row gives an overview of the strength and nature of the dependencies. The bottom row presents cycle information as explained in the previous subsection. The two large boxes in the middle detail dependencies going from the top box to the bottom box *i.e.,* from the *source package* to the *target package*. Each box contains squares that represent involved classes: referencing classes in the source package and referenced classes in the target package. Edges between squares link each source class (in top box) to its target classes (in bottom box) (Figure 4).

Colored information.: Enriched cells make use of colors to convey more information about the context in which dependencies occur. Our goal is to use preattentive

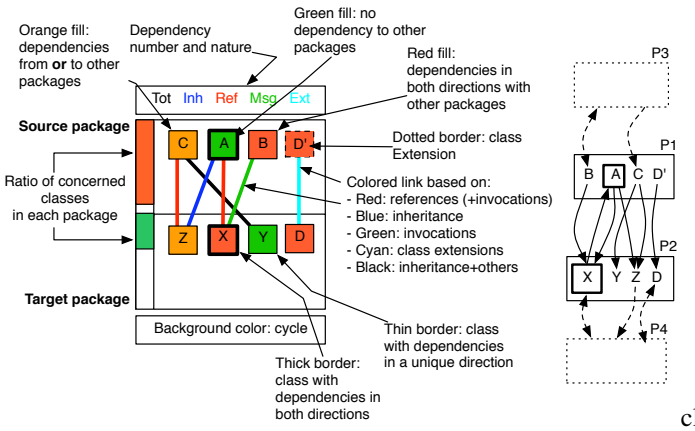


Figure 4. Enriched cell structural information.

visualization[4] as much as possible to help spotting important information [19], [12], [13], [21]. An enriched cell is composed of parts and shapes with different color schemas.

Cycle color (bottom row).: It is the first information to see in a cell. The bottom row represents cycle information using color as explained in Section III-A. The red/pink indicates a direct cycle between the two packages, yellow and cyan an indirect cycle, and gray an unidirectional access from the source package to the target package—which means that there is no cycle.

Dependency overview.: An enriched cell shows an overview of the strength, nature, and distribution of the dependencies from the source to the target.

- *Dependency strength and nature (top row).* The top row gives a summary of the number and nature of dependencies to get an idea of their strength. We show the total number of dependencies (Tot) in black, inheritance dependencies (Inh) in blue, references to classes (Ref) in red, invocations (Msg) in green, and class extensions (Ext) made by the source package to the target one in cyan[5]. A brighter color highlights the strongest dependency type. The colors are used to reinforce the comprehension of links between classes (see below). In Figure 6 there are 78 directed dependencies from *Morphic-Widgets* to *Morphic-Basic*. There are 12 inheritances, 24 references and 42 invocations (in bright green).
- *Dependency distribution (left bars).* For each package, we are interested in the ratio of classes involved in dependencies with the other package. We map the height of the left bar of each package box to the percentage of classes involved in the package. The bar color is also mapped to this percentage to reinforce its impact (from green for low values to red for 100% involvement). A package showing a red bar is fully involved with the other package.

Class color (middle boxes).: Each square represents a class and displays two types of information using its fill color as well as its border (Figure 4).

- *Border color and thickness: Internal usage.* A gray thin border means that the class has a unidirectional dependency with the other package *i.e.,* it either uses *or* is used by classes in the other package. In Figure 4, class B (resp. Z) has a thin border because it refers to X but is not referred in the target package (resp. is referred by A but does not refer to source package).
  A black thick border means that the class has a bidirectional dependency with the other package: it both uses and is used by classes in the other package of the cell (not necessarily the same classes). In Figure 4, class A has a thick border because it is referred by class X of the target package and because it refers to class Z.
  A dotted border represents a class extension as explained below.
- *Color fill: Total usage.* A class may be in dependency with other packages than the two represented by the cell, such as classes B or C in Figure 4. The color fill uses the colors of the traffic lights (green, orange, red) to qualify the relationships the class has with packages other than the two concerned. A class which has no dependency with external packages other than the concerned packages is displayed as green. A class which has dependencies in only one direction (*i.e.,* either incoming dependencies *or* outgoing dependencies) is displayed as orange. A class which has dependencies in both directions is displayed as red. A class which only has internal dependencies is never displayed in a DSM. Thus, moving a green class between the two concerned packages does not have any impact on their external dependencies, whereas moving a red or orange class can change their respective external dependencies.

Edge color.: Edges are the smallest details displayed by the eDSM. They give information on the nature and spread of dependencies between the classes in the cell (Figure 4). There are four basic natures, each one mapped to a primary color (synchronized with colors of information in top row of the cell): reference in red, inheritance in blue, invocation in green and class extension in cyan. When dependencies between two classes are of different natures, colors are mixed as follows: red is used for a dependency

[4]Researchers in psychology and vision have discovered a number of visual properties that are preattentively processed [12]. They are detected immediately by the visual system: viewers do not have to focus their attention on a specific region in an image to determine whether elements with the given property are present or absent. An example of a preattentive task is detecting a filled circle in a group of empty circles. Commonly used preattentive features include hue, curvature, size, intensity, orientation, length, motion, and depth of field. However, combining them can destroy their preattentive ability (in a context of filled squares and empty circles, a filled circle is usually not detected preattentively).

[5]A class extension is a method defined in a package, for which the class is defined in a different package [2]. Class extensions exist in Smalltalk, CLOS, Ruby, Python, Objective-C and C#3. 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 other packages. AspectJ inter-type declarations offer a similar mechanism.

with both references and invocations because a reference is often following by invocations (a new color would make it more difficult to understand the figure). Black is used for any dependency involving inheritance with references and/or invocations. Indeed, an inheritance dependency mixed with other dependencies can be quite complex and we choose not to focus on such a combination.

Representation of class extension.: A class extension represents a method which is in another package than its class. In a cell, a class extension is represented by a square with dotted border linked to the original class by a cyan link and the same color information than the original class (Figure 4). This convention exists because a class extension is not a class. In addition if class extensions are not handled adequately, they can be considered as wrong cyclic dependencies.

C. Interaction and detailed view

While the eDSM offers on overview at the package level as shown by Figure 5, extracting all the information from a cell is sometimes difficult. There is a clear tension between getting a small multiple effect and details readability. We offer zoom and fly-by-help to improve usability.

Zooming on two packages.: Each cell in a DSM represents a single direction of dependency. To get the full picture of interactions between two packages, we compare two cells, one for each direction. Despite DSM intrinsic symmetry, it is not always easy to focus on the two concerned cells. We provide a selective zoom with a detailed view on the two concerned cells, as shown in Figure 6. Thus, we focus on a direct cycle which seems interesting from the overview, and analyze the details with the zooming view.

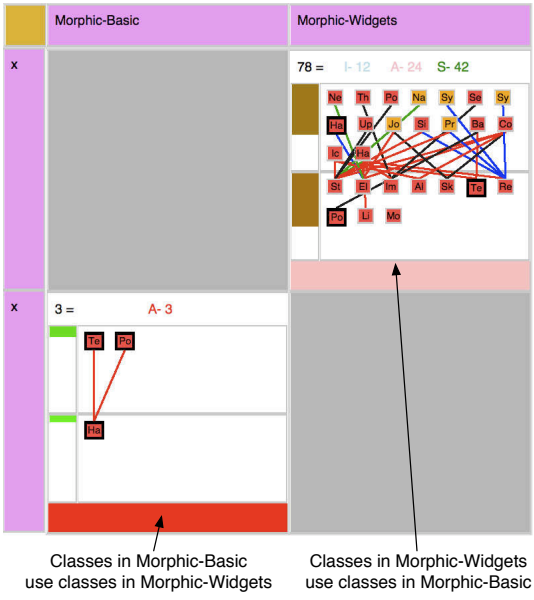


Figure 6. Zoom on two packages in cycle.

Fly-by help.: Complementary to the overview and the zooming facility, fly-by-help includes the full name of concerned packages, the name of classes and the name of each concerned method. Figure 7 shows the pop-up information of the cell linking *Morphic-Worlds* to *Morphic-Widget*: there is one reference to the class *HaloMorph* from the method *PasteUpMorph.acceptDroppingMorph:event:*.

D. Fixing a cycle with eDSM

We now detail a first example of cycle resolution through the analysis of zoomed cells.

In Figure 6, there is a direct cycle between *Morphic-Widgets* and *Morphic-Basic* (named *Widgets* and *Basic* below). We can see that *Widgets* have lots of dependencies to *Basic* (pink cell) while only two classes in *Basic* use one class of *Widgets* (red cell). Moreover, there are only red edges (class references) in the red cell, whereas in the pink cell they are of multiple colors.

At first glance, it is thus easier to investigate the dependencies of the red cell, from *Basic* to *Widgets*. Let us look at the red cell. There are two referencing classes and one referenced class. All three are colored in red, which means they use and are used in other packages. Thus, it would be difficult to move these classes without further investigation. Instead, we focus on the dependencies between classes in the red cell, which are only class references. The fly-by help displays for each class in the cell the concerned methods (methods in the source package making class references in the target package). There are three such methods: *PolygonMorph.customizeArrows:*, *TextMorph.setCurveBaseline:*, *TextMorph.changeMargins:*. This provides entry points in the source code to find precisely where the target class *HandleMorph* is referenced.

It appears that each of these methods contains the line *HandleMorph new* to create an instance of *HandleMorph*. A possible solution is to create class extensions for *TextMorph* and *PolygonMorph* in the package *Widgets* and to put the three referencing methods in it. Then the dependencies would be reversed effectively breaking the cycle.

IV. Small multiples at work

eDSM supports the understanding of the general structure of complex programs using structural element position. Since it is based on the idea of small multiples [20], the cell visual aspect generates visual patterns. While performing our *Morphic* experiment, we have detected some patterns stressing characteristic situations.

We applied eDSM to the Morphic framework of Squeak. Morphic is composed of 46 packages and 325 classes. It was never packaged in a modular way, hence showing a lot of cyclic dependencies. We use this case study to show eDSM in practice (Figure 5).

The first use of the eDSM is to get a system overview to scan packages not involved in cycles (not shown in Figure 5) and how they interact with other packages. Subsequently, we spot packages involved in direct and indirect cycles. Figure 5

Figure 5. An overview of a Morphic subset: Enriched cells in DSM provide a small multiple effect.

shows a large indirect cycle delimited by the pale blue area. At first glance, the bright red cells are good starting points for investigation of simple cyclic dependencies to resolve. In Figure 5 we can spot:

A **Packages in indirect cycles** (yellow bottom bar). In a first step, it is not really interesting to fix them because the cycle probably comes from a direct cycle between two other packages.

B **Packages communicating heavily.** There, the two packages interacts a lot, so intuitively it seems to be difficult to fix, the symmetric cell is probably less complex. A clear example of such situation was shown in Figure 6 where the symmetric cell is much simpler.

C **Packages referencing a lot of external classes** (a lot of red links and header with bright red number). This pattern shows direct references to classes between the two packages in cycle. The symmetrical may be easier to fix, we can assess it rapidly, since it can be marked in red. Figure 8 shows an example of such situation.

D **Packages where only two classes are referring to each other** (Thick border). Such pattern represents a direct cycle between two classes. In Figure 7, only one class of *Morphic-Worlds* is in cycle with only one class of *Morphic-Widgets*. In addition, they both have a thick border so it is clearly a direct cycle between these two classes. This pattern allows us to focus our attention on just two classes of the two packages.

E **Packages having a large percentage of classes involved in the dependency** (left bar in red). When this pattern shows a high ratio in the referencing package (top), changing it can be complex since many classes should be modified. In the case of a high ratio in referenced (bottom) package, a first interpretation is that this package is highly necessary to the good working of the referencing package.

F **Packages with direct cycles which seem easy to fix** (red bottom bar - low ratio of references). This pattern shows cycle created by a single class in one package. In Figure 8, the class labelled *Pa* is the only one appearing in *Morphic-Worlds* and both uses and is used by classes

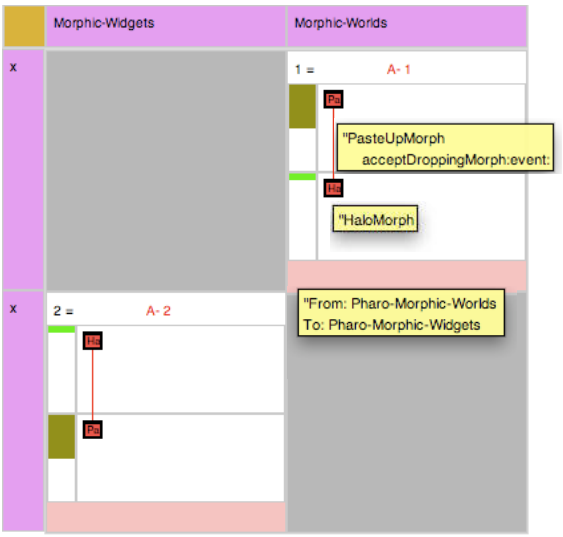


Figure 7. A twin-class cycle, with fine-grained information.

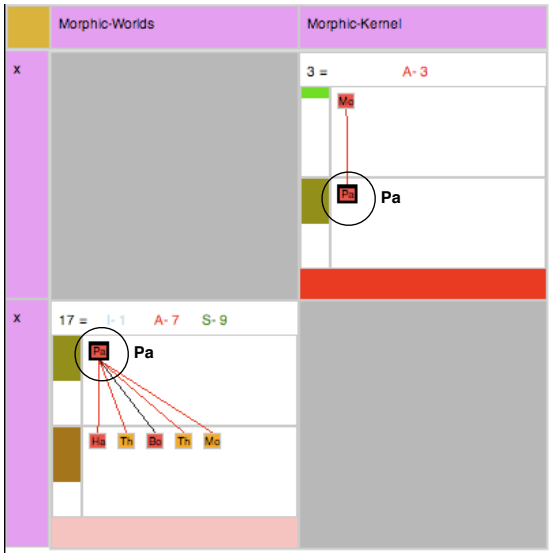


Figure 8. A one-hotspot cycle.

in *Morphic-Kernel* (as indicated by its thick border). Actually, there is a single class in *Morphic-Kernel* which links back to the *Pa* class. eDSM stresses that one class is the center of the cycle; in such a case we can focus on this class and its dependencies.

G **Packages containing classes performing a lot of invocations to other classes** (a lot of green links and header with the number in bright green).

H **Packages containing classes performing inheritance and invocation to other classes.** It means that the referencing package is highly dependent of the referenced package. Looking at the symmetric cell is good practice.

I **Packages in which a lot of classes refer to one class** (incoming funnel). This patterns shows that the dependency is not dispersed in the referenced package. It can be that the referenced class is either an important class (facade, entry point) or also simply packaged in the wrong package. If the color of the class is red, it is a central class because it is used by and it uses other packages. In Figure 9, a lot of classes reference one class which is essential for the referencing package.

J **Packages in which a lot of classes are referred by one class** (outgoing funnel). This pattern is the counterpart of the previous one. Therefore, it helps spotting important referencing classes. It is useful to check whether such a class in addition is referenced by other. In Figure 8, one class references a lot of classes.

V. Experimental Validation

We experimented and validated our approach with non-trivial case studies: ArgoUML [6], Seaside [7] and Morphic [8].

[6]http://argouml.tigris.org/
[7]http://www.seaside.st/
[8]http://wiki.squeak.org/squeak/30

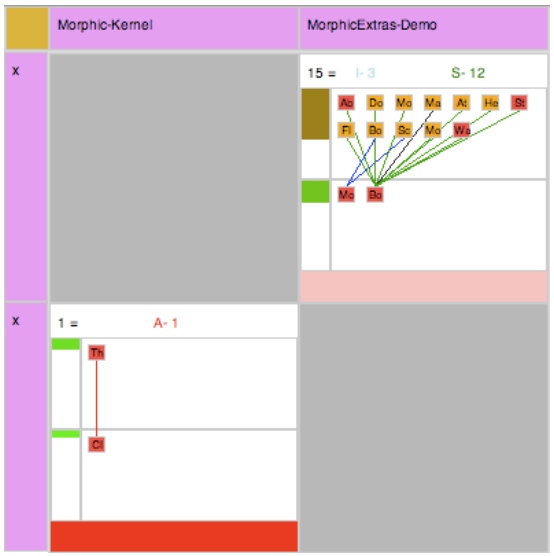


Figure 9. A funnelled cycle.

We applied DSM on ArgoUML, a Java project which had been imported in Famix3.0 with the tool iPlasma [9].

Seaside is a framework for dynamic web application development which matured over 8 years. Between version 2.8 and 2.9 the developers goal was to offer a more modular structure. They asked us to apply eDSM on the different versions. We reported the large improvement between the two versions: Seaside 2.9 contains 10 cycles when 2.8 contains 23 direct cycles. The work to fix residual cycles is underway.

[9]http://loose.upt.ro/iplasma/

ArgoUML is an open source UML modeling tool. We applied eDSM on the version 0.28 and the results include 90 direct cycles between 71 packages among the 97 imported packages. A visualization sample is available on *http://www.jannik-laval.eu/assets/files/softAnalysis/argoUml*. It shows that ArgoUML has one big indirect cycle (in blue) with 77 packages.

Morphic is a large Swing-like graphic framework comprising 46 packages and 325 classes. It supports complex interface building. It was not designed in a modular fashion and has a 12 years long history of evolution and extension, making it complex to analyze by exhibiting an abnormal number of dependencies between packages. It contains 45 direct cycles between 28 packages in the studied version. Based on the eDSM output, we proposed solutions to fix cycles. Pharo developers performed and integrated the simple fixes and acknowledged the more complex problems we identified.

In this section, we present a global analysis of *Morphic UI*. First, we explain the process to analyze the identified cycles and subsequently, we give their possible resolutions, our results as well as the validation (in terms of feedback) by the Pharo maintainers. One goal of the Morphic refactoring is to reorganize classes in simpler, conceptually cleaner packages to make its maintenance easier. Cycles are then primary targets to resolve to get layers of packages.

A. Experiment Protocol

We followed the following protocol: we analyzed each cycle found with eDSM and tried to quickly identify the opportunity to fix them. For each direct cycle, we looked for one simple solution or marked the cycle as too complex to be easily fixed. To assess the aid provided by eDSM we took as a criterion to find a solution in five minutes. Solutions include: moving a class between two packages, converting a method into a class extension, removing a useless class or method, merging packages. The solutions found were sent as fix propositions to Pharo developers for review.

We make a classification of direct cycles according to the perceived complexity. We consider that it is easier to break a cycle when there are 10 dependencies on one class instead of 10 classes with one dependency on each. Thus the classification is based on the number and type of dependencies. Note that this classification is presented here just as an indication of the situation we found. We may refine it and use it in future work for refactoring assistance.

- Monotype (Mn) cycle: a direct cycle where a cell has only a single edge between two classes, representing either reference, inheritance, or invocation. There are 21 of them in *Morphic UI*: all are class references.
- Simple (S) cycle: a direct cycle where a cell has only a single edge between two classes, representing multiple types of dependencies (reference and invocation or inheritance and others). There are 12 of them in *Morphic UI*.
- Direct cycle with two edges (2L). There are five of them in *Morphic UI*.
- Complex direct cycles—with more than two edges (CC). There are seven of them in *Morphic UI*.

Table I
RESULTS OF MORPHIC ANALYSIS

Cycle btw packages		Type	Proposition
Worlds	Xt-Flaps	S	merge packages
Worlds	Xt-Books	CC	merge packages
Worlds	Windows	S	delete method
Kernel	Xt-Flaps	S	create class extension
Kernel	Xt-Books	2L	move class to other package
Kernel	Worls	Mn	create class extension
Xt-SqueakPage	Xt-Books	S	merge packages
Xt-SqueakPage	Worlds	S	create class extension
Xt-SqueakPage	Kernel	S	create class extension
Widgets	Worlds	Mn	create class extension
FileList	Windows	S	merge packages
FileList	Kernel	2L	merge packages
Xt-Additional Widgets	Kernel	2L	move class to other package
Xt-Additional Widgets	Widgets	S	move class to other package
Xt-Support	Kernel	Mn	move class to other package
Xt-Demo	Kernel	Mn	move class to other package
Xt-Demo	Menus	Mn	delete method
Xt-PartsBin	Worlds	Mn	create class extension
Basic	Widgets	2L	create class extension
Basic	Menus	Mn	create class extension
Basic	Xt-Support	Mn	create class extension
Basic	Xt-PartsBin	Mn	create class extension
Balloon	Support	Mn	create class extension
Xt-Additionnal Support	Kernel	Mn	move method in a subclass
Xt-Additionnal Support	FileList	Mn	merge classes
Xt-Postscript Canvases	Kernel	Mn	create class extension

B. Results

We applied the previously described process on the 45 direct cycles identified in *Morphic UI* using eDSM. We proposed 25 cycle resolutions presented in Table I. Among the 20 cycles left, five cycles are judged visually too complex (with many dependencies on each side) and immediately left out; 15 cycles require a deeper exploration of the internals of *Morphic UI*, since we were not able to find a solution in five minutes. We checked our 25 proposals with one Pharo maintainer who commented, implemented or rejected them.

Among the 25 propositions, 18 have been accepted and integrated in the current Pharo release. The others are good propositions but have been replaced by merging large packages or by registration mechanisms. The propositions that were not integrated were also acknowledged as problems. However, their resolution requires more development. Here is a list of problems: missing menu registration mechanism in main tool bar, badly design button hierarchy and callback system.

Pharo developers acknowledge that the 20 cycles left out are real architectural problems: Kernel UI elements scattered over several packages with no particular reasons and left over of the monolithic architecture of Squeak.

VI. Discussion

Language Independent Approach.: eDSM has been implemented on top of the Moose reengineering environment [6] and it is based on the FAMIX language independent source code metamodel [3]. We applied eDSM on a Java case study and we expect it to work on C#, and C++ as well. Therefore while implemented in Smalltalk, eDSM can be applied to mainstream object-oriented languages.

A. Limits

There are still some limitations which we would like to overcome, with the objective to make eDSM more effective for reengineers.

When validating our proposals, the maintainer sometimes asked what was the impact of a merge or move between packages. He also asked to see other external references to packages in the cell before taking a decision. Currently we cannot show such valuable information. We plan to use a specific visualization, such as Package Blueprint [10], showing all dependencies from/to one single package in a pop-up view.

Another problem is screen space limitation. A DSM uses a lot of useless space when there are empty cells. An interactive filter on packages may be useful with respect to this.

Professional DSMs such as Lattix support layer specification and violation detection. This is orthogonal to our work but definitively relevant to add to our approach.

B. Impact and cost of small multiples

One critic about eDSM is that it loses the simplicity of the original DSM. Our experience on real and complex software showed that DSM is powerful but limited. We were constantly losing time browsing code to understand to what exact situation a number in a cell was referring to. eDSM gives such information in a glance.

A related critic about eDSM is that it looks too complex as it needs one page to describe cell design. Cells have been especially designed to work as *small multiples and micro-macro reading* [20] *i.e.,* that variations of the same structure reveal information. The presented eDSM is already the second large iteration of this work. We developed a first version of eDSM with only header informations (type and number of dependencies) and simple cell colors [1]. A fly-by-help text then gave the same information as the current cell design. However, it was cluttered: text was clearly not as efficient as little pictures. Then, it did not support small multiple effects or patterns. Comparing situations was tedious and understanding the problems too.

So, eDSM provides contextual information: in a global view, eDSM could be read similarly as the original DSM by looking the header for number of links and the bottom to see cycle context, it shows the global structure of the application. However, eDSM provides more information about the context of a dependency by displaying in a cell the complexity of the relation.

C. Comparison with other approaches

Package blueprint.: It takes the focus of a package and shows how such package uses and is used by the other package [10]. It provides a fine-grained view, however package blueprint lacks (1) the identification of cycles at system level and (2) the detailed focus on classes actually doing the cycles.

Distribution Map.: It is a visualization that can be applied to packages [7]. However, even if it removes the clutter due to edge representation Distribution Map is a generic visualization that focuses on how properties spread in a population of entities.

Oriented-graph.: Often graph-oriented visualizations are used to show dependencies among software entities. Several tools such as dotty/GraphViz, Walrus or Guess can be used. Using graph is intuitive and has a fast learning curve. One problem with oriented graph visualization is finding a good layout scaling well on large sets of nodes and edges: such a layout needs to preserve the readability of nodes, the ease of navigation following edges, and to minimize edge crossing. Even then, cycle identification is not trivial.

With DSM the visualization structure is preserved whatever the data size is, which enables the user to dive fast into the representation using the normal process. Cycles remain clearly identified by colored cells, there are no edges between packages, so this reduces clutter in the visualization. Moreover, eDSM enables fine-grained information about dependencies between packages. Classes in source package as well as in target package are shown in the cells of the DSM.

Dependence Clusters.: Brinkley and Harman proposed two visualizations for assessing program dependencies, both from a qualitative and quantitative point of view [**?**]. They identify global variables and formal parameters in software source-code. Subsequently, they visualize the effect dependencies. Additionally, the MSG visualization [**?**] helps finding *dependence clusters* and locating avoidable dependencies. Some aspects of their work is similar to our own. Granularity and the methodology employed differ: they operate on source-code and use slicing method, while we focused on coarse grained entities and use model analysis.

VII. Conclusion

This paper enhances Dependency Structure Matrix (DSM) using *small multiples*. First, colors are used to distinguish direct and indirect cycles. Second, cell contents are enriched with the nature and strength of the dependencies as well as with the classes involved. Such enhancements are based on small multiples [20] and preattentive visualization principles [19], [12], [13], [21]. Thanks to these improvements, package organization and cycles are made explicit. We applied the eDSM on a complex system and systematically checked and tried to fix the cycles. Out of 45 direct cycles, we could propose 25 solutions to break the cycles. 18 got accepted and

implemented by the maintainers of the Pharo open-source Smalltalk.

We believe this paper provides an appealing approach for identifying cycles. The experiment we conducted gave us the feeling that indirect cycles were more difficult to analyze than direct ones. This makes our future work focuses on getting better visualizations for indirect cycles. Currently, eDSM provides relevant indications for reengineers, but it appears that visualizing impact of changes in the matrix would greatly enhances reengineering tasks.

Acknowledgements: We gratefully acknowledge the sponsoring of ESUG (the European Smalltalk User Group) http://www.esug.org/.

REFERENCES

[1] A. Bergel, S. Ducasse, J. Laval, and R. Peirs. Enhanced dependency structure matrix for moose. In *Proceedings of FAMOOSr 2008*, 2008.

[2] A. Bergel, S. Ducasse, and O. Nierstrasz. Analyzing module diversity. *Jour. of Universal Computer Science*, 11(10):1613–1644, 2005.

[3] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

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

[5] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *ICSM 2006*. IEEE Comp. Society, 2006.

[6] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, pages 55–71. 2005.

[7] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *CSMR'06*, pages 35–44. IEEE Comp. Society Press, 2006.

[8] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, 2005.

[9] S. Ducasse, A. Lienhard, and L. Renggli. Seaside: A flexible environment for building dynamic web applications. *IEEE Software*, 24(5):56–63, 2007.

[10] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. pages 94–103. IEEE Comp. Society, 2007.

[11] D. Gebala, S. Eppinger, and M. Cambridge. Methods for analyzing design procedures. *Design Theory and Methodology*, 1991.

[12] C. G. Healey. Visualization of multivariate data using preattentive processing. Master's thesis, Department of Computer Science, University of Bristish Columbia, 1992.

[13] C. G. Healey, K. S. Booth, and E. J. T. Harnessing preattentive processes for multivariate data visualization. In *GI '93: Proceedings of Graphics Interface*, 1993.

[14] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030, 2006.

[15] J. Michaud, M.-A. Storey, and H. Muller. Integrating information sources for visualizing Java programs. In *ICSM'01*, pages 250–259. IEEE, 2001.

[16] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPSLA'05*, pages 167–176, 2005.

[17] D. Steward. The design structure matrix: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management*, 28(3):71–74, 1981.

[18] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[19] A. Treisman. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing*, 31(2):156–177, 1985.

[20] E. R. Tufte. *Visual Explanations*. Graphics Press, 1997.

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

[22] J. Wu and M.-A. D. Storey. A multi-perspective software visualization environment. In *CASCON '00*, page 15. IBM Press, 2000.

[23] A. Yassine, D. Falkenburg, and K. Chelst. Engineering design management: an information structure approach. *International Journal of Production Research*, 37(13):2957–2975, 1999.