Identifying cycle causes with Enriched Dependency Structural Matrix

Jannik Laval[1], Simon Denier[1], Stéphane Ducasse[1], Alexandre Bergel[2]
[1]RMoD Team, INRIA - Lille Nord Europe
USTL - CNRS UMR 8022, Lille, France
[2] Computer science department (DCC), University of Chile, Santiago, Chile
{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). eDSM cells are enriched with contextual information about (i) the type of dependencies (*e.g., *inheritance, class access), (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 has been implemented on top of the Moose reengineering environment and the Mondrian visualization framework. It has been applied to a non-trivial case study, the Morphic UI framework available in two open-source Smalltalks, Squeak and Pharo. Problems identified by eDSM have been performed and retrofitted in Pharo main distribution.*

1 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, metrics, their structure and their evolution. Distribution Map [4] shows how properties are spread over an application. Lanza et al. [6] propose to recover high-level views by visualizing relationships. Package Surface Blueprint reveals the package internal structure and relationships among other packages – surface represents relations between the analyzed package and its provider packages. Dong and Godfrey [3] 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 [14]. 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 [15, 16, 12]. MacCormack and al. [13] 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 [20]. Unfortunately, this algorithm inaccurately identifies cycles since independent cycles are merged. Another limit is that DSM current implementations do not take class extensions[1] into account, which are used in a number of languages including Objective-C, Ruby, Smalltalk, C# 3.0.

Our contribution is two-fold: first, we identify weaknesses of current DSM (Section 2); second, we address these weaknesses (Section 3). We propose a DSM with enriched cells[2]. eDSM cells contain contextual information which shows (i) the nature of dependencies (inheritance, class access, invocation, and class extension), (ii) the en-

[1]A class extension is one or more methods defined in a package on a class living in a different package.

[2]a DSM cell represents the intersection of two packages

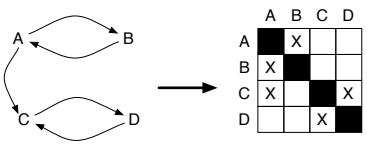


Figure 1. A simple DSM.

tities performing references, (iii) the entities being referenced. We distinguish independent cycles and differentiate cycles using colors. We applied eDSM on a large system, *Morphic UI* framework and the results of the analysis were integrated in Pharo[3], a new version of Squeak.

The paper is organized as follows: Section 2 introduces DSM and its limitations in existing implementations. Sections 3 and 4 present eDSM specifications and its usage, from overview of an application to detailed view of interpackage dependencies. Section 5 shows an experiment on *Morphic UI* and identifies some patterns and solutions found in the study. Section 6 discusses limitations of our solution and future work. Section 7 concludes.

2 Limitations of DSM

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 [14]. However, in their current form, DSMs must be coupled with other tools to offer fine-grained information.

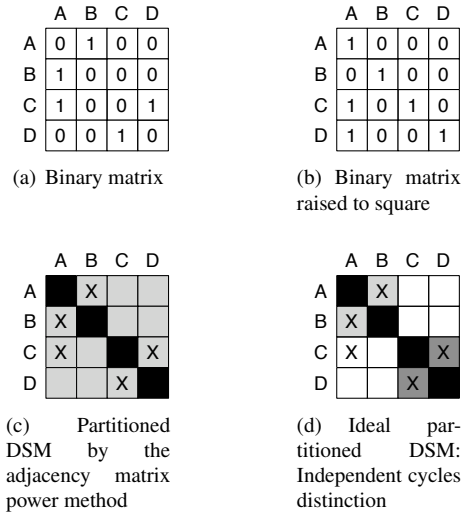
Figure 1 shows a sample dependency graph and its corresponding binary DSM. A binary DSM shows the existence/lack of a dependency (or reference) by a mark or "1/0". The rule for reading the matrix is: element in column header references element in row header if there is a mark. In our context, A, B, C, 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 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 a number of limitations with current DSM implementations: inaccurate merging of independent cycles by the adjacency matrix power method (Section 2.1), lack of fine grained-overview (Section 2.2) and lack of support for class extensions (Section 2.3).

2.1 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 [20]. A non-zero mark in the diagonal of the power matrix points to elements involved in a cycle of length n.

[3] http://www.pharo-project.org



(a) Binary matrix

(b) Binary matrix raised to square

(c) Partitioned DSM by the adjacency matrix power method

(d) Ideal partitioned DSM: Independent cycles distinction

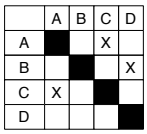
Figure 2. Limitation of the adjacency matrix power method. Cycles are shown in gray.

Figure 1 shows two distinct direct cycles: one between A and B and one between C and D. Figure 2(a) shows the binary matrix of the DSM and Figure 2(b) shows the matrix raised to the square. The diagonal of 1 indicates that all elements are involved in one or more direct cycles. However, the adjacency matrix power does not separately identify these different cycles, resulting in a single and inaccurate merged cycle. Figure 2(c) shows the partitioned matrix with a unique wrong cycle—a single gray zone representing the cycle. On the contrary, Figure 2(d) shows a correctly partitioned matrix with two distinct cycles.

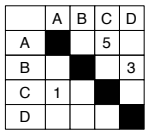
The adjacency matrix power method produces inaccurate results when used to identify independent cycles because it computes the number of edges to come back to an element without considering the cycling path [8]. Instead, path searching algorithms have also been used to detect cycles in DSM and should be systematically preferred when the problem of identifying independent cycles is important.

2.2 Lack of fine-grained information

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



(a) DSM with marks



(b) DSM with numbers

Figure 3. Examples of references in a DSM.

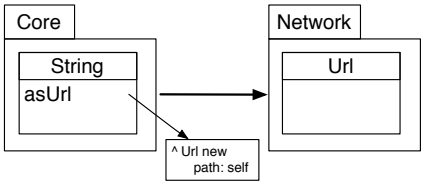
Dependency causes. Fixing a cycle often means changing some dependencies involved in the cycle. However, the cost of fixing a cycle may vary with the cause of dependency *e.g.,* changing a direct reference to a class is often easier than changing an inheritance relationship. Dependencies are of different natures (direct class access, method invocation, inheritance relationship, and class extension) and a binary matrix (Figure 3(a)) or a matrix providing the number of dependencies in each cell (Figure 3(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.

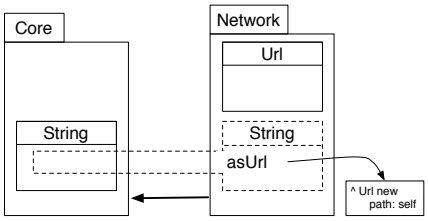
Dependency distribution. Knowing that a package has 78 references to another one (package *Morphic-Widgets* on *Morphic-Basic* in Figure 9) is a valuable but insufficient information. Such references could be done by a large number of classes or few classes and these 78 references could refer to a small number or a large number of classes. This additional information 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 9, 16 classes and 41 methods of package *Morphic-Widgets* reference 10 classes and 42 methods of package *Morphic-Basic*, while only two classes and three methods of *Morphic-Basic* reference one class and no method 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.

2.3 Class extensions not supported

A class extension is a method defined in a package, for which the class is defined in a different package [1]. Class extensions exist in Smalltalk, CLOS, Ruby, Python, Objective-C and now in C#. They offer a convenient way to incrementally modify existing classes when subclassing is inappropriate (Figure 4). AspectJ inter-type declarations offer a similar mechanism.



(a) Dependency without class extension (Core depends on Network)



(b) Reversed dependency with class extension (Network depends on Core)

Figure 4. Principle of class extension.

Note that extending by subclassing may be inappropriate when one can not modify the source code or client classes to make it refer to a new subclass. In addition class extensions are really powerful to layer applications because they allow one to revert package dependencies. In Figure 4, instead of creating a method in the package *Core* which creates a dependency between the packages *Core* and *Network*, we extend the class *String* in the package *Network* with the method *asUrl*. Consequently, the dependency between the two packages is reversed.

Thus, it is important that the tools supporting DSM construction correctly identify the direction of dependencies such as class extensions.

3 Enriched DSM (eDSM)

To address the problems previously mentioned, we enhance DSM with functionalities that are not present in the Lattix DSM implementation [14]. Our solution (i) isolates independent cycles using colors (Section 3.1), (ii) enriches contextual cell information (Section 3.2) and (iii) supports class extensions (Section 3.2).

In particular, enriched cells act as *small multiples* [18] where similar looking side by side little visualizations provide a differentiating effect (see Figure 7). 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 and the Mondrian visualization framework. Since it is based on the FAMIX meta-model

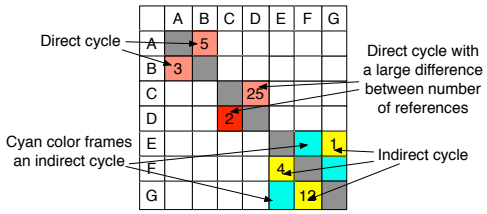


Figure 5. Cell color definition.

[2], our eDSM works for mainstream object-oriented programming languages [5].

3.1 Enhanced cycle detection

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

Cycle distinctions. eDSM distinguishes independent cycles using a path searching algorithm [8]. With this method, two independent cycles are detected separately and remain isolated from each other in the DSM (Figure 2(d)).

Indirect cycles. We use color in DSM cells to identify cycles. Indeed, as shown in Figure 5, DSM cells involved in a cycle have a yellow or red color. 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 frames all cells involved in an indirect cycle (visible in Figure 7). Its area visual indicates the number of packages in the cycle. On the contrary, rows and columns with white or gray colors indicate packages not involved in a 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 so far in the current version of our work.

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 5). 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 5 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. These cycles are distinct since there is no red box on rows A and B coming from columns 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 information displayed in a cell, as we explain in the following section.

3.2 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* [18] as shown in Figure 7.

Overall structure of an enriched cell. An enriched cell is composed of four parts (Figure 6). 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 to its target classes.

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

- *Dependency strength and nature (top row).* The top row gives a simple summary of the number and nature of dependencies to get an idea of their strength. We show the total number of dependencies (Tot), inheritance dependencies (Inh), direct accesses to classes (Acc), invocations (Msg), and method extensions (Ext) made by the source package to the target one. In Figure 9 there are 78 directed dependencies from *Morphic-Widgets* to *Morphic-Basic*.

- *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 in order 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, which makes it a candidate for merging both packages in some cases.

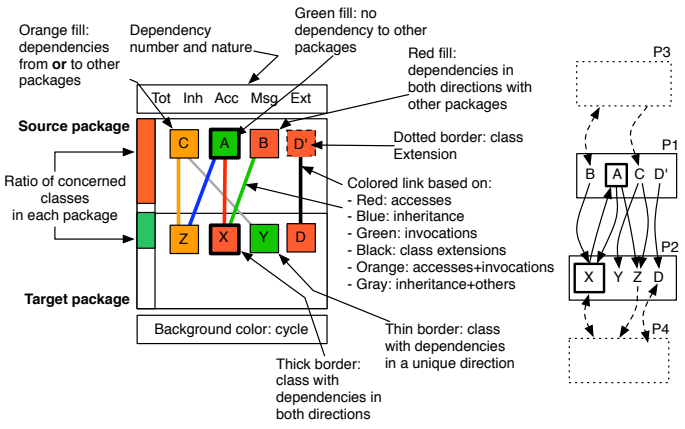


Figure 6. Enriched cell structural information.

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 visualization[4] as much as possible to help spotting important information [17, 9, 10, 19]. An enriched cell is composed of parts and shapes with different color schemas.

Cycle color (bottom row). The bottom row represents cycle information using color as explained in Section 3.1. 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.

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 6).

- *Color fill.* A class may be in dependency with other packages than the two represented by the cell, such as classes B or C in Figure 6. The color fill uses the metaphor 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 class can change their respective external dependencies.

- *Border color and thickness.* The square border thickness also conveys information: 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 6, 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 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 6, 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).

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 6). There are four basic natures, each one mapped to a primary color: access in red, inheritance in blue, invocation in green and class extension in black. When dependencies between two classes are of different natures, colors are mixed as follows: orange is used for a dependency with both accesses and invocations, and gray is used for any dependency involving inheritance with accesses and/or invocations. Indeed, an inheritance dependency mixed with other dependencies can be quite complex and we choose not to focus on such combination.

Representation of class extension. Class extension represents a method which is in another package than its class (Section 2.3). In a cell, a class extension is represented by a square with dotted border and the same color information than the original class. This convention exists because a class extension is not a class. The situation in Figure 6 shows all possible colors.

[4]Researchers in psychology and vision have discovered a number of visual properties that are preattentively processed [9]. 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).

Figure 7. An overview of a Morphic subset: Enriched cells in DSM provide a small multiple effect.

4 From overview to detailed views

Cells have been especially designed so that they work as *small multiples* [18] *i.e.,* that variations of the same structure reveal information. 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 as well as in the following sections.

4.1 Small multiples at work

Figure 7 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 break.

The first use of the eDSM is to get a system overview (Figure 7) to scan packages not involved in cycles (not shown in Figure 7) and how they interact with other packages. Subsequently, we spot packages involved in direct and indirect cycles: a pale blue area delimits cells in cycling packages. In Figure 7 we can spot:

A packages in indirect cycles (yellow bottom bar).

B packages communicating heavily.

C packages containing classes with a lot of accesses to other classes.

D packages where only two classes are referring to each other.

E packages having a large percentage of classes involved in the dependency (left bar).

F packages with direct cycles which seems easier to fix (low ratio of references - red bottom bar).

G packages containing classes performing a lot of invocations to other classes.

H packages containing classes performing inheritances and invocations to other classes.

I packages in which a lot of classes refer to one class (incoming funnel).

J packages in which a lot of classes are referred by one class (outgoing funnel).

Browsing the overview and accessing more detailed views is supported by direct interaction with the mouse. These views can be for example class blueprint or any polymetric views [11].

4.2 Interaction and detailed view

Fly-by help. More precise information about the dependencies is given using fly-by help on the cell and its elements. This information includes the full name of concerned packages, the name of classes and the name of each concerned method. Figure 8 shows the pop-up information of the cell linking *Morphic-Basic* to *Morphic-Widget*: there are three accesses to class *HandleMorph* from the methods *PolygonMorph.customizeArrows:*, *TextMorph.setCurveBaseline:*, and *TextMorph.changeMargins:*.

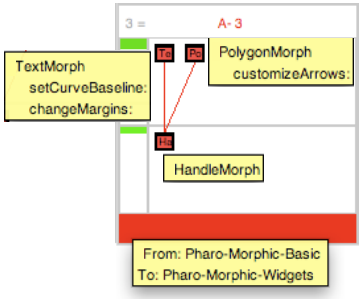


Figure 8. Fine-grained information in a cell.

Zooming on two packages. Each cell in a DSM represents a single direction of dependency. To get the full picture of a direct cycle, we compare two cells, one for each direction. Despite the symmetry intrinsic to a DSM, it is not always easy to focus on the two concerned cells. We provide a zoom which pops up a detailed view with the two concerned cells, as shown in Figure 9. Thus, we focus on a direct cycle which seems interesting from the overview, and analyze the details with the zooming view.

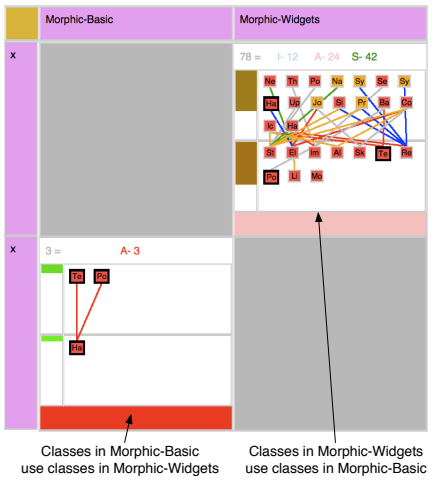


Figure 9. Zoom on two packages in cycle.

4.3 Visual patterns

eDSM supports the understanding of the general structure of complex programs. Since it is based on the idea of small multiples [18], the cell visual aspect generates visual patterns. While performing our *Morphic* experiment, we have detected some patterns stressing characteristic situations. We present three patterns.

One-hotspot cycle. A first pattern is a cycle created by a single class in one package. In Figure 10, the class labelled *Pa* is the only one appearing in *Morphic-Worlds* and both uses and is used by classes 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. We can focus on this class and its dependencies.

Twin-class cycle. The second pattern is a direct cycle between the same two classes. In Figure 11, 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 is more specific than the previous one. We focus our attention on just two classes of the two packages.

Funnelled cycle. The third pattern is a cycle with a funnelled cycle. It is a cycle with a lot of classes which reference one class (Figure 12) or one class which references a lot of classes (Figure 10).

This pattern shows the importance of a simple class in other packages. It is certainly a complex or a central class.

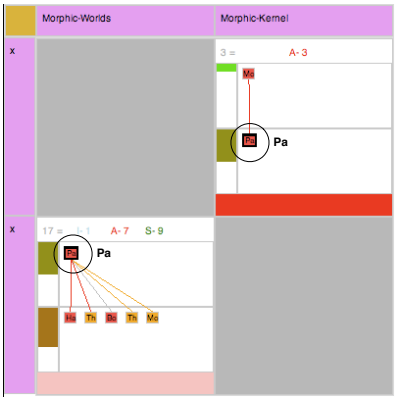


Figure 10. A one-hotspot cycle.

Figure 11. A twin-class cycle.

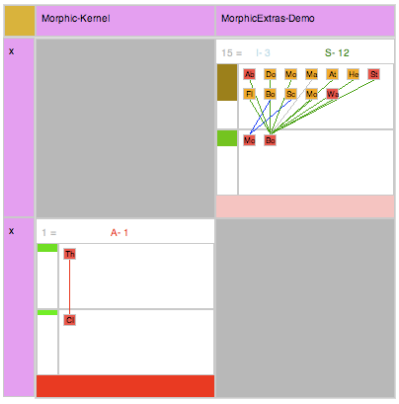


Figure 12. A funnelled cycle.

5 Morphic experiment

We experimented and validated our approach with a non-trivial case study. We refactored the Morphic framework in the Pharo open-source Smalltalk. Morphic is a graphic framework comprising 46 packages and 325 classes. 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 a great deal of dependencies between packages. It contains 45 direct cycles between 28 packages in the studied version.

In this section, we first present a global analysis of *Morphic UI*. Subsequently, we show how we proceed to analyze the identified cycles and as well as the results we obtained and validated by the maintainers of Pharo.

5.1 Morphic analysis

One goal of the Morphic refactoring is to reorganize classes in simpler, conceptually cleaner packages to make its maintenance easier. Cycles are then the primary target to remove to obtain layers of packages.

We analyze each cycle found with eDSM and try to quickly identify the opportunity to remove them. For each direct cycle, we look for one simple solution or mark the cycle as too complex to be easily fixed (the criterion is 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.

- Monotype (Mn) cycle: it is a direct cycle where a cell has only a single edge between two classes, representing either access, inheritance, or invocation. There are 21 of them in *Morphic UI*: all are direct class references.
- Simple (S) cycle: it is a direct cycle where a cell has only a single edge between two classes, representing multiple types of dependencies (access 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*.

5.2 Study of a cycle

The case study is based on Figure 9. It is a direct cycle between *Morphic-Widgets* and *Morphic-Basic* (named

Cycle between packages:		Type	proposition
Worlds	Extras-Flaps	S	merge packages
Worlds	Extras-Books	CC	merge packages
Worlds	Windows	S	delete method
Kernel	Extras-Flaps	S	convert a method into a class extension
Kernel	Extras-Books	2L	move class in another package convert a method into a class extension
Kernel	Worlds	Mn	convert a method into a class extension
Extras-SqueakPage	Extras-Books	S	convert a method into a class extension OR merge packages
Extras-SqueakPage	Worlds	S	convert a method into a class extension
Extras-SqueakPage	Kernel	S	convert a method into a class extension
Widgets	Worlds	Mn	convert a method into a class extension
FileList	Windows	S	merge packages
FileList	Kernel	2L	merge packages
Extras-AdditionalWidgets	Kernel	2L	move class in another package
Extras-AdditionalWidgets	Widgets	S	move class in another package
Extras-Support	Kernel	Mn	move class in another package
Extras-Demo	Kernel	Mn	move class in another package or delete class
Extras-Demo	Menus	Mn	delete method
Extras-PartsBin	Worlds	Mn	convert a method into a class extension
Basic	Widgets	2L	convert a method into a class extension
Basic	Menus	Mn	convert a method into a class extension
Basic	Extras-Support	Mn	convert a method into a class extension
Basic	Extras-PartsBin	Mn	convert a method into a class extension
Balloon	Support	Mn	convert a method into a class extension
Extras-AdditionnalSupport	Kernel	Mn	move method in a subclass
Extras-AdditionnalSupport	FileList	Mn	merge class
Extras-Postscript Canvases	Kernel	Mn	convert a method into a class extension

Figure 13. Results of Morphic analysis.

Widgets and *Basic* below).

We can see that the *Widgets* package has a lot 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 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 accesses. The fly-by help (Figure 8) displays for each class in the cell the concerned methods (methods in the source package making class accesses 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 accessed.

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 as explained in Section 2.3, effectively breaking the cycle.

5.3 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 Figure 13. 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 our proposal. Among the 25 propositions, 18 have been accepted and integrated in the current Pharo release.

6 Discussion

6.1 Limits

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

One limit is the lack of semantic information. This limit is a common defect of visualizations based solely on structural information. For now, cell information is only about class and method structure (inheritance, invocations). We plan to annotate dependencies with semantic information to improve the refactoring experience.

Our approach shows dependencies *between* packages. *Internal* dependencies between classes of one package are not displayed. To avoid overloading the eDSM overview, we consider using a zooming view showing the internals of the package.

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 can not show such valuable information. We plan to use a specific visualization, such as Package Blueprint

[7], 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.

6.2 Comparison with an oriented-graph

An oriented graph is generally used to show dependencies including cycles. It is intuitive and has a fast learning curve. One problem with oriented graph 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.

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. Moreover, eDSM enables fine-grained information about dependencies between packages. Classes in source package as well as in target package can be shown in the cells of the DSM.

7 Conclusion

This paper enhances Dependency Structure Matrix (DSM) using *small multiple*. 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 [18] and preattentive visualization principles [17, 9, 10, 19]. 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 a change forecast would greatly enhances reengineering tasks.

References

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

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

[3] X. Dong and M. Godfrey. System-level usage dependency analysis of object-oriented systems. In *ICSM 2007: IEEE International Conference on Software Maintenance*, pages 375–384, Oct. 2007.

[4] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[5] S. Ducasse, T. Gîrba, A. Kuhn, and L. Renggli. Meta-environment and executable meta-language using Smalltalk: an experience report. *Journal of Software and Systems Modeling (SOSYM)*, 2008.

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

[7] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

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

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

[10] 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.

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

[12] A. Lopes and J. L. Fiadeiro. Context-awareness in software architectures. In *Proceeding of the 2nd European Workshop on Software Architecture (EWSA)*, volume 3527 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2005.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

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