SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2014; **44**:235–257
Published online 20 November 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2164

Resolving cyclic dependencies between packages with enriched dependency structural matrix[‡]

Jannik Laval[1,*,†] and Stéphane Ducasse[2]

[1]*Université Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France*
[2]*RMoD Team, INRIA Lille Nord Europe, USTL, CNRS UMR 8022, Lille, France*

SUMMARY

Dependency structural matrix (DSM) is an approach originally developed for process optimization. It has been successfully applied to identify software dependencies among packages and subsystems. A number of algorithms have been 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 that is crucial to support remediation tasks. We propose enriched DSM (eDSM), which provides small-multiple views and micro–macro-readings by adding fine-grained information in each cell of the matrix. Each cell is enriched with contextual information about (i) the type of dependencies (inheritance, class reference, etc.), (ii) the proportion of referencing entities, and (iii) the proportion of referenced entities. We distinguish independent cycles and stress potentially simple fixes for cycles by using coloring information. This work is language independent and has been implemented on top of the Moose software analysis platform. In this paper, we expand our previous work. We improved the cell content information view on the basis of user feedback and performed multiple validations: two different case studies on Moose and Seaside software; one user study for validating eDSM as a usable approach for developers. Solutions to problems identified with eDSM have been performed and retrofitted in analyzed software. Copyright © 2012 John Wiley & Sons, Ltd.

Received 7 June 2011; Revised 9 October 2012; Accepted 10 October 2012

KEY WORDS: software visualization; reengineering; dependency structural matrix; package; dependency analysis

1. INTRODUCTION

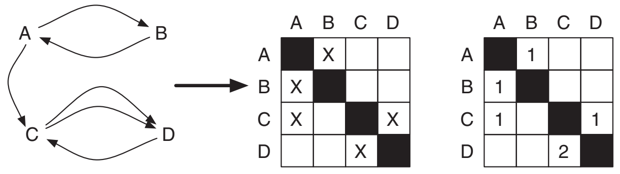
Understanding the package organization of an application is a challenging and critical task because it reflects the application structure. Many approaches have flourished to provide information on packages and their relationships, by visualizing software artifacts metrics, their structure, and their evolution [1]. Distribution maps [2] show how properties are spread over an application. Ducasse and Lanza [3] proposed to recover high-level views by visualizing relationships. Package surface blueprints [4] reveal the internal structure of a package and its relationships with other packages—surfaces represent relations between an analyzed package and its provider packages. Dong and Godfrey [5] proposed high-level object dependency graphs to represent and understand the system package structure.

These techniques do not provide information about package cycles. A package cycle is a strong coupling between multiple packages that prevents the developer from separating these packages.

*Correspondence to: Jannik Laval, Université Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France.

†E-mail: jannik.laval@mines-douai.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.*



$$\begin{array}{c|cccc} & A & B & C & D \\ \hline A & \blacksquare & X & & \\ B & X & \blacksquare & & \\ C & X & & \blacksquare & X \\ D & & & X & \blacksquare \end{array} \qquad \begin{array}{c|cccc} & A & B & C & D \\ \hline A & \blacksquare & 1 & & \\ B & 1 & \blacksquare & & \\ C & 1 & & \blacksquare & 1 \\ D & & & 2 & \blacksquare \end{array}$$

Figure 1. The graph on the left is represented by a DSM. Such DSM can be a binary one or one representing weight.

For example, in Figure 1 left, packages A and B depend on each other in a cycle. A similar cycle is present between C and D. This situation can cause important problems when maintaining a software system, particularly when a developer wants to replace a feature represented by a package by another one. Addressing cycles in software is not a new problem: Martin in his book defined heuristics to reduce cycles between layers as well as reconsider the size of packages. Some of the object-oriented reengineering patterns such as introducing registration mechanism instead of using conditional [6] also address the problems of cycles in software. Cycle understanding and identification is also an important activity (i) when specifying the quality of software systems [7] and (ii) when remodularizing existing software systems as shown by the work around Jigsaw,[§] the new module model for Java. In regression and integration testing domain, engineers have to know the order of integration. Consequently, cycles between classes or components to integrate cause problems, and engineers can have trouble to choose the classes or components to integrate first. Algorithms are proposed to order the class integration with the goal of minimizing the impact of cycles [8–11].

In the context of remodularization, Bavota *et al.* [12] proposed to compute metrics on the basis of relations at class level to reorganize packages. The problem of cycles was also treated at the class level by Melton and Tempero [13, 14]. The authors proposed to identify dependency cycles between classes to identify refactoring candidates. They affirmed that long cycles are difficult to understand and have a strong impact on the cost of maintenance. They particularly indicated that it is crucial to take into account the semantics of the software architecture to avoid breaking dependencies that should not be broken.

To highlight and understand the problem of package cycles, dependency structural matrix (DSM) provides an interesting dependency-based approach. DSM is an approach originally developed for process optimization. It highlights dependencies among tasks and patterns to detect problems. It has been successfully applied to identify software dependencies [15–19] and particularly cycles [19]. MacCormack *et al.* [20] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux. It provides a dependency-centric view possibly associated with color to perceive characteristics of dependencies [21].

Applied to package dependencies, DSM organizes packages in column and row headers and places in each cell of the matrix a dependency. A nonempty cell indicates that the package in the column depends on the package in the row.

In this paper, we improve DSM visualization to provide fine-grained information about package dependencies. We propose eDSM, a DSM with enriched cells. eDSM cells contain contextual information that shows (i) the *nature* of dependencies (inheritance, class reference, invocation, and class extension), (ii) the *referencing* entities, (iii) the *referenced* entities, and (iv) the *spread* of the dependency. We distinguish independent cycles and differentiate cycles by using colors. We applied eDSM on several large systems, the *Moose* software analysis platform,[¶] *Seaside 2.8*[‖] (a dynamic web framework), and *Pharo*[**] (an open-source implementation of Smalltalk programming language and environment). To evaluate the usability of the eDSM visualization, each study has a different context.

[§]http://openjdk.java.net/projects/jigsaw/
[¶]http://www.moosetechnology.org
[‖]http://www.seaside.st
[**]http://www.pharo-project.org

We expand on our previous work [22] with an improvement of cell visualization and with multiple validations of the approach. In [22], we proposed a first visualization and a first case study. In this paper, (i) we improve the visualization taking into account comments from users and results of the first validation, (ii) we perform two case studies to show that eDSM is usable as a valid information system for remodularization, and (iii) we perform a survey to measure and assess user experience feedback. This survey has been applied on nine different systems.

The tool is implemented on top of the Moose software analysis platform. Because it is based on the FAMIX meta-model [23], eDSM can work on mainstream object-oriented programming languages [24]. FAMIX includes multiple extractors (particularly for Java and C#) that allow one to match object-oriented systems with the FAMIX meta-model. The extraction of the source code is based on standard static analyst, which is not a part of our work and is not detailed in this paper.

The paper is organized as follows: Section 2 introduces DSM and its limitations in existing implementations. Section 3 presents our solution for macro-reading. Section 4 presents the structure of an enriched cell. Section 5 shows how eDSM support cycle understanding from overview to detailed view. Section 6 reports three different studies and shows the usability of eDSM visualization. Section 7 discusses about our solution. Section 8 concludes the paper with perspectives and future work.

2. DSM PRESENTATION AND LIMITATIONS

A DSM is an adjacency matrix built from graph. In our case, we consider packages as the nodes of the graph. A link from a source package A to a target package B means that A (i.e., the client package) depends on B (i.e., the provider package).

The rule for reading the matrix is as follows: elements in column depend on elements in row when there is a mark. In Figure 1, A, B, C, and D are packages. An element in the column is also called the client and the one in the row the provider. In Figure 1, A references B and C, B references A, C references D, and D references C.

In a DSM, rows and columns are ordered using algorithms - *partitioning algorithms* [25] or *clustering algorithms* [26]. They optimize the organization of elements in the matrix. We structure the matrix to show core level packages on bottom right and dependent packages on top left. To order packages in this configuration, we use simple *partitioning algorithms*. In case of cyclic dependencies, we compute a weight for each line and column, which is the sum of nonempty cells, and we push the highest weighted package on the bottom right of the cycle. An example of this ordering is presented in Figure 3. It is not the topic of this paper, so we do not write anymore about it.

The use of a DSM gives pertinent results for software component independence verification [19]; however, in their current form, a DSM must be coupled with other tools to offer fine-grained information and support corrective actions. A DSM provides poor level of information related to the scope and reason of identified cycles.

2.1. Dependency information

A traditional DSM offers a readable general overview of dependencies but does not provide details about the situation it describes. The software engineer cannot use the dependency matrix to get answer to questions such as the cause of this cycle. Other authors reported weak points of DSM. Cai and Huynh [27] used DSM to represent modular structure before and after changes. They pointed three weaknesses of DSM: first, it is not expressive enough to support precise design analysis; second, it only represents high-level structure dimensions; and third, a DSM does not reveal the multiple ways to perform a change. Current implementations of DSM allow one to perform high-level inventory of a situation, but they are limited to coarse-grained understanding—tools just offer drop-down lists to show classes and methods creating dependencies between packages.

We identify the weaknesses around two points that will be addressed by eDSM: lack of information on dependency causes and lack of information on dependency distribution.

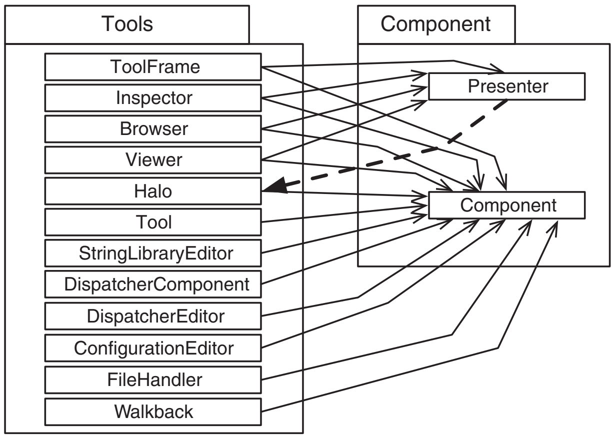


Figure 2. A cycle between Tools and Component, two packages of Seaside 2.8.

2.2. *Dependency cause evaluation*

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, for example, 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 1 middle) or a matrix providing the number of dependencies in each cell (Figure 1 right) does not provide such information.

Annotating a DSM with the type 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, providing fine-grained information for understanding cycles, without sacrificing the overall view of architecture.

2.3. *Dependency distribution evaluation*

Knowing that a package has 31 dependencies to another one is valuable but insufficient information (Figure 3). The ratio of concerned classes in a package is important because 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 many classes with few dependencies. This simple heuristic is used on our DSM to help reengineers to fix cycles.

For example in Figure 2, 12 classes of package *Tools* refer to two classes of package *Component* using 31 different dependencies, whereas only one class of *Component* refers to one class of *Tools* (the dashed arrow in Figure 2). Consequently, it should be easier to focus the dependencies from *Component* to *Tools* rather than the ones in the opposite direction.

3. MACRO-READING: COLORED DSM

To address the lack of fine-grained information mentioned in Section 2, we introduce enriched cells in DSM. We enhance DSM with functionalities that are not present in current DSM implementations such as Lattix [19]. Our solution provides a micro–macro-reading [28] by (i) highlighting independent cycles using colors for a *macro-reading* as presented in this section and (ii) understanding interpackage dependencies with a *micro-reading* visualization presented in Section 4.

Enriched cells resolve DSM limitations informing reengineers about the dependency cause and the dependency distribution evaluation. An important design feature is the use of color to focus on dependencies where it seems easier to resolve a package cycle. Therefore, we use brighter colors for places having fewer dependencies. Our approach enhances the traditional matrix by providing

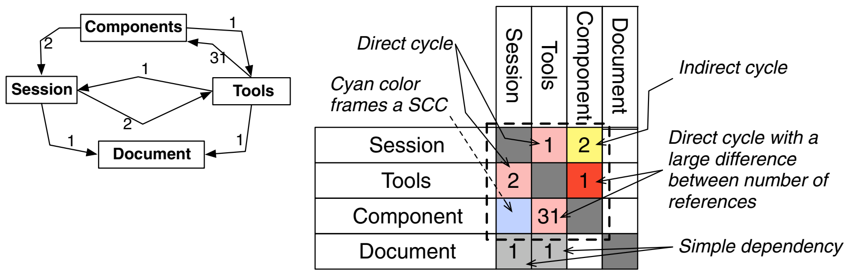


Figure 3. A graph and its colored DSM representation (the dashed area represents a strongly connected component).

a number of new features: *cycle distinctions*, *direct and indirect cycle identification*, and hints for *fixing cycles*.

First, the set of nodes in a cycle is named a strongly connected component (SCC) in graph theory. eDSM distinguishes them using the Tarjan algorithm [29], which has a linear complexity.

Then, we use color in DSM cells to identify cycles. A pale blue square surrounds the area in the matrix representing the SCC (visible in Figure 4). Each pale blue cell means that the packages are involved in an SCC even if there is no dependency between these exact two packages. This area is a visual indication of the number of packages in the cycle. We use a dashed box in Figure 3 to represent it.

Finally, on top of this square, a dependency (i.e., a nonempty cell) involved in an SCC has a red/pink or yellow color (Figure 3).

A red/pink cell means that the two concerned packages refer each other and thus create a direct cycle. In a direct cycle, the two red/pink cells are symmetric against the diagonal. We differentiate red and pink cells as follows: we indicate some places where it seems easier to start addressing the cycles (red cells). We define a special rule to highlight cells of primary focus when resolving direct cycles. 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). To differentiate red and pink cells, we compute the dependency weight, and we use a ratio of 1–3 to highlight a lighter dependency with red cell. This rule is only applied 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.

A yellow cell means that the dependencies from one package to the other participate in an SCC: it represents the fact that the dependency takes place in an indirect cycle.

Finally, rows and columns with white or gray colors indicate packages not involved in any cycle. A white cell means that there is no dependency and that the two packages are not involved in an SCC. A gray cell means that there is a dependency but no cycle. The diagonal of the matrix, where a package may reference itself, is colored in gray to highlight the symmetry axis.

A real example is presented in Figure 4. The Moose software analysis platform is composed of 78 packages. The figure shows that the system contains eight SCCs of different size (eight multicolored squares all along the diagonal of the matrix).

4. ENRICHED CONTEXTUAL CELL INFORMATION

Each cell is the intersection between a client package and a provider package. To give a detailed overview of dependencies, we propose to build a small, specific view in each cell.

The goal is to create *small-multiples* as shown in Figure 8. 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* [28]. Our goal is also to use preattentive visualization as much as possible to help spotting important information [30–33]. An enriched cell is composed of parts and shapes with different color schemas.

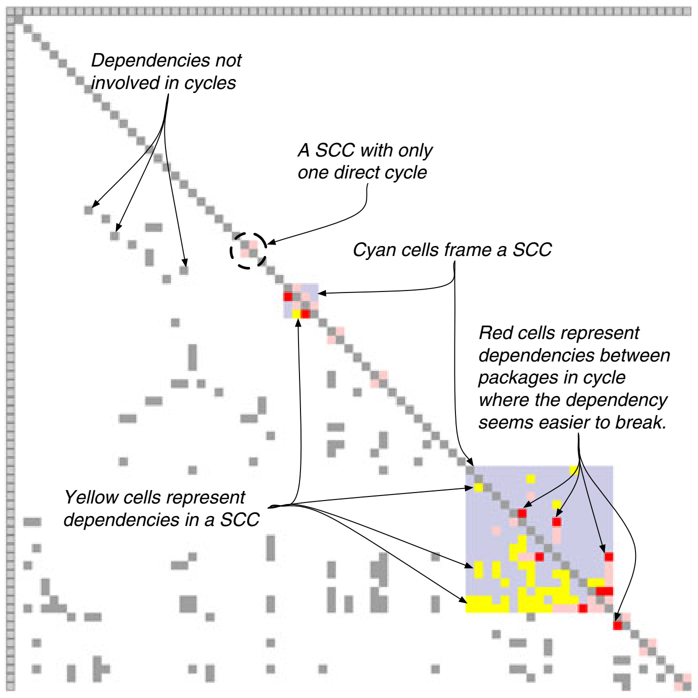


Figure 4. A DSM with the 78 packages of the Moose software analysis platform (case study presented in Section 6.1).

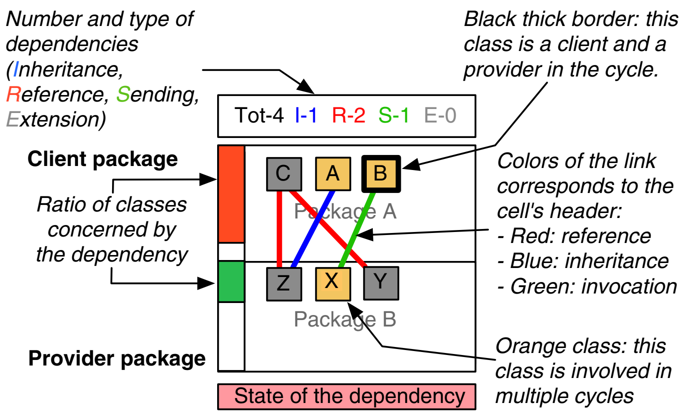

Number and type of dependencies (Inheritance, Reference, Sending, Extension)
Black thick border: this class is a client and a provider in the cycle.
Tot-4 I-1 R-2 S-1 E-0
Client package
Ratio of classes concerned by the dependency
C
A
B
Package A
Colors of the link corresponds to the cell's header:
- Red: reference
- Blue: inheritance
- Green: invocation
Z
X
Y
Package B
Provider package
Orange class: this class is involved in multiple cycles
State of the dependency

Figure 5. Enriched cell structural information.

Each cell represents a small context, which enforces comparison with others. Each cell represents a small dashboard with indicators about the situation between the client and the provider. It makes use of colors to convey more information about the context in which dependencies occur. This section focuses on the cell contents.

4.1. *Overall structure of an enriched cell*

The cell content displays all dependencies at class level from the client package (header of the column) to the provider package (header of the row).

An enriched cell is composed of three parts (see Figure 5): (i) the identification of cycles (explained in Section 3) is represented by a colored frame on the bottom row (i.e., red, pink, yellow, or gray), (ii) the top row gives an overview of the strength and nature of dependencies between classes into the two involved packages, and (iii) the two large boxes in the middle detail class dependencies going from the top box to the bottom box (i.e., from the *client package* to the *provider package*). Each box contains squares that represent involved classes: referencing classes in the client package and referenced classes in the provider package. Dependencies between squares link each client class (in top box) to its provider classes (in bottom box) (Figure 5).

4.2. *Dependency overview (top row)*

An enriched cell shows an overview of the strength, nature, and distribution of the dependencies from the client to the provider.

The top frame gives a summary of the number and nature of dependencies to get an idea of their strength. It shows the total number of dependencies (Tot) in black, inheritance dependencies (I) in blue, references to classes (R) in red, invocations (M) in green, and class extensions (E) made by the client package to the provider one in gray. A stronger color highlights the lightest dependency type to help reengineers targeting the minimal effort to do. The colors are used to reinforce the comprehension of links between classes (see the following text). In Figure 6, the cell representing the dependency from *Platform* to *HTTP* has eight directed dependencies: one inheritance (in blue) and seven references (in bright red).

4.3. *Content of the dependency (middle boxes)*

The core of a cell is composed by two boxes: one (on top) for the client package and one (on bottom) for the provider package. Classes in the client package refer to classes in the provider package. Each of these boxes is structured as follows:

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.

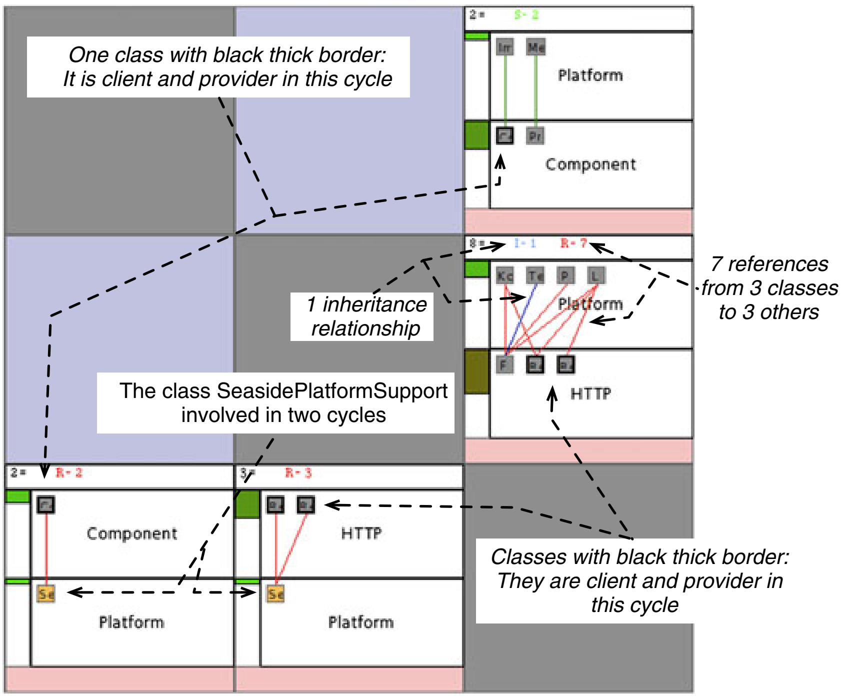

One class with black thick border: It is client and provider in this cycle
Platform
Component
7 references from 3 classes to 3 others
1 inheritance relationship
Platform
HTTP
The class SeasidePlatformSupport involved in two cycles
Component
Platform
HTTP
Platform
Classes with black thick border: They are client and provider in this cycle

Figure 6. Zoom on three packages in cycles.

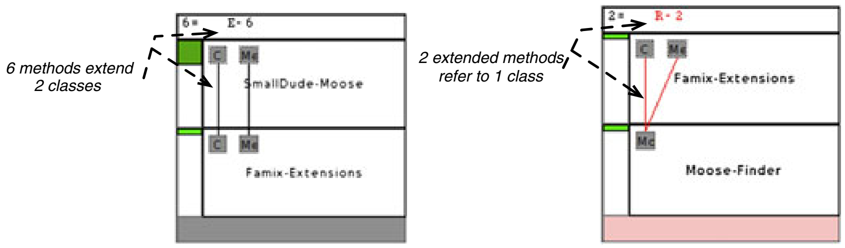


Figure 7. Left—the package *SmallDude-Moose* extends two classes from *Famix-Extensions*. Right—two class extensions in *Famix-Extensions* refer to one class in *Moose-Finder*.

Class color. Each square represents a class and displays two types of information by using its border as well as its fill color (Figure 5). (i) The class border color and thickness represent the internal usage. A gray thin border means that the class has a unidirectional dependency with the other package, that is, it either uses *or* is used by classes in the other 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 enriched cell (not necessarily the same classes). In Figure 6, three classes (WAResponse and WARequest in one cycle and WAComponent in the other one) have a thick border because they reference a class from *Platform* and they are referenced by two classes from *Platform*. (ii) The class color fill represents the impact of the class in the system. A class may be in dependency with other packages than the two represented by the cell, such as class SeasidePlatformSupport in Figure 6. The color fill uses orange to qualify the relationships the class has with packages other than the two concerned. A class that is implied in other cycles is displayed as orange. Classes that do not correspond to this description are in gray. Thus, reengineering orange classes can have an impact on several cycles.

Edge color. Edges are the smallest details displayed by enriched cell. They give information on the nature and spread of dependencies between classes (Figure 5). There are four basic natures, each one mapped to a primary color (synchronized with colors of information in top row of the enriched cell): reference in red, inheritance in blue, invocation in green, and class extension in gray. When dependencies between two classes are of different natures, colors are mixed as follows: red is used for a dependency with both references and invocations because a reference is often followed 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 complex, and we choose not to focus on such a combination.

Representation of class extension. A class extension[††] represents a method that is in another package than its class. In an enriched cell, a class extension is represented by a square with dotted border (Figure 7) because it represents methods in another package than the class definition. We differentiate two pieces of information about extensions: first, a client package has an extended method of a class defined in a provider package. In this case, there is an extension link between the class and its extension (in gray). In Figure 7, the two classes are extended in package *SmallDude-Moose*. Here, six methods from these classes are defined in package *SmallDude-Moose*. Second, a client package uses an extended method whose class is defined in a third package. In this case, there is no extension link but could have access or invocation dependencies. In Figure 7, the two class extensions in *Famix-Extensions* refer to one class in *Moose-Finder*.

[††]A class extension is a method defined in a package, for which the class is defined in a different package [34]. 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 intertype declarations offer a similar mechanism.

4.4. Interaction

We presented the structural elements of eDSM both at macro and at micro levels. Now, we show that to make the approach usable, a good interaction is offered. Although the eDSM offers an overview at the package level as shown by Figure 8, extracting all the information from an enriched cell is sometimes difficult. There is a clear tension between getting a small-multiple effect and detail readability. We offer three kinds of zoom and fly-by-help to improve usability. (i) We can zoom 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 9. Thus, we focus on a direct cycle that seems interesting from the overview and analyze the details with the zooming view. This functionality is available by double clicking on a cell. (ii) We can zoom on dependencies from a particular package: each package has several dependencies. To get the full picture of interactions with a specific package, we watch all cells in the concerned column to know the outgoing dependencies or the concerned row to know the incoming dependencies. For a big DSM, it is not always easy to focus on a specific package. We provide a selective zoom with a detailed view on a package and all its dependencies. This functionality is available by double clicking on a package cell (i.e., row or column header). (iii) We can zoom on an SCC: the analyzed system can have multiple SCCs. When the reengineer needs to focus on fixing dependencies in a specific SCC, he does not need to see the entire eDSM. We provide a selective zoom with a detailed view on an SCC. It selects an SCC and shows it in a new eDSM. This functionality is available by double clicking on an SCC cell (i.e., a blue cell).

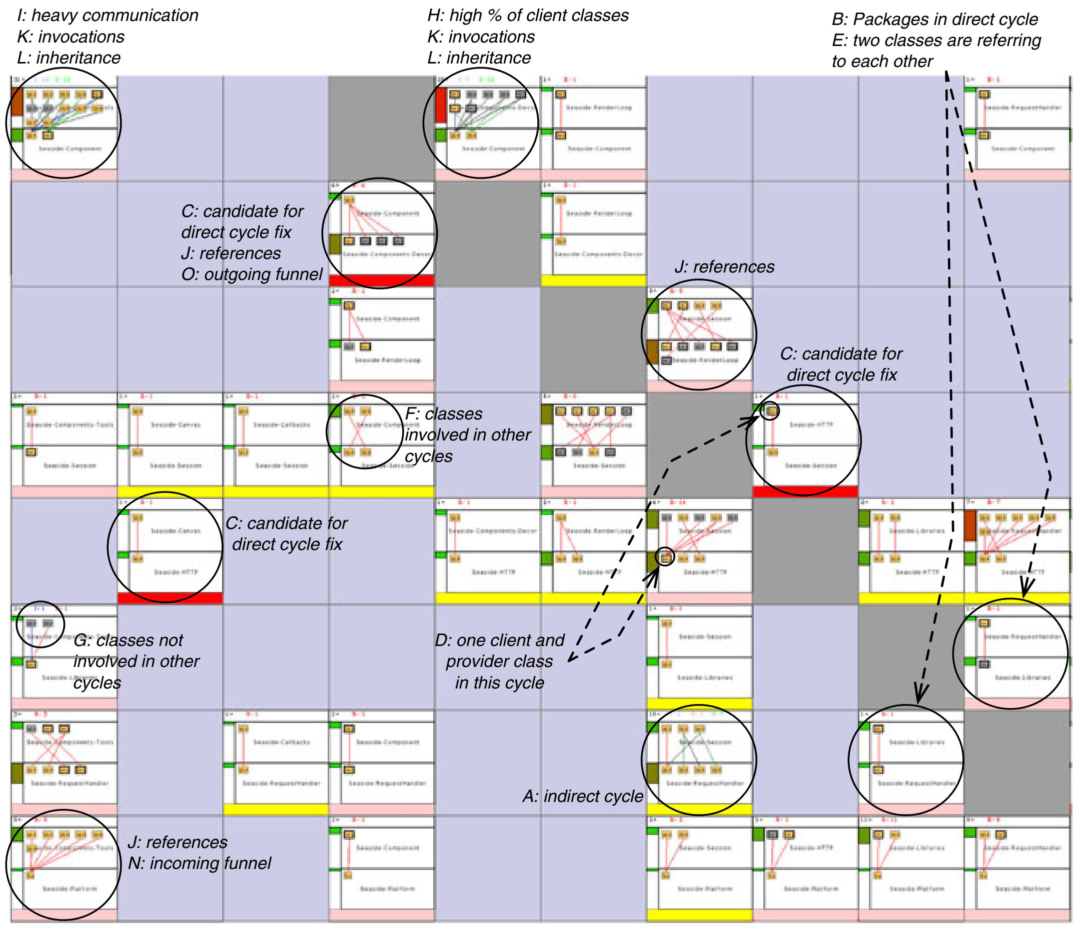

I: heavy communication
K: invocations
L: inheritance
H: high % of client classes
K: invocations
L: inheritance
B: Packages in direct cycle
E: two classes are referring to each other
C: candidate for direct cycle fix
J: references
O: outgoing funnel
J: references
C: candidate for direct cycle fix
F: classes involved in other cycles
C: candidate for direct cycle fix
G: classes not involved in other cycles
D: one client and provider class in this cycle
A: indirect cycle
J: references
N: incoming funnel

Figure 8. An overview of a Seaside subset: enriched cell in DSM provides a small-multiple effect.

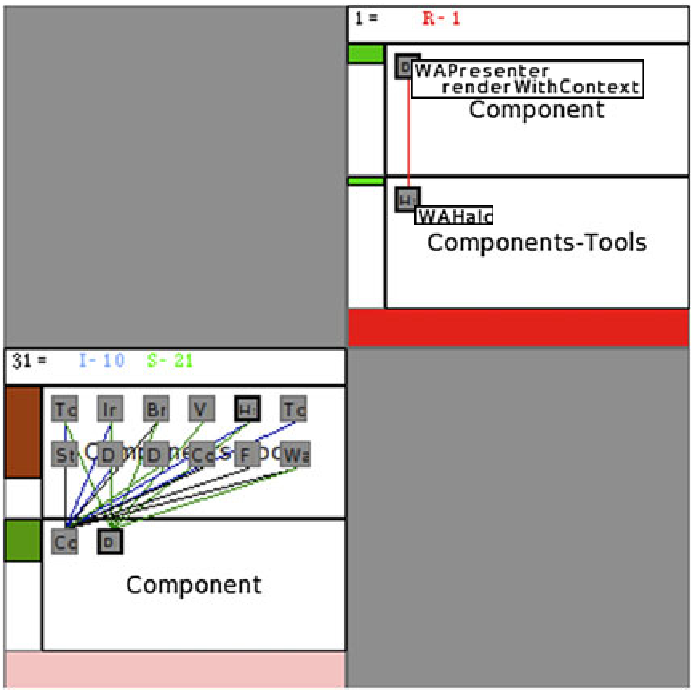


Figure 9. A cycle with good candidate dependency to remove.

Complementary to the overview and the zooming facility, tooltips (shown in Figure 9) on a class include the name of the class and the name of each concerned method. Moreover, a Tooltip is available on edges showing the source code of the class or/and method that create the dependency.

5. ENHANCED DSM AT WORK

In this section, we show how eDSM helps to get a first overview and then understand more detailed aspects of a cycle.

5.1. *Small-multiples at work*

Enriched DSM (eDSM) supports the understanding of the general structure of complex programs using structural element position. Because it is based on the idea of small-multiples [28], the enriched cell visual aspect generates visual patterns.

To show eDSM in practice, we applied it to the Seaside 2.8 open-source dynamic web framework[‡‡] (Figure 8). The case study is reported in Section 6.2. It is composed of 33 packages and 358 classes. It has a large number of cyclic dependencies.

The first goal of the eDSM is to get a system overview to identify packages not involved in cycles (not shown in Figure 8) and how they interact with other packages. Subsequently, we spot packages involved in direct and indirect cycles. In Figure 8, we spot some visual patterns.

A. *Packages with indirect cycles* (yellow bottom bar). It is not a good starting point to fix them because the cycle could be resolved by removing a direct cycle between two other packages.
B. *Packages with direct cycles* (pink bottom bar). These dependencies are diagonally symmetric. These dependencies should be fixed, but the reengineer should select manually the best candidate dependency to remove.

[‡‡]http://www.seaside.st/

C. *Packages with direct cycles with a good candidate dependency to fix* (red bottom bar—low ratio of references). This pattern shows a cycle created by a single class in one package. In Figure 9, the class labeled *WAPresenter* is the only one appearing in *Component* and both uses and is used by classes in *Components-Tools* (as indicated by its thick border). Actually, there is a single class in *Components-Tools* that links back to the *WAPresenter* 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.
D. *One class involved as client and provider in a cycle* (black thick border). The cycle is created by a single class. Figure 6 shows an example of such situation. It means that the reengineer should invest in analyzing this class before the rest of the classes involved in the cycle.
E. *Packages where only two classes are referring to each other* (thick border). Such pattern represents a direct cycle between two classes. The both classes have a thick border, so it is a direct cycle between them. This pattern allows us to focus our attention on just two classes of the two packages.
F. *Classes involved in multiple cycles.* When a class is involved in multiple cycles, its background is orange. It means that when we change this class, it could probably impact other cycles. In Figure 6, the class SeasidePlatformSupport is involved in two cycles.
G. *Classes not involved in other cycles.* When a class is not involved in other cycles, its background is gray. It means that when we change this class, there will be no impact on existing cycles. In Figure 6, there are several classes in gray.
H. *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 because many classes should be modified. In the case of a high ratio in the referenced (bottom) package, a first interpretation is that this package is highly necessary to the good working of the referencing package.
I. *Packages communicating heavily.* The two packages interact heavily, so intuitively, it seems to be difficult to fix this dependency. The opposite may be easier to fix.
J. *Packages referencing a large number of external classes* (many red links and header with bright red number). This pattern shows direct references to classes between the two packages.
K. *Packages containing classes performing numerous invocations to other classes* (many green links and header with the number in bright green).
L. *Packages containing classes inheriting from other classes.* It means that the referencing package is highly dependent of the referenced package. Looking at the opposite cell is a good practice.
M. *Packages with a large number of extensions.* It means that the referencing package extends the referenced package. It represents additional features for the referenced package (absent in Figure 8).
N. *Packages in which a large number of classes refer to one class* (incoming funnel). This pattern shows that the dependency is not dispersed in the referenced package. An interpretation is that the referenced class is either an important class (facade, entry point) or simply packaged in the wrong package.
O. *Packages in which a large number 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.

Browsing the overview and accessing more detailed views are supported by direct interaction with the mouse. These views can be for example class blueprint or any polymetric views [35] as well as source code.

5.2. Fixing a cycle with eDSM

We now detail an example of cycle resolution using eDSM. It begins with understanding eDSM in the large to select a good candidate. Then, we detail how to understand a couple of direct cycle enriched cells.

First, using full eDSM, we look for red enriched cells because they represent situation where the number of dependencies in a cycle is unbalanced. Red cells are good candidates because they are in direct cycle and they have much less dependencies than their counterpart does. In Figure 8, three red enriched cells seem to be good candidates for removing cyclic dependencies. Among these three dependencies, two have single dependency between classes.

When there is no red cell in a direct cycle, it is more difficult to define the better candidate dependency to remodularize. Therefore, for each direct cycle, the reengineer should select manually the best candidate.

Now, let us take an example. In Figure 9, there is a direct cycle between *Component* and *Components-Tools* packages (named *Component* and *Tools* in the following text). We can see that *Tools* has several dependencies to *Component* (pink enriched cell), whereas only one class in *Component* uses one class of *Tools* (red enriched cell). Moreover, there is one red edge (class reference) in the red enriched cell, whereas in the pink cell, they are multiple inheritances and multiple invocations.

At first glance, it is thus easier to investigate the dependency of the red enriched cell, from *Component* to *Tools*. Let us look at the red enriched cell. There is one referencing class and one referenced class. The two classes have a bold border, which means they are involved in the two directions of the cycle. It means these two classes represent the core of the cycle. In Figure 9, it is visible that these two bold classes are present in the two enriched cells.

More precisely, the dependency is composed by only one reference in the method WAPresenter.renderWithContext: to the class WAHalo. This provides entry points in the source code to find precisely where the provider class WAHalo is referenced. It appears that the method WAPresenter.renderWithContext: contains the creation of an instance of WAHalo. A possible solution is to create a class extension for WAPresenter in the package *Tools* and to put the referencing method in it. Then, the dependency would be reversed, effectively breaking the cycle. This solution was proposed and accepted by Seaside developers (see Section 6.2).

6. VALIDATION

We report on three different studies performed to collect feedback and validate our approach. Such studies have been performed on small and large software applications. The first case study confirms that eDSM is useful when a reengineer knows the source code and knows how to use eDSM. The second study shows that eDSM helps understanding and fixing a software system without knowing the source code. The third study is a user study that shows that enriched cell is a comprehensive tool for developers. The situations evaluated are resumed in Table I.

6.1. Moose case study

Goal. The Moose open-source analysis platform is a well-maintained platform with multiple libraries. We expect few cycles, and such cycles should be simple to fix because developers are highly sensitive to the problem. The goal of this experiment is to validate the approach on a well-known case study.

Presentation. In this case study, we used eDSM to analyze cycles between packages in the Moose reengineering system version 4b4. The system has 78 packages and 723 classes. There are 25 packages in cycle creating eight SCCs. The SCCs involve respectively 2, 2, 2, 2, 2, 2, 4,

Table I. Summary of the situation evaluated.

	Participants are experts of the system	Participants are eDSM experts
Study 1	Yes	Yes
Study 2	No	Yes
Study 3	Yes	No

and 9 packages. There are 17 direct cycles. Figure 4 shows the DSM of this system. During this case study, we used the whole functionalities of eDSM.

Protocol. The goal is to remove from the architecture all cycles using eDSM. We perform the study in two steps: (i) as direct maintainers of the Moose platform, we could readily validate whether each cycle was a problem or not, find the problematic dependencies, and propose a solution to eliminate these dependencies; and (ii) before implementing the solution, each proposition was sent to, and validated by, the Moose community.

Topology. At the end of the study, there were four direct cycles left. We did not remove them because they are required for testing some tools such as eDSM.

For the rest of the system, we provide 22 propositions to remove cycles to the community (in Table II). All these propositions have been accepted and integrated in the source code.

Conclusion of the case study. The results of this case study show that eDSM is adapted to understand problems in a known system. The Moose reengineering platform had cycles that are not critical. They have been removed with simple actions on the source code.

6.2. *Controlled comparative study: Seaside*

Goal. In this comparative study, the goal is to validate the information provided by eDSM, by comparing conclusion given using eDSM and the remodularization work performed by engineers without eDSM.

Presentation. Seaside 2.8 is an open-source dynamic web framework. Between version 2.8 and 3.0, the developers explicitly reengineer the application in modular packages. One of the goals of Seaside 3.0 was to refactor the architecture so that it can be deployed on seven different Smalltalk systems in a modular way. In particular, Seaside maintainers wanted to have no package cycles. This is why

Table II. Propositions provided to the Moose community.

Type: iterative development
extend method FAMIXClass.browseSource() in Moose-Finder.
extend method FAMIXMethod.browseSource() in Moose-Finder.
move class MPImportSTCommand in Moose-Wizard.
move class MPImportJavaSourceFilesWithInFusionCommand in Moose-Wizard.
extend method FAMIXNamedEntity.isAbstract() in Famix-Extensions.
extend method FAMIXNamedEntity.isAbstract:(Object) in Famix-Extensions.
extend method FAMIXClass.isAbstract() in Famix-Extensions.
extend method CompiledMethod.mooseName() in Famix-Implementation.
extend method CompiledMethod.mooseNameWithScope:(Object) in Famix-Implementation.
extend method FAMIXPackage.definedMethods() in Famix-Extensions.
extend method FAMIXPackage.extendedClasses() in Famix-Extensions.
extend method FAMIXPackage.extendedClassesGroup() in Famix-Extensions.
extend method FAMIXPackage.extensionClasses() in Famix-Extensions.
extend method FAMIXPackage.extensionClassesGroup() in Famix-Extensions.
extend method FAMIXPackage.extensionMethods() in Famix-Extensions.
extend method FAMIXPackage.localMethods() in Famix-Extensions.
extend method FAMIXPackage.localClasses() in Famix-Extensions.
extend method FAMIXPackage.localClassesGroup() in Famix-Extensions.
Type: evolution
extend method MooseModel.mseExportationTest() in Moose-SmalltalkImporterTests.
move class MooseScripts in Moose-SmalltalkImporter.
Type: message not sent
remove reference checkClass() refers to MooseModel.
remove method FAMIXClass.ascendingPathTo(Object).

we choose Seaside to perform a comparative study. Seaside 2.8 contains 33 packages, 358 classes, 2 SCCs (one with 3 packages and one with 22 packages), and 25 direct cycles. During this case study, we used eDSM without accessing to the source code.

Protocol. In this case study, we import a model of Seaside 2.8, and we do not access the source code. The goal is to use only eDSM to remove all cycles in the model. Then, we sent to Seaside developers our propositions for cycles removal. They analyzed the validity of the propositions. They provided four types of answer: (i) *accepted and integrated in Seaside 3.0* (true positive): it represents the best validity for our case study as the developers have already detected and integrated; (ii) *accepted but not integrated*: it represents a good proposition, but the developers have implemented another solution. The difference is due to our lack of knowledge of the system; (iii) *refused* (false positive): engineers refuse the change because proposed changes break the semantic of the system; and (iv) *no control on the package*: some packages we analyzed are not controlled by the Seaside team. Consequently, the developers cannot evaluate the validity of the proposition.

Results. We proposed 71 actions to be performed. It took us 7 hours to remove all cycles in the system. Table III shows a summary of our propositions by types of action. We proposed to extend 42 methods, to move 22 classes in other packages, to merge 5 packages, and to create 2 packages and move classes or extend methods in them.

Seaside developers accepted 39 propositions (Table IV): 17 propositions were already integrated in Seaside 3.0. Twenty-two propositions were accepted but not integrated because developers have implemented other solutions. Six propositions have not been evaluated because developers have no control on the packages involved, so they have no idea of the structure. Finally, 26 propositions (37%) have been refused particularly because of our lack of knowledge of the studied system. These propositions would break the meaning of the expected architecture.

Conclusion of the case study. In this case study, we wanted to stress that contrary to the first case study, we performed the study without any knowledge of the architecture of Seaside and without access to the source code. In our opinion, this study points to the data-to-information quality of eDSM, both extracting the global picture and showing the right amount of details.

In this study, Seaside was already remodularized, which offered a good feedback. Results prove that eDSM helps detecting structural problems and provides enough information to resolve them. The Seaside developers have accepted 55% of the propositions, and 8% were out of control. The part of refused proposition (37%) is acceptable considering the challenge of reengineering an unknown system.

Table III. Summary of proposed actions.

Proposition type	Number
Extend a method	42
Move a class	22
Merge two packages	5
Add a package	2

Table IV. Validity of propositions.

Validity type	Number
Accepted and already integrated in Seaside 3.0	17
Accepted, not integrated in Seaside 3.0	22
Refused	26
No control on the structure	6

Enriched DSM provides information about problems; it does not inform about possible solutions. Solutions are proposed manually. We can consider that eDSM provides enough information to help remodularizing a system but does not replace the engineer knowledge or source code browser.

6.3. User study

Goal. In this case study, we perform a user study to validate eDSM as a usable tool for nonexpert in visualization. Contrary to the two first case studies, which validate eDSM features by an expert, this user study targets uses on a wider range of systems. We evaluate each feature of eDSM.

Used tools. For this user study, we wrote a small tutorial about eDSM (available on http://www.moosetechnology.org/docs/eDSM) and a questionnaire. The developers use eDSM on their own developed software systems and answer questions. They can only use eDSM and the software source code (i.e., no other tool for software analysis). The participants were recruited on a Smalltalk developer mailing list.

Protocol. We provide the users with the eDSM tool, the questionnaire, and the tutorial. The questionnaire has 36 questions. We organize the questionnaire in eight parts.

1. Participant characteristics: requesting general information about the user experience. There are four questions. (i) Are you aware of the package structure of your application? (ii) Are you an expert of the system you will analyze with our tool? (iii) Are you skilled using visualizations? (iv) Do you use software visualizations in general?
   Possible answers are: *strongly disagree*, *disagree*, *agree*, or *strongly agree*.
2. Characteristics of the system: we ask the name of the system and five simple metrics that are provided by a small script in eDSM. They are as follows: number of packages of the software, number of classes, number of packages in cycle, number of SCCs, and size of each SCC.
3. Time spent using eDSM: this single question asks the time users used eDSM.
4. Usefulness of simple eDSM to understand the system: as eDSMs are known to help understanding a structure and the adding of colors should help identifying structural problems, we ask if the eDSM helps to see the structure of the application and if it helps to identify critical dependencies.
   Possible answers are as follows: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.
5. Pattern identification in eDSM: this part of the questionnaire tests the usefulness of eDSM. We are particularly interested in which small-multiple patterns the user paid attention. We ask if the developer identify places where he could cut a cycle, packages where most of the classes are involved in cycles, packages where only one class is creating a cyclic dependency, packages that should be merged, and cycles he want to keep.
   Possible answers are as follows: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.
6. Usefulness of enriched cell in general. Enriched cell has two goals: understanding and resolving. We ask if enriched cells are useful to understand dependencies and if they are useful to fix a dependency.
   Possible answers are: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.
7. Use and usefulness of enriched cell features: in this part, we want to investigate the usefulness of each enriched cell feature in details. For eight features (color of enriched cell, header of enriched cell, name of package in background of enriched cell, ratio of concerned classes, color of classes, border of classes, color of edges, and popup information), we ask if he uses this feature and if he considers this feature useful.
   Possible answers are as follows: the first question is a yes–no question. The second question needs an answer between 1 (not useful) and 5 (very useful) on a Likert scale.
8. Open questions: we ask some opened questions to collect more information about the use of eDSM and the perception of the user. The questions are asked in two times: (i) at the beginning of the questionnaire, what is your goal in this experiment for your application (identify cycles,

layers, and hidden dependencies)? What are your expectations in using this tool? What is the size of your screen? (ii) At the end of the questionnaire, did you complete your goal? If not, why? Did DSM help you? Which applications do you use to see your software structure and its problems? Which features of the eDSM need to be improved? Which features of the tool are useless?

Results. The user study was conducted with nine participants unsupervised, from master students to experience researchers, with various programming skills and experienced in software projects. We selected them with two criteria: the time they have to investigate eDSM and the size of the maintained system. In fact, when the maintained system has one package, eDSM is useless. Two users performed the study on Pharo, which explains why Pharo appears two times in Figure 11 and the following.

In this section, we detail the results of each part of the questionnaire.

1. Participant characteristics (Figure 10): answers show that answering engineers are experts of these software applications. They are also concerned by the package structure. Their experience with visualization tools exists but is not excessive.
2. Characteristics of the systems (Table V): there are eight software systems evaluated with eDSM. There are two developers working on the Pharo Smalltalk environment and one developer for each of other systems. All these systems are developed in Smalltalk. Table V shows that systems have different size. All of them have cycles.
3. Time spent using eDSM (Figure 11): in this study, we do not differentiate the learning time from the use time. The longer the participants work on eDSM, the more precise the analysis is.

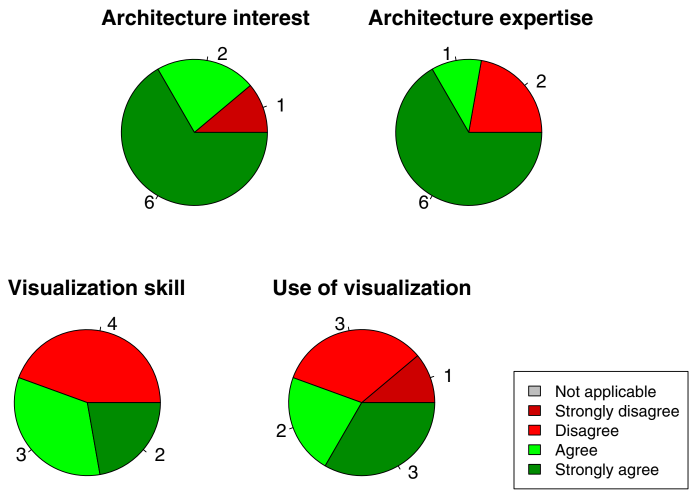


Figure 10. Characteristics of participants.

Table V. Some metrics about studied software applications.

Software name (kind of software)	Packages (number in cycle)	Classes	SCCs (size)	Direct cycles
Pharo (Smalltalk environment)	104 (68)	1558	1 (68)	98
Mondrian v.480 (visualization engine)	20 (8)	149	2 (2–6)	7
Merlin (wizard library)	5 (4)	31	1 (4)	3
Moose (software analysis platform)	108 (21)	1428	6 (2–2–2–2–4–9)	19
Seaside 3.0 (web framework)	93 (5)	1408	2 (2–3)	2
SqueakDBX (database)	3 (3)	88	1 (3)	1
XMLParser (XML library)	8 (4)	33	1 (4)	4
PolyMorph (UI library)	12 (4)	152	1 (4)	4

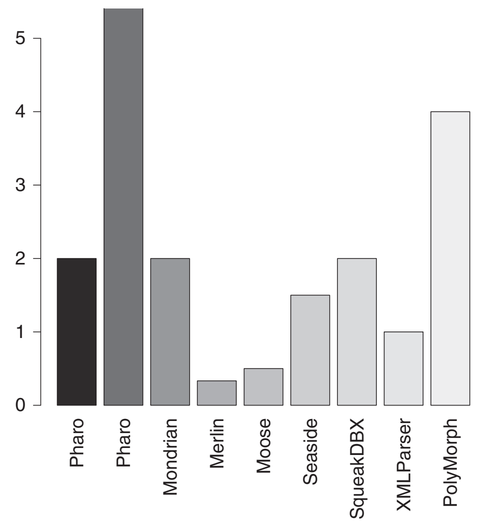


Figure 11. Time spent for each project (in hours).

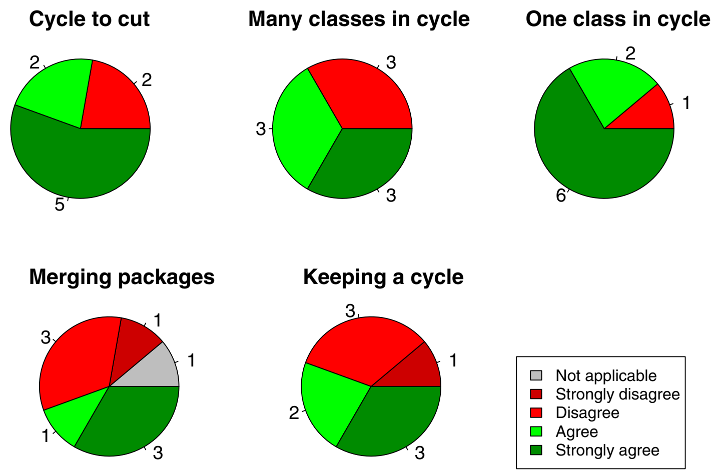


Figure 12. Pattern identification in eDSM.

The time varies between tens of minutes to more that 5 hours. We can note that the time spent using eDSM is not related to the size of the system.

4. Usefulness of simple eDSM to understand the system: the overview of eDSM (as presented in Figure 4) is clearly useful to retrieve cycles: all but three developers agree or strongly agree that eDSM is useful to understand the structure, and all developers agree or strongly agree that eDSM is useful to identify cycles. The developers have more difficulties to retrieve the structure (providers are on bottom/right, and clients are on top/left) because it needs more learning time of the visualization, and it was not explained in the tutorial.
5. Pattern identification in eDSM (Figure 12): eDSM promotes the use of small-multiple views. Answers show that developers can detect these patterns in eDSM. Patterns do not appear in all systems, especially in Smalltalk ones. Hence, one answer is *not applicable*.
6. Usefulness of eCell (enriched cell) in general: results show that eCell helps developers to understand cycles (all but one developers agree or strongly agree that eCell is useful to understand cycles). Fixing a cycle is a bit more difficult, as sometimes developers need to access source code to understand dependencies, but with eCell, developers have no problem to access the source code (all but one developers agree or strongly agree that eCell is useful to fix cycles).

7. Use and usefulness of eCell features (Table VI and Figure 13): enriched cell is a complex visualization. It provides a large quantity of information; some of which is only useful in specific cases. Results confirm that the main features of enriched cell are used (the cell color, the header, the name of packages in the cell, and the popup information view). Specific features, reserved for special understanding of the package, are less used.
8. Open questions: participants use eDSM globally for analyzing these systems and detecting cycles between packages. The main idea is to easily detect cycles, visualize them, and try to build a better structure than the existing one. Generally, the goal has been reached. One person did not like the matrix representation, and two others (*PolyMorph* with eight packages and *XMLParser* with 12 packages) declared that the learning time is too big for the studied system because they have few packages.

   All but one participant consider eDSM as a useful tool. The one who does not like it prefers node-link visualization, which needs less learning time. This developer uses its own iterative tools such as scripts and node-link visualization in an iterative process. Other participants do not use any other tools and probably let their systems deteriorate, which is shown by this experiment where all studied systems have cycles.

Table VI. Use of enriched cell features.

Enriched cell feature	Pharo	Pharo	Mondrian	Merlin	Moose	Seaside	SqueakDBX	XMLParser	PolyMorph	Rate (%)
Cell color	X		X	X	X	X	X	X	X	88
Header	X		X	X	X	X	X	X	X	88
Name	X		X	X	X	X	X	X	X	88
Ratio					X	X	X	X		44
Class color	X				X		X	X	X	55
Class border						X		X	X	33
Edge color				X	X		X	X	X	55
Popup	X		X	X	X	X		X	X	77

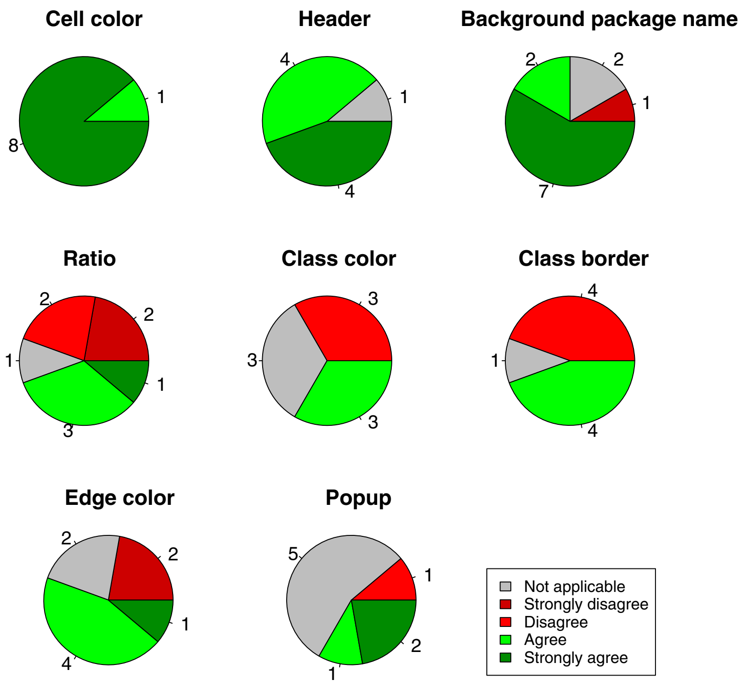


Figure 13. Usefulness of enriched cell features.

About the improvement that could be integrated in eDSM. Globally, participants like eDSM, but the learning time seems to be a problem. There are multiple colors with multiple meanings. The repeated answer is the need of a nomenclature and a better interaction with the source code.

Conclusion of the user study. Results are good enough to say that eDSM is a useful tool to understand and to help breaking cycles between packages. The part of problem encountered by participants is due to a lack of information about the nomenclature.

Enriched DSM encompasses the principle of micro–macro-reading by showing the details of cycles at class level within the package structure. However, enriched cells are too complex for users who are not experts, especially because of the use of many colors.

One point to highlight is that all studied software applications have cycles between packages. This problem could mean two issues: first, the lack of tools leads too many acyclic dependency principle (proposed by Martin [36]) violations, and researchers should provide better approaches to avoid this kind of problem; and second, the definition of a package is not correct, and one should think about what is a package.

6.4. Threats to validity

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

Construct validity. We highlighted two threats of the construct validity. The first threat is related to the study of Seaside (Study 2). The human assessment of the proposals made with eDSM was made by only one architect. This architect is the main developer of the system and clearly knows the system, but he could evaluate wrongly some propositions. To avoid this threat, we propose to do the evaluation to another maintainer of Seaside.

The second threat is related to the user study (Study 3): there is no other tool to compare. This threat can impact the human evaluation during the questionnaire. As the developers have no other tools to compare to, they could have difficulties to give their point of view.

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

External validity. eDSM should work on object-oriented language. However, in the three studies, only Smalltalk software applications were analyzed. eDSM has been implemented on top of the Moose reengineering environment [24] and the FAMIX language independent source code meta-model [23]. The source code model extracted from Smalltalk code is similar to the one of Java, and the extracted dependencies between packages are the same: class references, inheritances, and accesses. We believe that this thread does not impact the approach. Smalltalk code meta-model is closed to Java, C++, and C#. In addition, the information we use is available in mainstream object-oriented languages: packages, classes, inheritance, access, and method invocation. An exception is about class extensions, which is not supported by Java. In this case, using Java simplifies eDSM.

Reliability. One threat to validity is related to the dependency extraction. eDSM takes into account four kinds of dependency. Three of them (inheritance, class reference, and class extension) are easy to detect and to extract using static analysis. There is a threat to validity with the case of method invocation. Smalltalk is dynamically typed. It is not possible to know the type of an object before the execution. To avoid this threat, eDSM uses only the sure invocations: the invocations that the receiver is known statically. With this strategy, we are sure to not display false-positive invocations, but eDSM do not provide all invocations links. This problem could be avoided by coupling the static analysis with a dynamic analysis, but we are not sure the benefits of this kind of strategy are interesting because of the cost of the analysis.

Another threat to validity is related to the user study (Study 3). It has needed participants without conflict of interest with the authors. It is important for the validity of the human evaluation.

We recruit developers on a Smalltalk mailing list, where authors are known as active developers. This context could have an impact on the user study. To avoid this threat, we would send the questionnaire to another mailing list. We would use a Java developer mailing list, which can avoid also the threat related to the use of only Smalltalk software applications.

7. DISCUSSION

7.1. Comparison with other approaches

Often node-link visualizations are used to show dependencies among software entities. Several tools such as dotty/GraphViz [37], Walrus [38], or Guess [39] can be used. Using a node-link visualization is intuitive and has a fast learning curve. One problem with node-link visualizations is finding a layout scaling on large sets of nodes and dependencies: such a layout needs to preserve the readability of nodes, the ease of navigation following dependencies, and to minimize dependency crossing. Even then, modularity identification is not trivial.

Holten proposed hierarchical edge bundles, an approach to improve the scalability of large hierarchical graph visualizations. Edges are bundled together on the basis of hierarchical information, and it uses color schema to represent the flow of information [40]. It has been applied to see the communication between classes grouped by packages in large systems, and the bundling of edges produces less cluttered display. However, it is difficult to identify package dependency patterns, as nodes are positioned in circle, creating many link crossings. Henry *et al.* [41] have reported this limit. With eDSM, the visualization structure is preserved whatever the data size is. A matrix provides a clear structure in comparison with a node-link visualization: a line or a column represents all interactions with a package. This is a spatial advantage because there are no edges between packages, so this reduces clutter in the visualization. Cycles remain identified by colored cells, there is no edge between packages, and so this reduces clutter in the visualization. Moreover, eDSM enables fine-grained information about dependencies between packages. Classes in client package as well as in provider package are shown in the cells of the DSM.

Package blueprint is a visualization that takes the focus on a package and shows how such package uses and is used by other packages [4]. It provides a fine-grained view; however, package blueprint lacks of (1) the identification of cycles at system level and (2) the detailed focus on classes actually involved in the cycles.

In terms of dependency clusters, Binkley and Harman proposed two visualizations for assessing program dependencies, both from a qualitative and quantitative points of view [42]. They identify global variables and formal parameters in software source code. Subsequently, they visualized the effect dependencies. Additionally, the Monotonic Slice-size Graph (MSG) visualization [43] helps finding *dependence clusters* and locating avoidable dependencies. Some aspects of their work are similar to our own. The granularity and the methodology employed differ: they operate on procedure and use slicing analysis, whereas we focused on coarse-grained entities and use model analysis.

7.2. Use of small-multiple

Small-multiples and micro–macro-reading [28] are important features of eDSM. They provide 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. However, eDSM provides more information about the context of a dependency by displaying in an enriched cell the complexity of the relationship. Also seeing simultaneously the multiple contexts of dependencies allows the programmer to compare and assess the complexity of each. eDSM also provides browsing actions such as detailed zoom and focusing on SCC or subset.

One critique about eDSM is that it loses the simplicity of the original DSM. Our experience on real and complex software showed that DSM is powerful at high level but limited for details, which are crucial to understand problems. With a simple DSM, we were constantly losing time browsing code to understand what a cell was referring to. eDSM gives such information at a glance. A related critic about eDSM is that it looks too complex. However, one does not need to know all the features. The main features are easy to catch to start with eDSM.

7.3. Limits of eDSM

There are still limitations that we would like to overcome, with the goal to make eDSM more effective for reengineers. A problem is the limitation of the screen space. A DSM requires much unused space when there are empty cells. This limitation is visible in Figure 8, where many blue cells are empty. To overcome this issue, the reengineer can use the simple DSM visualization (i.e., without enriched Cell) and use the interactions to focus on few cells.

The experiment we conducted gave us the feeling that indirect cycles were more difficult to analyze than direct ones. It makes our future work focus 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 enhance reengineering tasks.

One common problem of visualization, which is applied to eDSM, is the problem of having enough information to understand a system but not too much information to still have a clean visualization. In the second case study, on the validation of Seaside proposition, many propositions have been refused because of a lack of information about the programming strategy, but in the user study, we show that eDSM provides enough information, which could be not used every time.

A limit that we faced is the problem of using squares and not full class names. This problem is not trivial because the solution should be consistent across views and in presence of cells with couple of squares and cells fully packed. Our experiences showed us that having the fly-by-help is a first possible solution but having the names all the time present would be much better. A possible solution would be to have a semantical zoom that adapts itself to the level of detail and available space.

8. CONCLUSION

This paper enhances dependency structure matrix (DSM) using *micro–macro-reading*. 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 [28], micro–macro-reading, and preattentive visualization principles [30–33]. Thanks to these improvements, package organization, cycles, and cycle details are made explicit. We applied eDSM on several complex systems to demonstrate that eDSM is a concrete tool that allows developers to get a better understanding of cycles between packages, even helping to resolve them.

In the user study, we discovered that all studied software applications had cycles between packages. This situation shows that all these systems break the acyclic dependency principle proposed by Martin [36] and probably that many systems are in a similar situation. First, we should analyze the reason of cycles. It seems that cycles are mainly due to structural problems and can be fixed by reengineering source code. A part of these cycles has similar characteristics, and it should be interesting to analyze them and discover particular patterns that can help in detection of unwanted dependencies. Second, some cycles seem not to be breakable because of their semantics. For a human convenience, engineers organize classes in multiple small packages instead of one big. This organization creates package cycles, but they are not critical.

These two observations open some questions about the problem of cycle: what are the patterns for package cyclic dependencies? Are they good or bad dependencies? Packages can be in cycle because of a semantic division that makes sense for the engineer and is not critical for the modularity of the application.

REFERENCES

1. Lungu M, Lanza M, Gîrba T. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*. IEEE Computer Society Press: Los Alamitos CA, 22–24 March 2006; 185–196, DOI: 10.1109/CSMR.2006.39. Available from: http://scg.unibe.ch/archive/papers/Lung06aPackagePatterns.pdf
2. Ducasse S, Gîrba T, Kuhn A. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*. IEEE Computer Society: Philadelphia, PA, 24–27 September 2006; 203–212, DOI: 10.1109/ICSM.2006.22. Available from: http://scg.unibe.ch/archive/papers/Duca06cDistributionMap.pdf

3. Ducasse S, Lanza M. The class blueprint: visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)* January 2005; **31**(1):75–90. Available from: http://scg.unibe.ch/archive/papers/Duca05bTSEClassBlueprint.pdf
4. Ducasse S, Pollet D, Suen M, Abdeen H, Alloui I. Package surface blueprints: visually supporting the understanding of package relationships. *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, Paris, France, 2–5 October 2007; 94–103. Availabe from: http://scg.unibe.ch/archive/papers/Duca07cPackageBlueprintICSM2007.pdf
5. Dong X, Godfrey M. System-level usage dependency analysis of object-oriented systems. In *ICSM 2007*. IEEE Computer Society, 2007, DOI: 10.1109/ICSM.2007.4362650.
6. Demeyer S, Ducasse S, Nierstrasz O. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM Press: New York NY, 15–19 October 2000; 166–178, DOI: 10.1145/353171.353183. Available from: http://scg.unibe.ch/archive/papers/Deme00aFindingRefactoring.pdf, Also in ACM SIGPLAN Notices 35 (10).
7. Martin RC. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall: Upper Saddle River, New Jersey, 2002.
8. Kung DC, Gao J, Hsia P, Toyoshima Y, Chen C. On regression testing of object-oriented programs. *Journal of Systems and Software* January 1996; **32**:21–40. Available from: http://portal.acm.org/citation.cfm?id=218153.218155.
9. Tai KC, Daniels F. Test order for inter-class integration testing of object-oriented software. *Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International*, Washington, DC, 1997; 602–607, DOI: 10.1109/CMPSAC.1997.625079.
10. Le Traon Y, Jeron T, Jezequel JM, Morel P. Efficient object-oriented integration and regression testing. *IEEE Transactions on Reliability* 2000; **49**(1):12–25.
11. Briand LC, Feng J, Labiche Y. Using genetic algorithms and coupling measures to devise optimal integration test orders. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*, SEKE '02. ACM: New York, NY, USA, 15–19 July 2002; 43–50, DOI: http://doi.acm.org/10.1145/568760.568769. Available from: http://doi.acm.org/10.1145/568760.568769
12. Bavota G, De Lucia A, Marcus A, Oliveto R. Software re-modularization based on structural and semantic metrics. *17th Working Conference on Reverse Engineering (WCRE) 2010*, Beverly, MA, 2010; 195–204, DOI: 10.1109/WCRE.2010.29.
13. Melton H, Tempero E. Identifying refactoring opportunities by identifying dependency cycles. In *Proceedings of the 29th Australasian Computer Science Conference – Volume 48*, ACSC '06. Australian Computer Society, Inc.: Darlinghurst, Australia, Australia, 16–19 January 2006; 35–41. Available from: http://dl.acm.org/citation.cfm?id=1151699.1151703
14. Melton H, Tempero E. An empirical study of cycles among classes in java. *Empirical Software Engineering* 2007; **12**(4):389–415.
15. Steward D. The design structure matrix: a method for managing the design of complex systems. *IEEE Transactions on Engineering Management* 1981; **28**(3):71–74.
16. Sullivan KJ, Griswold WG, Cai Y, Hallen B. The structure and value of modularity in software design. *ESEC/FSE 2001*, Vienna, Austria, 2001; 99–108.
17. Lopes A, Fiadeiro JL. Context-awareness in software architectures. In *Proceeding of the 2nd European Workshop on Software Architecture (EWSA)*, Vol. 3527, Lecture Notes in Computer Science. Springer: Pisa, Italy, 2005; 146–161, DOI: 10.1007/11494713_10.
18. Sullivan KJ, Griswold WG, Song Y, Cai Y, Shonle M, Tewari N, Rajan H. Information hiding interfaces for aspect-oriented design. *Proceedings of the ESEC/SIGSOFT FSE 2005*, Lisbon, Portugal, 2005; 166–175.
19. Sangal N, Jordan E, Sinha V, Jackson D. Using dependency models to manage complex software architecture. *Proceedings of OOPSLA'05*, San Diego, CA, USA, 2005; 167–176.
20. MacCormack A, Rusnak J, Baldwin CY. Exploring the structure of complex software designs: an empirical study of open source and proprietary code. *Management Science* 2006; **52**(7):1015–1030.
21. Heer J, Bostock M, Ogievetsky V. A tour through the visualization zoo. *Queue* 2010; **8**(5):20–30.
22. Laval J, Denier S, Ducasse S, Bergel A. Identifying cycle causes with enriched dependency structural matrix. *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 13–16 October 2009; 113–122. Available from: http://rmod.lille.inria.fr/archives/papers/Lava09c-WCRE2009-eDSM.pdf
23. Ducasse S, Anquetil N, Bhatti U, Cavalcante Hora A, Laval J, Girba T. Mse and famix 3.0: an interexchange format and source code model family. *Technical Report*, RMod – INRIA Lille-Nord Europe, November 2011. Available from: http://rmod.lille.inria.fr/archives/reports/Duca11b-Cutter-deliverable11-SoftwareMetrics.pdf
24. Ducasse S, Gîrba T, Lanza M, Demeyer S. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series. Franco Angeli: Milano, 2005; 55–71. Available from: http://scg.unibe.ch/archive/papers/Duca05aMooseBookChapter.pdf
25. Warfield JN. Binary matrices in system modeling. *IEEE Transactions on Systems, Man, and Cybernetics* 1973; **3**(5):441–449.
26. Browning TR. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. *IEEE Transactions on Engineering Management* 2001; **48**(3):292–306.

27. Cai Y, Huynh S. An evolution model for software modularity assessment. In *WoSQ '07: Proceedings of the 5th International Workshop on Software Quality*. IEEE Computer Society: Washington, DC, USA, 2007; 3–8, DOI: 10.1109/WOSQ.2007.2.
28. Tufte ER. *Visual Explanations: Images and Quantities, Evidence and Narrative*. Graphics Press: Cheshire, CT, USA, 1997.
29. Tarjan RE. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1972; **1**(2):146–160.
30. Treisman A. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing* 1985; **31**(2):156–177.
31. Healey CG. Visualization of multivariate data using preattentive processing. *Master's Thesis*, Department of Computer Science, University of Bristish Columbia, 1992.
32. Healey CG, Booth KS, Enns JT. Harnessing preattentive processes for multivariate data visualization. *GI '93: Proceedings of Graphics Interface*, Vancouver, BC, Canada, 1993; 107–117.
33. Ware C. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2000.
34. Bergel A, Ducasse S, Nierstrasz O. Analyzing module diversity. *Journal of Universal Computer Science* 2005; **11**(10):1613–1644. Available from: http://www.jucs.org/jucs_11_10/analyzing_module_diversityhttp://scg.unibe.ch/archive/papers/Berg05cModuleDiversity.pdf
35. Lanza M, Ducasse S. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)* September 2003; **29**(9):782–795. Available from: http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf
36. Martin RC. Design principles and design patterns, 2000. Available from: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf, www.objectmentor.com
37. Gansner ER, North SC. An open graph visualization system and its applications to software engineering. *Software Practice Experience* 2000; **30**(11):1203–1233.
38. Munzner TM. Interactive visualization of large graphs and networks. *Ph.D. Thesis*, Stanford University, Stanford, CA, USA, 2000. AAI9995264.
39. Adar E. Guess: a language and interface for graph exploration. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '06. ACM: New York, NY, USA, 24–27 April 2006; 791–800, DOI: http://doi.acm.org/10.1145/1124772.1124889. Available from: http://doi.acm.org/10.1145/1124772.1124889
40. Holten D. Visualization of graphs and trees for software analysis. *Ph.D. Thesis*, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven: Eindhoven, 2009. ISBN 978-90-386-1882-1.
41. Henry N, Fekete JD, McGuffin MJ. Nodetrix: a hybrid visualization of social networks. *IEEE Transactions on Visualization and Computer Graphics* 2007; **13**(6):1302–1309.
42. Binkley D, Harman M. Analysis and visualization of predicate dependence on formal parameters and global variables. *IEEE Transactions on Software Engineering* 2004; **30**(11):715–735.
43. Binkley D, Harman M. Locating dependence clusters and dependence pollution. In *ICSM '05*. IEEE Computer Society: Washington, DC, USA, 2005; 177–186, DOI: 10.1109/ICSM.2005.58.