Ecole Nationale Supérieure des Ingénieurs des Etudes et Techniques d'Armement (ENSIETA)

Final year project

Enhanced Dependency Structure Matrix

Institut National de Recherche en Informatique et Automatique (INRIA)

Author:
Romain Peirs

Supervisor:
Dr. Stéphane Ducasse

September 26, 2009

Abstract

Dependency Structure Matrix (DSM) is an approach taken from process optimization and it exists a variety of algorithms to help organizing the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies. However, the existing DSM implementations have a lack of information in their visualization. That is why, we enhanced DSM by adding written and colored information in the visualization to improve the architecture understanding of a software system and the support of its remodularization. This visualization has been implemented on top of Moose which is a collaborative research platform for Software Analysis and Information Visualization.

This report presents my internship done at the research centre INRIA Lille - Nord Europe with the RMoD Team led by S. Ducasse.

Contents

Chapter 1

Introduction

1.1 Context

It is well-known that 50% to 75% of the overall cost of a software system is devoted to its maintenance [LS80]. During maintenance, software professionals spend at least half their time reading and analyzing software in order to understand it. The maintenance of object-oriented applications is harder than the ones written in procedural languages because the presence of inheritance and late-binding greatly increases the number of potential dependencies within a program. Use correctly, visualization provides a faster way to analyze software. The main purpose for using visualization is to help understanding and refactoring the application.

1.2 Software Evolution

To cope with the complexity of large software systems, applications are structured in subsystems or packages [DPS+07]. It is now frequent to have large object-oriented applications structured over large number of packages. Ideally a package should contain classes highly cohesive be as less as couple with the rest of the application. This help the maintenance, as far as the coupling between deployment unit is less important you can easily remove it from the system and deploy it for other people, but as systems inevitably become more complex, their modular structure must be maintained. It is thus useful to understand the concrete organization of packages and their relationships. Packages are important but complex structural entities that can be difficult to understand since they play different development roles (i.e., class containers, code ownership basic structure, architectural elements...). Packages provide or require services. They may play core role or contain accessory code features. Maintainers of large applications face the problem of understanding how packages are structured in general and how packages are in relation with each others in their provider/consumer roles. This problem

was experienced first-hand while preparing the 3.9 release of Squeak, a large open-source Smalltalk [DD07]. In addition, approaches that support application remodularization succeed in producing alternative views for system refactorings, but proposed changes remain difficult to understand and assess. There is a good support for the algorithmic parts but little support to understand their results. Hence it is difficult to assess the multiple solutions.

1.3 Understanding Software Structure

Several previous works provide information on packages and their relationships, by visualizing software artefacts, metrics, their structure or their evolution. Metric can be somehow difficult to understand. They are project dependent and subject to change. In a visualization like Distribution Map [DGK06], we saw the package properties but we don't learn about the structure of the package. Lanza et al. [DL05] propose a way to recover architecture by visualizing relationship but do not provide a fine-grained view. However, while these approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the package shapes (the number of classes it defines, the inheritance relationships of the internal classes, how the internal class inherit from external ones,...) and help identifying their roles within an application.

It exists a visualization called Package Surface Blueprint which already reveals package structure and relationships. A package blueprint is structured around the concept of surface, which represents the relationships between the observed package and its provider packages. The Package Surface Blueprint reveals the overall size and internal complexity of a package, as well as its relation with other packages, by showing the distribution of references to classes within and outside the observed package. However, this visualization has limitations. Firstly, for the analysis of a large project the visualization is not easy to use because of the number of packages. Moreover, with a package blueprint visualization, we need 3 views to know what are the dependencies which exist between the packages (1 for inheritance, 1 for outgoing dependencies and 1 for incoming dependencies). Finally, we do not know the number of dependencies which exist. For all these reasons, we need another visualization which synthesize the different views of Package Surface Blueprint and specify the dependencies: their number and their kind (access, invocation, inheritance).

DSM visualization already exists. Indeed, it exists a promising approach taken from process optimization that propose Dependency Structure Matrix (DSM) [Ste81]. In addition, a variety of algorithms are available to help organize the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies. The potential significance of the DSM for software was noted by Sullivan et al [SGCH01], in the context of

evaluating design trade-offs, and has been applied by Lopes et al [LB05] in the study of aspect-oriented modularization. MacCormack et al [MRB06] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux.

1.4 Contributions

While DSMs have proven solution to reveal software structure, DSMs have weaknesses too. They lack providing certain information and their visualization, notably concerning the dependencies cycles it often unprecise. The DSM current implementations produce blurry cycles with the power of adjacency matrix method, (2) they lack of fine grained-overview, detected cycles are not focused on an entity and lack of support for class extension.

Our contribution is to address such weaknesses. We added 2 kinds of information to DSM cells: first, a written information in cells which provides detailed information about the kinds of references and the number of concerned classes and methods by these references; second, a color information which provides a visual help to the user. We enrich cell information with contextual information, we isolate independent cycles and coloring information. We provide entity focused oriented view and level cycle coloring.

Outline. Chapter 2 is a presentation of a dependency structure matrix (DSM). Chapter 3 describes the limitations of the current DSM. Chapter 4 shows the solutions to some of the problems outlined above and describes the new functionality offered by our DSM. Finally, we are concluding in chapter 5.

Chapter 2

Dependency Structure Matrix

In this chapter, we show in Section 2.1 what is a Dependency Structure Matrix (DSM) and in what is it useful. Section 2.2 describes several algorithms which can be used to partition a DSM, reordering so properly the matrix.

2.1 Presentation

The dependency structure matrix (DSM) was invented for optimizing product development processes. Since its invention, the concept has been extended to many other fields, including software engineering and reengineering. I will describe a DSM in its original context to reduce its purpose to its original intention [SJSJ05].

Earlier works [War73] [Ste81] started with the use of graphs for system modeling. For example, consider a system that is composed of two elements (or sub-systems): element "A" and element "B". [The two elements are assumed to completely describe the system and characterize its behavior]. A graph may be developed to represent this system pictorially. The system graph is constructed by allowing a vertex/node on the graph to represent a system element and an edge joining two nodes to represent the relationship between two system elements. The directionality of influence from one element to another is captured by an arrow instead of a simple link. The resultant graph is called a directed graph or simply a digraph.

The matrix representation of a digraph is a binary square matrix with m rows and columns, and n non-zero elements, where m is the number of nodes and n is the number of edges in the digraph. The matrix layout is as follows: the system elements names are placed down the side of the matrix as row headings and across the top as column headings in the same order. If there exists an edge from node i to node j, then the value of element ij (column i, row j) is unity (or marked with an X). Otherwise, the value of

the element is zero (or left empty). In the binary matrix representation of a system, the diagonal elements of the matrix do not have any interpretation in describing the system, so they are usually either left empty or blacked out. A major advantage of the matrix representation over the digraph is in its compactness and ability to provide a systematic mapping among system elements that is clear and easy to read regardless of size.

If the system is a project represented by a set of tasks to be performed, then off-diagonal marks in a single row of the DSM represent all of the tasks whose output is required to perform the task corresponding to that row. Similarly, reading down a specific column reveals which task receives information from the task corresponding to that column. Marks below the diagonal represent forward information transfer to later (i.e. downstream) tasks. This kind of mark is called forward mark or forward information link. Marks above the diagonal depict information fed back to earlier listed tasks (i.e. feedback mark) and indicate that an upstream task is dependent on a downstream task.

There are three basic building blocks for describing the relationship amongst system elements: parallel, sequential and coupled (Figure 2.1).

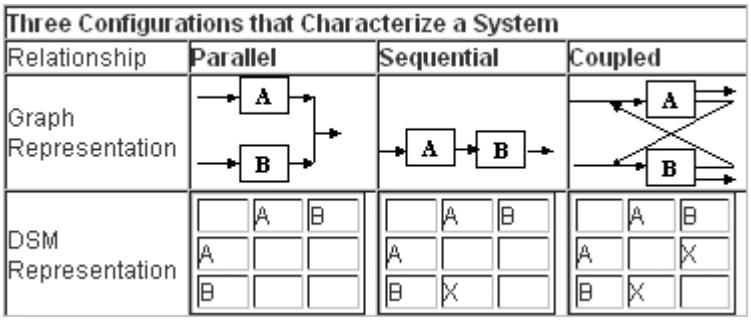


Figure 2.1: DSM Presentation

In the parallel configuration, the system elements do not interact with each other. Understanding the behavior of the individual elements allow us to completely understand the behavior of the system. If the system is a project, then elements would be project tasks to be performed. As such, activity B is said to be independent of activity A and no information exchange is required between the two activities.

In the sequential configuration, one element influences the behavior or decision of another element is a unidirectional fashion. That is, the design parameters of system element B are selected based on the system element A design parameters. Again, in terms of project tasks, task A has to be performed first before task B can start.

Finally, in the coupled system, the flow of influence or information is intertwined: element A influences B and element B influences A. This would

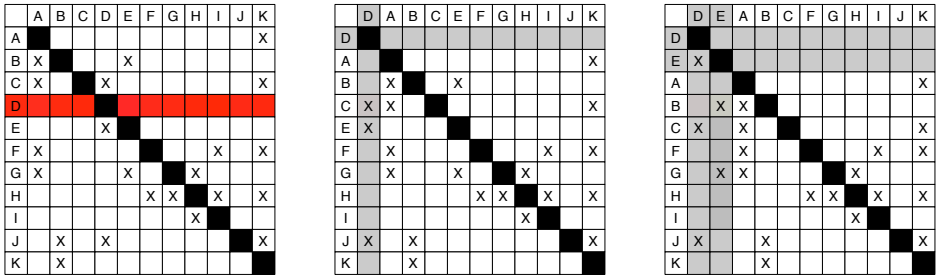
occur if parameter A could not be determined (with certainty) without first knowing parameter B and B could not be determined without knowing A. This is called a cyclic dependency.

2.2 Partitioning Algorithms

Partitioning is the process of manipulating (i.e. reordering) the DSM rows and columns such that the new DSM arrangement does not contain any feedback marks. Thus, transforming the DSM into a lower triangular form. For complex engineering systems, it is highly unlikely that simple row and column manipulation will result in a lower triangular form. Therefore, the analyst's objective changes from eliminating the feedback marks to moving them as close as possible to the diagonal (this form of the matrix is known as block triangular). In doing so, fewer system elements will be involved in the iteration cycle resulting in a faster development process[htt].

Several algorithms exists to partition a DSM [GEC91] [War73]. However, some of these algorithms are similar during the first steps [GEC91]. The difference is made during the identification of the cycles. In Section 2.2.1 and Section 2.2.2, I will present several algorithms which perform the identification of the cycles. But now, I am explaining how the partitioning algorithms proceed during the first steps:

1. Identify elements in the DSM which are used by none of the other elements in the matrix. These elements are easily identified because they have an empty row in the DSM (Figure 2.2(a)). Move all these empty rows to the top of the matrix and the corresponding columns to the left of the matrix (Figure 2.2(b)) and omit those elements for further consideration. Repeat this step on the remaining elements until there are no empty rows in the matrix (Figure 2.2(c)).



(a) DSM (b) First sort (c) Second sort

Figure 2.2: First step of partitioning algorithm

2. Identify elements in the DSM which use none of the other elements in the matrix. These elements are easily identified because they have an

empty column in the DSM. Move all those empty columns to the left of the matrix and the corresponding rows to the bottom of the matrix and omit those elements for further consideration. Repeat this step on the remaining elements until there are no empty columns in the matrix (Figure 2.3).

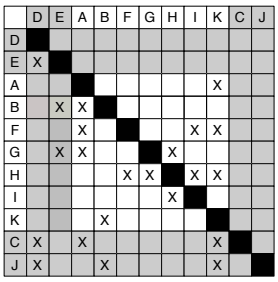


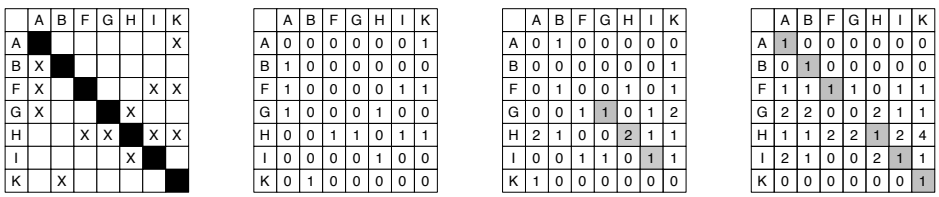
Figure 2.3: Second step of partitioning algorithm

These two steps are called the topological sorting.

3. If after the topological sorting there are no remaining elements in the DSM, then the matrix is completely partitioned; otherwise, the remaining elements contain cycles (at least one). This is at this moment that the identification of cycles is performed.

2.2.1 Identifying Cycles by Powers of the Adjacency Matrix

The adjacency matrix is a binary DSM where an empty cell is replaced with a 0 and a non-empty cell is replaced by 1 (Figure 2.4(a) and Figure 2.4(b)). Raising the DSM to the n-th power shows which element can be reached from itself in n steps. That is why when non-zero values are part of the diagonal of the matrix, it implies that the corresponding elements are involved in a cycle. In Figure 2.4(c) the DSM shows that elements G, H and I are involved in a two-step cycle and in Figure 2.4(d) elements A, B, F, H, I and K are involved in a three-step cycle.



(a) DSM (b) Binary DSM (c) Power 2 (d) Power 3

Figure 2.4: Powers of the adjacency matrix

Thus, the procedure to identify all the cycles and so to partition the DSM is the following:

1. Convert the DSM matrix into a binary matrix (Figure 2.5(a)).
2. Raise the power of the adjacency matrix until you identify elements with a non-zero value along the diagonal (Figure 2.5(b)).
3. Merge all those elements together. The merged matrix forms the new active matrix (Figure 2.5(c)).
4. Repeat the topological sorting on the new active matrix. If it still remains elements, repeat the identification of the cycles (Figures 2.5(d) to 2.5(j)). If not, the merged elements represent blocks and the resultant matrix is the partitioned DSM (Figure 2.5(k)).

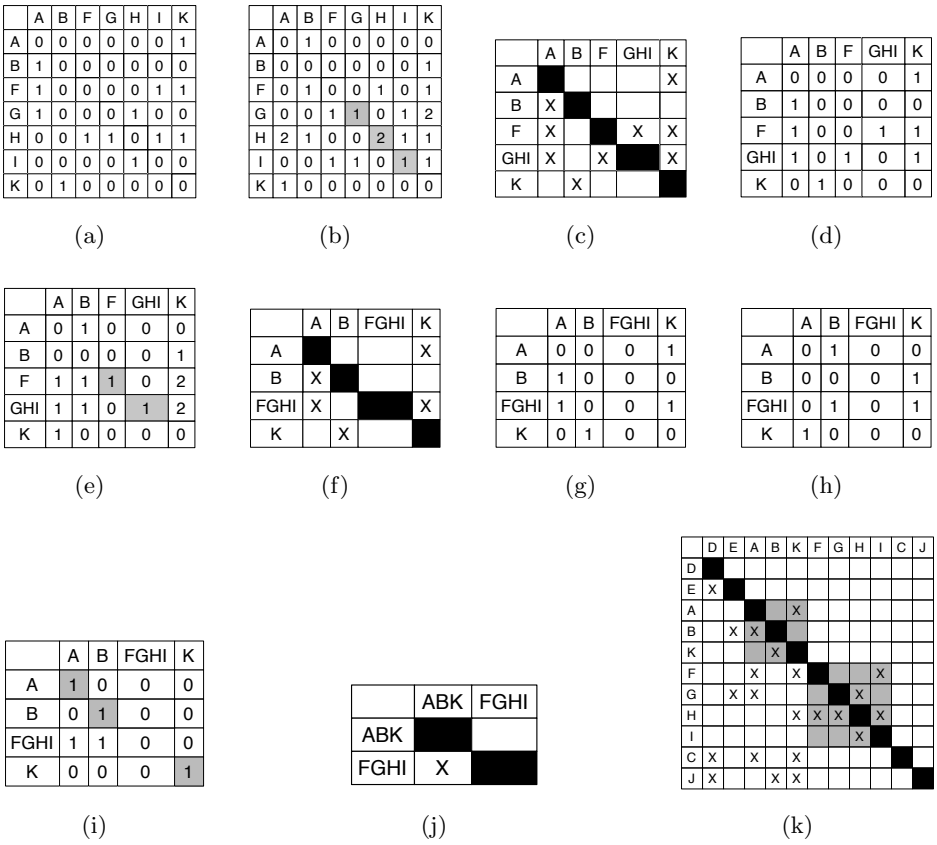


Figure 2.5: Formation of partitioned DSM with powers of the adjacency matrix method

2.2.2 Identifying Cycles by Path Searching

In path searching, information flow is traced either backwards or forwards until an element is encountered twice [SW64]. All elements between the first and second occurrence of the task constitute a cycle of information flow. When all cycles have been identified, and all elements have been scheduled, the sequencing is complete and the matrix is in a block triangular form.

We know that in the active matrix, all elements are involved in at least one cycle. So, a cycle can be traced starting with any of the elements. That is why the following algorithm is used to detect all cycles:

1. Choose arbitrary an element and trace its dependencies until this element is encountered a second time (Figure 2.6(a))
2. Merge together all elements encountered to form a unique element (Figure 2.6(b))
3. Repeat the topological sorting on the new active matrix (Figure 2.6(c)
4. Repeat those 3 steps until there are no elements in the active matrix. The merged elements represent blocks and the resultant matrix is the partitioned DSM (Figure 2.6(d))

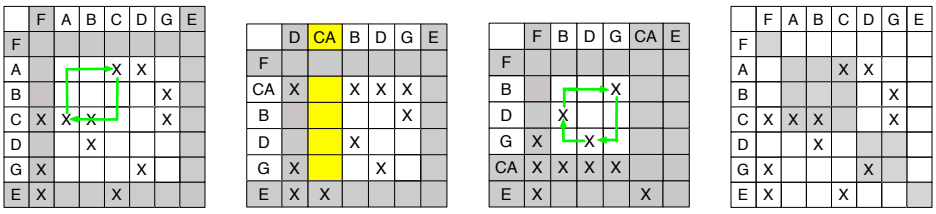


Figure 2.6: Formation of partitioned DSM with path searching method

2.2.3 DSM Partitioning Using Reachability Matrix Method

The reachability matrix is a binary DSM with the diagonal elements equal to 1 (Figure 2.7(a)) [War73]. This diagonal of 1 does not add information to the matrix but is required to this algorithm.

The method calls for finding a multi-level hierarchical decomposition for the matrix. The top level in this hierarchy is composed of all elements that require no input or are independent from all other elements in the matrix. Any two elements at the same level of the hierarchy are either not connected to each other or are part of the same circuit at that level. Once the top level

set of elements is identified, the elements in the top level set and their corresponding from/to connections are removed from the matrix leaving us with a sub-matrix that has its own top level set. The top level set of this sub-matrix will be the second level set of the original matrix. Proceeding in this manner, all the levels of the matrix can be identified.

The steps of this method are:

1. Construct a table with 4 columns (Figure 2.7(b)):
   (a) In the first column, list all the elements in the matrix.
   (b) In the second column, list the set of all the input elements for each row in your table. This set can easily be identified by observing an entry of 1 in the corresponding row in the DSM.
   (c) In the third column, list the set of all output elements for each row in your table. This set can easily be identified by observing an entry of 1 in the corresponding column in the DSM.
   (d) In the fourth column, list the intersection of the input and output sets for each element in your table.
2. Identify top level elements and remove them from the table. An element is in the top level hierarchy of the matrix if its input set is equal to the intersection set.
3. Repeat the two first steps until the DSM is fully partitioned (Figure 2.7(c)).

Conclusion

We saw in this chapter that it exists several algorithms which can bring to light the architecture of an application and its defects. But, there is still a lack of information which could be brought to a reenginner. That is what we show in the next chapter.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
3	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0
4	1	1	0	1	1	0	0	0	1	0	0	0	0	0	0
5	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
6	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
7	1	1	1	1	1	0	1	1	1	0	0	0	0	0	0
8	1	1	1	1	1	0	0	1	1	0	0	0	0	0	0
9	1	1	0	1	1	0	0	0	1	0	0	0	0	0	0
10	0	1	0	0	1	1	0	0	0	1	0	0	0	0	0
11	1	1	1	1	1	0	1	1	1	0	1	0	0	0	0
12	1	1	1	1	1	0	1	1	1	0	1	1	0	0	0
13	1	1	0	1	1	0	0	0	1	0	0	0	1	0	0
14	0	1	0	0	1	1	0	0	0	1	0	0	0	1	0
15	0	1	0	0	1	1	0	0	0	1	0	0	0	0	1

(a) Reachability matrix

TABLE I

Element s	Reachability Set R(s)	Antecedent Set A(s)	Set Product R(s)A(s)
1	1	1, 3, 4, 7, 8, 9, 11, 12, 13	1
2	2	2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15	2
3	1, 2, 3, 4, 5, 8, 9	3, 7, 8, 11, 12	3, 8
4	1, 2, 4, 5, 9	3, 4, 7, 8, 9, 11, 12, 13	4, 9
5	2, 5	3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 14, 15	5
6	2, 6	6, 10, 14, 15	6
7	1, 2, 3, 4, 5, 7, 8, 9	7, 11, 12	7
8	1, 2, 3, 4, 5, 8, 9	3, 7, 8, 11, 12	3, 8
9	1, 2, 4, 5, 9	3, 4, 7, 8, 9, 11, 12, 13	4, 9

(b) Reachability table

	2	1	5	6	10	9	4	14	15	13	3	8	7	11	12
2	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
6	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
10	1	0	1	1	1	0	0	0	0	0	0	0	0	0	0
9	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0
4	1	1	1	0	0	1	1	0	0	0	0	0	0	0	0
14	1	0	1	1	1	0	0	1	0	0	0	0	0	0	0
15	1	0	1	1	1	0	0	0	1	0	0	0	0	0	0
13	1	1	1	0	0	1	1	0	0	1	0	0	0	0	0
3	1	1	1	0	0	1	1	0	0	0	1	1	0	0	0
8	1	1	1	0	0	1	1	0	0	0	1	1	0	0	0
7	1	1	1	0	0	1	1	0	0	0	1	1	1	0	0
11	1	1	1	0	0	1	1	0	0	0	1	1	1	1	0
12	1	1	1	0	0	1	1	0	0	0	1	1	1	0	1

(c) Partitioned reachability matrix

Figure 2.7: Formation of partitioned DSM with reachability method

Chapter 3

DSM Limitations

DSM are powerful and have been used to analyze the overall structure of large systems, however they have some limits that we present now: merging independent cycles (Section 3.1), lack of fine grained-overview (Section 3.2), cycles not focused on an entity (Section 3.3:) and lack of support for class extension (Section 3.4).

3.1 Blurry Cycles with the Power of Adjacency Matrix Method

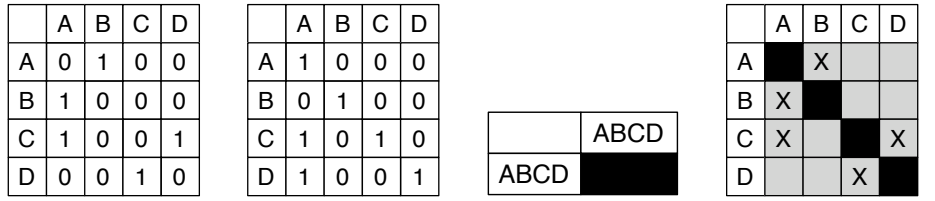
A way to compute cycle in DSM is to use the technique based on powering the adjacency matrix. The principle of this approach is to raise a binary DSM to its n-th power to indicate which elements can be traced back to themselves in n steps; thus constituting a cycle [YFC99]. However, the indicated elements do not automatically belong to the same cycle. Indeed, it can exist several cycles with the same number of steps and the power of the adjacency matrix method cannot differentiate these different cycles, so we have blurry cycles.

	A	B	C	D
A	■	X		
B	X	■		
C	X		■	X
D			X	■

Figure 3.1: A DSM

On Figure 3.1, we see that the elements A and B constitute a direct cycle and the elements C and D constitute another one. But if we raise

the binary DSM (Figure 3.2(a)) to the square by following the adjacency matrix technique, a non-zero value appears in the diagonal for every element (Figure 3.2(b)). These non-zero values mean that any of the elements A, B, C and D are involved in at least one direct cycle but these non-zero values do not inform what these direct cycles are made of. Moreover, with the partitioning algorithm based on this method, we merge these 4 elements together (Figure 3.2(c)) which means that in the partitioned matrix these elements will appear as one cycle (Figure 3.2(d)). So, the partitioned matrix provides wrong information by indicating a unique cycle (the grey area in Figure 3.2(d)) whereas the matrix should show two direct cycles as shown in Figure 3.2(e).



	A	B	C	D
A	0	1	0	0
B	1	0	0	0
C	1	0	0	1
D	0	0	1	0

(a) Binary matrix

	A	B	C	D
A	1	0	0	0
B	0	1	0	0
C	1	0	1	0
D	1	0	0	1

(b) Binary matrix to square

	ABCD
ABCD	

(c) Merged DSM

	A	B	C	D
A		X		
B	X			
C	X			X
D			X	

(d) Partitioned DSM with adjacency matrix method

	A	B	C	D
A		X		
B	X			
C	X			X
D			X	

(e) Ideal partitioned DSM

Figure 3.2: Limitation of the power of adjacency matrix method

So, the power of adjacency matrix method does not allow us to determine precisely the different cycles. However, if we combine this method with a path searching method we can identify all the different cycles.

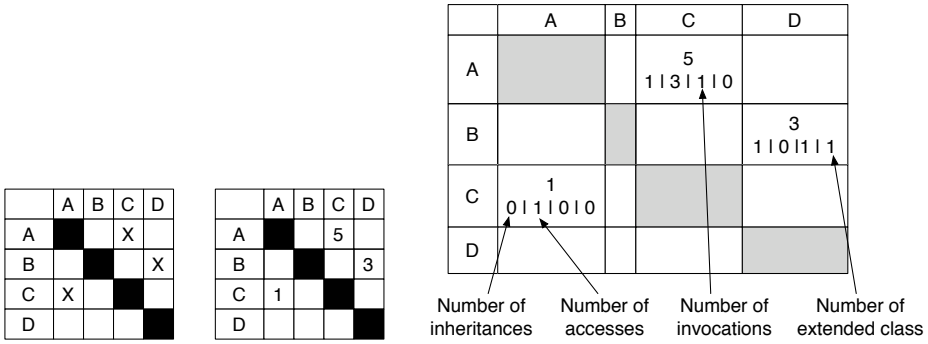
Notice that the DSM software Lattix does not use this partitioning algorithm but is using reachability matrix method.

3.2 Lack of Fine-grained Information

A traditional DSM offers a general overview but lacks from providing precise information about the situation it describes. We identify two kinds of weaknesses: lack of dependency causes and lack of dependency impacts.

3.2.1 Lack of Causes

Dependencies can be due to several sources: class direct accesses, class extensions (see Section 3.3), inheritance relationships and method invocation. Fixing cycles may be different based on the source of linking *e.g.,* changing a direct reference to a class is often simpler than changing an inheritance relationship. That is why, it is not enough to just indicate the dependencies in a DSM by a mark (Figure 3.3(a)) or even by a number representing the number of dependencies that exist (Figure 3.3(b)). Indeed, indicating at least a number for each kind of dependency (Figure 3.3(c) would give a more fine-grained information and so permit a better analysis.



(a) DSM with marks (b) DSM with numbers (c) DSM with kind of dependencies

Figure 3.3: Examples of references in a DSM

3.2.2 Lack of Impacts

In addition, the information provided is often too summarized and does not give an idea of potential problems. For example, knowing that a package has 99 references to another one is a valuable information. Such references could be done by 20 or 3 classes and these 99 references could refer to a small subset or a large number of classes. The same remark can be done for methods instead of classes.

But knowing that these references originate only from 3 classes and 15 methods and that only 4 classes and 6 methods are referenced by these 99 dependencies makes a large difference. In addition, having access to this information without having to look at every class in the concerned packages give more impact to the analysis.

For example, if we have a package A which has 123 references to the package B (Figure 3.4(a)), it has definitely not the same impact if a reengineer knows that these references are due to 3 classes and 25 methods in the

package A which are referencing only 1 class and 3 methods in the package B (Figure 3.4(b)).

(a) Summarized information

(b) Full information

Figure 3.4: The importance of providing adequate information

3.3 Cycles Not Focused on an Entity

Cycles are shown in the context of complete system and the ordering within a level is the result of the partitioning algorithm. It makes it hard to understand the cycle (and not the level) in which a given package is involved. In particular when cycles inside the same level are merged, we obtain not precise information.

For example, if we have a package A involved in a direct cycle with the package B and this package B involved in a direct cycle with the package C (Figure 3.5(a)), package A is also involved in a cycle with package C but the length of this cycle is not the same that the length of the cycle between package A and package B. This is why it is a valuable information to see the difference between the lengths of the cycles (Figure 3.5(b)) because the used methods to break them will not be the same. In Figure 3.5(b) direct cycles started from the entity A are displayed: yellow shows the direct cycles, then red the second level.

(a) Cycles not focused on an entity

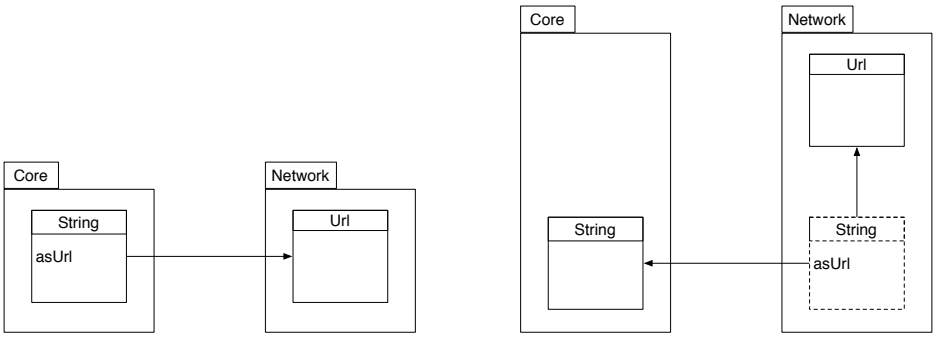
(b) Cycles focused on an entity

Figure 3.5: The importance of cycles focused on an entity

3.4 No Support for Class Extensions

A class extension is a method that is defined in a package, but whose class is defined in another package [BDNW05, BDN05]. Class extensions offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. Indeed, class extensions offer a good solution to the dilemma that arises when one would like to modify or extend the behavior of an existing class, and subclassing is inappropriate because that specific class is referred to, but, one cannot modify the source code of the class in question. A class extension can then be applied to that specific class [BDNW04].

Class extensions is offered in languages such as Objective-C, CLOS, Smalltalk, Ruby and C#3.0 limited to static methods. However, up to now DSMs were mainly applied to Java and C++ which do not support class extensions. That is why DSM softwares designed for these languages do not take into account this kind of dependency. But in the case of a DSM software designed for multi languages, this is important to consider class extensions. Indeed, extending a class in another package automatically create a reference from the package where the class is extended to the package where the class is defined because the extended class just modifies the behavior but do not define it; so you need the definition of the class if you want to modify it.



(a) Dependencies without class extension (b) Dependencies with class extension

Figure 3.6: Meanings of class extension

In addition, we have a better modularity with class extension as shown in Figure 3.6. Indeed, we take a class *String* which is defined in the package Core and a class *Url* which is defined in the package Network. Now, we take a method named *asUrl* and defined in the class *String* which accesses to the class *Url*. So, we have a reference from the method *asUrl* to the class *Url*. Without a class extension, this method is packaged in the package Core (Figure 3.6(a)) and so we have a reference from the package Core

to the package Network. This reference is very bad because the package Core should not reference any package (since it is meant to be part of a low architectural layer). But if we extend the class *String* in the package Network and move the method *asUrl* in this extension (Figure 3.6(b)), the reference is moved inside the package Network and another reference is created from the package Network to the package Core. So, there is only a reference from the package Network to the package Core, that is much more logical.

Conclusion

In this chapter, we identified the limits of existing DSMs and thus, we will show the solutions that we bring to solve these limitations in the next chapter.

Chapter 4

Enhanced DSMs

Our implementation has taken into account the previous described limitations and adds some functionality that does not seem to be offered by DSM software like Lattix [SJSJ05] or other:

- Enriched contextual cell information (Section 4.1),
- Isolating independent cycles and coloring information (Section 4.2),
- Entity-Focused oriented view and level cycle coloring (Section 4.3).

4.1 Enriched Contextual Cell Information

As described in the previous chapter, current DSM softwares do not provide a fine-grained information. That is why, we address this issue in our enhanced DSM (Figure 4.1). The enriched contextual cell information shows the following information:

1. Strength of the dependency.
   On the first line, we show the number of references from a package to another one. This number gives us the strength of the link which exists between these packages but give no more information.

2. Dependency Kinds.
   On the second line, we show the kinds of reference (Inheritance (H), Access (A), Invocation (I) and class Extension (E)) and how many references there are for each. These numbers give us a first information about how the studied software is built.

3. Spread of the Dependency.
   Finally, we show how the references are spread in the packages. Indeed, we give the numbers of classes in every package on the fourth line. This information gives us an idea of the size of the package. On the fifth

line, we indicate for every package the number of classes and methods which are involved in the references and finally on the sixth line, we give the percentage that the involved classes represent in every package and the percentage that the involved methods represent in the involved classes. These information are very important because they show us the part of the packages which are concerned by the references. So, we know whether it is a small part of a package or a big one which is concerned by the dependencies and gives an approximation about the necessary engineering effort to remove these dependencies.

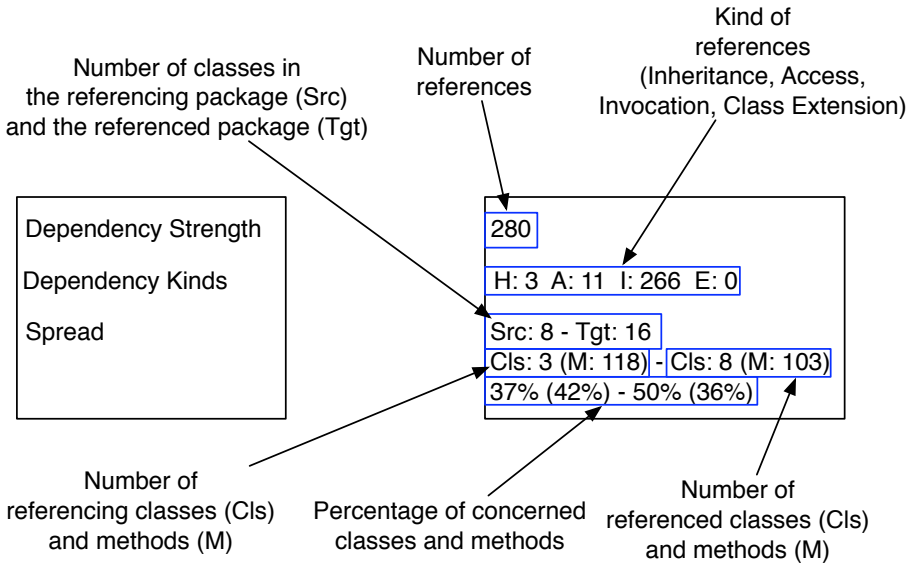


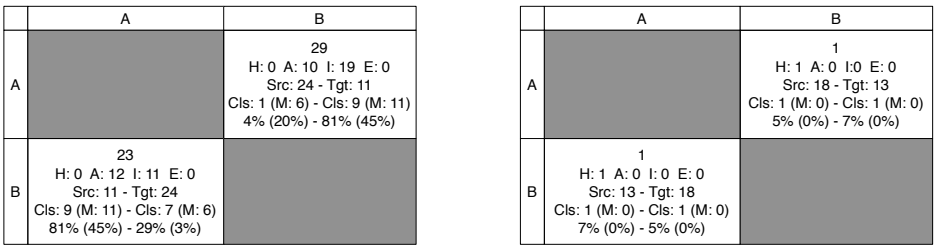
Figure 4.1: DSM cell with enriched contextual information

This enriched contextual view supports the identification of situation by just glancing over the matrix.

4.1.1 Some Examples

In Figure 4.2(a), we can observe a direct cycle. This cycle is due to 29 references from the package B to the package A and 23 references from the package A to the package B. Moreover, the dependencies from B to A are due to 10 accesses and 19 invocations. Finally, there is only 1 class and 6 methods which are referencing the package A. So we can suggest that the best to break the cycle is to move the referencing methods from package B to package A and thus extending the class containing these methods in the package A.

In Figure 4.2(b), we can also see a direct cycle but this time it is due to 1 reference from package B to package A and 1 reference from A to B. Moreover, in both cases, the dependency is due to an inheritance relationship. This case is a very bad scenario because if we want to break the cycle we have to move an entire class from the package A to the package B, for example. But if there are classes in package A which inherits from the moved class, we have to also move these classes to the package B. But, these changes could reveal other dependencies which are not wished. So, we notice that even if the number of references is very little, it can be difficult to remove these references. That is why, an enriched contextual cell is very important.



(a) Example1 (b) Example2

Figure 4.2: Examples of using enriched contextual cell information

4.1.2 At the Tool Support Level

In addition to the enriched contextual cell information, our tool implementation offers two features to get more information: first a large fly-by-help pop-up window presents the enriched information with more details. Second the cell entity can be navigated using the environment navigation facility. Hence offering the possibility to reach the classes, methods and packages under analysis.

1. A pop-up window that adds information to the original content in DSM cells (Figure 4.3): names of the concerned packages, classes and methods.

2. The possibility to inspect the entity which is represented by a DSM cell (Figure 4.4). This inspection gives access to the more precise information. Indeed, with this inspection, you exactly know, for example, which class inherits from another one, or which method invokes another one.

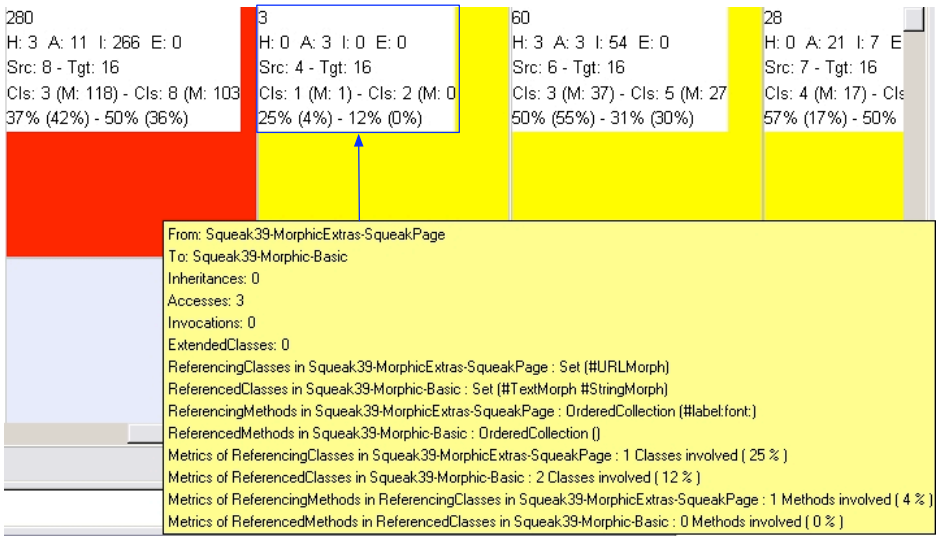


Figure 4.3: DSM cell with a pop-up window

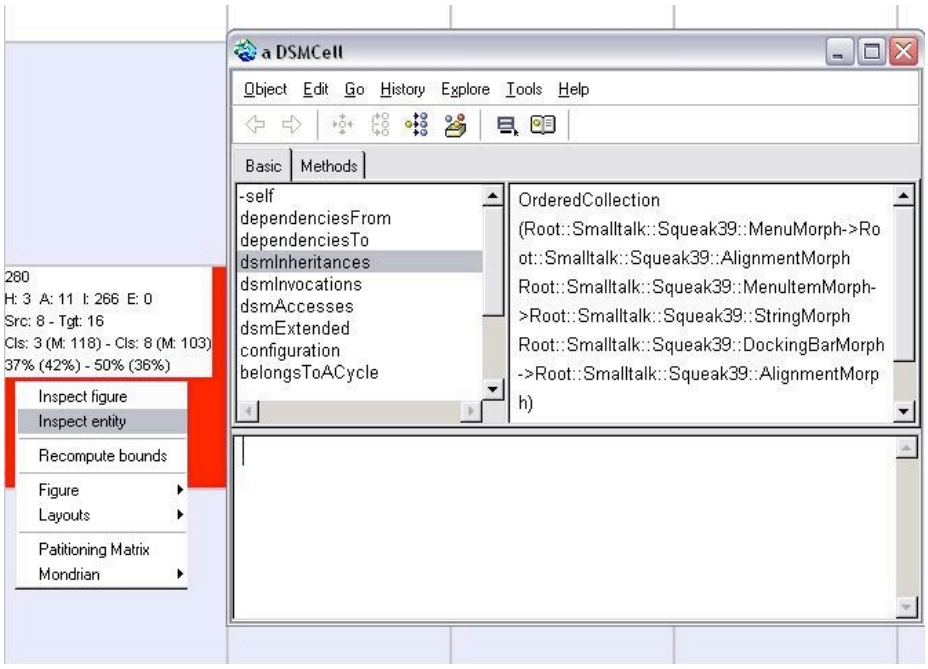


Figure 4.4: DSM cell inspection

4.2 Enhanced Cycle Detection

Cycle identification is important in large entangled software. The cycles detection is often made using path searching algorithm (see Section 2.2.2). This algorithm allows one to detect separately every independent cycle. Indeed, as explained in Section 3.1, it can exist a situation where, for example, package A and B are in a direct cycle and package C and D are in another direct cycle. In this case, these cycles are separately identified with path searching algorithm, which cannot be achieved with an algorithm based on the powers of adjacency matrix.

Our approach enhances the traditional matrix by three aspects: cycle distinctions (Section 4.2.1), cycle nesting identification (Section 4.2.2), and hints for cycling fixing (Section 4.2.3)).

4.2.1 Cycle Distinctions

Our approach distinguishes independent cycles. It is based on the path searching algorithm [GEC91]. With the path searching method, 2 independent cycles are separately detected and can so be isolated from each other in the DSM (Figure 4.5).

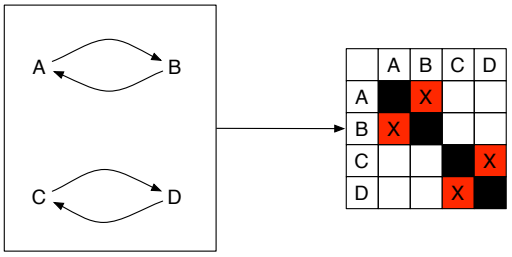


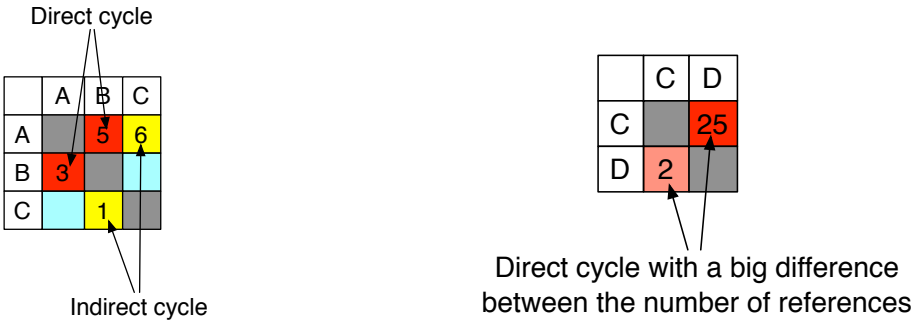
Figure 4.5: Independent cycles distinction

4.2.2 Cycle Nesting

We added color information in DSM cells to give information about cycles. Indeed, as you can see on Figure 4.6(a), after partitioning, DSM cells involved in a cycle have a yellow or red color. The yellow color means that the 2 concerned packages (a DSM cell is the intersection of 2 packages) are involved in an indirect cycle while the red color means they are involved in a direct cycle.

4.2.3 Cycle Fixing Hints

In addition, in the case of a direct cycle, if there is a big difference between the number of references between the 2 packages (by default, the ratio is 3), the cell that contains the least references has a light red color (Figure 4.6(b)). This information allows the user to focus his attention on the references which should be solved a priori in a first time. I say a priori because as said in Section 4.1.1, it could sometimes happen that the least numerous references do not correspond to these that are the simplest to remove if you want to break a cycle (for example, in the case of inheritance).



(a) Cycles representation

(b) Cycle fixing hints

Figure 4.6: Direct and indirect cycles coloring

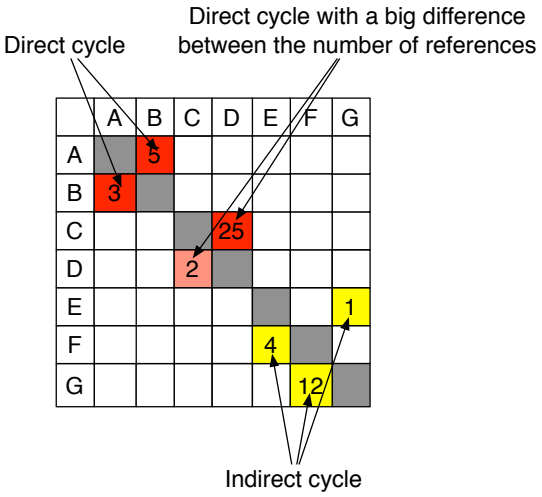


Figure 4.7: Cell color definition

4.2.4 An Example

You can observe in Figure 4.8 a big independent cycle. Without any color, this would be difficult to quickly distinguish direct and indirect cycles. Indeed, direct and indirect cycles would not clearly appear, so the observer could not easily separate them. But with cells color, at first glance, it is very easy to identify direct and indirect cycles. In addition, you can focus your attention on the light red cells because they indicate what are the cells which should probably disappear to eliminate the direct cycles. So, adding color information in DSMs really facilitate refactoring work.

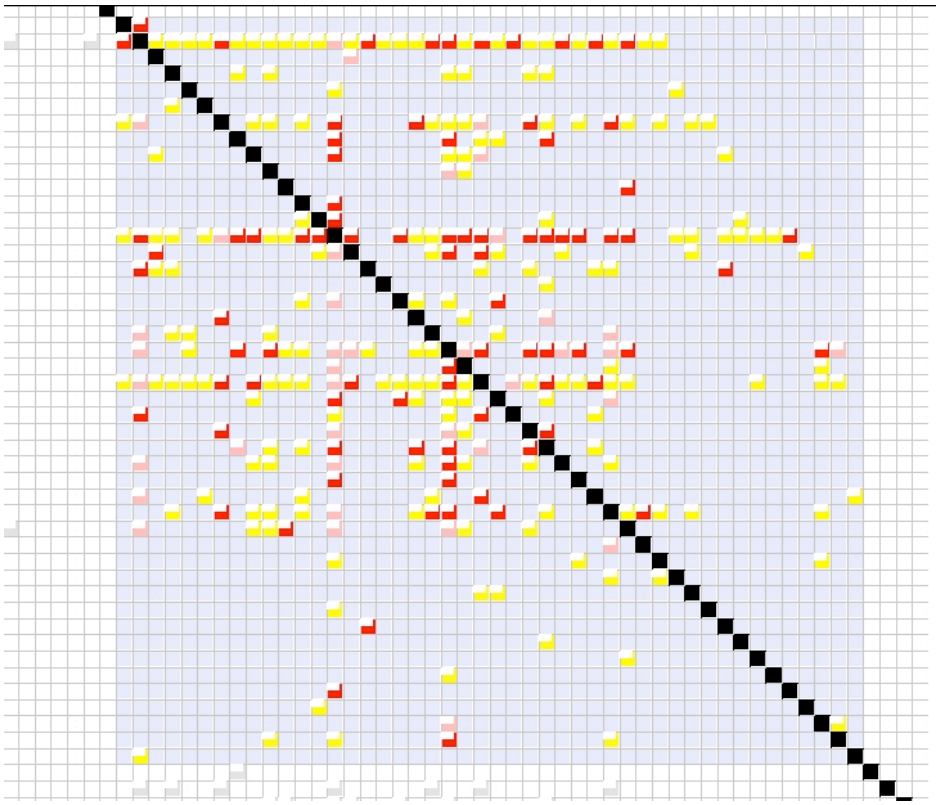


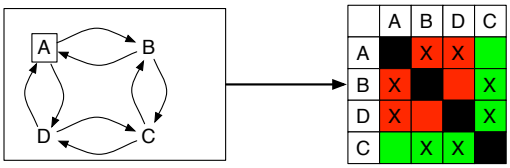
Figure 4.8: Example of cycle detection

4.3 Entity-focused Cycle Centric View

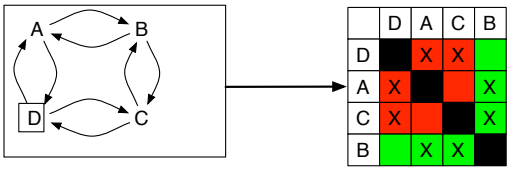
We have seen in the previous section that our approach isolates independent cycles and add them information by coloring cells to visualize better the direct and indirect cycles. However, if it is easy to see the different cycles

which exist into an independent cycle when there are a few number of packages, it is much more difficult when the number of packages is much bigger. Indeed, you do not easily know if the length of a cycle between 2 packages is short or long. But, what it is complicated with cycles, it is that an element can belong to several cycles which do not have the same length, so how to show in the matrix the length of cycles to which belongs an element?

To solve this problem, we added the notion of focused element. Indeed, the length of a cycle between an element and another one is relative to the first one. So, we represent the length of the cycles relative to a chosen element that we call the focused element. For example, in Figure 4.9, you can see that relative to the element A, the elements B and D are involved in a direct cycle with A; and the element C is in an indirect cycle with A Figure 4.9(a). But if you focus your attention on the element D, that are the elements A and C which are involved in a direct cycle with D and the element B which is in an indirect cycle with D Figure 4.9(b).



(a) Situation with the focused element A



(b) Situation with the focused element D

Figure 4.9: Visualization of the different cycles relative to the focused element

4.3.1 Cycle Level

So, to help the visualization of the different cycles in which your focused element is involved, we introduce a color information for every cycle level. Indeed, as shown in Figure 4.10, the elements involved in the first cycle level with the focused element (*i.e.,* the elements which are in cycle with the focused element and which the length of the cycle is the shortest) are in red color. This process is repeated for every element which is in cycle with the focused element: the elements which belong to a same cycle level are

in a same color. Thanks to that color information, first, you can easily see whether 2 elements are close in a cycle or not and second, you can count the number of elements in every cycle level.

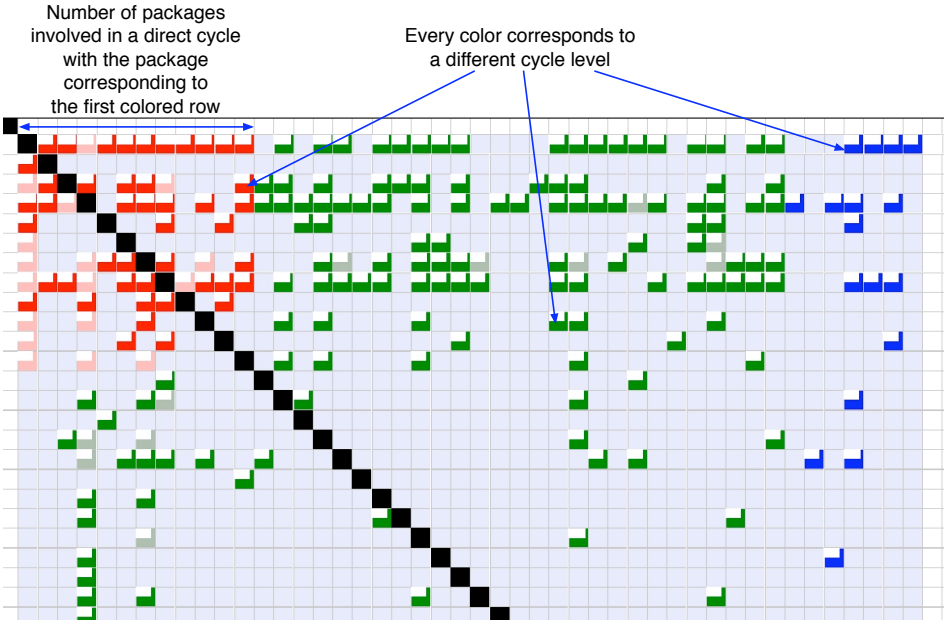


Figure 4.10: Visualization of different cycle levels

Chapter 5

Conclusion

During this final year project, I hence used an existing approach taken from process optimization to help optimizing software. This approach called Dependency Structure Matrix (DSM) allows one to collect all information about references between packages that compose a software inside a matrix. In addition, several algorithms already exist to reorder the matrix and so reveal the software architecture.

However, the existing DSM implementations like Lattix do not provide enough information about the references between the packages and about the cycles which can exist between them. That is why, the goal of my project was to fill this lack and so improve the DSM concept. To do that I added several information in the matrix. First, a written information in cells which provides detailed information about the kinds of references and the number of concerned classes and methods by these references. Second, a color information which provides a visual help to the user.

Thanks to these improvements, understanding and analysis of software applications are easier and thus faster. However, in future work, it would be very helpful to be able to see directly in the matrix the consequences of the changes brought to the application without having to change the code and may be from that a code generator could make itself the changes into the code from the changes realized into the matrix.

Bibliography

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[BDNW04] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. Technical Report IAM-04-003, Institut für Informatik, Universität Bern, Switzerland, June 2004.

[BDNW05] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling visibility of class extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.

[DD07] Marcus Denker and Stéphane Ducasse. Software evolution from the field: an experience report from the Squeak maintainers. In *Proceedings of the ERCIM Working Group on Software Evolution (2006)*, volume 166 of *Electronic Notes in Theoretical Computer Science*, pages 81–91. Elsevier, January 2007.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[DL05] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, January 2005.

[DPS+07] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

[GEC91] D.A. Gebala, S.D. Eppinger, and M. Cambridge. Methods For Analyzing Design Procedures. *Design Theory and Methodology, DTM'91: Presented at the 1991 ASME Design Technical Conferences, 3rd International Conference on Design Theory and Methodology, September 22-25, 1991, Miami, Florida*, 1991.

[htt] http://www.dsmweb.org. The Design Structure Matrix (DSM).

[LB05] Cristina Videira Lopes and Sushil Krishna Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM.

[LS80] Bennett Lientz and Burton Swanson. *Software Maintenance Management*. Addison Wesley, Boston, MA, 1980.

[MRB06] Alan MacCormack, John Rusnak, and Carliss Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Manage. Sci.*, 52(7):1015–1030, 2006.

[SGCH01] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[Ste81] 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.

[SW64] P.R.W.H. Sargent and AW Westerberg. "Speed-up" in Chemical Engineering Design. *Chemical Engineering Research and Design*, 42(a):190–197, 1964.

[War73] J.N. Warfield. Binary Matrices in System Modeling. *IEEE Transactions on Systems, Man, and Cybernetics*, 3(5):441–449, 1973.

[YFC99] A. Yassine, D. Falkenburg, and K. Chelst. Engineering design management: an information structure approach. *International Journal of Production Research*, 37(13):2957–2975, 1999.