Package Dependencies Analysis and Remediation in Object-Oriented Systems

Jannik Laval

Université des Sciences et Technologies de Lille – Lille 1

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Package Dependencies Analysis and Remediation in Object-Oriented Systems

THÈSE

présentée et soutenue publiquement le 17 Juin 2011

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille

(spécialité informatique)

par

Jannik LAVAL

Composition du jury

Président :	Laurence DUCHIEN	(Professeur – Université de Lille 1)
Rapporteurs :	Jean-Marc JEZEQUEL	(Professeur – Université de Rennes)
	Oscar NIERSTRASZ	(Professeur – Université de Berne)
Examinateurs :	Theo D'HONDT	(Professeur – Vrije Universiteit Brussel)
	Laurence DUCHIEN	(Professeur – Université de Lille 1)
Directeur de thèse :	Stéphane DUCASSE	(Directeur de recherche – INRIA Lille)
Co-Encadreur de thèse :	Nicolas ANQUETIL	(Maître de Conférence – Université de Lille 1)

Laboratoire d'Informatique Fondamentale de Lille — UMR USTL/CNRS 8022
INRIA Lille - Nord Europe

Numéro d'ordre: 40515

Acknowledgments

This thesis is a project that has resulted thanks to those who encouraged me. They mainly speak french, so I write this part in this language.

Cette thèse est un projet qui a pu aboutir grâce aux personnes qui m'ont encouragé. Celles-ci parlant principalement français, je rédige donc cette partie dans cette langue.

Cette thèse a pu voir le jour grâce à l'aide et au soutien de toutes les personnes motivées par ce projet. Je souhaiterais tout d'abord remercier Stéphane Ducasse et Nicolas Anquetil, mes encadrants de thèse. Stéphane m'a accordé beaucoup de confiance et d'énergie, et m'a ainsi permis d'aller au bout du projet, tout en m'ouvrant le monde de la recherche. Nicolas m'a aidé en me donnant de nombreux conseils et remarques pertinentes, il m'a permis de prendre du recul sur mes travaux. Je le remercie également pour le temps et l'énergie passés dans la lecture de ma thèse.

J'aimerais remercier les chercheurs ayant accepté d'être membre de mon jury: les rapporteurs, Jean-Marc Jezequel et Oscar Nierstrasz; et les examinateurs, Theo D'Hondt et Laurence Duchien.

Merci à Alexandre Bergel, Simon Denier et Jean-Rémy Falleri. Alexandre a été Chargé de Recherche durant mon Master et les six premiers mois de ma thèse, Simon, post-doc durant mes deux premières années de thèse, et Jean-Rémy, post-doc durant ma deuxième année de thèse. Chacun d'eux m'a énormément apporté de par leur façon de travailler, de penser et par leur vision.

Je tiens également à remercier Hervé Verjus. C'est grâce â lui que j'ai pris goût à la recherche. Il m'a poussé durant mon Master et son objectivité m'a permis d'apprécier le métier de la recherche.

Plus généralement, j'aimerais remercier les communautés que j'ai cotoyées. Les communautés Moose et Pharo sont agréables, dynamiques et réactives. J'apprécie tous leurs efforts dans mes expérimentations.

Je souhaite remercier tous mes collègues. Nous sommes dans une équipe avec une forte valeur internationale. J'apprécie particulièrement l'ouverture d'esprit et l'énergie qui peut se dégager de cette équipe.

Un grand merci à Babette, Michel, Cathy, Julien, Christian et Lili pour leur soutien depuis le début de cette aventure. Merci à Guillaume pour toute l'aide à la construction des éléments périphériques, mais néanmoins importants, de ce projet.

Finalement, j'aimerais remercier Audrey mon épouse. Elle m'a suivi dans cette aventure. Elle m'a toujours soutenu par son amour, son énergie et sa patience. Elle m'a encouragé et a cru à la réussite de ce projet. Son soutien m'a été précieux.

Cette thèse est dédiée à ma fille, Éléa.

To Eléa,

Abstract

Software evolves over time with the modification, addition and removal of new classes, methods, functions, dependencies. A consequence is that behavior may not be placed in the right packages and the software modularization is broken. A good organization of classes into identifiable and collaborating packages eases the understanding, maintenance, test and evolution of software systems. We argue that maintainers lack tool support for understanding the concrete organization and for structuring packages within their context.

Our claim is that the maintenance of large software modularizations needs approaches that help (i) understanding the structure at package level and assessing its quality; (ii) identifying modularity problems; and (iii) take decisions and verify the impact of these decisions.

In this thesis, we propose ECOO, an approach to help reengineers identify and understand structural problems in software architectures and to support the remodularization activity. It concerns the three following research fields:

- Understanding package dependency problems. We propose visualizations to highlight cyclic dependency problems at package level.
- Proposing dependencies to be changed for remodularization. The approach proposes dependencies to break to make the system more modular.
- Analyzing impact of change. The approach proposes a change impact analysis to try modifications before applying them on the real system.

The approaches presented in this thesis have been qualitatively and quantitatively validated and results have been taken into account in the reengineering of analyzed systems. The results we obtained demonstrate the usefulness of our approach.

Keywords: remodularization; dependency analysis; visualization; change impact analysis; package dependency

Résumé

Les logiciels évoluent au fil du temps avec la modification, l'ajout et la suppression de nouvelles classes, méthodes, fonctions, dépendances. Une conséquence est que le comportement peut être placé dans de mauvais paquetages et casser la modularité du logiciel. Une bonne organisation des classes dans des paquetages identifiables facilite la compréhension, la maintenance, les tests et l'évolution des logiciels.

Nous soutenons que les responsables manquent d'outils pour assurer la remodularisation logicielle. La maintenance des logiciels nécessite des approches qui aident à (i) la compréhension de la structure au niveau du paquetage et l'évaluation de sa qualité; (ii) l'identification des problèmes de modularité, et (iii) la prise de décisions pour le changement.

Dans cette thèse nous proposons ECOO, une approche qui aide la remodularisation. Elle concerne les trois domaines de recherche suivants:

- Comprendre les problèmes de dépendance entre paquetages. Nous proposons des visualisations mettant en évidence les dépendances cycliques au niveau des paquetages.
- Proposer des dépendances qui devraient être changées. L'approche propose des dépendances à changer pour rendre le système plus modulaire.
- Analyser l'impact des changements. L'approche propose une analyse d'impact du changement pour essayer les modifications avant de les appliquer sur le système réel.

L'approche présentée dans cette thèse a été validée qualitativement et les résultats ont été pris en compte dans la réingénierie des systèmes analysés. Les résultats obtenus démontrent l'utilité de notre approche.

Mots clés: remodularisation; analyse de dépendance; visualisation; analyse d'impact; dépendance de paquetage

Contents

CHAPTER 1

Introduction

Contents

In a progressive country change is constant; ...change... is inevitable.

[Benjamin Disraeli]

At a Glance

This chapter introduces the domain and the context of our research. We explain the remodularization problems in large software systems. In this context, we place our approach and the solutions offered. We finish this chapter with a summary of the contributions.

1.1 Context

Software systems, and in particular, Object-Oriented systems are representations of the real world. Like the real world, these systems are not static, they should evolve with new features, new rules, new paradigms... and have to deal with constraints like memory consumption, response delay, user interface, new features...

Software systems must be continuously updated or should risk becoming outdated and irrelevant [Lehman 1996]. Moreover, it becomes difficult to analyze the complete software systems because of their increasing size. For example, the Windows operating system consists of more than 60 millions lines of code (500,000 pages printed double-face, about 16 times the Encyclopedia Universalis).

Most of the effort during a software system lifecycle is spent in supporting its evolution [Sommerville 1996]. It is well known that up to 80% of the total cost of software development project is spent in maintenance and evolution of existing features [Davis 1995, Erlikh 2000].

Software maintenance and evolution is hard because maintainers have to deal with large source code systems. Reengineers have to spend a large part of their time understanding the system. Corbi [Corbi 1989] estimates the portion of time invested in program comprehension to be between 50 and 60 %.

Maintaining such large applications is a trade-off between having to change a model that nobody can understand in details and limiting the impact of changes. Purushothaman and Perry found that 40% of bugs are introduced while fixing other bugs, because understanding the complete implications of a change in a large source code system is not really possible [Purushothaman 2005]. A simple change can be scattered over the system because of code duplication for example.

In short, software systems evolving during years are difficult to understand and change. Maintainers need help maintaining them.

1.2 A Significant Modularization Problem

Software evolves over time with the modification, addition and removal of new classes, methods, functions, and dependencies. A consequence is that some classes may not be placed in the right packages and the software modularization is broken [Eick 2001, Griswold 1993].

The studies of Eick [Eick 2001] shows that software code decays: as software systems evolve over time to meet requirements and environment changes, with the modification, addition and removal of new classes and dependencies, the software systems become more complex and their modularization looses quality [Lehman 1985].

The following two laws of Lehman and Belady illustrate this situation [Lehman 1996]. They stress that software must continuously evolve to stay useful and that this evolution is accompanied by an increase of complexity.

- Continuous Changes. *"An E-type program (i.e., a software system that solves a problem or implements a computer application in the real world) that is used must be*

continually adapted else it becomes progressively less satisfactory." [Lehman 1996]

- Increasing Complexity. *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."* [Lehman 1996]

These two laws, deduced from empirical studies, show the need and the difficulty of software evolution. In addition, companies face the problem that most of the changes cannot be predicted (unanticipated changes) because they are driven by the market or emerging technologies.

As a consequence, software modularization must be maintained. In that respect, it is then important to understand, to assess and to optimize the concrete organization of packages and their relationships.

1.3 Package Granularity

Classes and their relationships represent the static structure of the software system and maintainers need to understand this structure for maintaining and upgrading software systems. To solve the problem of complexity of large object-oriented software systems, software developers organize classes into subsystems using the concepts of package. A package is a unit of reuse and deployment: it is built, tested, and released as a whole as soon as one of its classes is changed, or used elsewhere [Martin 2000]. We name software modularization the organization of classes into multiple packages. Developers organize them by different criteria, they are complex entities that represent code ownership, feature containment, team organization, deployment entities [Abreu 2001]. Modularity implies that releasing a new package should impact the dependent packages in the building process.

A package provides and requires services from other packages. They can play central roles or peripheral [Ducasse 2007]: some packages act as reference hubs, others as authorities. Packages have different usage patterns, often depending on the clients that use them [Abdeen 2008]. These multiple views of packages do not ease the understanding and the maintenance. Researchers in object-oriented programming, have claimed that a good organization of classes into identifiable and collaborating packages eases the understanding, maintenance, test and evolution of software systems [DeRemer 1976, Myers 1978, Yourdon 1979, Pressman 1994, Ponisio 2006a].

As we explained before, software system evolution makes the software modularization complex and unclear. A reason of their complexity is that the source code is composed of a large collection of interdependent classes that mutually communicate to produce desired behavior.

It makes packages complex entities, they contain classes communicating inside and outside their scopes. Understanding them and their role (*e.g.,* core package, UI class container, tests package ...) is important for reengineering because making changes to a package may impact the entire system depending on its role. Modifying a core package can have a large impact, whereas modifying a peripheral package should have a low impact on other packages. Maintainers need to know the role of a package before modifying it.

Software remodularization, as a software maintenance task, is done by maintainers and more precisely by reengineers. This task is to make a better package organization after a structure deterioration. Reengineers have to: (i) understand the structure at package level and at class level and assess its quality; (ii) understand where are structural problems; and (iii) take decisions and verify the impact of these decisions.

Package organization represents the backbone of large software systems. It is largely acknowledged that packages should form layered structures [Bachmann 2000,Demeyer 2002]. One of the problems of remodularization is package cycles. Cycles between packages have a high impact on the modularity of applications and break layered structure. Indeed, a cycle in the package dependency graph implies all packages in the cycle to be tested, and released together as they depend on each other. Martin [Martin 2000] proposes the Acyclic Dependencies Principle (ADP), which states that there should not be any cycle between packages.

While it is easy to detect package cycles as soon as they are introduced (as JooJ does on classes [Melton 2007b]) and correct them, the problem is more difficult in legacy software where no cycle assessment has been performed during the software development. It is difficult to understand the interweaving of dependencies and difficult to devise an efficient plan for breaking cycles.

Although languages such as Java make dependencies between packages explicit (i.e., via the import statement), reengineers lack tool support to understand the concrete organization and structure of packages within their context.

Existing approaches for understanding the structure of packages and software modularization [Ducasse 2005c, Lungu 2006, Ponisio 2006a, Ponisio 2006b, Ducasse 2007, Dong 2007a] do not provide a fine-grained view of packages and do not provide information about the remodularization effort. In these approaches, it is not easy to understand the place of a package in the system, particularly when this system grows.

Moreover, developers need to make choices about future system structure such as changing the dependencies between packages.

Existing approaches lack of (i) a deep understanding fine-grained package structure and dependencies; (ii) an identification of package dependency problems; and (iii) an analyze of the impact of a change on the package structure.

> **Research question:** *How can we identify structural problems in software applications and help reengineers in their remodularization task?*

1.4 Our Approach in a Nutshell

> **Thesis:** *In large software architectures, we need to identify unwanted dependencies between packages causing structural problems and help proposing solutions to avoid modularity problem.*

Many current software systems are being (or already are) implemented in object-oriented languages (e.g., Java, C++ and Smalltalk); as a consequence, those systems will represent

future legacy software systems to maintain. This thesis focuses on object-oriented software systems.

Particularly, we work on the notion of packages, which is supported by Object oriented languages such as Java, Smalltalk and C++. The idea behind package is to improve the quality of software, *e.g.,* adaptability and changeability, by organizing classes into identifiable groups. It helps engineers to easily understand, maintain, test and evolve large object-oriented software systems [DeRemer 1976, Myers 1978, Yourdon 1979, Pressman 1994].

In this thesis, we propose new analyses to help engineers to understand and restructure large existing applications (adapted visualizations and layer identifications) for software evolution. In a first part, we propose approaches to support the understanding of software applications at the level of packages. It provides information to highlight modularity problems (*i.e.,* cycles between packages). Based on these analyses, an algorithm provides an identification of layers based on a semi-automatic classification of package dependencies. The semi-automatic approach is important because it supports the knowledge of the maintainer. Finally we provide an approach that allows maintainers to try transformations on model of the software system and provide feedback on the impact of the suggested transformations.

The approach is named ECOO for ECell, Ozone, Orion. It is centered on the need of the reengineer to control changes. It provides visualizing tools and semi-automatic algorithm to provide propositions for changes. Figure 1.1 provides the overview of the approach. In this approach, we propose two visualizations EDSM and CYCLETABLE, which help in understanding structural problems at package level, including ECELL to have a detailed view of a dependency. We also propose OZONE an algorithm that computes dependencies involved in cycles and propositions to remodularize software architecture. It uses a strategy based on EDSM and CYCLETABLE heuristics. In addition, we propose ORION which provides an infrastructure to analyze impact of changes in large software architecture. On top of ORION, we already plugged EDSM, CYCLETABLE and OZONE.

These steps will be implemented on top of the Moose[1] open-source reengineering environment and validated on multiple open-source applications.

1.5 Contributions

The novelty of this dissertation resides in providing a solution to resolve cycles between packages in putting the reengineer at the center of the solution. In fact, the reengineer is the sole "entity" which can analyze different solutions and take final decisions. In this context, we made the following contributions:

1. *Identifying cycle causes with Enriched Dependency Structural Matrix* [Laval 2009a]. In this paper, we propose EDSM, an approach to enrich Dependency Structural Matrix with ECELL, a view displaying the internals of a package dependency. We adapt EDSM for software reengineering with contextual information about (i) the type of

[1] More information about Moose, see: http://www.moosetechnology.org/

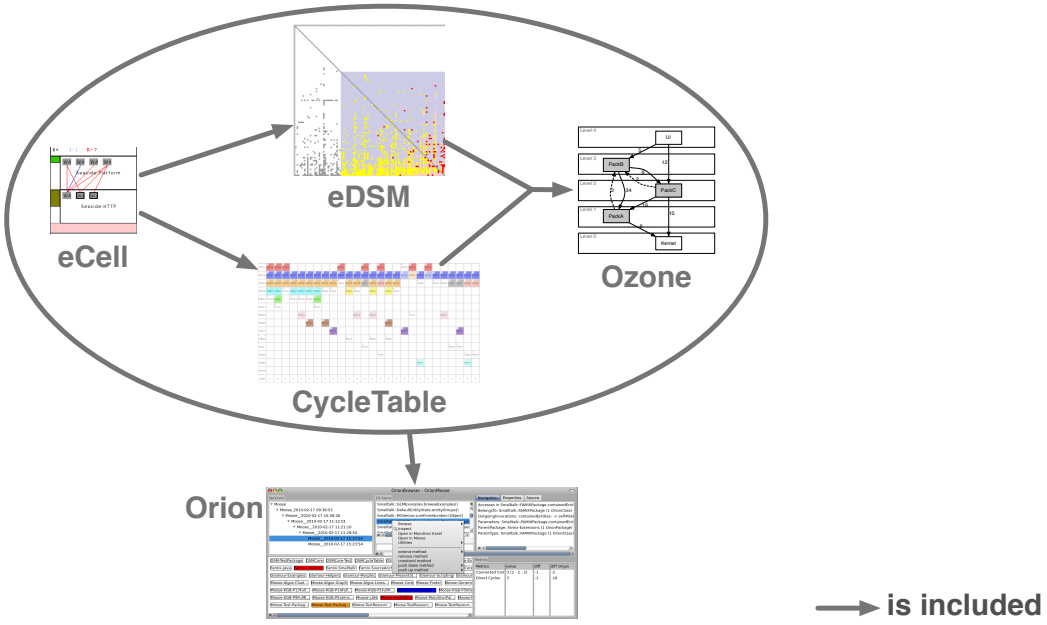


Figure 1.1: Our approach.

dependencies (inheritance, class reference, ...); (ii) the proportion of referencing entities; and (iii) the proportion of referenced entities. We highlight strongly connected component (SCC) and stress potentially simple fixes for cycles using coloring information.

2. *Cycles Assessment with CycleTable* [Laval 2010b]. CYCLETABLE presents (i) a heuristic to focus on *shared* dependencies between cycles in SCC; and (ii) a visualization highlighting dependencies to efficiently remove cycles in the system. This visualization is completed with ECELL (small views displaying the internals of a dependency).

3. OZONE*: Package Layered Structure Identification in presence of Cycles* [Laval 2010a]. OZONE is an approach which provides (i) a strategy to highlight dependencies which break Acyclic Dependency Principle; and (ii) an organization of package in multiple layers even in presence of cycles. While our approach can be run automatically, it also supports human inputs and constraints.

4. *Supporting Simultaneous Versions for Software Evolution Assessment* [Laval 2010c]. We propose ORION, an interactive prototyping tool for reengineering to simulate changes and compare their impact on multiple versions of software source code models. Our approach offers an interactive simulation of changes, reuses existing assessment tools, and has the ability to hold multiple and branching versions simultaneously in memory. Specifically, we devise an infrastructure which optimizes

memory usage of multiple versions for large models. This infrastructure uses an extension of the FAMIX source code meta-model but it is not limited to source code analysis tools since it can be applied to models in general.

1.6 Structure of the Dissertation

This thesis is organized as follows:

Chapter 2 (p.9) analyzes the problem of the maintenance of large and complex object oriented software modularizations. It also evaluates existing approaches that tried to solve these problems.

Chapter 3, ECOO (p.25), presents the overall approach. It defines the terminology used in the thesis. It also argues that a cycle between packages cannot be easily removed and shows an overview of the complete approach explained in the next chapters.

Chapter 4, ECELL (p.39), proposes a visualization to understand at fine-grained level the content of a package dependency. It provides information to help the reengineer to understand at a glance the complexity of the dependency. The user study shows that it is helpful to highlight dependency problems.

Chapter 5, EDSM (p.55), proposes a matrix-based visualization built around ECELLs to provide micro-macro reading. It allows reengineers to identify some package architecture problems by highlighting cycles between packages. It also helps during reengineering task by providing information about easy-to-fix dependencies. The validation shows that EDSM is a helpful approach that gives enough information to help reengineers take decisions about potentially easy-to-fix dependencies.

Chapter 6, CYCLETABLE (p.83), proposes a cycle-centric visualization. By the decomposition of Strongly Connected Component, it highlights dependencies that have a high impact on cycles to help reengineers remove these cycles with minimal effort. The validation shows that CYCLETABLE is better than a node-link visualization for the detection of this kind of dependency.

Chapter 7, OZONE (p.103), provides a semi-automatic algorithm based on our experiments performed with EDSM and CYCLETABLE. It proposes dependencies that can be removed to have a system without cycles. A layered view in presence of cycles is shown to help engineers to evaluate these propositions. The case study shows that this approach provides good results but can be improved with more specific parameters.

Chapter 8, ORION (p.127), proposes an infrastructure to analyze impact of changes on large software models. It provides an architecture where multiple versions of a software model are loaded jointly and can be compared. ORION uses an optimized representation that is shown to improve memory usage while the validation shows that the use of memory is low and the access speed to entities is a bit slower than a sole model.

Chapter 9 (p.155) summarizes how our proposals satisfy the requirements identified in Chapter 2 (p.9) for the remodularization of large software systems. The chapter ends with an outlook on the opened future work.

CHAPTER 2

State of the Art

Contents

One never notices what has been done; one can only see what remains to be done.

[Marie Curie]

At a Glance

This chapter introduces the work related to our research. We detail four domains: (i) software visualization for structure assessment; (ii) remodularization approaches; (iii) regression and integration testing; and (iv) software change impact analysis. We show that these approaches do not provide enough information to help reengineers remodularize software systems.

Keywords: Software visualization, analysis for remodularization, change analysis.

2.1 Introduction

In Chapter 1 (p.1) we argue that a legacy system reengineer must perform four goals: he should (i) identify problems; (ii) solve these problems; (iii) avoid the degradation of the system; and (iv) minimize change costs. In this chapter we show that there is a lack of approaches to perform these tasks. We organize this chapter in three sections corresponding to the three domains related to our work: (i) software visualization for structure assessment; (ii) remodularization approaches; and (iii) software change impact analysis.

Structure of the Chapter

In the next section we introduce related work to software visualization for structure assessment. In Section 2.3 (p.14) we introduce related work to software analysis for remodularization approaches. In Section 2.4 (p.17) we present related work to regression and integration testing. In Section 2.5 (p.20) we detail related work to software change impact analysis. Finally, Section 2.6 (p.23) summarizes the chapter.

2.2 Software Visualization for Structure Assessment

We identify two main types of approaches to obtain information about packages and their relationships: metrics and visualizations. Metrics allow one to measure the quality of a software entity (*i.e.,* global software, package, class, method...). Visualizations provide visual information about the state of the software application. Our work is about visualizing information, consequently, we do not provide a state of the art of metrics approaches. But we participated in Squale deliverable [Balmas 2009b, Ducasse 2009a, Balmas 2009a, Denier 2010a, Denier 2010b, Mordal-Manet 2011], which provide a quality model based partly on metrics.

There is a significant effort to create efficient software visualizations to support the understanding and analyses of applications [Langelier 2005, Maletic 2001, Marcus 2003, Wettel 2007a]. Lanza and Ducasse worked on system level understanding combining metrics and visualization [Lanza 2003b] and class understanding support [Ducasse 2005b].

Many approaches, mainly based on visualization techniques [Healey 1992, Tufte 1997, Tufte 2001, Ware 2000], have flourished to produce a representation of the software structure [Ducasse 2005b, Dong 2007b, Langelier 2005, Lanza 2003a, Lungu 2006, Sangal 2005, Storey 1997, Wettel 2007a, Wettel 2007b, Wysseier 2005]; to show how properties are spread in a population of packages [Ducasse 2006]; to identify software bugs and to understand software evolution [D'Ambros 2006, D'Ambros 2007, Lanza 2001]. Other approaches have also used metrics to assess software design quality [Lanza 2002, Martin 2002, Ponisio 2006b, Pinzger 2005]; to assess the effort of maintaining package structure [Hautus 2002]. Few of these approaches are for addressing the problem of package understanding [Ducasse 2005c, Lungu 2006, Martin 2002, Ponisio 2006b].

In this section, we detail the main important visualization approaches related to our work.

Package Blueprint. It takes the focus of a package and shows how such package uses and is used by other packages [Ducasse 2007]. It shows three different kinds of visualizations: (i) packages used by a selected package; (ii) packages using a selected package; and (iii) packages inheriting from a selected package. Figure 2.1 shows the principle (left) and a view (center and right) of package blueprint.

It provides a fine-grained view because it shows the class relationships within packages. However, package blueprint lacks (1) a global view of the system structure and (2) consequently, the identification of cycles at system level.

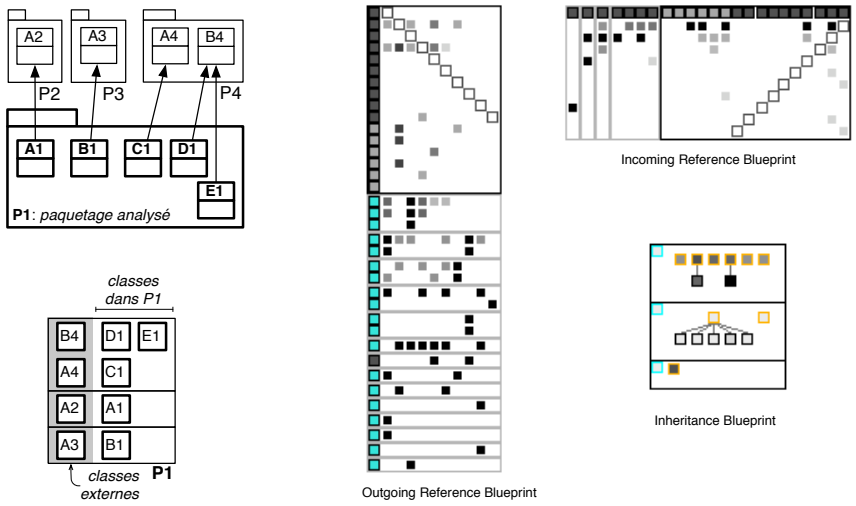


Figure 2.1: (left) Principle of Package Blueprint with 4 packages. (right) The three visualization provided by Package Blueprint at work.

Node-link Representation. Often node-link visualizations are used to show dependencies among software entities. Several tools such as dotty/GraphViz [Gansner 2000], Walrus [Munzner 2000] or Guess [Adar 2006] can be used. Using node-link visualization is intuitive and has a fast learning curve. One problem with node-link visualization 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.

Figure 2.2 shows the dependencies inside the core of Pharo1.0, composed of 70 packages. With a node-link visualization, it is difficult to see where are the problems. Here package cycles are represented by red links. Figure 2.4 shows the same information provided in a Dependency Matrix. It is difficult to find a single layout which present well in all cases.

Holten proposed Hierarchical Edges Bundles (HEB) (Figure 2.3), an approach to improve the scalability of large hierarchical graph visualizations. Edges are bundled together based on hierarchical information and it uses color schema to represent the flow of information [Holt 2006, Holten 2009]. It has been applied to see the communication between

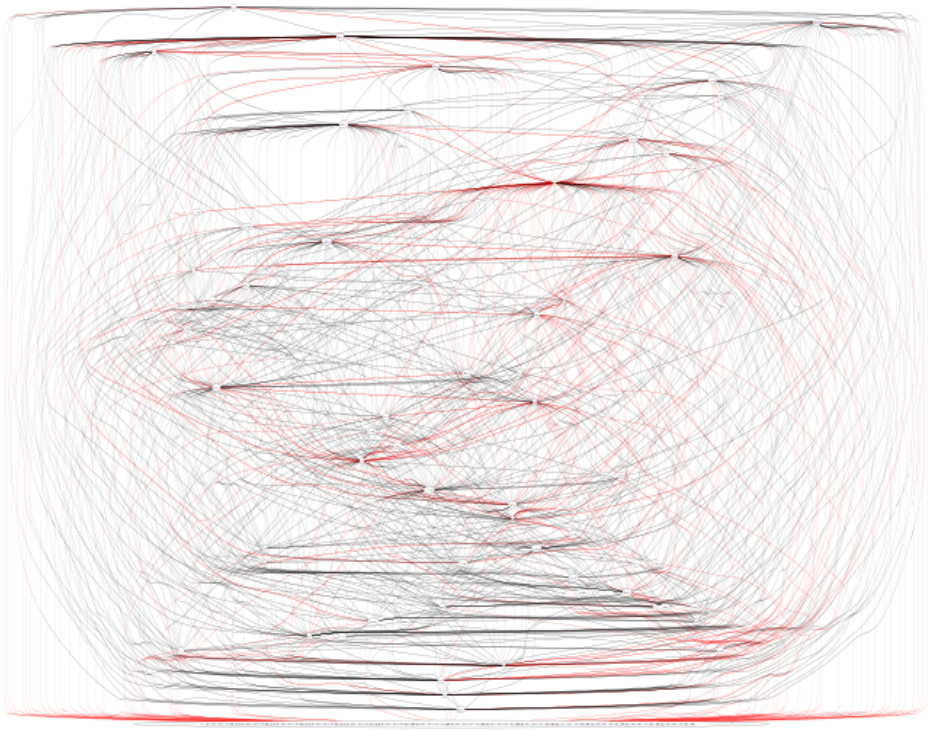


Figure 2.2: Dependencies between 70 packages of Pharo 1.0.

classes grouped by packages in large systems and the bundling of edges produces less cluttered display. But, it is difficult to identify package dependencies patterns, as nodes are positioned in circle, creating many link crossings. Henry et al [Henry 2007] have also reported this limit.

Dependency Structural Matrix (DSM). Node-link visualizations are known to be intuitive and to work well on sparse graphs, while matrices perform better especially for detailed analysis of dense graphs [Ghoniem 2005]. Contrary to node-link, a DSM visualization preserves the same structure whatever the data size. This feature enables the user to dive fast into the representation using the normal process. Cycles remain clearly identified by colored cells, there is no edge between packages, and so this reduces clutter in the visualization. However, DSM does not provide fine-grained information about dependencies between packages. Classes involved in a dependency are not shown in cells of a DSM. Figure 2.4 shows the same information as in previous node-link view (Figure 2.2) in a colored DSM.

Hybrid Approaches. Henry et al. propose NodeTrix (Figure 2.5), a hybrid graph visualization mixing matrices for local details and node-link visualization for the overall structure [Henry 2007]. It is tailored for the globally sparse but locally dense graphs.

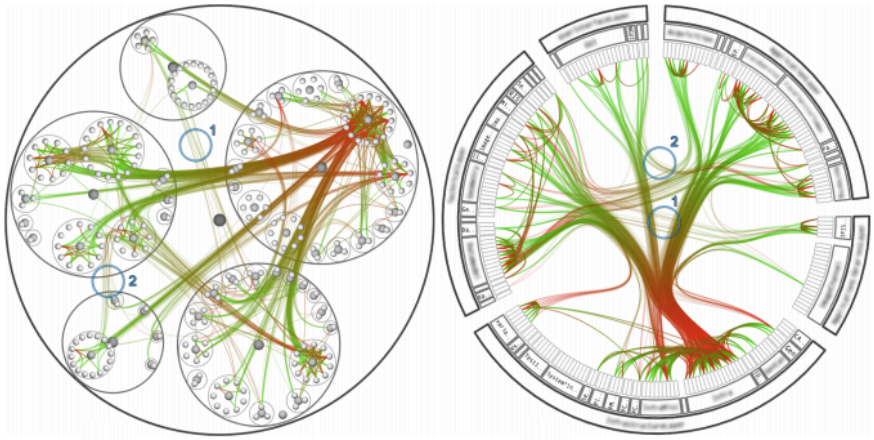


Figure 2.3: Hierarchical Edges Bundles (HEB) where green represents caller and red represents called.

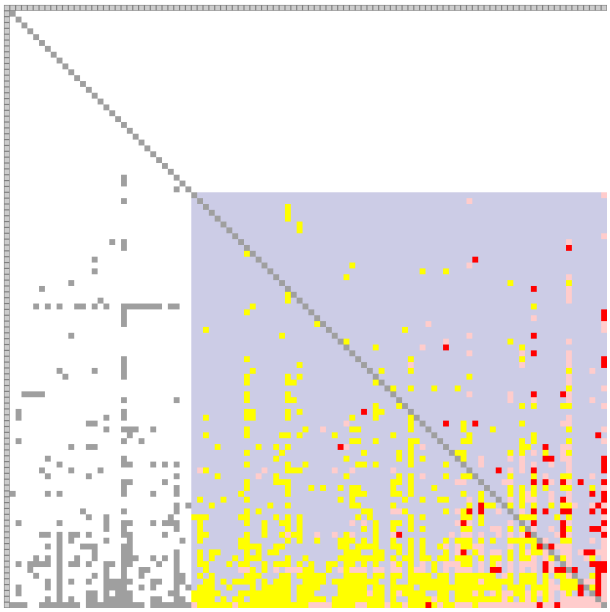


Figure 2.4: A Dependency Structural Matrix of Pharo 1.0: the blue square represents packages in cycle, the red cells packages in direct cycles.

Abello and van Ham present Matrix Zoom, a scalable hybrid visualization for hierarchical sets of data, using matrices only for fine-grained details [Abello 2004]. The user navigates into the hierarchy and can get some details on the focused subset in the matrix.

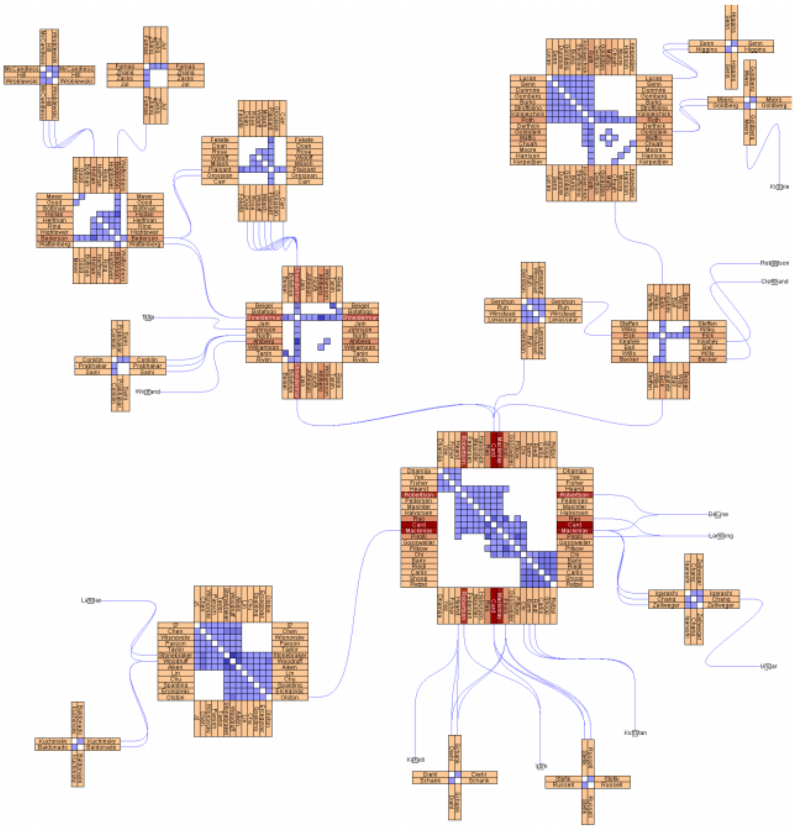


Figure 2.5: NodeTrix is a visualization mixing matrices and node-link.

As a Conclusion

> *Existing visualizations lack details for remodularization. These approaches do not provide a fine-grained view of packages that would help maintainers understand the causes of the problems: understanding package dependencies; identifying unwanted dependencies; identifying package roles within a system*

2.3 Software Analysis for Remodularization

It is a well-known practice to layer applications with bottom layers being more stable than top layers [Martin 2002]. Until now few approaches have been proposed in practice to identify layers: Mudpie [Vainsencher 2004] and Sotograph [Bischofberger 2004] represent a first cut at identifying cycles between packages as well as package groups potentially

representing layers. From the side of remodularization algorithms, lots of them where defined for procedural languages [Koschke 2000]. However object-oriented programming languages bring some specific problems linked with late-binding and the fact that a package does not have to be systematically cohesive since it can be an extension of another one [Wilde 1992].

Melton et al. [Melton 2007a] proposes an empirical study of cycles among classes. They compute metrics to define better candidate dependencies to remove. They particularly indicate that it is crucial to take into account the semantic of the software architecture to avoid breaking dependencies that should not be broken.

2.3.1 Dependency Analysis

On Packages. Dong and Godfrey [Dong 2007a] propose an approach to study dependencies between packages and to give a new meaning to packages with (1) characterization of external properties of components, (ii) usage of resource and (iii) connectors. It helps the maintainers to understand the nature of package dependencies. This kind of tool is useful to understand a global system.

Lungu et al. [Lungu 2006] propose a collection of package patterns to help reengineers understand large software system. They propose to recover architecture based on package information and an automatic process to recover defined patterns. Then they propose a UI to interact with the package structure. This approach is useful to understand the behavior of a package in the system. It can provide information about the position of a package in a layered organization. This kind of pattern could be used to add more information on a package and to propose more information about the breaking of a dependency, for example knowing that a package is autonomous is valuable information.

Bunch [Mancoridis 1999, Mitchell 2006] is a tool which remodularizes automatically software. It proposes to decompose and to show an architectural-view of the system structure based on classes and function calls. It helps maintainers to understand relationships between classes. This tool breaks the package concerns and does not provide the information we need to make a layered organization of a package system.

PASTA [Hautus 2002] is a tool for analyzing the dependency graph of Java packages. It focuses on detecting layers in the graph and consequently provides two heuristics to deal with cycles. One considers all packages in the same strongly connected component as a single package. The other heuristic selectively ignores some undesirable dependencies until no more cycle is detected. Thus, PASTA reports on these undesirable dependencies that should be removed to break cycles. The undesirable dependencies are selected by computing a class relationship weight and selecting the minimal ones.

On Classes. The Kleinberg algorithm [Kleinberg 1999] defines authority and hub values for each class in a system. A high authority means the class is used by a big part of the system, and the hub value means the class uses multiple other classes in the system. Scanniello et al. [Scanniello 2010] propose an approach to build layers of classes based on this algorithm. They identify relationships between classes and use the Kleinberg algorithm to group them into layers. They propose a semi-automatic approach that allows the maintainer

to manipulate the architecture and adds its own meaning of the system. Zaidman [Zaidman 2006] uses this algorithm on a dynamic analysis to measure the runtime coupling and retrieve classes that need to be understood early on in the program comprehension process.

JooJ [Melton 2007b] is an eclipse plugin (not released) to detect and remove as early as possible cycles between classes. The principle of JooJ is to highlight statements creating cycles directly in the code editor. It computes the strongly connected components to detect cycles among classes. It also computes an approximation of the minimal set of edges to remove in order to make the dependency graph totally acyclic. This problem is called minimum feedback arc set in the graph literature (explained in the next paragraph). It highlights the minimum number of statements that one needs to remove in order to break all cycles among classes. However, no study is made to validate this approach for cycle removal. It is possible that the selected dependencies should not be removed because they are valid in the domain of the program.

Feedback Arcset. In graph theory, a feedback arcset is a collection of edges that should be removed to obtain an acyclic graph. The *minimum* feedback arcset is the minimal collection of edges to remove to obtain an acyclic graph. This theoretical approach cannot be used for three particular reasons: (i) it is an NP-complete problem (optimized by Kann [Kann 1992] to become APX-hard). Some approaches propose heuristics to compute the Feedback Arc Set Problem in reasonable amount of time [Eades 1993]; (ii) it does not take into account the semantic of the software structure. Optimizing a graph is not equivalent to an acceptable solution at the software level; and (iii) the goal of breaking cycles in software applications is not to break a minimal set of links, but the more pertinent ones.

2.3.2 Search-based Software Engineering

Some approaches based on Formal Concept Analysis [Snelting 1998] show that such an analysis can be used to identify modules. However the presented examples are small and not representative of real code. Other clustering algorithms [Jain 1988, Jain 1999] have been proposed to identify modules [Mancoridis 1999, Mitchell 2006]. Once again, the specific characteristics of object-oriented programming are not taken into account, like late binding.

Lutz [Lutz 2001] proposes a hierarchical decomposition of a software system. It uses a genetic algorithm to find the best way to group components of the system into coarse-grained components. Mudpie [Vainsencher 2004] is a tool to help the maintenance of software system by bringing out Strongly Connected Components (defined in Section 3.3, p.28) and focusing on dependencies in cycle. Approaches exist to decompose a system by using genetic heuristics [Abdeen 2009].

Mancoridis and Mitchell [Mancoridis 1998, Mancoridis 1999] introduced a search-based approach based on hill-climbing clustering technique to cluster software modules (classes in our context). Their approach starts with an initial random modularization. The clustering algorithm clusters each of the result and selects the result with the highest quality as the suboptimal solution. Recently, they used Simulated Annealing technique to optimize resulting clusters [Mitchell 2002, Mitchell 2006, Mitchell 2008]. Their optimization ap-

proach creates new modularizations by moving randomly some classes (a block of classes) to new clusters. The goal of their approach is increasing cluster internal dependencies. The main problem is the random move of classes, it breaks the coarse-grained architecture of the system [Abdeen 2009].

Harman et al. [Harman 2002] proposed genetic algorithms to partition software classes into subsystems (*i.e.,* packages). Their algorithms start with an initial population of modularizations. These algorithms apply genetic operators on packages to modify current modularizations and/or create new modularizations into the population.

The goal of both approaches is increasing package internal dependencies. For that, they forget the idea that a package is a structural entity that engineer use to group classes. These approaches move classes without taking into account the package structure.

As a Conclusion

> *These approaches are often not customized for object-oriented applications. For example, they do not take into account the late binding paradigm. Existing solutions produce new architecture of the software application automatically, which is quite impossible to analyze. They break the meaning of the architecture and reengineers need to learn a new architecture. For example, Falleri et al. [Falleri 2008] show that Relational Concept Analysis, applied to rich UML descriptions including references between concepts produce a huge number of artifacts.*

2.4 Regression and Integration Testing

Regression and Integration testing consists on combining and testing multiple components as a group. This phase is important for the evolution of software application. For object-oriented programs, one of the problems is the interaction between components. These components can be at different levels: algorithmic level, class level, cluster level and system level [Chen 2001, Frankl 1994].

When integrating components, it is sometimes difficult to know the order of integration because of heavy interaction amongst them. Consequently, engineers can have trouble integrating components, which affect the quality and reliability of the application.

Researchers worked on strategies to find an optimal order of components for testing. It means first of all understanding communication and coordination between components. Then, based on this knowledge, strategies will find an order of components, which will create stubs to ensure the integration process.

Chan et al. [Chan 2002] propose an overview of integration testing techniques at the cluster level (*i.e.,* multiple classes in interaction). The conclusion is that all the listed approaches used for object-oriented programs are simply extensions of techniques used in the imperative programming techniques. Shashank et al. provide a literature survey of integration testing in component-based software engineering [Shashank 2010]. The authors show that there are four main challenges: (i) lack of source code, (ii) heterogeneity of

the components, (iii) incomplete interface specification, and (iv) difficulty in identifying dependencies.

For the last challenge, researchers work on approaches close to remodularization work. This challenge is now detailed.

2.4.1 Understanding Interactions between Components

In direct relation with our work, two challenges are related: (i) understanding communication and coordination between components, and (ii) finding the better strategy to order component integration [Briand 2003].

In remodularization domain, we want to understand the interaction between components to make them more modular (*i.e.,* changing the structure of the application). Whereas in regression and integration testing researchers want to understand the interaction between components to find an integration order (*i.e.,* defining components more important than others).

To ensure the integration process when classes communicate heavily, the system build class stubs to avoid integration errors, then replace these stubs by the real classes. Researchers work on minimizing the number of these stubs. Numerous approaches provide algorithms to order dependencies among classes [Briand 2001, Kung 1996, Labiche 2000, Le Traon 2000, Tai 1997] and minimize the number of stubs needed. Kung et al. propose the first paper about the integration problem [Kung 1995]. Limited to a class dependency graph with no cycles, they propose a topological sorting of classes. If there are cycles between classes, they propose to break the cycles performing a random selection.

2.4.2 Solving Integration Problems

When there are cycles between classes or components to integrate, the problems are to choose the classes to integrate first. Some approaches in the regression and integration testing domain have been proposed to minimize the testing effort. This effort can be computed from the number of stub creation, and from stub complexity.

Kung et al. After proposing a simple strategy to order classes [Kung 1995], Kung et al. worked on a better strategy for integration testing [Kung 1996]. They differentiate three kinds of dependencies: inheritance, aggregation and association, and they argue that in each cycle, there is at least an association dependency. They propose to search all strongly connected components (SCC) [Tarjan 1972], and break cycles by removing associations because it is the weakest of the three kinds of dependencies.

Tai and Daniels propose an algorithm that gives two numbers to each class of the system: a *major* number based on aggregation and inheritance, and a *minor* number based on associations [Tai 1997]. It uses the same idea as Kung et al. [Kung 1996] that already differentiate association from inheritance and aggregation. The strategy build a layered architecture based on the *major* number where classes in a layer *n+1* depends on classes in layer *n*. Then to break cycles, a weight function is computed on each layer with the sum

of incoming and outgoing association of classes in the same layer. The strategy is to create stubs for classes with a high weight, because the higher the weight is, the more cycles are broken.

Le Traon et al. propose a model to break strongly connected components (SCC) and build stub based on the generated acyclic graph [Le Traon 2000]. The algorithm is an improvement of the Bourdoncle algorithm [Bourdoncle 1993]. It computes a weight for each vertex in a SCC based on incoming and outgoing dependencies inside the cycle. Then, in each SCC, it selects the vertex with maximal weight and removes incoming dependencies. The goal of this algorithm is to minimize stub creation in testing.

Briand et al. propose a strategy that combines the two previous strategies to address some of their drawbacks [Briand 2001]. The strategy computes Strongly Connected Components like Le Traon et al. [Le Traon 2000]. Consequently, it computes the weight of each association dependency based on the Tai and Daniels algorithm [Tai 1997], and finally break the dependency with the highest weight.

Genetic Algorithm. Briand et al. propose an algorithm using inter-class coupling measurements to define an optimal order of class integration [Briand 2002]. This algorithm tries to recover the best choice for stub minimization. In [Da Veiga Cabral 2010], authors propose to optimize this algorithm by refining the cost function of the genetic algorithm with a multi-objective optimization algorithm.

Comparative Studies. Le Hanh et al. [Hanh 2001] propose an experimental comparison of four approaches [Kung 1996, Tai 1997, Le Traon 2000, Briand 2002] to break SCC for stub minimization. In [Briand 2003], Briand et al. review and compare three strategies [Tai 1997, Le Traon 2000, Briand 2001] for breaking cycles based on graph. This study shows: (i) Tai and Daniels's algorithm [Tai 1997] creates unnecessary stubs, (ii) Le Traon et al.'s algorithm [Le Traon 2000] optimizes the number of stubs but produces arbitrary choice, and (iii) Briand et al.'s algorithm [Briand 2001] has not these kind of problems.

In [Bansal 2009], Bansal et al. propose a comparison of various graph-based strategies. They show that genetic algorithm find optimal solutions but must be run many times. They also show that Le Traon et al.'s algorithm [Le Traon 2000] minimize the number of stubs compared to other algorithm, which is the goal of the algorithm.

As we will show in Chapter 7 (p.103) such algorithms are close to our solution.

As a Conclusion

In Regression and Integration Testing domain, researchers work particularly on minimizing the testing effort by stubbing elements that have the minimal impact on the system. Researchers from this domain produce algorithm that could be used to understand remodularization at package level.

2.5 Software Change Impact Analysis

The domain of change impact analysis deals with computing (and often predicting) the effects of changes on a system. We structure this domain in two parts in relation with our work that we want to compare to: change impact analysis and versioning mechanism.

2.5.1 Program Slicing

Program slicing is an approach extracting all instructions that have an impact on the value of a specific variable at a specific point of a program. It is used to identify change impact at low level (*i.e.,* variable and instruction).

CodeSurfer [Anderson 2001, Anderson 2005] is a tool that provides a number of analyses, including pointer analysis, and creates a dependency graph of the program. It allows the user to browse dependencies. It uses program slicing to understand which part of a program is impacted by a variable, and which are the parts of a program that can impact a particular variable or a given source-code element in general [Welser 1984, Tip 1995, Bohner 1996]. In the context of software maintenance, such program slicing techniques have been widely adopted for procedural source code [Gallagher 1991].

Program slicing is dedicated to low-level analysis. Our work is at the architecture level (*i.e.,* classes and packages).

2.5.2 Change Impact Analysis

Compared to Software Configuration Management (SCM) and Revision Control System, which supports change persistence and comparison, the domain of change impact analysis deals with computing (and often predicting) the effect of changes on a system. Our approach is orthogonal to change impact analysis. Tools performing change impact analysis can be used complementary to our infrastructure to perform change assessment on a version and guide the reengineer when creating new versions and testing new changes. We structure the domain in two parts: change model and change assessment, which provides tools and analyses (metrics, visualization, change prediction).

Change Model. Smalltalk basic mechanism to record changes dynamically is called a changeset. A changeset captures a list of elementary changes that can be manipulated, saved to files and re-applied if necessary. However, in Smalltalk systems, only one version of a system can be refactored at a time, even if changeset containing several versions can be manipulated. The same happens with Cheops and change-oriented methodology and IDES [Ebraert 2009]. In [Robbes 2008], the author argues that managing changes as first-class entities is better than traditional approaches. The implementation of this approach records fine-grained changes and provides a better comprehension of changes history. This approach is applied on a single version of source code.

Some models exist to support changes as a part of development. The Prism Model [Madhavji 1992] proposes a software development environment to support change impact analysis. This work introduces a model of change based on deltas that supports incremental

changes. As it is based on deltas, it is not really possible to analyze different models in parallel.

Another work in change management system is Worlds [Warth 2008]. It is a language construct that reifies the notion of program state. The author notes that when a part of a program modifies an object, all elements referencing this object are affected. Worlds has been created to control the scope of side effects. With the same idea to control side-effect but restricted to source code, ChangeBoxes [Denker 2007] propose a mechanism to make changes as first-class entities. ChangeBoxes support concurrent views of software artifacts in the same running system. We can manipulate ChangeBoxes to control the scope of a change at runtime.

Other work [Johnson 1988, MORmSETT 1993] exists in this domain, but they manage only a single branch. Worlds manage several parallel universes. The limitation of Worlds is that it only captures the in-memory side effects. This work is a source-code-based approach.

A history-based approach is Hismo [Gîrba 2005a], a meta-model for software evolution. This approach is based on the notion of history as a sequence of versions. A version is a snapshot taken at a particular moment. It makes version from the past based on a copy approach: each version is a model. Hismo is a study of the past. In [Gîrba 2005a], the author proposes some metrics to compare elements that have changed.

Han [Han 1997] considers system components (variable, method, and class) that will be impacted by a change. The approach is focused on how the system reacts to a change. Links between these components are association (S), aggregation (G), inheritance (H), and invocation (I). Change impact is computed based on the value of a boolean expression. For example a change is formalized as S~H+G, which means that a change is applied on all element associations without inheritance plus all element aggregations. This work has been reused in [Abdi 2006]. The class-based change impact model [Chaumun 2002] is based on the same semantics, with a more general model. It analyses history and identifies classes that are likely to change often.

In [Mens 2006], the authors propose taxonomy of model transformation for helping developers who want to choose a model transformation approach. In the enumeration of characteristics of a model transformation, there is no information about multiple models management.

Other models exist as [Abdi 2009] which proposes a technique based on a probabilistic model, a Bayesian network is used to analyze the impact of an entry scenario.

Change Assessment. [Li 1996] and [Lee 1998] propose an algorithm to analyze change impact based on the detection of inheritance, encapsulation, and polymorphism changes. The algorithm proposes an order of changes based on the repercussion on self, children and clients. This method is the first one applied to an object model. It is also reserved to classes.

Some approaches try to predict changeability, they assess the impact of a change to a code location by looking at previous change impact upon this location. [Cai 2007] presents a decision-tree-based framework to assess design modularization for changeability. It

formalizes design evolution problem as decision problems, model designs and potential changes using augmented constraint networks (ACN). [Antoniol 1999] uses metrics comparison to try to predict the size of evolving Object-oriented systems based on the analysis of classes impacted by a change. A change is computed as a number of lines of code added or modified, but they do not provide the possibility to compare some versions and to choose one.

Other approaches propose change impact analysis based on test regression [Ryder 2001]. [Kung 1994] proposes a change impact analysis tool for regression tests. In this paper they define a classification of changes based on inheritance, association and aggregation. They also define formal algorithms to compute impacted classes and ripple effects. The Chianti tool [Ren 2004] is able to identify tests, which run over the changed source code. They can be run in priority to test regression in the system. For each affected test, Chianti reports the set of related changes.

2.5.3 Software Configuration Management and Revision Control

Software Configuration Management (SCM) is the discipline of managing the evolution of a software system. It integrates Revision control, which is the management of changes. It is the predominant approach to save software evolution. It allows one to manage high-level abstraction evolution.

The majority of revision control systems uses a diff-based approach. They only store changes so they are efficient in memory. The domain of revision control does not provide a model that allows us to navigate between multiple versions of a model. In fact, this is not a real goal of the revision control domain.

In [Buckley 2005], three tools are compared: Refactoring Browser, CVS and eLiza. A refactoring browser transforms source code. It has basic undo mechanism but does not manage versions. So, it is really useful for refactoring source code but it works on a current model of source code. It is not really adapted for the application of various analyses on different versions. CVS (Concurrent Versions System) works on file system and supports parallel changes. However since CVS does not include a domain model of the information contained in the files they manipulate, it is difficult to use a CVS model to perform multiple analyses on various versions. It is possible but limited. The third system compared is eLiza. It has been created to provide systems that would adapt to changes in their operational environment. eLiza provides only one active configuration, which provides a sequential versioning system.

Molhado [Nguyen 2005a] is a SCM framework and infrastructure, which provides the possibility to build version and SCM services for objects, as main SCM systems provide only versioning for files. As it is flexible, the authors work on several specific SCM built on Molhado: web-based application [Nguyen 2005b], refactoring aware [Dig 2007] to manage changes and merge branches. The main topic of Molhado is to provide a SCM system based on logical abstraction, without the concrete level of files management.

As a Conclusion

Remodularization requires engineers to make choices about future system structure such as changing the dependencies between packages. While software maintainers would greatly benefit from the possibility to assess different choices to select the most adequate changes before changing the source code. There is no approach that supports the navigation and manipulation of multiple versions simultaneously in memory.

2.6 Summary

In this chapter we show that the problems of software evolution are many and varied. Software system remodularization is a challenge and needs specific approaches to ensure the evolution of the system. We consider three particular domains to help reengineers in their task: (i) visualization approaches; (ii) software analysis approaches; and (iii) change impact analysis approaches.

Current visualizations do not provide coarse and fine-grained information to help engineers (i) understand the system; and (ii) solve modularity problems (*i.e.,* cycles between packages). Software analysis approaches do not take enough into account the high level structure of the software application. Regression and integration testing provides interesting algorithms but they are not used for helping reengineers. Change impact analysis approaches do not provide a good enough infrastructure to help reengineers taking decisions.

The next chapter introduces ECOO, our approach to answer these problems. In this chapter, we will explain the challenges stressed by the approach, the terminology used and the four part of the approach.

CHAPTER 3

ECOO, An approach for package remodularization

Contents

The most exciting phrase to hear in science, the one that heralds the most discoveries, is not 'Eureka!' but 'That's funny...'

[Isaac Asimov]

At a Glance

This chapter introduces the global approach defended in this thesis. This approach is a composition of four parts. We explain each part and how together they represent the ECOO approach. We also explain the terminology used in the following of the document.

Keywords: Package granularity, modularity problem, Cycle understanding, reengineering help.

3.1 Introduction

In Chapter 2 (p.9) we show the lack of approach to perform the four reengineering goals (identify problems, solve these problems, avoid the regression of the system and minimize change costs). Our research is related to these four points. In this chapter, we present our global approach named ECOO to answer these four points at package level.

Structure of the Chapter

In the next section we introduce the challenges addressed by the ECOO approach. In Section 3.3 (p.28) we introduce the terminology used in the whole thesis. Section 3.4 (p.31) introduces the ECOO approach, explains the principle of the approach and the four handled axes. We explain in Section 3.5 (p.36) how the approach is implemented and which tools are used. Section 3.6 (p.37) explains the validation methodology, which is a validation for each axis rather than a global validation. Finally, Section 3.7 (p.37) summarizes the chapter.

3.2 Package Level Challenges

We develop an approach to make reengineering, and particularly remodularization more effective. Our goal is to provide a global approach for software architecture remodularization. Our work provides solution to five challenges of software remodularization:

Architecture Analysis. Legacy systems are often systems that have evolved for many years with several changes of development team. The two factors, age of the software and turnover of the development teams involve a significant loss of knowledge and architecture deterioration: code duplication, undesirable dependencies between entities, architecture less modular. Therefore, understanding how the package architecture is built, the internal and external package communications is a challenge with the aim of understanding the overall application structure.

The bigger an application is, the less easy to find are problems of structure. It becomes difficult to identify sources of problems. It is therefore important to understand the organization of packages and their relationships. Packages are complex structures because they contain classes and methods in relation to themselves and with the outside.

> Challenge: *Architecture becomes less modular with time. We need to understand the software architecture to ensure the evolution.*

The approach proposed in this thesis is based on static analysis of the architecture. We provide visualizations to highlight (i) architectural layers; and (ii) architectural problems.

Cycle Understanding. A cycle between multiple packages means that these packages are not independent of each other. In a modular system, it is considered as an architectural problem. The challenge is how to understand the reasons of a cycle between multiple packages to help engineers to understand and resolve modularity problems.

> Challenge: *Cycles break package modularity. We need to understand causes of cycles to remove them.*

The approach in this thesis proposes a visualization allowing the developer to identify cycles and understand dependencies in details.

Our visualization is based on the principle that a link between two packages is defined by a set of classes and methods linked by four types of dependency: inheritance, method invocation, class reference, and class extension.

Package Layers Understanding. Package organization represents the backbone of large software systems. It is usually agreed that packages should form layered structures. However, identifying such layered structures is difficult since packages are often in cycles (when there is a cycle, we cannot differentiate multiple layers).

Ideally, software is composed of several layers of packages: the first layer, named "Kernel", then the layers above depend on the layers below. The challenge here is to understand the package layer structure in presence of cycles and to propose solutions to recover modularity.

> Challenge: *A layered Architecture ensures a good modularity. We need to identify layers even in presence of cycles.*

The approach in this thesis identifies unwanted dependencies (*i.e.,* dependencies which break the layered structure) in such systems and build the layered structure considering unwanted dependencies.

Change Impact Analysis. When reengineering software systems, maintainers should be able to assess and compare *multiple* change scenarios for a given goal, so as to choose the most pertinent one. When changing a legacy system, it is difficult to predict the impact on the functionality and structure of the software. Thus, remodularization is made by iterating small changes followed by audits.

> Challenge: *To ensure that a change in the software structure is adapted to the reengineer goal, the challenge is to provide a system that can give feedback on changes and can manage multiple parallel changes.*

The approach in this thesis proposes the ORION meta-model which allows one to create versions of a same model combined with changes in the structure to compare and analyze the impacts of change.

Help for Decision. With the architecture analysis, cycle analysis, layers analysis and impact assessment tools, there is only one step to reach the help for decision. Thus, the goal is to offer possible solutions to developer, that he can validate or invalidate based on his knowledge of the software.

> Challenge: *The best way of evolution is when the reengineer take the decision and understand the evolution. The challenge is to help the reengineer to take the best decision for a specific problem*

The approach in this thesis has for goal to help engineers in their task. We provide tools that analyze software system and provide help for decision to remodularize it by targeting cycles between packages.

3.3 Terminology

In this section, we define the terminology used in this thesis.

Kinds of Dependencies. At the package level in object-oriented system, we concentrate on the following kinds of dependencies: method invocation, class access or reference, class inheritance, and class extension.

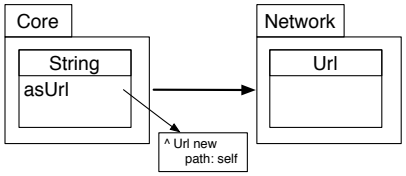
- Method *invocation*. There is a method invocation that goes from a class A to a class B if there is at least one method in class A invoking one method of class B. In dynamic typed languages (e.g., Smalltalk), we often cannot statically determine the class of the invoked method (*i.e.,* the class of the target object in the run-time). Our strategy consists in resolving candidate classes (*i.e.,* every class within which there is a method that has the invoked method signature) and filtering them by pertinence. In real case, invocations are commonly associated with a class reference.

- Class *access* and *reference*. There is a class access going from a class A to a class B if class B is explicitly used in class A code as a type of an instance and/or a class variable, or a variable/parameter type.

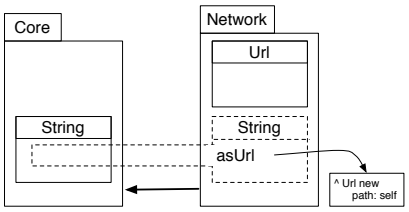
- Class *inheritance*. There is a class inheritance dependency between class A and class B if class A is a subclass of class B.

- Class *extension*. A class extension is a method defined in a package, for which the class is defined in a different package [Bergel 2005]. Class extensions exist in Smalltalk, CLOS, Ruby, Python, Objective-C and C#3. They offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. They support the layering of applications by grouping with a package its extensions to other packages. AspectJ inter-type declarations offer a similar mechanism. Figure 3.1 shows the class extension principle, particularly the inversion of the dependency when there is a class extension.

Package Dependency. A dependency from package A to package B is the union of all kinds of dependencies from classes in package A to classes in package B. In this context, A is called *client* of package B and package B is called *provider* of package A.



(a) Dependency without class extension.



(b) Dependency with class extension.

Figure 3.1: Class extension principle.

Architecture Remodularization. It is the set of techniques to make a more modular system. The goal is to make the software flexible, configurable and scalable. The issue of modularity is particularly true for legacy systems: critical systems that have evolved over the years.

Acyclic Dependency Principle (ADP). Martin defines the Acyclic Dependencies Principle (ADP) [Martin 2000]. ADP proposes that the graph of dependencies between packages should be a directed acyclic graph: there should not be any cycle between packages.

Layered Architecture. For Bachmann *et al.* [Bachmann 2000], there are two properties in a layered architecture: (i) a layer B is below a layer A if elements (*i.e.,* packages) in layer A can use elements in layer B; and (ii) layer A can use only packages below it (in Figure 3.2, layer B). In Figure 3.2 the dotted dependency from *Kernel* to *pA* should not exist. Szyperski [Szyperski 1998] and Bachmann [Bachmann 2000] make a distinction between *Closed* layering and *Open* layering. In closed layering, the implementation of one layer should only depend on the layer directly below. In this case, in Figure 3.2 the dependency from *pE* to *Kernel* should not exist. In open layering, any lower layer may be used.

A layered system offers good properties of modifiability and portability. It means that there are no cycles between packages and that dependencies do not cross non-contiguous layers [Bachmann 2000].

Strongly Connected Component (SCC). In a graph, it is the maximal set of nodes (here, packages) that depend on each other ones. In Figure 3.3, all nodes are in a single SCC. We use the Tarjan SCC algorithm [Tarjan 1972] as a reference for our implementation.

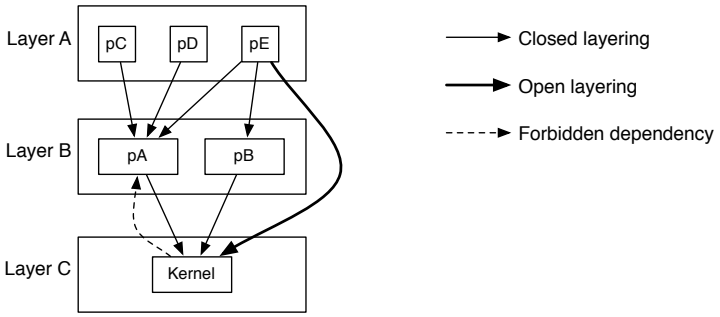


Figure 3.2: Layer Description - a dashed arrow represents an *unwanted* dependency, a thick arrow represents a dependency allowed in *opened* layering.

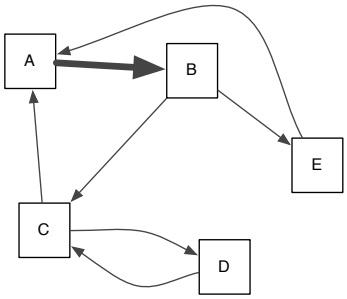


Figure 3.3: Sample SCC.

Cycle. It is a circular dependency between two or more packages. We distinguish a SCC and a cycle. A SCC is a collection of nodes, a cycle is a path that comes back to its origin. We distinguish two kinds of cycles:

- *Direct cycle*. It represents a cycle between two packages. In Figure 3.3, C and D are in a direct cycle because there is one dependency from C to D, and one dependency from D to C.
- *Indirect cycle*. It represents a cycle between more than two packages. In Figure 3.3, A, B and C are in indirect cycle. A, B and E are also in indirect cycle.

***Layer-breaking Dependency*.** It is a dependency that (i) belongs to a cycle; and (ii) seems the best dependency to remove to be able to build an "adequate" layered organization. In Figure 3.2, the dependency from "Kernel" to "pA" breaks the layered architecture.

***Minimal* Cycle.** It is a cycle with no node (here no package) appearing more than once when listing the sequence of nodes in the cycle. In graph theory, it is named a simple

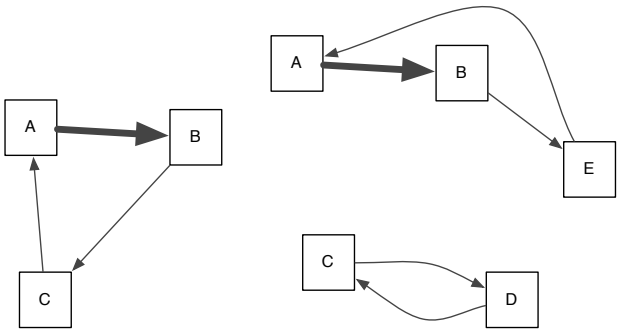


Figure 3.4: Sample SCC (Figure 3.3) decomposed into three minimal cycles.

cycle. In Figure 3.4, A-B-E and A-B-C are two different minimal cycles, but A-B-C-D-C is not because C is present twice. A-B-C-D-C can be reconstructed with the two minimal cycle A-B-C and C-D. To retrieve minimal cycles, some algorithms exist as [Tiernan 1970, Weinblatt 1972]. We use the algorithm proposed by Falleri et al. [Falleri 2010], which is the most recent.

***Shared* Dependency.** It is a dependency presents in at least two minimal cycles. In Figure 3.4, the edge between A and B is shared by the two minimal cycles A-B-E and A-B-C.

3.4 ECOO: Package Cycle Remediation

ECOO (for ECELL, OZONE, ORION) provides a global approach to understand and resolve modularity at the package level in software architecture. First, we provide ECELL, a visualization to understand a package dependency at a glance but with adequate details. It is integrated in two different visualizations (EDSM and CYCLETABLE) to provide information to resolve cycle issues. Second, OZONE is a help for decision. It proposes package dependencies to remove for a better modularity of package architecture (*i.e.,* an architecture without cycles). Third, ORION provides a change impact analysis system. It helps reengineers choose which change is the best for the expected goal by allowing them to compare multiple change impacts.

3.4.1 Principle

ECOO is an approach structured around four ideas grouped to solve the four problems enunciated previously (identify problems, solve these problems, avoid the regression of the system and minimize change costs)

Figure 3.5 shows key points treated by the ECOO approach. ECOO is package-oriented and deals with dependencies between packages. These dependencies are complex and difficult to understand without a specific approach. The analysis of these dependencies is made with two different approaches: (i) analysis of cycles; and (ii) analysis of layers. These two

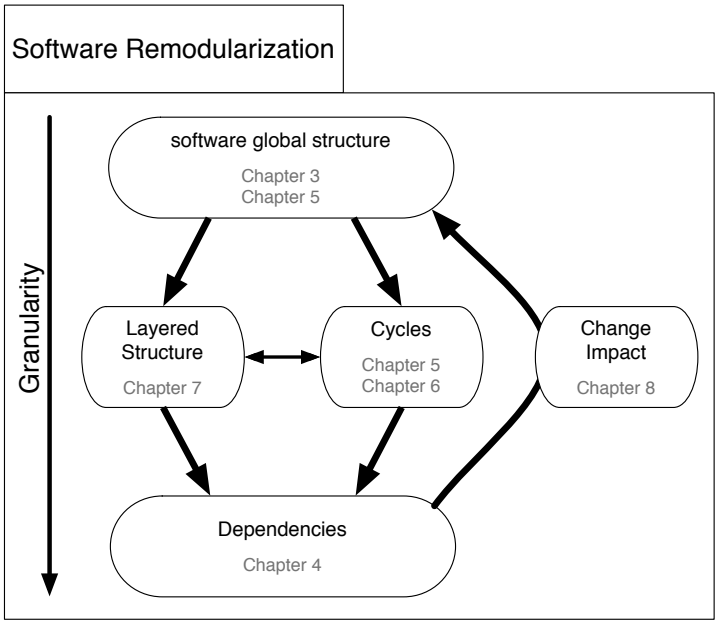


Figure 3.5: Key points treated by ECOO approach.

approaches are strongly correlated because a cycle cannot be arranged in multiple layers. These two analyses allow reengineers to understand the global software structure and take decision about the actions to do. We build a meta-model for change-impact analysis to help reengineers in their decisions.

In the rest of this chapter, we present an overview of the four parts of the approach.

3.4.2 Package Dependency Oriented Approach

In software systems, relationships between packages are complex because they depend on fine-grained relationships between classes. Understanding software architecture is not trivial and understanding package dependency is not easy with standard visualizations. A dependency between two packages is in reality a collection of relationships between classes.

ECELL (for Enriched Cell) is a dependency-oriented visualization. It builds a view of dependency from a package to another. ECELL shows only information needed to understand the dependency. It is based on four kinds of dependencies: inheritance, class reference, invocation, and class extension.

ECELL overview. An ECELL is composed of four parts (Figure 3.6): a header provides the number of all and each kind of dependencies. At the center, there are two rectangles representing relationships between classes from the client package (*i.e.,* the source package) to the provider package (*i.e.,* the target package). Then at the bottom, there is a frame

that is used in EDSM (Chapter 5, p.55) and CYCLETABLE (Chapter 6, p.83) for different meaning: EDSM shows the kind of cycle, CYCLETABLE shows the shared dependencies.

3.4.3 Understanding Architecture

To understand package architecture, we improve Dependency Structural Matrix (DSM) by adding colors and including ECELL in each cell of the matrix.

This visualization allows one to see the structure of a software application in a matrix. Using colors, it allows reengineers to spot package cycle issues. With integration of ECELL it helps reengineers to understand and fix cycles.

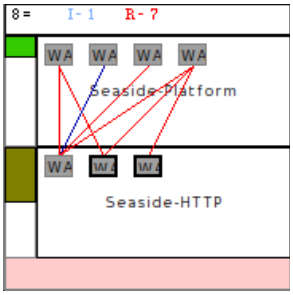


Figure 3.6: ECELL at work.

EDSM overview. It provides information using colors: red/pink represents a dependency in a direct cycle. Yellow represents a dependency in an indirect cycle. A blue square represents a Strongly Connected Component. Grey represents standard dependency (*i.e.,* not in a cycle). Figure 3.7 shows a visualization of EDSM with colors but without ECELL.

3.4.4 Targeting Cycle Problems

EDSM provides information about direct cycles, but not on complex cycles due to the matrix structure. To resolve complex cycles, it is useful to know the implication of a dependency in cycles.

We propose CYCLETABLE, which decomposes each cycle in minimal cycles and highlights dependencies implied in multiple of them. We introduce ECELL to provide fine-grained information for each dependency.

CYCLETABLE overview. A CYCLETABLE is a matrix where a row represents a package and a column represents a minimal cycle. In this matrix, a cell represents a dependency. A colored cell is a shared dependency (*i.e.,* shared by each cycle where there is the same colored cell). Figure 3.8 shows a CYCLETABLE. For example, we can see, in the second row, a lot of blue cells, which represent the same dependency in multiple minimal cycles (*i.e.,* multiple columns). If we remove this particular dependency, all implied cycles disappear.

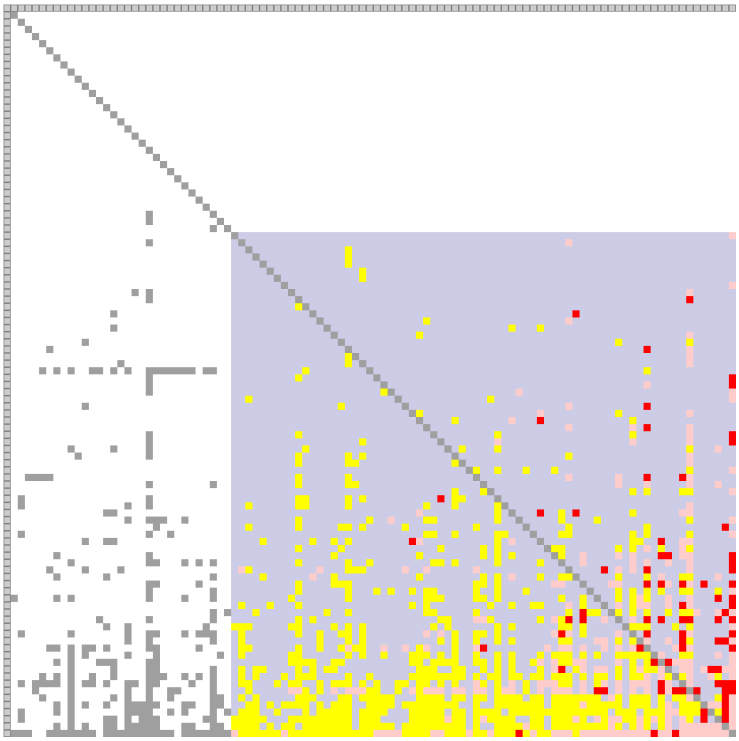


Figure 3.7: EDSM at work.

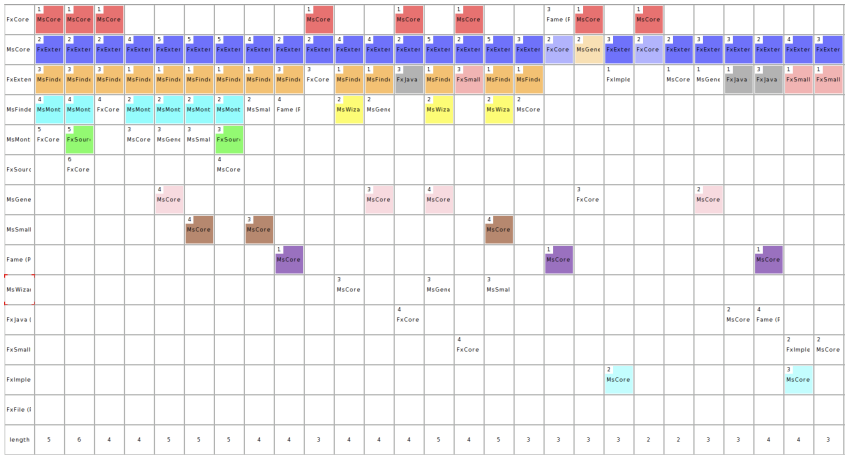


Figure 3.8: CYCLETABLE at work.

3.4.5 Proposing Changes

Based on our visualizing tools and our case studies, we concluded that some cycles have similar characteristics. To help reengineers in their work, we provide a list of dependencies that have these properties. It provides (i) an approach to propose changes at package level; and (ii) an approach to build package layer architecture in presence of cycles. These two points allow a reengineer to understand how packages interact with the others and what is the place of each package in the system.

OZONE overview. OZONE is a semi-automatic approach which computes a list of *unwanted* dependencies and builds a layered architecture using this list to interpret complex cycles. The reengineer can add its own evaluation of dependencies to minimize false-positive results. Figure 3.9 shows OZONE principle. It shows that packA, packB and packC are in a cycle but on multiple layers.

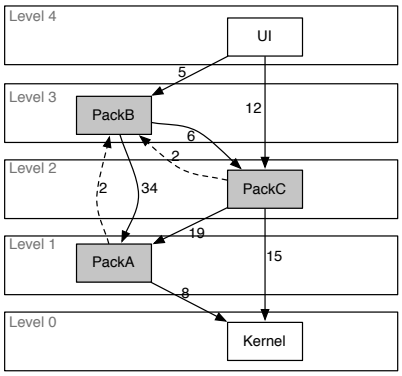


Figure 3.9: OZONE principle. Dotted arrows are highlighted dependencies. Grey shapes represent packages in a cycle.

3.4.6 Analyzing Change Impact

Providing help for reengineering with ECELL, EDSM, CYCLETABLE and OZONE is not enough. In practice, removing a dependency is not trivial and it can have a large impact on the system architecture. For example moving a class from a package to another can remove a dependency, but create multiple other dependencies (explained in detail in Chapter 8, p.129).

We provide ORION, an approach to simulate changes in multiple model versions without impacting the real system. It allows reengineers to try changes without breaking the system and having feedback from reengineering tools (*i.e.,* visualizations, metrics).

ORION overview. We provide a meta-model for change impact analysis applicable to existing models. ORION is an interactive prototyping tool for reengineering to simulate changes and compare their impact on multiple versions of software source code models. It

reuses existing assessment tools, and has the ability to hold multiple and branching versions simultaneously in memory. Its infrastructure optimizes memory usage of multiple versions for large models. Figure 3.10 shows an experimental browser built especially to manipulate multiple versions of a system for our experiments.

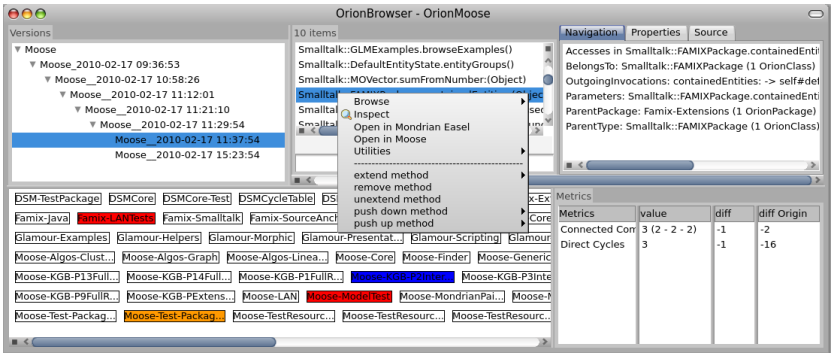


Figure 3.10: ORION at work.

3.5 Implementation

Language Independent Approach. All our implementations have been implemented on top of the Moose reengineering environment [Ducasse 2005a] and are based on FAMIX [Demeyer 2001], a family of meta-models, which are customized for various aspects of code representation (static, dynamic, history). FAMIX-core describes the static structure of software systems, particularly object-oriented software systems[1].

Our tools can be applied to object-oriented languages, when there is a parser for it to build FAMIX model. Currently, we can use our tools on Smalltalk, Java, C-Sharp and C++ source code.

ECELL, EDSM and CYCLETABLE. (Chapter 4, p.39, Chapter 5, p.55, Chapter 6, p.83) These three approaches are implemented on top of Mondrian. It is a visualization engine, which provides a scripting system for visualization. It allows us to build visualization quickly.

OZONE. (Chapter 7, p.103) It uses a graph representation of a package system. We use the graph library available in Moose reengineering environment, which is sufficient for the behavior we need (*i.e.,* node-link representation and add/remove edges). The tool (browser and visualization) is build on top of Moose by using Mondrian and Glamour. Glamour is an engine for scripting browsers.

[1] see http://www.moosetechnology.org/docs/famix

ORION. (Chapter 8, p.127) It is an extension of FAMIX. Extending FAMIX is a major asset of ORION as it allows us to reuse tools and analyses developed on top of FAMIX [Ducasse 2009b].

The browser used for validation was built on top of the Moose, Mondrian and Glamour engines.

3.6 Validation

This approach has a large spectrum, it is therefore better to validate each part of the approach separately. Each part has a specific validation adapted to the goal.

- ECELL. It has been validated at the same time as EDSM in a user survey performed with 9 different projects, that provides a qualitative assessment of the approach.
- EDSM. We provide three validations of EDSM: a case study performed on Moose 4beta4, a controlled experiment performed on Seaside2.8 (a web framework), a user survey performed with 9 different projects.
- CYCLETABLE. It has been validated with a controlled experiment on a graph of Moose package dependencies with 11 developers.
- OZONE. It has been validated with a case study performed on Moose 4beta4.
- ORION. It has been validated with a benchmark study on memory used and speed accessing elements and a case study performed on Moose 4beta4.

These multiple validations provide a complete evaluation of the global approach.

3.7 Summary

In this chapter we introduced ECOO, our approach to help reengineers to resolve modularity problems at package level. The approach is a combination of four axes (understanding architecture, targeting cycle problems, proposing changes, and analyzing impact), which will be explained in following chapters. We also introduced challenges stressed by the approach and explained terminology used in the whole document.

The next chapters explain in detail the approach. We begin with a review of ECELL (Chapter 4, p.39) before integrating it in EDSM (Chapter 5, p.55) and CYCLETABLE (Chapter 6, p.83). These two approaches introduce the axis of proposing changes provided by OZONE (Chapter 7, p.103), which can be used in the ORION approach (Chapter 8, p.127).

CHAPTER 4

ECELL, Understanding package dependencies

Contents

Nothing in life is to be feared, it is only to be understood.

[Marie Curie]

At a Glance

This chapter introduces a visual representation of package dependency. A package dependency is complex because it represents multiple class relationships (inheritance, class reference, class extension, method invocation). The visualization shows information needed to understand the package dependency. A user study validation proves that this visualization can be used at different levels of expertise.

Keywords: Dependency analysis, fine-grained information, package dependency visualization.

4.1 Introduction

In the previous chapter, we introduced ECOO, a global approach to help reengineers to remodularize software systems. This chapter explains that a package dependency is composed of multiple class relationships that make it difficult to understand. We stress this point and provide a visualization of package dependency. We can identify patterns in this visualization that help engineers to analyze dependency problems.

Structure of the Chapter

In the next section we introduce the challenge of the ECELL approach and which kind of information we want to see. In Section 4.3 (p.42) we provide an overview of ECELL that is detailed in Section 4.4 (p.43). Section 4.5 (p.47) shows and explains some visual patterns we already see in experiments. Section 4.6 (p.49) provides a validation, which is a part of validation made in context of EDSM (Chapter 5, p.73). Finally, Section 4.7 (p.53) summarizes the chapter.

4.2 Dependency Information

In a software system, package relationships are complex because they depend on more fine-grained relationships between classes. Seeing a software architecture is not trivial and understanding package dependency is not easy with standard visualizations. For example, Figure 4.1 shows 14 core packages of Moose Reengineering System. A Node-link visualization cannot provide fine-grained information of the dependencies between these nodes (*i.e.,* here packages) because links do not provide enough space to add information.

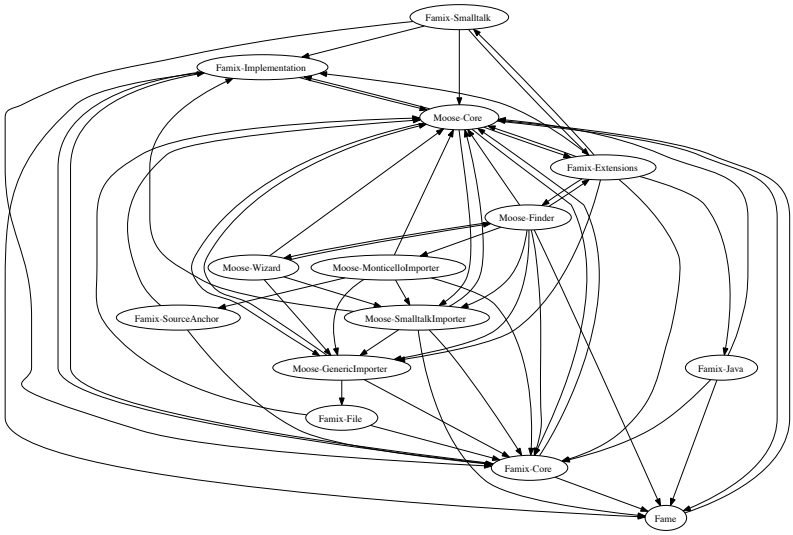


Figure 4.1: Package dependencies in the core of Moose Reengineering System.

A dependency between two packages is actually a collection of relationships between classes. Figure 4.2 illustrates this idea. In Package A and Package B there are multiple classes communicating together.

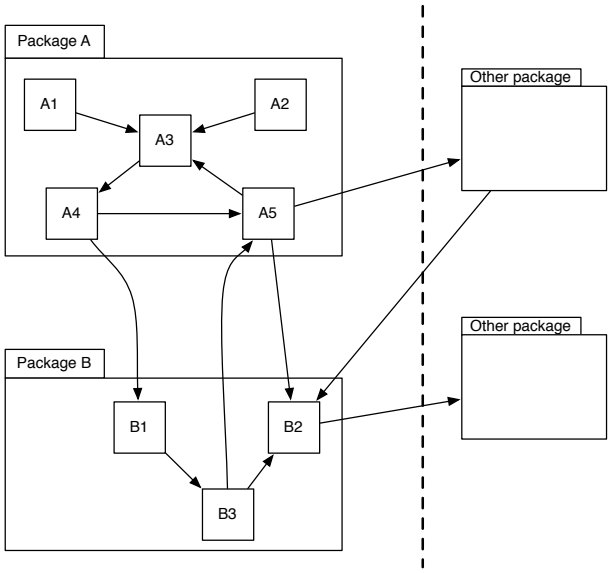


Figure 4.2: An illustration of package dependency.

To provide a good view of package dependency, we should provide fine-grained information. It is important to understand a dependency and to help engineers to remodularize their system.

We point out two pieces of information that should be provided:

- *Evaluating the Causes of Dependencies*: Removing a package dependency often means changing some class relationships involved in the cycle. However, the cost of this action vary with the type of relationship *e.g.,* changing a reference to a class is often easier than changing an inheritance relationship.

  ECELL gives this fine-grained information and it supports a better understanding of the situation.

- *Evaluating the Distribution of Dependencies*: Knowing that a package has multiple dependencies to another one is valuable but insufficient information. For a dependency between packages, valuable information is to know how many classes are involved in the dependency and the distribution of the relationships between these classes. Figure 4.3 illustrates three kinds of relationship distributions. The first dependency involves multiple classes in the two packages. The second dependency shows that one class in the client package is involved and uses all classes in the provider package. The third case shows that multiple classes in the client package use one class in the provider package. These three kinds of relationships do not provide the same information about the package dependency.

This information is important for understanding package dependency and for package remodularization. We provide a solution to see it in ECELL.

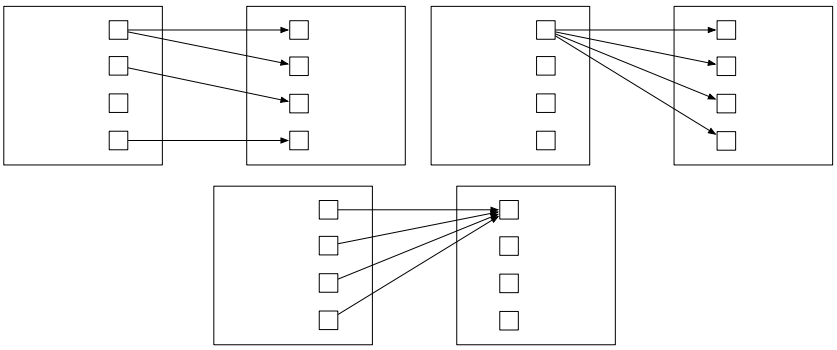


Figure 4.3: Three kinds of relationship distribution between packages.

4.3 Overview of an ECELL

4.3.1 ECELL Goal

ECELL is a package-dependency-oriented visualization. It builds a view of a selected dependency from a package to another. ECELL shows only information needed to understand the dependency. Figure 4.4 illustrates the kind of information provided in ECELL. It shows only classes involved in the dependency and relationships from client package classes to provider package classes. Here only relationships $A4 \rightarrow B1$ and $A5 \rightarrow B2$ are displayed.

The goal of ECELL is to provide a small dashboard of the package dependency with indicators about the situation between the client and the provider packages. It contains contextual information, which 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.

ECELL uses colors to convey information about the context in which dependencies occur. Our goal is to use preattentive visualization[1] as much as possible to help spot important information [Treisman 1985, Healey 1992, Healey 1993, Ware 2000]. An ECELL is composed of parts and shapes with different color schemas.

[1] Researchers in psychology and vision have discovered a number of visual properties that are preattentively processed [Healey 1992]. They are detected immediately by the visual system: viewers do not have to focus their attention on a specific region in an image to determine whether elements with the given property are present or absent. An example of a preattentive task is detecting a filled circle in a group of empty circles. Commonly used preattentive features include hue, curvature, size, intensity, orientation, length, motion, and depth of field. However, combining them can destroy their preattentive ability (in a context of filled squares and empty circles, a filled circle is usually not detected preattentively).

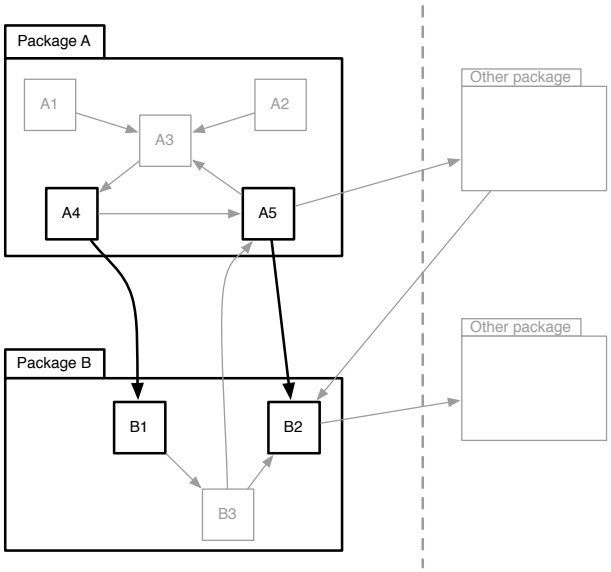


Figure 4.4: ECELL displays only the concerned relationships.

4.3.2 ECELL Construction

An ECELL content displays all dependencies at class level from a client package to a provider package. An ECELL is composed of three parts (see Figure 4.5).

- The top row gives an overview of the strength and nature of dependencies between classes into the two involved packages.
- 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*). The *client package* depends on 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 4.5).
- At the bottom, a colored frame represents the state of the dependency used to provide information in EDSM (Chapter 5, p.55) and CYCLETABLE (Chapter 6, p.83).

4.4 Details of an ECELL

4.4.1 Cycle Color (bottom row)

The bottom row is the first information to see in an ECELL. It represents the more coarse-grained information using color. This frame is used to provide either cycle information in EDSM (Chapter 5, p.55) or shared dependencies information in CYCLETABLE (Chapter 6, p.83). Information about this feature is provided in the respective chapters.

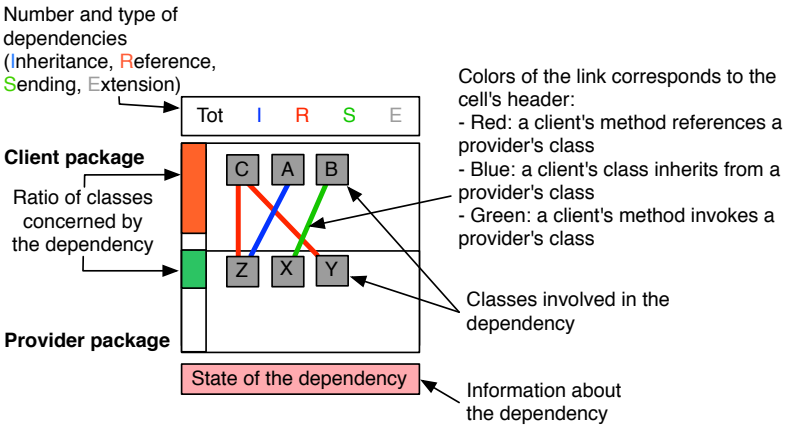


Figure 4.5: ECELL structural information.

4.4.2 Dependency Overview (top row)

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

The top row 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 (S) in green, and class extensions (explained in Chapter 3, p.28) (E) made by the client package to the provider one in gray. A stronger color highlights the strongest dependency type to help reengineers targeting the minimal effort to do. The colors are used to reinforce the comprehension of links between classes (see below). In Figure 4.6 there are 8 directed dependencies from *Seaside-Platform* to *Seaside-HTTP*: 1 inheritance and 7 references (in bright red).

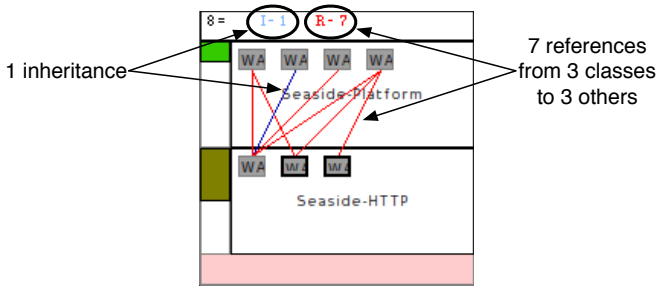


Figure 4.6: An ECELL representing dependency from *Seaside-Platform* to *Seaside-HTTP*

4.4.3 Content of the Dependency (middle boxes)

A dependency is represented by two boxes: one (on top) for the client package, and one (on bottom) for the provider package. Classes in the client package reference classes in the provider package.

4.4.3.1 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.

4.4.3.2 Class Representation

Each square represents a class. A square has particular color and border in EDSM (Chapter 5, p.55). For example, Figure 4.6 shows two classes with black thick border, which represent two classes strongly involved in a direct cycle. As it concerns a specific behavior of EDSM, it is explained in detail in Chapter 5 (p.60).

4.4.3.3 Edge Color

Edges are the smallest details displayed by ECELL. They give information on the nature and spread of dependencies between the classes (Figure 4.5). There are four basic natures, each one mapped to a primary color (synchronized with colors of information in top row of the ECELL): reference in red, inheritance in blue, invocation in green and class extension in gray. When dependencies between two classes are multiples, only one link is displayed. For example, in Figure 4.6, there are 7 references but only 6 red link are displayed. It means that one on the client's classes makes 2 references to the same provider's class. 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.

4.4.3.4 Representation of Class Extension

A class extension (explained in Section 3.3 p.28) represents a method that is in another package than its class. In an ECELL, a class extension is represented by a square with dotted border (Figure 4.7 and Figure 4.8) because it represents methods in another package than the class definition. We differentiate two pieces of information about extensions:

- 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 grey).

In Figure 4.7, The two classes are extended in package *SmallDude-Moose*. Here, six methods from these classes are defined in package *SmallDude-Moose*.

- 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 4.8, the two class extensions in *Famix-Extensions* refer to one class in *Moose-Finder*.

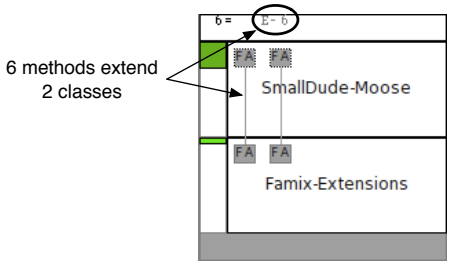


Figure 4.7: The package *SmallDude-Moose* extends two classes from *Famix-Extensions*.

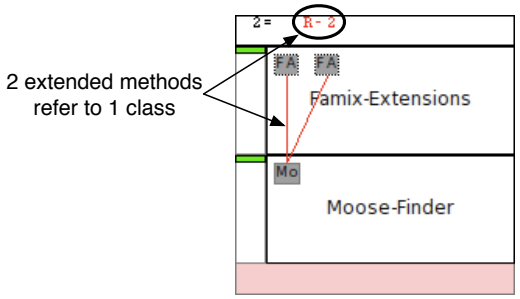


Figure 4.8: Two class extensions in *Famix-Extensions* refer to one class in *Moose-Finder*.

4.4.4 Tooltip

Complementary to the overview and the zooming facility, Tooltips on a class include the name of the class and the name of each concerned method. Figure 4.9 shows the pop-up information of ECELL linking *Seaside-Platform* to *Seaside-Components*: it shows the tooltips from the class *WAImageTest* involving method *renderImageOn:*.

Moreover, a Tooltip is available on edges showing the source code of the class or/and method that create the dependency. Figure 4.9 does not show a tooltip on an edge because in this example we used a model without its source code.

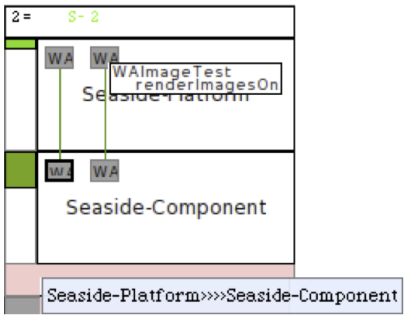


Figure 4.9: ToolTips in an ECELL.

4.5 Visual Patterns

The ECELL visual aspect generates visual patterns. While performing experiments, we have detected some patterns stressing characteristic situations listed in this section. Figures of this section show some specific characteristics of class shapes (specific colors for background and border) coming from EDSM (Chapter 5 p.55). This behavior is not used in this section but is explained in Section 5.3.2 (p.60).

A. **Packages having a large percentage of classes involved in the dependency** (left bar in red). When this pattern shows a high ratio in the referencing package (top), changing it can be complex since many classes should be modified. In the case of a high ratio in referenced (bottom) package, a first interpretation is that this package is highly necessary to the good working of the referencing package (Figure 4.10).

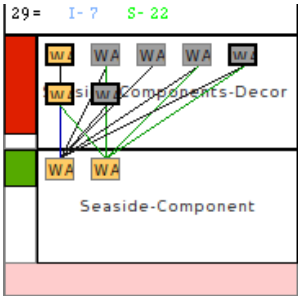


Figure 4.10: ECELL showing visual patterns A, B, D, E.

B. **Packages communicating heavily.** The two packages interact heavily, so intuitively it seems to be difficult to fix this dependency (Figure 4.10). The opposite may be easier to fix.

C. **Packages referencing a large number of external classes** (a lot of red links and header with bright red number). This pattern shows direct references to classes between the two packages (Figure 4.11).

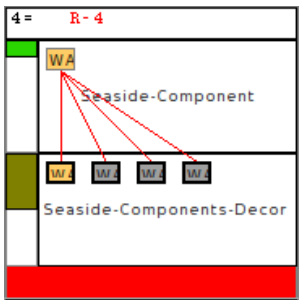


Figure 4.11: ECELL showing visual patterns C, H.

D. **Packages containing classes performing numerous invocations to other classes** (a lot of green links and header with the number in bright green) (Figure 4.10).

E. **Packages containing classes inheriting from other classes.** It means that the referencing package is highly dependent of the referenced package (Figure 4.10).

F. **Packages with a large number of extensions.** It means that the referencing package extends the referenced package (Figure 4.7).

G. **Packages in which a large number of classes refer to one class** (incoming funnel). This patterns shows that the dependency is not dispersed in the referenced package. It can be that the referenced class is either an important class (facade, entry point) or also simply packaged in the wrong package (Figure 4.12).

H. **Packages in which a large number of classes are referred to 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 (Figure 4.11).

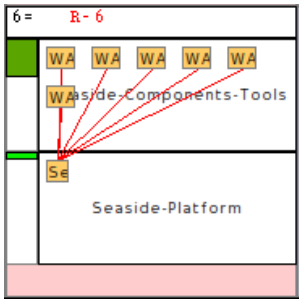


Figure 4.12: ECELL showing visual pattern G.

4.6 Validation

4.6.1 Goal

This case study is a part of the user study performed to validate EDSM (Chapter 5 p.73) as a usable tool for non-expert developers. In this study, two groups of questions concern ECELL. We provide here the analysis of these two questions. This section is a copy of the two concerned questions from Section 5.5.3 (p.73). EDSM is an approach to understand cycles, so in this study we use the term *cycle* as a group of particular dependencies.

4.6.2 Used Tools

For this user study, we wrote a small tutorial about EDSM[2] where a part is dedicated to ECELL and a questionnaire. The developers use EDSM (*i.e.,* ECELL) on their own developed software system and answer questions. They can only use EDSM (*i.e.,* ECELL) and the software source code (*i.e.,* no other tool for software analysis)

4.6.3 Protocol

We provide the users with the EDSM tool, the questionnaire and the tutorial. The questionnaire has 36 questions organized in eight parts. Two parts concern ECELL.

1. Usefulness of ECELL in general: we propose ECELL as a view of a dependency for reengineering. It has two goals: understanding and resolving dependencies. We ask two questions related to these features: is ECELL useful to understand cycles? Is ECELL useful to fix a cycle?

   Possible answers are: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.

2. Use and usefulness of ECELL features: In this part, we want to investigate the usefulness of each ECELL feature in details. For 8 features (Color of ECELL, Header of ECELL, Name of package in background of ECELL, Ratio of concerned classes, Color of classes, Border of classes, Color of edges, Popup information), we ask two questions: Do you use this feature? Do you consider this feature useful?

   Possible answers are: the first question is a *yes-no* question. The second question needs an answer *between 1 (not useful) and 5 (very useful)*.

4.6.4 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 a sole package, EDSM is useless. In the following figures, we use a color for each studied system provided

[2]available on http://www.moosetechnology.org/docs/eDSM

in Figure 4.13. In this section we detail the result of the two parts of the questionnaire concerned by ECELL.

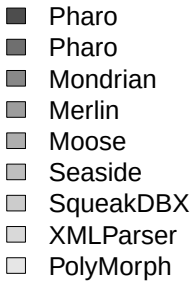


Figure 4.13: Color used for each project.

1. Characteristics of the system (Table 4.1): they are eight software systems evaluated with EDSM. There are two developers working on the Pharo Smalltalk Environment and one developer for each of following system: Mondrian Visualization Engine, Merlin, Moose, Seaside 3.0, SqueakDBX, XMLParser, Polymorph. All these systems are developed in Smalltalk. Table 4.1 shows there are systems with different size. All these software applications have cycles between packages.

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

Table 4.1: Some metrics about studied Software Applications

2. Usefulness of ECELL in general (Figure 4.14): ECELL in the context of large EDSM should be useful to understand a cycle and to fix it. Results show that ECELL helps developers to understand cycles. Fixing a cycle is a bit more difficult, as sometimes developers need to access source code to understand dependencies. It is available in ECELL.

3. Use and usefulness of ECELL features (Table 4.2 and Figure 4.15): ECELL is a complex visualization. It provides a lot of information, some of which is useful for

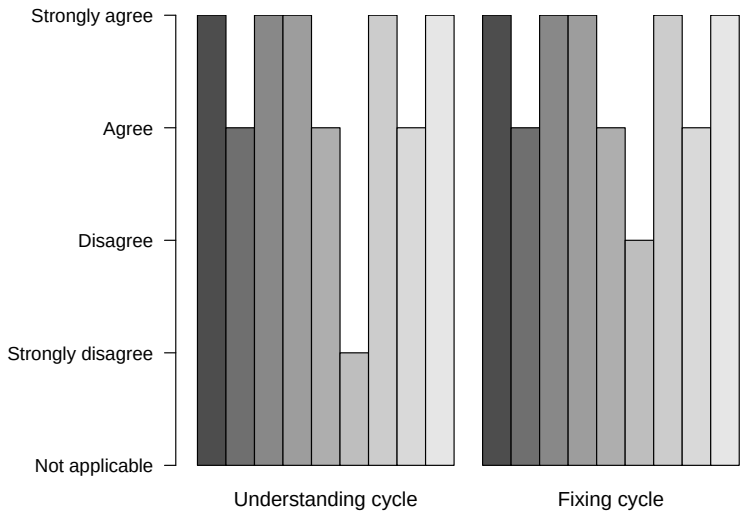


Figure 4.14: Usefulness of ECELL as rated by the 9 participants in our validation experiment.

specific uses. Results confirm that the main features of ECELL are used (the cell color, the header, the name of packages in the cell and the popup information view). Only one user did not use ECELL because of a bug during his experiment time.

Specific features, reserved for special understanding of the packages were less used. Figure 4.15 shows that main features (*i.e.,* Cell Color, Header of ECELL, Name of packages in ECELL, Popup) are useful, whereas specific ones (*i.e.,* Ratio, Class color, Class border, Edge color) were considered less useful.

ECELL 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%

Table 4.2: Use of ECELL feature.

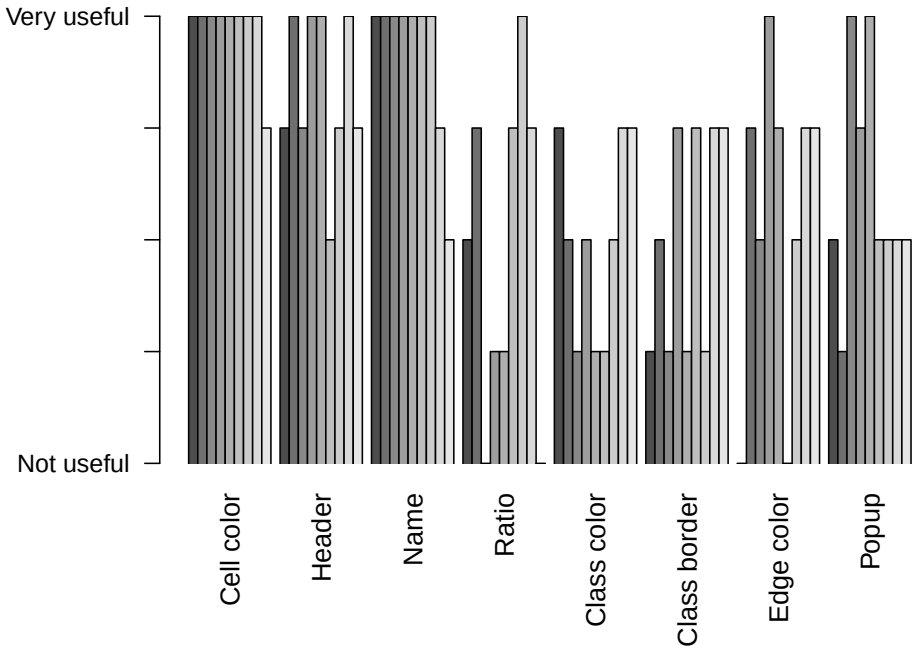


Figure 4.15: Usefulness of ECELL features as evaluated by the 9 participants in our validation experiment.

4.6.5 Conclusion of the User Study

This user study was built to validate EDSM, which explains the notion of cycles in the first question. ECELL is useful to understand dependency at a glance. It allows one to point problems and help resolving them.

This study shows that depending of the user, features are more or less useful. All developers used the coarse grained information provided by ECELL. Less developers used the fine grained information. But each feature was used and considered useful by at least one developer. As a consequence, we think that ECELL should integrate a system to manage granularity to match with the need of developers.

Finally, ECELL is not a replacement for code browsers. The goal is to provide visual information, which is not easy to find with a standard browser. ECELL is complementary to a browser and the future implementation of ECELL will probably be integrated with a source code browser.

4.7 Summary

In this chapter we introduced ECELL. It provides contextual information on package dependencies. ECELL provides information about the context of a dependency by displaying in a cell the complexity of the relationship. We validate this visualization by a user study. Results show that the main features of ECELL are used and useful whereas more specific features are used for particular task and are less used by users.

The next two chapters introduce EDSM and CYCLETABLE. The first one is a visualization of a whole software system and highlight cycles. The second one is a cycle-centric visualization, which highlight dependencies highly implied in cycles. These two visualizations are improved with ECELL placed in cells of the visualization.

CHAPTER 5

EDSM, Understanding package dependencies

Contents

It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change.

[Charles Darwin]

At a Glance

In this chapter we present EDSM, a Dependency Structural Matrix where cells are enriched with ECELL (Chapter 4). We distinguish Strongly Connected Components and stress potentially simple fixes for cycles using coloring information. We validate the approach with different studies: two different case studies on Moose and Seaside software; one user study to validate EDSM as a usable tool for developers.

Keywords: Software visualization, dependency structural matrix, structure analysis.

5.1 Introduction

We show in Chapter 2 (p.9) that understanding the package organization of an application is a challenging and critical task since it reflects the application structure. Many approaches have flourished to provide information on packages and their relationships, by visualizing software artifacts [Wu 2000], metrics, their structure and their evolution. Distribution Map [Ducasse 2006] shows how properties are spread over an application. Lanza et al. [Ducasse 2005b] propose to recover high-level views by visualizing relationships. Package Surface Blueprint [Ducasse 2007] reveals the package internal structure and relationships with other packages – surfaces represent relations between an analyzed package and its provider packages. Dong and Godfrey [Dong 2007a] propose high-level object dependency graphs to represent and understand the system package structure.

Dependency Structure Matrix (DSM) is an approach originally developed for process optimization. It highlights patterns and problematic dependencies among tasks. It has been successfully applied to identify software dependencies [Steward 1981, Sullivan 2001, Lopes 2005, Sullivan 2005, Sangal 2005] and particularly cycles [Sangal 2005]. It provides a dependency-centric view possibly associated with color to perceive more rapidly some values [Heer 2010]. MacCormack *et al* [MacCormack 2006] have applied the DSM to analyze the value of modularity in the architectures of Mozilla and Linux.

Applied to package dependencies, DSM has for header package names and each cell represents a dependency. A non-empty cell at the intersection indicates package in column depends on package in row.

In this chapter, we improve DSM visualization to provide fine-grained information about package dependencies. We propose EDSM, a DSM including ECELL (Chapter 4 p.39). We distinguish independent cycles and differentiate cycles using colors. We applied EDSM on several large systems, the *Moose* reengineering framework, *Seaside 2.8*, a dynamic web framework and *Pharo*[1], an open-source implementation of Smalltalk programming language and environment. Each study has a different context to study the usability of the EDSM visualization.

Structure of the Chapter

The chapter is organized as follows: Section 5.2 (p.56) introduces DSM and its limitations in existing implementations. Section 5.3 (p.58) presents EDSM specifications and Section 5.4 (p.64) presents its usage, from overview of an application to detailed view of inter-package dependencies. Section 5.5 (p.69) reports three different validations and shows the usability of EDSM visualization. Section 5.6 (p.80) discusses about our solution. Section 5.7 (p.82) summarizes the chapter.

5.2 DSM Presentation and Limitations

DSM is an adjacency matrix built from a collection of components. In our case, we consider packages, to understand the structure of an application.

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

The use of DSMs gives pertinent results for the verification of the independence of software components [Sangal 2005], however, in their current form, DSMs must be coupled with other tools to offer fine-grained information and support corrective actions. Figure 5.1 shows a sample dependency graph and its corresponding binary DSM. A binary DSM shows the existence/lack of a dependency (or reference) by a mark or "1/0".

The rule for reading the matrix is: elements in column reference elements in row when there is a mark. In our context, A, B, C, and D are packages. The element in column is also called the client (*i.e.,* the source) and the one in row the provider (*i.e.,* the target). In Figure 5.1, A references B and C, B references A, C references D and D references C.

To optimize the DSM visualization some algorithms are known as *partitioning algorithms* [Warfield 1973] or *clustering algorithms* [Browning 2001]. They optimize the organization of elements in the matrix. We use them and we structure the matrix to show the package at core level on bottom and on right, and the packages on higher level on top and on left. It is not the topic of this chapter, so we do not write anymore about it.

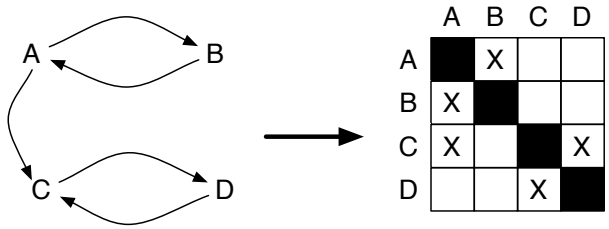


Figure 5.1: A simple DSM.

5.2.1 Dependency Information

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

Y. Cai *et al* [Cai 2007] use DSMs to represent modular structure before and after changes. The authors points three weaknesses of DSM: first, they are not expressive enough to support precise design analysis, second, it only represents design dimensions and third, DSM does not reveal the multiple ways to do a change. Current DSM implementations 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.

For example, current DSM implementations do not provide detailed information about inter-package dependencies. Cycles, which constitute a special target for dependency resolution, are commonly identified using the adjacency matrix power method [Yassine 1999].

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

(a) DSM with marks.

	A	B	C	D
A	■		5	
B		■		3
C	1		■	
D				■

(b) DSM with numbers.

Figure 5.2: Examples of DSM.

5.2.2 Dependency Causes 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 *e.g.,* changing a reference to a class is often easier than changing an inheritance relationship. Dependencies are of different natures (class reference, method invocation, inheritance relationship, and class extension) and a binary matrix (Figure 5.2(a)) or a matrix providing the number of dependencies in each cell (Figure 5.2(b)) do not provide such information.

Annotating a DSM with the types of dependencies can give more fine-grained information and it supports a better understanding of the situation. However, a challenge with this solution is that the matrix should remain readable, providing fine-grained information for understanding cycles, no sacrificing the overall view of architecture.

5.2.3 Dependency Distribution Evaluation

Knowing that a package has 31 dependencies to another one is valuable but insufficient information. The ratio of concerned classes in a package is important since it allows one to quantify the effort to fix a cycle. The intuition is that it is easier to target few classes with some dependencies rather than a lot of classes with few dependencies. This simple heuristic is used on our DSM to help reengineers to fix cycles.

For example in Figure 5.3, 12 classes of package *Components-Tools* reference 2 classes of package *Component*, while only one class of *Component* references one class present in *Components-Tools* (the large gray arrow in Figure 5.3). Consequently, it should be easier to focus the dependencies from *Component* to *Components-Tools* rather than the ones in the opposite direction.

5.3 Micro-macro Reading with EDSM

5.3.1 Macro-reading: Colored DSM

To address the lack of fine-grained information mentioned in Section 5.2.1 (p.57), we enhance DSM with ECELL. We enhance DSM with functionalities that are not present in current DSM implementation such as Lattix [Sangal 2005]. Our solution provides a micro-macro reading by (i) highlighting independent cycles using colors (Section 5.3.1.1,

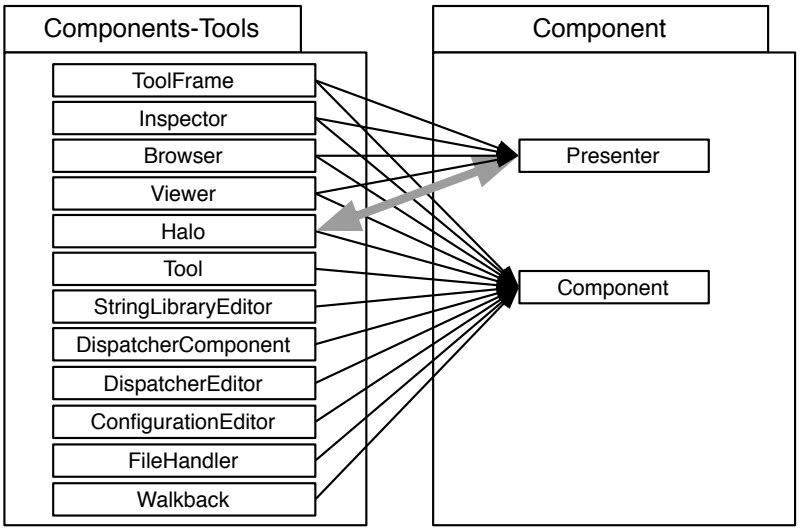


Figure 5.3: A cycle between two packages of Seaside 2.8.

p.59) for a macro reading; and (ii) understanding inter-package dependencies with a micro-reading visualization (Section 5.3.2, p.60).

ECELL resolves DSM limitations (Section 5.2, p.56) by informing reengineers about the dependency cause and the dependency distribution evaluation. An important design feature is the use of color to focus on packages where it seems easier to resolve a package cycle. Therefore we use brighter colors for places having fewer dependencies. The tool is implemented on top of the Moose open-source reengineering environment. Since it is based on the FAMIX meta-model [Demeyer 2001], our EDSM works for mainstream object-oriented programming languages [Ducasse 2005a].

5.3.1.1 Cycle Detection

Our approach enhances the traditional matrix by providing a number of new features: cycle distinctions, direct and indirect cycle identification, and hints for fixing cycles. EDSM distinguishes strongly connected components (SCC) using a Tarjan algorithm [Tarjan 1972]. This method is efficient to detect SCCs.

We use color in DSM cells to identify cycles. A DSM cell (*i.e.,* a dependency) involved in a Strongly Connected Component has a red or yellow color (Figure 5.4). The red color means that the two concerned packages reference each other and thus create a direct cycle. Two packages in a direct cycle have two red cells symmetric against the diagonal. The yellow color means that the dependencies from one package to the other participate in a SCC. The pale blue background color frame all cells involved in a SCC (visible in Figure 5.5). Its area is a visual indication of the number of packages in the cycle. On the contrary, rows and columns with white or gray colors indicate packages not involved in any cycle. The diagonal of the matrix, where a package may reference itself, is colored in gray to highlight

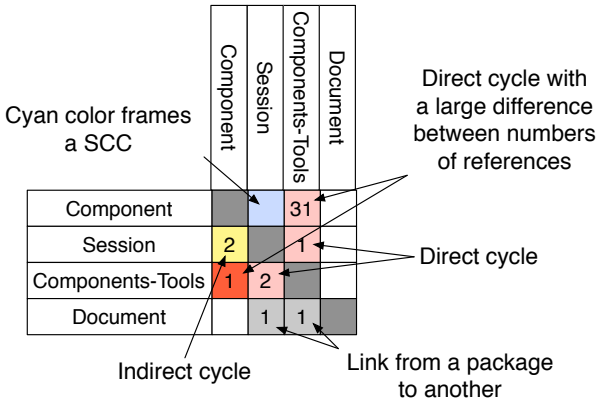


Figure 5.4: Cell color definition.

the symmetry axis but is not used in the current version.

5.3.1.2 Color Hint for Targeting Direct Cycle

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 5.4). The ratio we use is 1 to 3 of the weight of package dependency. This rule only applies to direct cycles as it is easier to compare two packages side by side than an arbitrary number of packages involved in an indirect cycle.

Figure 5.4 illustrates in a simple DSM the rules for cycle colors in cells. It shows two direct cycles with the red color, one between *Session* and *Components-Tools* and one between *Component* and *Components-Tools*. The bright red color in the *Component to Components-Tools* enables one to quickly focus on the dependencies from *Component* to *Components-Tools*, since there is only one dependency against 31 in the opposite direction. Finally, *Session*, *Component* and *Components-Tools* are involved in the same strongly connected component highlighted by the yellow and cyan cells. The yellow cell means particularly that there is a dependency from *Component* to *Session* creating an indirect cycle.

5.3.2 Micro-reading: ECELL Integration

Each cell shows the intersection between a client and a provider. To give a detailed overview of dependencies from a client package to a provider package, we propose to integrate ECELL (Chapter 4, p.39) in each cell. The goal is to create *small multiples* [Tufte 1997] as shown in Figure 5.6.

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” [Tufte 1997]. In EDSM, each ECELL represents a small context, which enables com-

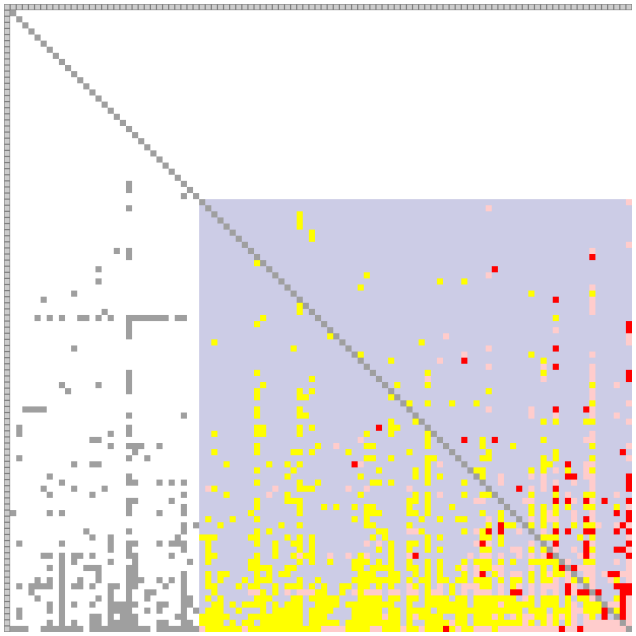


Figure 5.5: a DSM with a blue square representing a SCC.

parison with others. Each ECELL can be used as a small dashboard with indicators about the situation between the client and the provider.

Our goal is to use preattentive visualization as much as possible to help spotting important information [Treisman 1985, Healey 1992, Healey 1993, Ware 2000].

As explained in Chapter 4 (p.39), an ECELL contents displays all dependencies at class level from a client package to a provider package. We adapt ECELL to EDSM with three new features:

- The bottom part of ECELL (*State of the dependency* in Figure 5.7) represents the type of dependency proposed in Section 5.3.1.2 (p.60): pink/red represent direct cycle, yellow represents indirect cycle, gray represents non-cyclic dependencies.

- *Class Color fill: impact of the class in system.* A class may depend on other packages than the two represented by the cell, such as class SeasidePlatformSupport in Figure 5.8. The color fill uses strong orange and light 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 light orange. A class which has some methods involved in other cycles is displayed as strong orange. The classes which do not correspond to this description are in gray. Thus, reengineering strong or light orange class can have impacts on several cycles.

- *Class Border color and thickness: Internal usage.* A gray thin border means that the class has a unidirectional dependency with the other package *i.e.,* it either uses *or* is

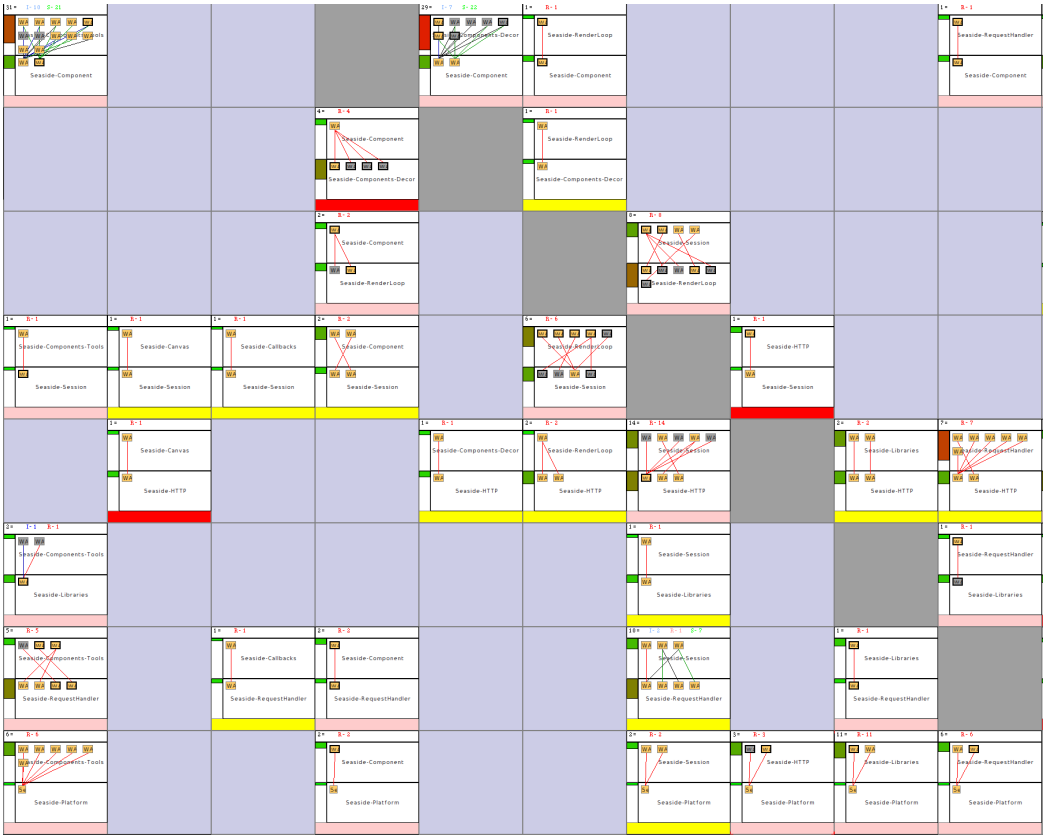


Figure 5.6: A part of DSM presented in Figure 5.5 including ECELL. ECELL in DSM provides a small-multiple effect.

used by classes in the other package. In Figure 5.8, the class SeasidePlatformSupport has a thick border because it is referenced by classes from *Seaside-HTTP*.

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 ECELL (not necessarily the same classes). In Figure 5.8, three classes (WAResponse and WARequest in one cycle and WAComponent in the other one) have a thick border because they reference a class from *Seaside-Platform* and they are referenced by 2 classes from *Seaside-Platform*

ECELL included in EDSM provides enough information to avoid limitations listed in Section 5.2 (p.56). It provides information about type and number of dependencies in the header of the cell, about dependency causes by showing classes involved in the cycle, about dependency distribution by showing ratio of classes and links between classes.

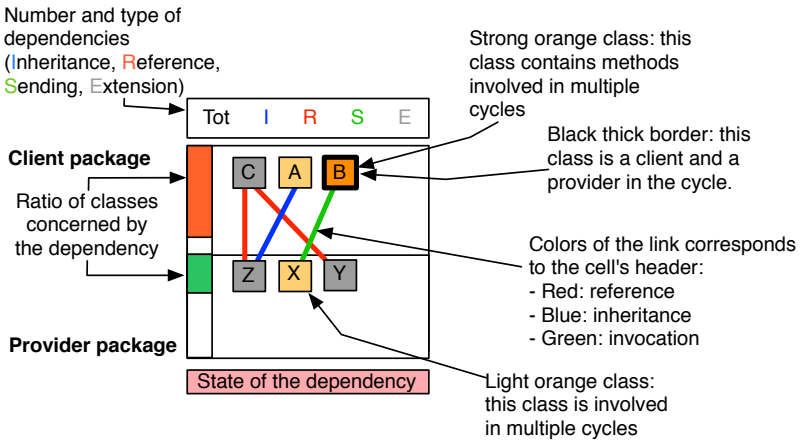


Figure 5.7: Enriched cell structural information.

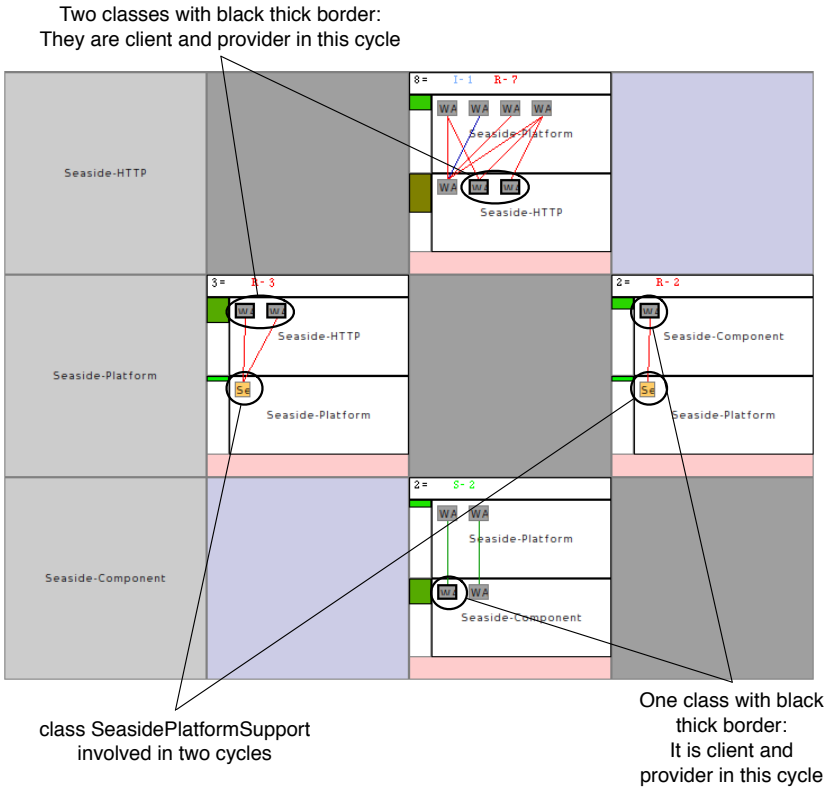


Figure 5.8: Zoom on three packages in cycles.

5.4 Using EDSM

5.4.1 Interaction and Detailed View

While the EDSM offers an overview at the package level as shown by Figure 5.6, extracting all the information from an ECELL is sometimes difficult. There is a clear tension between getting a small multiple effect and detail readability. We offer zoom and fly-by-help to improve usability.

Zooming on Two Packages. Each cell in a DSM represents a single direction of dependency. To get the full picture of interactions between two packages, we compare two cells, one for each direction. Despite DSM intrinsic symmetry, it is not always easy to focus on the two concerned cells. We provide a selective zoom with a detailed view on the two concerned cells, as shown in Figure 5.8. Thus, a new window is opened to focus on a direct cycle that seems interesting from the overview, and analyze the details with the zooming view.

Zooming 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.

Zooming on a 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 a SCC. It selects a SCC and shows concerned packages in a new EDSM.

5.4.2 Fixing a Cycle with EDSM

We now detail an example of cycle resolution through the analysis of EDSM. It begins with understanding EDSM in the large to select a good candidate for remodularization. Then we detail how to understand a couple of direct cycle.

First, when we see the full EDSM, we should watch the red ECELL. They seem to be the better candidates because they are in direct cycle and they have much fewer dependencies than their counterpart. In Figure 5.6, there are 3 red ECELLs which seem to be good candidates for remodularization. They are extracted in Figure 5.9. Among these three dependencies, two have single dependency between classes (*i.e.,* in Figure 5.9, the second and third ECELL).

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

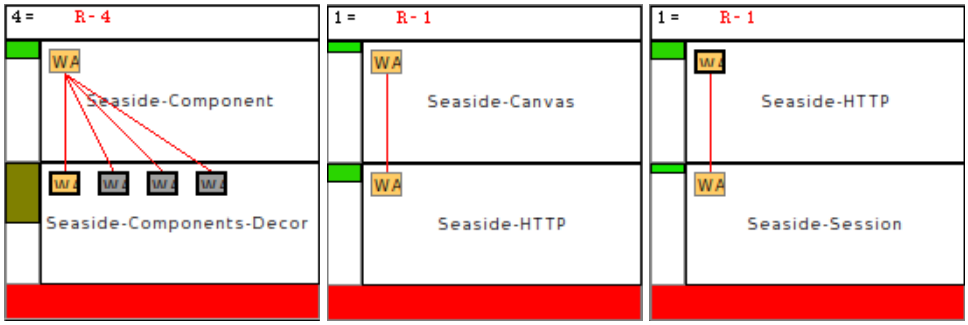


Figure 5.9: Three red cells to be fixed.

The criteria to identify the best candidate for a remodularization are in ECELL. As we show in Section 5.3.2 (p.60), ECELL displays a lot of information which can be read in this order: (i) the state of the dependency, which is represented by a color bar at the bottom of the ECELL; (ii) the number and the type of dependencies, which display the type of remodularization. For example, for extension, it is easy to move the extended method, whereas for inheritance, the reengineering task would be more difficult; and (iii) the content of the ECELL, where the class involvement is displayed. Here, we see in detail the work to do to break the dependency.

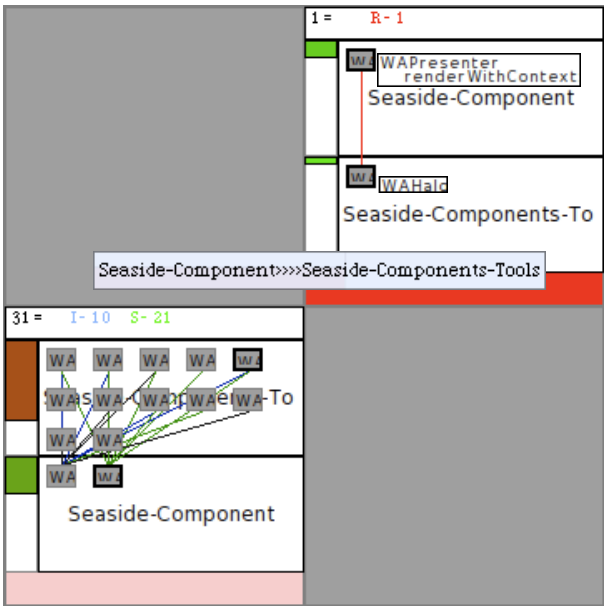


Figure 5.10: A cycle with good candidate dependency to remove.

In Figure 5.10, there is a direct cycle between *Seaside-Component* and *Seaside-Components-Tools* (named *Component* and *Tools* below). We can see that *Tools* have lots of dependen-

cies to *Component* (pink ECELL) while only one class in *Component* uses one class of *Tools* (red ECELL). Moreover, there is one red edge (class reference) in the red ECELL, whereas in the pink ECELL they are multiple inheritances and multiple invocations.

At first glance, it is thus easier to investigate the dependencies of the red ECELL, from *Component* to *Tools*. Let us look at the red ECELL. 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. In Figure 5.10 it is visible that these two bold classes are present in the two ECELL. It means these two classes represent the core of this cycle.

There is only one reference in one 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 class extensions for *WAPresenter* in the package *Tools* and to put the referencing method in it. Then the dependencies would be reversed, effectively breaking the cycle.

5.4.3 Small Multiples at Work

EDSM supports the understanding of the general structure of complex programs using structural element position. Since it is based on the idea of small multiples [Tufte 1997], the ECELL visual aspect generates visual patterns.

We applied EDSM to the Seaside 2.8 framework, the case study is available in Section 5.5.2 (p.70). It is composed of 33 packages and 358 classes. It has a large number of cyclic dependencies. We use this case study to show EDSM in practice (Figure 5.11).

The first use of the EDSM is to get a system overview to scan packages not involved in cycles (not shown in Figure 5.11) and how they interact with other packages. Subsequently, we spot packages involved in direct and indirect cycles. In Figure 5.11 we can spot some visual patterns.

A. **Packages in indirect cycles** (yellow bottom bar). It is not a good starting point to fix them because the cycle probably comes from 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.

C. **Packages with direct cycles with a good candidate dependency to fix** (red bottom bar - low ratio of references). This pattern shows cycle created by a single class in one package. In Figure 5.10, the class labeled *WAPresenter* is the only one appearing in *Seaside-Component* and both uses and is used by classes in *Seaside-Components-Tools* (as indicated by its thick border). Actually, there is a single class in *Seaside-Components-Tools* which 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.

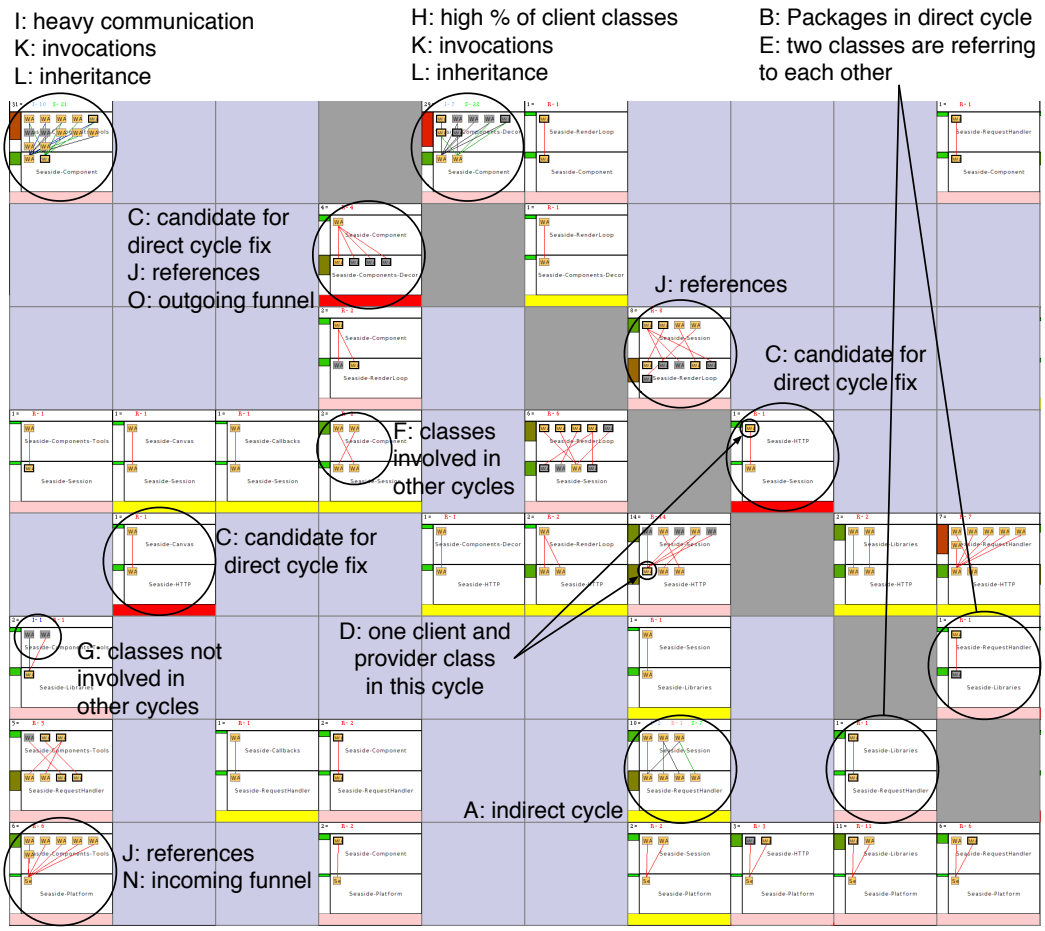


Figure 5.11: An overview of a Seaside subset: ECELL in DSM provides a small-multiple effect.

D. **One class involved as client and provider in a cycle** (black thick border) The cycle is created by a single class. Figure 5.8 shows an example of such situation.

E. **Packages where only two classes are referring to each other** (Thick border). Such pattern represents a direct cycle between two classes. In Figure 5.12, only one class of *Seaside-Component* is in cycle with only one class of *Seaside-RequestHandler*. In addition, they both have a thick border so it is a direct cycle between these two classes. This pattern allows us to focus our attention on just two classes of the two packages.

F. **Classes involved in other cycles.** When a class is involved in other cycles, its background is orange. It means that when we change this class, it could probably impact other cycles. In Figure 5.8, 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 5.8, there are several classes in gray.

The following patterns (H to O) have been already explained in Chapter 4 (p.39). We show here these ECELL patterns in EDSM context.

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 since many classes should be modified. In the case of a high ratio in referenced (bottom) package, a first interpretation is that this package is highly necessary to the good working of the referencing package.

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** (a lot of 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** (a lot of 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 on the referenced package. Looking at the opposite cell is good practice.

M. **Packages with a large number of extensions.** It means that the referencing package extend the referenced package. It represents additional feature for the referenced package (Absent in Figure 5.11).

N. **Packages in which a large number of classes refer to one class** (incoming funnel). This patterns shows that the dependency is not dispersed in the referenced package. It can be that the referenced class is either an important class (facade, entry point) or also simply packaged in the wrong package.

O. **Packages in which a large number of classes are referred to 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 is supported by direct interaction with the mouse. These views can be for example class blueprint or any polymetric views [Lanza 2003b].

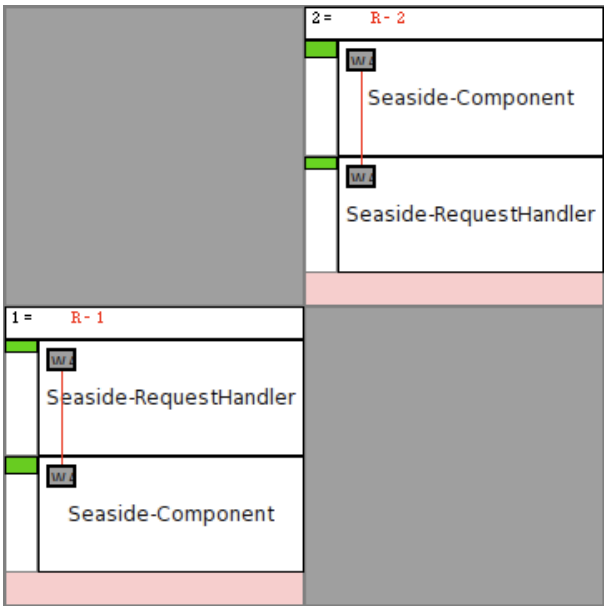


Figure 5.12: A class-to-class cycle.

5.5 Validation

We report on three different studies performed to enhance and validate our approach dealing with package dependencies. Such studies have been performed on small and large software architecture. The first one is on the Moose open-source reengineering environment. This case study confirms that EDSM is useful when a reengineer knows the source code and knows how to use EDSM. The second study is on a model of Seaside 2.8 Framework. It 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 ECELL is a comprehensive tool for developers.

5.5.1 Moose Case Study

5.5.1.1 Goal

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

5.5.1.2 Presentation

In this case study, we used EDSM to analyze cycles between packages in the Moose Reengineering System version 4b4. The system has 108 packages and 1428 classes. There

are 21 packages in cycle creating 6 SCCs. SCCs involve respectively 2, 2, 2, 2, 4 and 9 packages. There are 19 direct cycles.

During this case study, we used mainly EDSM. We also used ORION (Chapter 8, p.127). This approach provides a tool allowing us to make modifications on a model and to analyze impact of changes.

5.5.1.3 Protocol

The goal is to remove from the model all cycles based on our experiment with 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 dependency and propose a solution to eliminate the problematic dependency; and (ii) before implementing the solution, each proposition was sent to, and validated by, the Moose community.

5.5.1.4 Topology

At the end of the study, there were 4 direct cycles left made by test packages. We did not remove them, as they are required for testing some tools like EDSM.

For the rest of the system, we provide 22 propositions to remove cycles to the community (in Table 5.1). All these propositions have been accepted and integrated in the source code.

5.5.1.5 Conclusion of the Case Study

In this case study we wanted to show that EDSM is adapted to understand problems in a known system. The Moose Reengineering Framework has cycles that are not critical. They have been removed with simple actions on the source code.

5.5.2 Seaside

5.5.2.1 Goal

In this user study, the goal is to validate the approach, in particular the information provided by EDSM, by comparing results of our work with the same work done by engineers without EDSM.

5.5.2.2 Presentation

We analyze Seaside 2.8, a web framework in Smalltalk. This system receives the major revision 3.0 because of some refactoring after the version 2.8. A major goal of the Seaside 3.0 revision was to reengineer the application in modular packages.

Seaside 2.8 contains 33 packages, 358 classes, 2 SCCs (one with 3 packages and one with 22 packages), and 25 direct cycles between packages. During this case study, we used mainly EDSM. We also used ORION (Chapter 8, p.127). This approach provides a tool allowing us to make modifications on a model and to analyze impact of changes.

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).

Table 5.1: Propositions provided to the Moose Community

5.5.2.3 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 and ORION to remove all cycles in the Seaside model. We send to Seaside developers our propositions for cycles removal, they analyze the validity of the propositions. They provide four types of answers: (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 validity but the proposition has not been integrated because of better solutions due to our lack of knowledge of the system; (iii) *refused* (false positive), it represents the worst validity: engineers refuse the change because proposed changes break the semantic of the system; and (iv) *refused for lack of control on the package*, this case is particular because some packages we analyzed are not controlled by the Seaside team.

5.5.2.4 Results

We proposed 71 actions to perform. It took us seven hours to remove all cycles in the system. Table 5.2 shows a summary of our propositions by type of actions. We proposed to extend 42 methods, to move 22 classes in other packages, to merge 5 packages and to

create 2 packages to move classes or extend methods in them. Seaside developers accepted 39 propositions (Table 5.3).

Proposition type	Number
extend a method	42
move a class	22
merge 2 package	5
add a package	2

Table 5.2: Summary of proposed actions.

- 17 propositions were already integrated in Seaside 3.0.
- 22 propositions were accepted but not integrated because developers have implemented other solutions. Due to evolution of the system and not only remodularization, they have probably evolved and refactored some packages, with an impact on the architecture.
- 6 propositions have been refused because developers have lack of 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. Engineers refused multiple propositions because these propositions would break the meaning of the expected architecture.

Validity type	Number
accepted and already integrated in Seaside 3.0	17
accepted, not integrated in Seaside 3.0	22
refused	26
refused for lack of control	6

Table 5.3: Validity of propositions.

5.5.2.5 Conclusion of the Case Study

In this case study, we would like to highlight that contrary to the first case study, we performed the study without any knowledge of the architecture of Seaside and without the source-code. In our opinion, this study points to the data-to-information quality of EDSM, both extracting the global picture showing the right details.

In this case, Seaside was already remodularized which offered a good feedback for the study. Results of this case study prove that EDSM helps detecting structural problems and provides enough information to fix them, as a developer can do without EDSM. The Seaside developers have accepted 55% of propositions and 8% were refused for lack of control on the packages. The part of refused proposition (37%) is acceptable considering we did not know the system and we did not have access to the source code.

EDSM 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.

5.5.3 User Study

5.5.3.1 Goal

This case study was partly explained in the user study performed to validate ECELL (Chapter 4 p.49). In this case study, we perform a user study to validate EDSM as a usable tool for non-expert developers. 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.

5.5.3.2 Used Tools

For this user study, we wrote a small tutorial about EDSM (Appendix A, p.163 and 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)

5.5.3.3 Protocol

We provide the users with the EDSM tool, the questionnaire and the tutorial. The questionnaire has 36 questions. We can organize the questionnaire in 8 parts.

1. Participant characteristics: requesting general information about the user experience. There are 4 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 5 simple metrics which are answered by a small script in EDSM. They are: Number of packages of the software, Number of classes, Number of packages in cycle, Number of SCCs, Size of each SCC.

3. Time spent using EDSM: This single question asks the time users used EDSM.

4. Usefulness of simple DSM to understand the system: As DSM are known to help understanding a structure and the adding of colors should help identifying structural problems, we ask two questions: (i) Did the DSM help you to see the structure of your application? (ii) Did the DSM help you to identify some critical dependencies? The simple DSM from our approach is used. It provides colored cells for fast detecting cycles.

   Possible answers are: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.

5. Pattern identification in EDSM:

   This part of the questionnaire tests the usefulness of EDSM in large. We are particularly interested in which small multiple patterns the user paid attention. These patterns correspond to dependencies that could be removed. We ask 5 questions: Do you identify places where you could cut a cycle? Do you identify packages where most of the classes are involved in cycles? Do you identify packages where only one class is creating a cyclic dependency? Do you identify packages which should be merged? Do you identify cycles you want to keep?

   Possible answers are: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.

6. Usefulness of ECELL in general (already explained in Chapter 4, p.39): we propose ECELL as a view of a dependency for reengineering. It has two goals: understanding and resolving. We ask two questions related to these features: is ECELL useful to understand dependencies? Is ECELL useful to fix a dependency?

   Possible answers are: *strongly disagree*, *disagree*, *agree*, *strongly agree*, or *not applicable*.

7. Use and usefulness of ECELL features (already explained in Chapter 4, p.39): In this part, we want to investigate the usefulness of each ECELL feature in details. For 8 features (Color of ECELL, Header of ECELL, Name of package in background of ECELL, Ratio of concerned classes, Color of classes, Border of classes, Color of edges, Popup information), we ask two questions: Did you use this feature? Do you consider this feature useful?

   Possible answers are: the first question is a yes-no question. The second question needs an answer between 1 (not useful) and 5 (very useful).

8. Open questions: We ask some opened question 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, 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? (lack of time, too complex, tool useless) 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?

5.5.3.4 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. In the following figures, we use a color for each studied system provided in Figure 5.13

In this section we detail the result of each part of the questionnaire.

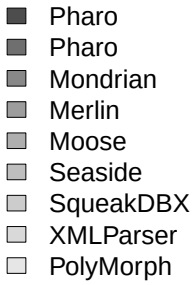


Figure 5.13: Color used for each project.

1. Participant characteristics (Figure 5.14): answers show that answering engineers are experts of these software applications. They are also concerned by the package structure. Their interaction with visualization tools exists but is not excessive.

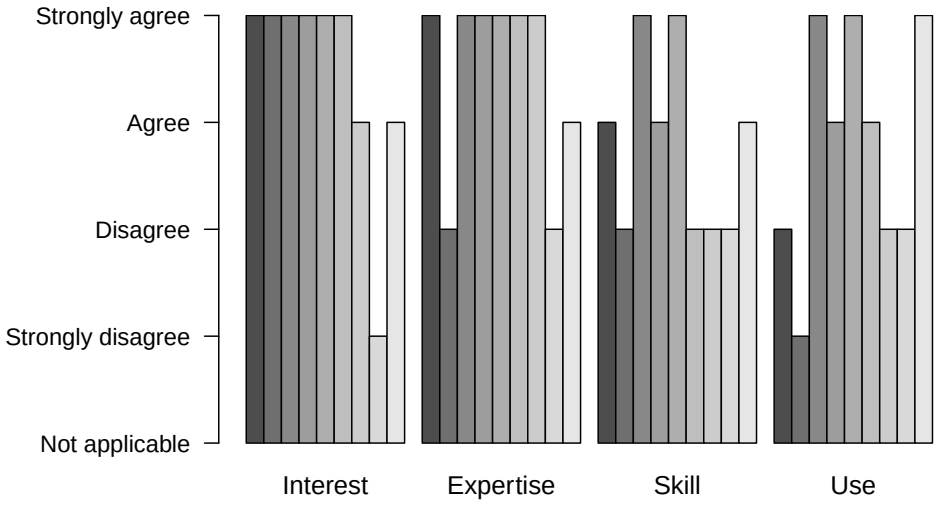


Figure 5.14: State of participants.

2. Characteristics of the system (Table 5.4): they are eight software systems evaluated with EDSM. There are two developers working on the Pharo Smalltalk Environment

and one developer for each of following system: Mondrian Visualization Engine, Merlin, Moose, Seaside 3.0, SqueakDBX, XMLParser, Polymorph. All these systems are developed in Smalltalk. Table 5.4 shows there are systems with different size. All these software applications have cycles between packages.

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

Table 5.4: Some metrics about studied Software Applications

3. Time spent using EDSM (Figure 5.15): the time spent using EDSM is important to know. The longer the participants work on EDSM, the more precise the analysis is. It is a good metric to know the time to understand EDSM and the time to get the goal. We do not differentiate the learning time from the use time.

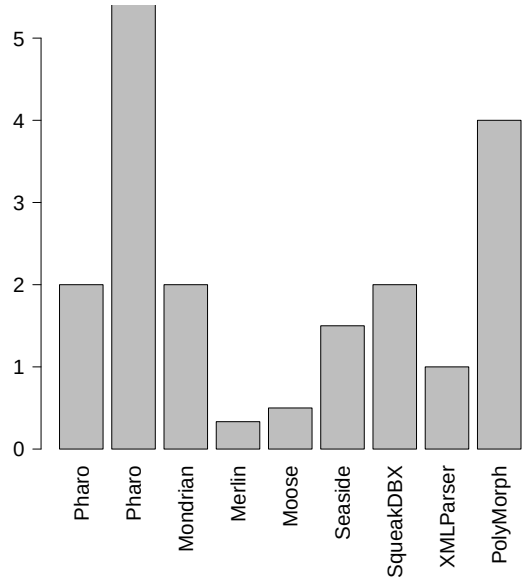


Figure 5.15: Time spent for each project.

4. Usefulness of simple DSM to understand the system (Figure 5.16): Figure 5.11 shows a simple DSM with only background color and weight of dependencies. This view is useful to have a preview of the system. Previous work (as [Sangal 2005])

promotes DSM for retrieving software structure. The participants seems to have difficulties to retrieve the structure (providers are on bottom/right and client are on top/left). They identify cycles between packages and detect critical dependencies.

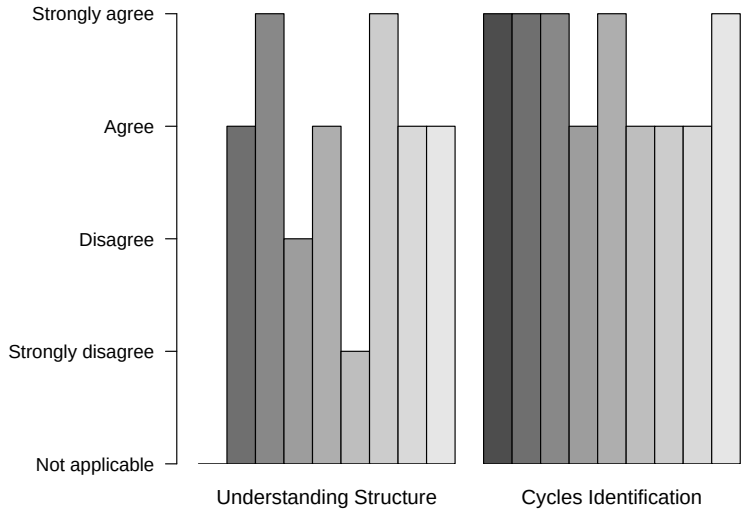


Figure 5.16: Usefulness of DSM.

5. Pattern identification in EDSM (Figure 5.17): EDSM promotes the use of small multiple view. We ask five simple pattern identifications for expert of a system. Answers show that developers can detect these patterns in EDSM. Patterns do not appear in all systems, especially in Smalltalk ones. Hence, some answers are "not applicable".

6. Usefulness of ECELL in general (Figure 5.18) (already explained in Chapter 4, p.39): ECELL in the context of large EDSM should be useful to understand a cycle and to fix it. Results show that ECELL helps developers to understand cycles. Fixing a cycle is a bit more difficult, as sometimes developers need to access source code to understand dependencies. It is available in ECELL.

7. Use and usefulness of ECELL features (Table 5.5 and Figure 5.19) (already explained in Chapter 4, p.39): ECELL 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 ECELL 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. Figure 5.19 shows that main features are useful, where specific ones are considered less useful. Maybe the class border feature is useless.

8. Open questions: Participants use EDSM globally for analyzing these systems and detecting cycles between packages. The main idea is to detect easily cycles, visualize

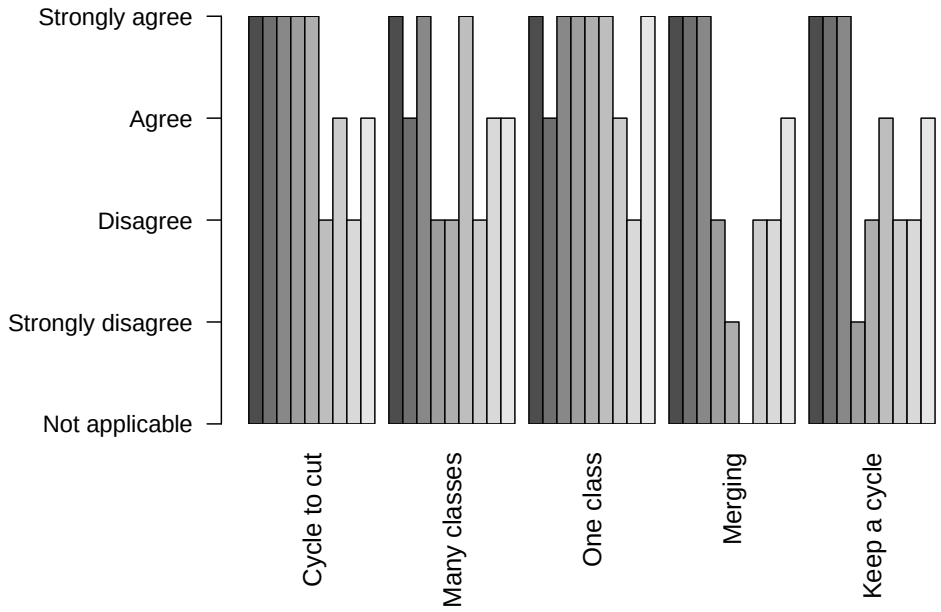


Figure 5.17: Patterns identification in EDSM.

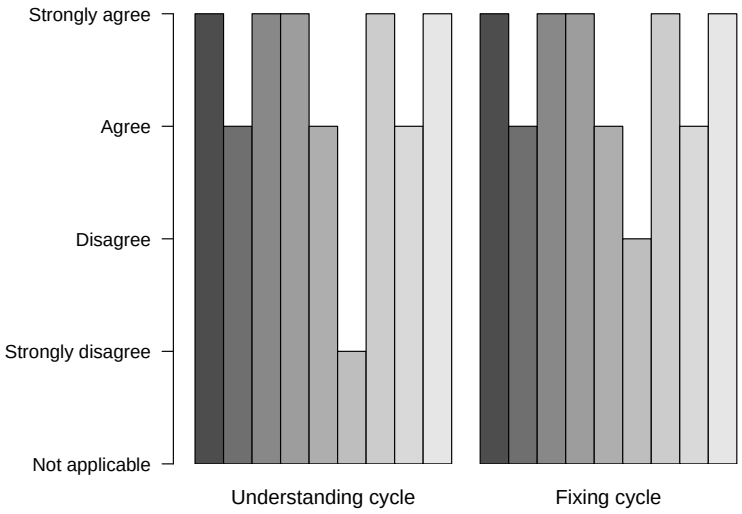


Figure 5.18: Usefulness of ECELL.

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

ECELL 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%

Table 5.5: Use of ECELL features.

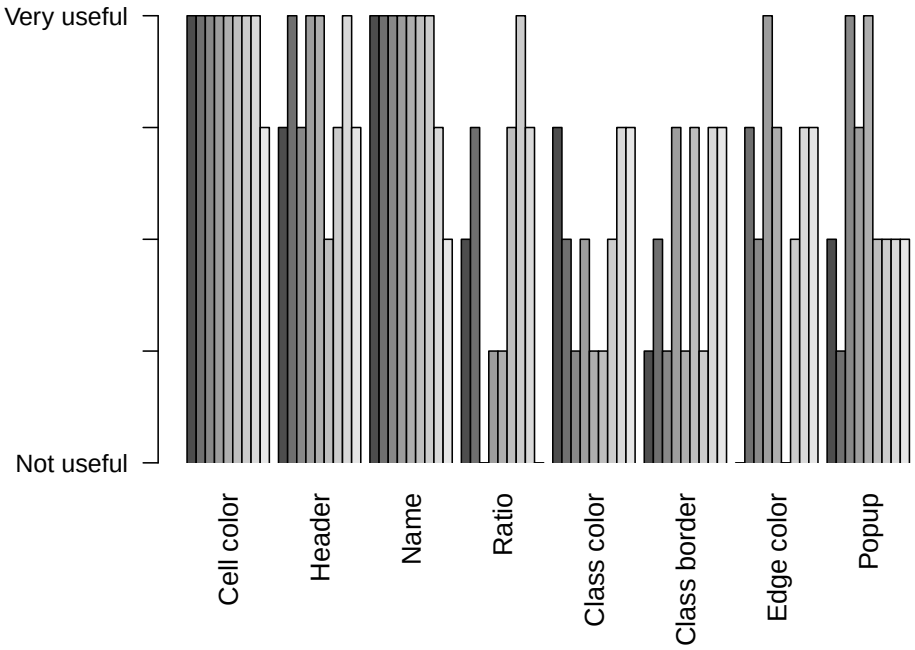


Figure 5.19: Usefulness of ECELL features.

(PolyMorph with 8 packages and XMLParser with 12 packages) declared that the learning time is too big for the studied system.

All but one participant consider EDSM as a useful tool. The one who does not like it prefers graph visualization. This developer uses iterative tools like scripts and simple graph visualization. In fact, EDSM is less useful in this case. Other participants do

not use any other tool and probably let their systems deteriorate, which is shown by this experiement where all studied systems have cycles. This result is important because there is a lack of tools to analyze software application.

About the improvement which could be integrated in EDSM the answer which is repeated is the need of a nomenclature and a better interaction with the source code. Globally, participants like EDSM but the learning time seems to be a problem. There are multiple colors with multiple meaning. To improve the reading of ECELL and EDSM, a nomenclature is a possible solution.

5.5.3.5 Threats to Validity

No Other Tool to Compare to. One of the problems of our user study is that there is no tool for comparison. It is a difficult state because, we cannot compare our approach with others developed at level of classes.

Class Extensions. All studied software applications are developed with the Smalltalk language. It has some particular features such as class extension, which makes it easier to modularize software applications, but which also make easier to create cycles. More experimentation with java should be done.

5.5.3.6 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.

EDSM encompasses the principle of micro-macro reading by showing the details of cycles at class level within the package structure. But ECELL is too complex for users who are not experts, especially due to use of many colors. We already integrated a basic mode with colors reserved for most useful features.

A point to highlight is that all studied software applications have cycles between packages. This problem is important because it could mean two issues: either lack of tools lead too many ADP (Acyclic Dependency Principle proposed by Martin [Martin 2000]) violation (due to iterative) and we should provide a better structure to avoid this kind of problem either the definition of a package is not correct and we should think about what is a package.

5.6 Discussion

5.6.1 Comparison with Other Approaches

Package Blueprint. It takes the focus of a package and shows how such package uses and is used by other packages [Ducasse 2007]. It provides a fine-grained view, however package blueprint lacks (1) the identification of cycles at system level and (2) the detailed focus on classes actually involved in the cycles.

Node-link Representation. Often node-link visualizations are used to show dependencies among software entities. Several tools such as dotty/GraphViz, Walrus or Guess can be used. Using node-link representation is intuitive and has a fast learning curve. One problem with node-link visualization is finding a good layout scaling well on large sets of nodes and edges: such a layout needs to preserve the readability of nodes, the ease of navigation following edges, and to minimize edge crossing. Even then, cycle identification is not trivial. In our opinion it is difficult to find a single layout which works well in all cases.

With DSM the visualization structure is preserved whatever the data size is, which enables the user to dive fast into the representation using the normal process. Cycles remain 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.

Dependence Clusters. Brinkley and Harman proposed two visualizations for assessing program dependencies, both from a qualitative and quantitative point of view [Binkley 2004]. They identify global variables and formal parameters in software source-code. Subsequently, they visualize the effect dependencies. Additionally, the MSG visualization [Binkley 2005] helps finding *dependence clusters* and locating avoidable dependencies. Some aspects of their work are similar to our own. Granularity and the methodology employed differ: they operate on procedure and use slicing analysis, while we focused on coarse-grained entities and use model analysis.

5.6.2 Advantages of EDSM

EDSM brings two benefits for software structure analysis.

The first one is about the matrix structure. A matrix provides a clear structure in comparison to 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.

A second benefit arises from small-multiples and micro-macro reading [Tufte 1997]. It provides contextual information: in a global view, EDSM could be read similarly as the original DSM by looking the header for number of links and the bottom to see cycle context. However, EDSM provides more information about the context of a dependency by displaying in an ECELL 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 like detailed zoom and focusing on SCC or subset.

5.6.3 Limits of EDSM

There are still some limitations that we would like to overcome, with the objective to make EDSM more effective for reengineers. A problem is the limitation of screen space. A DSM

requires a lot of unused space when there are empty cells.

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, a lot of propositions has been refused because of a lack of information about programming strategy, but in the user study we show that EDSM provides a lot of information, which could be not used every time.

5.6.4 Impact and Cost of Small Multiples

One critic about EDSM is that it loses the simplicity of the original DSM. Our experience on real and complex software showed that DSM is powerful at high level but limited for details, which are crucial to understand problems. With 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 criticism 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 (ECELL to work as *small multiples and micro-macro reading i.e.,* that variations of the same structure reveal information).

5.7 Summary

This chapter presented an enhancement of Dependency Structure Matrix (DSM) using *micro-macro reading* thanks to ECELL. First, colors are used to distinguish direct and indirect cycles. Second, ECELL 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, micro-macro reading and preattentive visualization principles. 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 useful tool for detecting, understanding and fixing cycles between packages.

EDSM is integrated in the Moose Reengineering system and we are integrating changes proposed by user study participants (*i.e.,* nomenclature).

The next chapter introduces CYCLETABLE. It is a cycle-centric visualization. Because EDSM does not provide enough information about cycles (*i.e.,* it only shows Strongly Connected Components and direct cycles), CYCLETABLE decomposes Strongly Connected Components and highlight high impact dependencies.

CHAPTER 6

CYCLETABLE, Visualization for Cycles Assessment

Contents

Crash programs fail because they are based on theory that, with nine women pregnant, you can get a baby a month.

[Wernher Von Braun]

At a Glance

In this chapter we present CYCLETABLE, a visualization highlighting dependencies to efficiently remove cycles in the system. It decomposes Strongly Connected Component into minimal cycles to focus on *shared dependencies* between minimal cycles. This visualization is completed with ECELL (Chapter 4) to provide fine-grained view of dependency. We performed (i) a case study, which shows that the shared dependency heuristic highlights dependencies to be removed; and (ii) a comparative study, which shows that CYCLETABLE is useful for the task of breaking cycles in a SCC compared to a normal node-link representation.

Keywords: Software visualization, cycle decomposition, breaking cycle.

6.1 Introduction

In Chapter 5 (p.55), we propose EDSM, which enhances Dependency Structural Matrix (DSM) for a better understanding of cycles at the package level. EDSM shows Strongly Connected Components (SCC) and highlights cycles between two packages in the SCC. However, eDSM is not well adapted when one cycle involves more than two packages (indirect cycles).

We devised a new approach based on the decomposition of one SCC into a set of *minimal cycles* (defined in Section 3.3, p.28). Together the set of minimal cycles cover all dependencies in the SCC and allows the engineer to come up and assess plans to remove all cycles in the system.

In this chapter, we present an approach highlighting dependencies named *shared dependencies* (defined in Section 3.3, p.28) combined with a visualization entirely dedicated to cycle assessment, called CYCLETABLE. CYCLETABLE layout displays all cycles at once and shows how they are intertwined through one or more shared dependencies. CYCLETABLE combines this layout with ECELL (Chapter 4, p.39) to present details of dependencies at class level, which allows the engineer to assess the weight of the dependency.

Structure of the Chapter

Section 6.2 (p.84) introduces the background and the challenges of cycle analysis with the traditional node-link representation of graphs and with DSM. Section 6.3 (p.87) and Section 6.4 (p.89) explain CYCLETABLE layout and usage. Section 6.5 (p.90) presents ECELL in CYCLETABLE and discusses on a sample case the criteria to break cycles as highlighted by the visualization. Section 6.6 (p.92) presents some validations based on a case study and a comparative study. Section 6.7 (p.100) lists related work and Section 6.8 (p.101) concludes the chapter.

6.2 Cycle Understanding Problems

In this section, we present important points related to cycle understanding and what methods exist to fix them. We take an example (Figure 6.1 and Figure 6.2) already presented in Chapter 3 (p.30)

6.2.1 Feedback Arcset

In graph theory, a feedback arcset is a collection of edges we should remove to obtain an acyclic graph. The *minimum* feedback arcset is the minimal collection of edges to remove to break the cycle. This approach could produce good results working on package dependencies because it does not break so much the structure. This method is not usable for two important reasons:

- It is a NP-complete problem (optimized by Kann [Kann 1992] to become APX-hard).

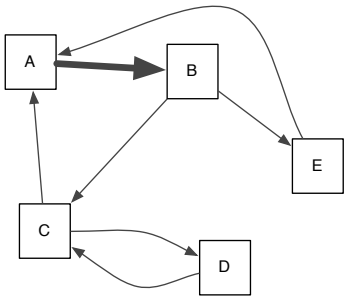


Figure 6.1: Sample SCC.

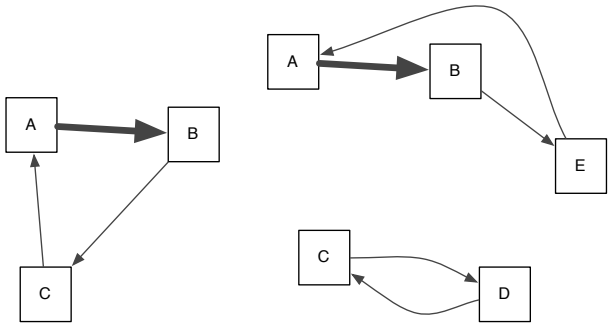


Figure 6.2: Sample SCC (Figure 6.1) decomposed into three minimal cycles.

- It does not take into account the semantic of the software structure. Optimizing a graph is not equivalent to a possible realistic at the software level.

6.2.2 Cycle Visualization

6.2.2.1 Graph Visualization

Figure 6.1 shows a sample graph with five nodes and three minimal cycles. Notice that cycle A-B-C and A-B-E share a common dependency (in bold) from A to B. This shared dependency is interesting to spot since it joins two cycles and by removing it we would break those cycles.

Graph layouts offer an intuitive representation of graphs, and some handle cyclic graph better than others. On large graphs, complex layouts may reduce the clutter but this is often not simple to achieve.

6.2.2.2 DSM and EDSM Visualization

In Chapter 5 (p.55), we show that DSM (Dependency Structural Matrix) provides a good approach to see software dependencies [Steward 1981, Sullivan 2001, Lopes 2005, Sullivan 2005, Sangal 2005] and particularly cycles [Sangal 2005]. It provides a dependency-

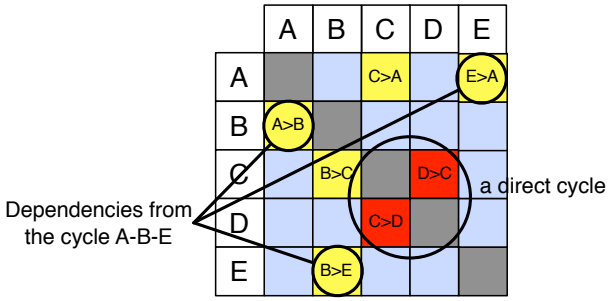


Figure 6.3: DSM corresponding to the graph of Figure 6.1. Each cell represents a dependency. See also Chapter 5, p.55.

centric view possibly associated with color to perceive more rapidly interesting values [Heer 2010]. We use DSM (Dependency Structural Matrix) to see direct cycles: a direct cycle is displayed in red and the two cells representing the two dependencies of the direct cycle are symmetric along the diagonal [Sangal 2005]. Seeing indirect cycles is more difficult, as the visualization is not adapted for it. The main reason for this problem is that it is difficult to read an indirect cycle in the matrix, *i.e.,* to follow the sequence of cells representing the sequence of dependencies in the cycle. The order can appear quite arbitrary as one has to jump between different columns and rows (this problem does not exist with direct cycles as there is only two cells involved, mirroring each other along the diagonal). The cycle A-B-E composed by the three dependencies A>B, B>E and E>A has been circled in Figure 6.3 to show the complexity of reading indirect cycles, intertwined with direct cycles.

In Figure 6.3, the whole matrix displays a pale blue background, indicating that A, B, C, D, and E are in the same SCC. We can see the direct cycle between C and D (in red) and in yellow the other dependencies in the SCC.

We proposed EDSM (Chapter 5, p.55), an improvement of DSM, which shows the relationships between classes in package dependencies. It shows all classes involved in a dependency and which types of dependency exist, providing a good understanding of the dependency and support for breaking the dependency when necessary. While EDSM allows us to analyze direct cycles comfortably, it does not address the problem of indirect cycles left over after removal of direct cycles.

6.2.3 Lack of Solutions

In this section, we present the problem of understanding and breaking cycles and we explain why existing approaches are not up to the task. Solving cycles in legacy systems with several packages and large SCCs is difficult. Feedback Arcset is not necessarily adapted. Node-link representations become unreadable with a large number of packages and dependencies crossing each other. DSM does not provide enough information about indirect cycles. Based on our experience, we propose to focus on shared dependencies in order

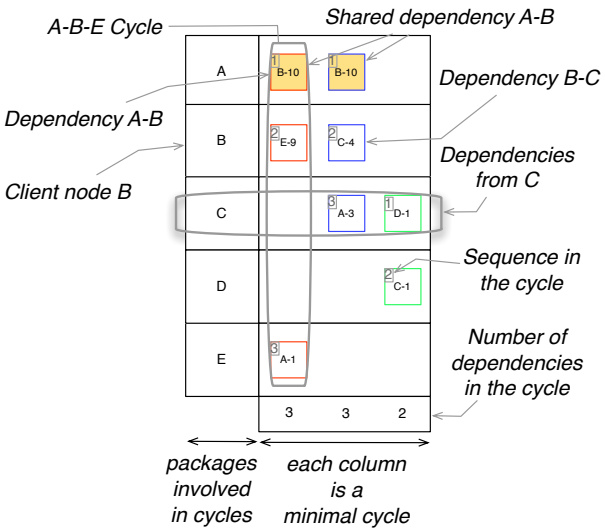


Figure 6.4: CYCLETABLE for Figure 6.1 sample graph.

to efficiently understand and break cycles. The next section shows how this heuristic is embodied and used in the CYCLETABLE visualization.

6.3 CYCLETABLE

During our experiments with EDSM, we have noted that a dependency can be part of multiple cycles. These "shared" dependencies should be highlighted because when we remove them, all involved cycles disappear. Our intuition is that the more shared a dependency is, the more likely it is unwanted in the architecture and should be removed.

We propose a visualization to help reengineers to identify dependencies involved in cycles and to highlight shared dependencies. This visualization shows all minimal cycles ordered by shared dependencies.

6.3.1 CYCLETABLE in a Nutshell

We design CYCLETABLE with the purpose of visualizing intertwined cycles. CYCLETABLE is a rectangular matrix where packages are placed in rows and cycles in columns. CYCLETABLE (i) shows each minimal cycle independently in columns; (ii) highlights shared dependencies between minimal cycles; and (iii) uses ECELL (Chapter 4, p.39) to provide information about dependency internals, enabling *small multiples and micro-macro reading* [Tufte 1997] *i.e.,* variations of the same structure to reveal information. We detail each of these points now.

6.3.2 CYCLETABLE Layout

The CYCLETABLE layout is presented in Figure 6.4. This figure shows a sample CYCLETABLE layout for the graph in Figure 6.1 and Figure 6.2. The first column contains the name of packages involved in cycles, then each column represents one minimal cycle. A row represents all dependencies involved in cycles coming from the package. A non-empty cell, at the intersection of one row and one column, indicates that the package of this row has a dependency involved in the column cycle.

6.3.2.1 One Cycle per Column

Except for the first, each column represents a minimal cycle. In Figure 6.4, the first column involves packages A, B and E in a cycle (first cell represents the dependency from A to B, the second cell from B to E and the last one from E to A).

6.3.2.2 One Package per Row

Each row contains dependencies (represented as boxes) from a package. In Figure 6.4, the first row represents package A, with a dependency to B involved in two different minimal cycles. The second row represents package B, which depends on E and C.

6.3.2.3 Shared Dependencies

Cells with the same background color represent the same dependency, shared by multiple cycles. In Figure 6.4, first row contains two boxes with a yellow background color. It represents the same dependency from A to B, involved in the two distinct cycles A-B-E and A-B-C. It is valuable information for reengineering cycles. Indeed, removing or reversing A-B can solve two cycles.

6.3.2.4 Size of the Cycle

The last line of CYCLETABLE displays the size of each cycle (*e.g.,* each column). This information is valuable when there are multiple cycles. A first approach could be to fix the smaller cycles because there are fewer dependencies to understand.

6.3.3 Cycle Sequence

Cycle sequence represents a relative order between dependencies. This number is sometimes necessary to retrieve the exact order of dependencies in a cycle. Let's take the example of cycle A-B-E. In Figure 6.4, the first relative dependency is A>B (there is the number 1 in top-left corner). The second dependency of the cycle is B>E (number 2 in top-left corner). The third and last dependency of the cycle is E>A (number 3 in top-left corner). In this particular case, it is not useful as cycle sequence follows the top-down order.

To understand the usefulness of this information, Figure 6.5 provides a real example. The 13^{th} column displays a cycle that cannot be read from top to bottom. As the dependencies are not in the right order, it is useful to have the sequence of the cycle. The cycle should

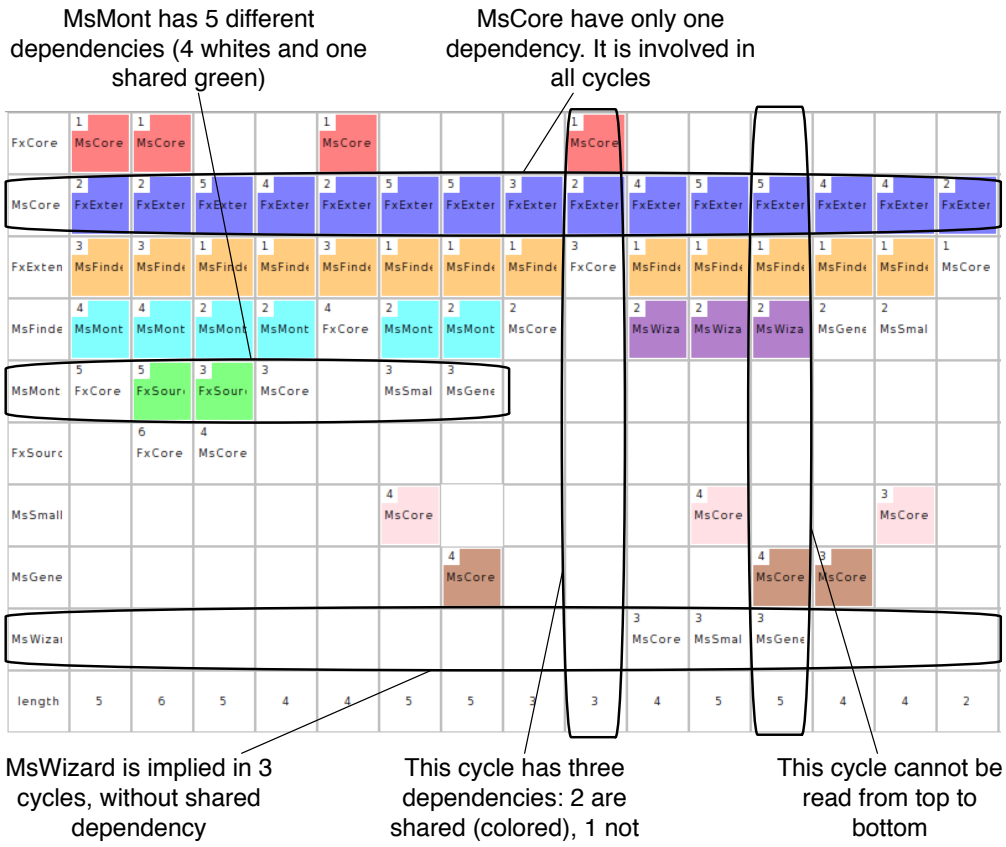


Figure 6.5: A subset of Moose in CycleTable with simple cells.

be read from number 1 to number 5, 1: FxExtension>MsFinder, 2: MsFinder>MsWizard, 3: MsWizard>MsGene, 4:MsGene>MsCore and 5: MsCore>FxExtension.

6.4 Reading a CycleTable

Figure 6.5 shows a sample CycleTable with 9 packages involved in 15 minimal cycles.

6.4.1 Reading a Package

There are three visualization patterns for a package.

- There is one color in the row: the package has one shared dependency to another package but it is involved in multiple cycles. For example in Figure 6.5, the package MsCore (row 2) has one shared dependency to FxExtension, this dependency is shared by all cycles displayed in this CycleTable.

- There are white cells in the row: a white dependency is not a shared dependency. The package is involved in multiple cycles with many dependencies. In Figure 6.5, the package MsWizard (last row) has three different dependencies involved in three different cycles. There is no shared dependency.

- When there are multiple colors in the row, there are multiple cycles with multiple shared dependencies. It is the common visualization pattern. The goal is to look at the most present color. For example in Figure 6.5, the line MsMont has six cells: four cells are white and two cells are green. The two green cells are the same dependency, the white cells represents four different dependencies. MsMont has five different dependencies, involved in six cycles.

6.4.2 Reading a Cycle

A column represents a cycle. Cycle length is displayed at the bottom of the table. A cycle with colored cells has shared dependencies with other cycles. For example in Figure 6.5, the 9^{th} cycle between FxCore, MsCore and FxExtensions has two shared dependencies (red and blue) and one non-shared dependency.

6.4.3 Reading Colors

The more cells share the same color, the more the same dependency is involved in multiple cycles. Then this dependency is interesting for cycle removing. We do not say that this dependency must be removed, but when we remove a shared dependency, all cycles involving this dependency are removed. For example, in Figure 6.5, if the blue dependency from MsCore to FxExtension could be removed, all presented cycles would be removed.

6.5 CYCLETABLE with ECELL

6.5.1 Integrating ECELL in CYCLETABLE

Showing the details at class level of a dependency from one package to another is also important to understand the dependency and assess its weight. We use ECELL (Chapter 4, p.39) adapted to CYCLETABLE (Figure 6.6). Figure 6.7 shows how ECELL is included in CYCLETABLE and provides a closed context to understand each dependency separately. This section shows the specific behavior for CYCLETABLE.

An ECELL contents displays all dependencies at class level from a client package to a provider package. Only the bottom part of ECELL (*State of the dependency* in Figure 6.6) has a particular meaning. Its color represents the identification of shared dependencies and the number is the position in the cycle (see Section 6.4, p.89).

6.5.2 Breaking Cycles with CYCLETABLE and ECELL

Figure 6.7 shows a CYCLETABLE with four packages of Moose core: FxCore, MsCore, FxExtensions and MsFinder. Six minimal cycles are immediately visible. It also appears

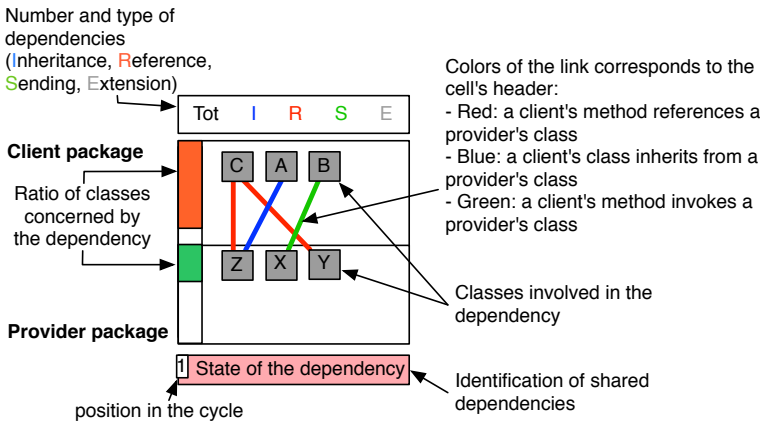


Figure 6.6: ECELL structural information. See also Chapter 4, p.39.

that three dependencies are each involved in multiple cycles (with red, blue, and orange markers at the bottom of the ECELLs).

An important asset of CYCLETABLE is that it does not focus on a single solution to break cycles. It rather highlights different options as there are many ways to resolve such cycles. Only the reengineer can select what he thinks is the best solution. We now discuss how CYCLETABLE allows one to consider solutions for solving cycles in Figure 6.7.

The first point to consider in CYCLETABLE is the notion of shared dependencies, the number of cycles that are involved, and their weight. For example, the red ECELL linking FxCore to MsCore (first row) is in two indirect cycles and one direct cycle. It has a weight of two dependencies and involves four classes (two in each package) as well. But one dependency is an inheritance which can require some work to remove. Finally, from a semantic point of view, MsCore is at the root of many functions in the system so it seems normal to have such dependencies from FxCore.

Instead, we turn our focus to the blue ECELLs, linking MsCore to FxExtensions. It has a weight of five dependencies and involves two classes. From a semantic point of view, FxExtensions represents extended functionalities of the system so it seems that the dependency from MsCore is misplaced: it is just a single method referencing a class in FxExtensions. Moving the method to package FxExtensions is enough in this case to remove the dependency. This single action breaks four cycles.

Two direct cycles remain: (FxExtensions - MsFinder) named A in Figure 6.7 and (FxCore - MsCore) named B in Figure 6.7. The cycle A has a dependency shared with previously fixed cycles (yellow dependency) and is small (two internal dependencies). But the other dependency is also made of two internal dependencies. The situation is balanced. In this case the reengineer has to rely first on his knowledge of the system architecture to detect the improper dependency (FxExtensions >MsFinder). CYCLETABLE is still useful to explore the involved classes and methods.

We assessed before that the dependency from FxCore to MsCore is acceptable. Hence, the dependency from MsCore to FxCore should be removed to resolve the cycle labeled B

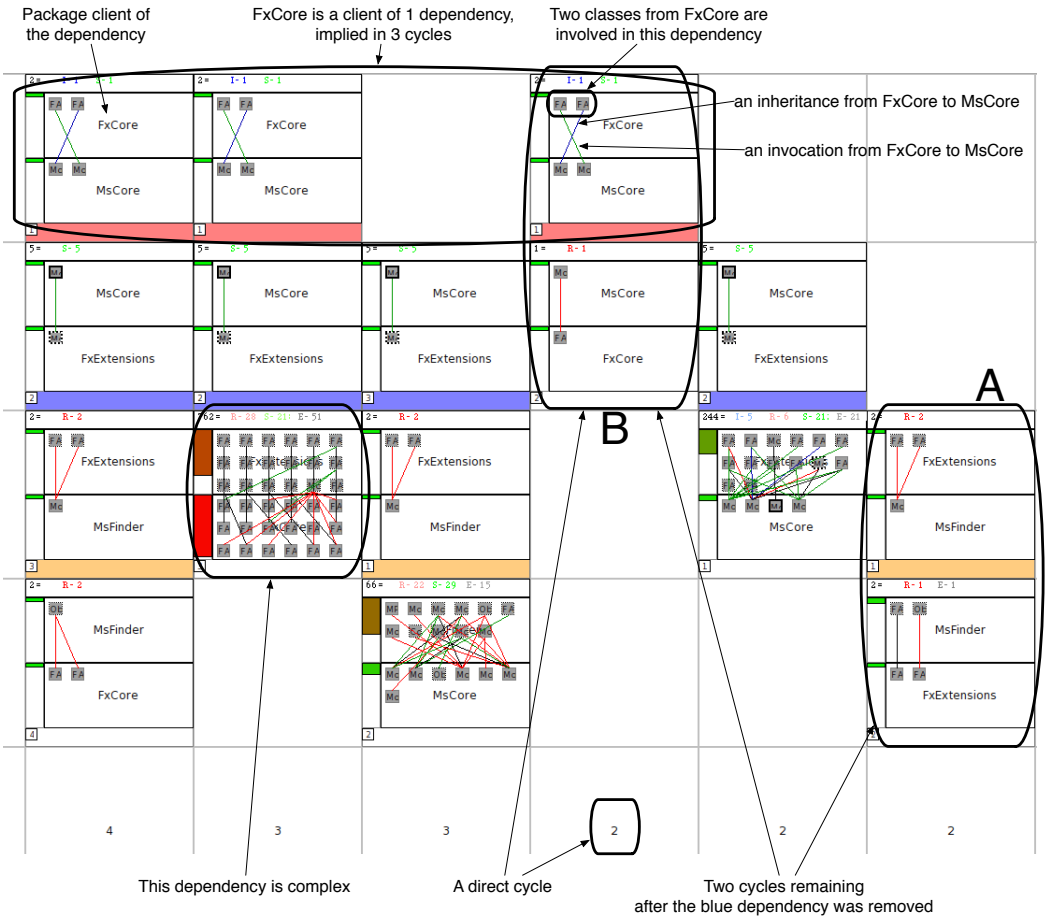


Figure 6.7: CYCLETABLE with ECELL. There are 4 packages involved in cycles: FxCore, MsCore, FxExtensions and MsFinder.

(Figure 6.7). As for the first case, a single method making a reference was misplaced in package MsCore and should become a class extension.

6.6 Validation

We performed two studies to validate our approach. First, we show on a case study that unexpected dependencies in the architecture, which should be removed, often reveal themselves as shared dependencies and are given the primary focus in CYCLETABLE. Second, we perform a comparative study of CYCLETABLE with a normal node-link visualization to validate the efficiency of CYCLETABLE when understanding and fixing large sets of intertwined cycles.

6.6.1 Case Study on Unexpected Dependencies

6.6.1.1 Presentation and Hypothesis

The case study was realized on the core of *Moose version 4beta4* (33 packages). The rest of Moose is not included in this case study because it does not have cycles. A developer from the Moose team evaluated all package dependencies of the system (106 dependencies), regardless of their involvement in cycles. The goal was to retrieve an objective evaluation of each dependency. The possible values that the developer can give are: the dependency is expected in the system architecture, purpose of the dependency requires deep investigation, and the dependency is unexpected and should probably be removed.

After this step, we match all shared dependencies from CYCLETABLE against the evaluation given by the developer. We assessed two hypotheses: the probability that unexpected dependencies are often shared, and prominence of unexpected dependencies in CYCLETABLE, given by their positions in the matrix.

6.6.1.2 Results—shared dependencies as primary targets for removal investigation

Table 6.1 summarizes the results of the case study. The first three lines show some characteristics of the system: there are 14 packages involved in 42 minimal cycles, themselves including 17 different shared dependencies. Then, the assessment performed by the Moose developer returned 11 unexpected dependencies, which should be removed. Finally, we perform the intersection between unexpected and shared dependencies: 9 out of the 11 unexpected dependencies are also shared by various cycles. The two other unexpected but not shared dependencies are actually independent direct cycles *i.e.,* they are direct cycles forming each one SCC, with no intertwined cycles. These two dependencies are not critical in the system architecture.

The 11 unexpected dependencies retrieved by the developer cover the 42 minimal cycles: in other words, fixing those 11 dependencies would break all cycles. It is remarkable that fixing the 9 shared dependencies actually break 40 out of 42 minimal cycles (the two remaining cycles being the independent direct cycles). This case study shows that i) unexpected dependencies are often shared dependencies, and that ii) removing shared dependencies can break multiple cycles with minimal effort, as only a handful of dependencies need to be assessed.

6.6.1.3 Results—prominence of unexpected dependencies in CYCLETABLE

CYCLETABLE uses a heuristic to order packages and cycles in the matrix. This heuristic tries to place cycles sharing common dependencies next to each other. In this study, we show that the ordering given by the heuristic effectively also puts forward unexpected dependencies, given that they are often shared. Starting with the set of unexpected dependencies retrieved by the developer, we looked up the position of the client package in CYCLETABLE. This position corresponds to the row where the dependency is displayed.

Table 6.2 shows that 9 out of 11 unexpected dependencies (80%) appear in the first three lines (3 out of 15 packages, 20%). Thus issues with cyclic dependencies relate mostly to

Characteristics	Moose
number of packages	33
number of packages in cycles	14
number of dependencies	106
number of minimal cycles	42
number of shared dependencies	17
number of unexpected dependencies	11
unexpected ∩ shared	9
cycles coverage by unexpected ∩ shared	40 / 42

Table 6.1: Results of Shared Dependency Validation.

three packages. This result shows that just by focusing on the first part of the visualization, a great deal of work can be done in breaking cycles.

Unexpected dependency	Position in CYCLETABLE (line number)
Famix-Core » Famix-Implementation	**1**
Moose-Core » Famix-Core	**2**
Moose-Core » Moose-SmalltalkImporter	**2**
Moose-Core » Famix-Extensions	**2**
Moose-Core » Moose-GenericImporter	**2**
Moose-Core » Famix-Implementation	**2**
Famix-Extension » Famix-Smalltalk	**3**
Famix-Extensions » Moose-Finder	**3**
Famix-Extension » Famix-Java	**3**
Fame » Moose-Core	9
Moose-Wizard » Moose-Finder	10

Table 6.2: Results of Unexpected Dependency Position.

6.6.2 Comparative Study with Node-link Representation

6.6.2.1 Presentation and Hypothesis

In this comparative study, we validate CYCLETABLE as a useful visualization to understand and break a large set of cycles intertwined together. The precise goal of the study was to validate the effectiveness of CYCLETABLE layout in matrix, compared to a common node-link layout. We measure the time taken by participant to reason about cycles and the quality of their answer.

The setup is the following: first the participant is given a tutorial about the task and the tool with a small example, questions and correct answers to train himself. Second he performs the same questions on the real case study. For the case study, we use a subset of Moose (the 14 packages in cycles, see Table 6.1). Since we focus on assessing the tool, we replace all package names by arbitrary letters from A to O and we do not use

ECELL (Figure 6.9). Hence, participants could not use prior Moose background (some had already worked as developers in Moose) or package names to guide their intuition. The assessment of multiple intertwined cycles is impractical when one uses a single node-link representation showing the full SCC (as shown in Figure 6.8).

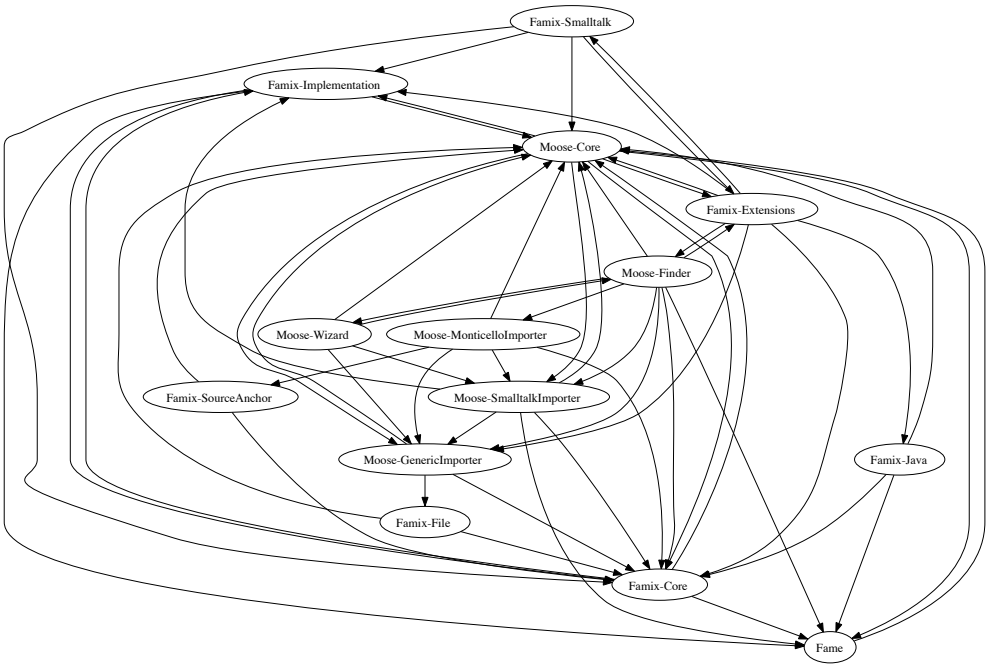


Figure 6.8: Node-Link representation of the Moose 14 packages in cycle.

Instead, we choose to display a series of node-link representations, one for each minimal cycle. This allows us to map the same data in CYCLETABLE and node-link. In particular, a shared dependency was also displayed with the same color across multiple node-links.

One group of seven participants answers questions using CYCLETABLE. The other group of six participants answers questions using the node-link visualization.

6.6.2.2 List of Questions

Here are the eight questions that the users have to answer. We also give the rating of the answer, based on the distance of the answer to the correct one. A 0 rating indicates a good answer.

Q1: Give 2 packages that are in a direct cycle (cycle between two packages). *Rating:* 0 when the answer represents a direct cycle, 1 else.

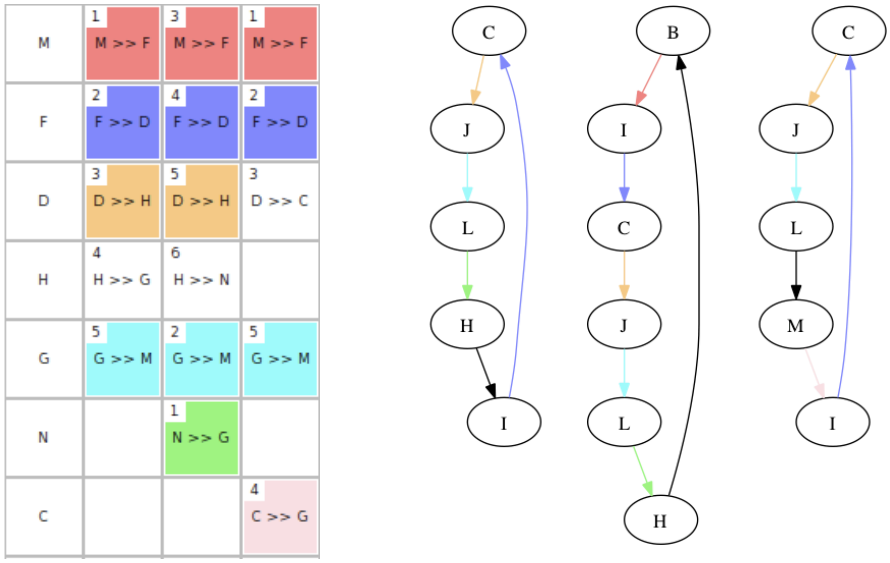


Figure 6.9: Sample of CYCLETABLE (left) and node-link visualization (right) representing the same cycles used in the study. Colors and letters are randomly generated.

Q2: Give a minimal cycle involving N and O (enumerate packages in order). *Rating:* 0 when the answer is {O, G, N}, +1 for each false value.

Q3: How many minimal cycles go through package F? *Rating:* Computed as the difference between the answer and 16, the correct answer.

Q4: How many shared dependencies exit package F? *Rating:* Computed as the difference between the answer and 2, the correct answer.

Q5: How many dependencies should be removed to break all cycles involving package F? *Rating:* Computed as the difference between the answer and 8, the correct answer.

Q6: What is the biggest shared dependency in the system? *Rating:* 0 when the answer is {G, M}, 1 else.

Q7: How many minimal cycles are broken by removing the biggest shared dependency? *Rating:* Computed as the difference between the answer and 24, the correct answer.

Q8: Give the minimum number of edges to remove in order to break all cycles in the system. *Rating:* Computed as the difference between the answer and 10, the correct answer.

6.6.2.3 Results

13 participants performed the study, from bachelors students to experienced researchers with various programming skills and experience in visualizations. We distinguish three parts in the results (Figure 6.10 and Figure 6.11).

The first part relates to questions Q1 and Q2. For these two easy questions, the user should identify cycles. Results show that it is faster to identify a cycle with a node-link visualization. We can consider that it is still more intuitive than CYCLETABLE.

The second part relates to questions 3 to 7, where the user should recognize shared dependencies or packages involved in multiple cycles. Here, CYCLETABLE performs better and faster. Q7 appears as an exception: actually participants in both groups confused two very similar colors, which is a mistake on our part for the choice of colors. These questions validate the design of CYCLETABLE compared to node-link, for the purpose of reasoning about shared dependencies.

Finally question 8 evaluates the capacity to assess the full complexity of the graph. It builds upon the preparatory work made by answering the previous questions as one needs to assess a minimal set of dependencies, mostly based on the impact of removing shared dependencies. Results show it takes in average more than 90 seconds with CYCLETABLE and more than 3 minutes with node-link visualization. While both groups gave similar answers, it highlights the ease to read CYCLETABLE for this task.

6.6.3 Threats to Validity

Rating. We compute a rating based on the distance to the expected answer. We can see that CYCLETABLE provides better answers than a node-link visualization. But false-answers are due to a visualization without software meaning for participants. In the case of real reengineering session, results could be different. It is a part of future work.

Removable Dependency. We suppose for CYCLETABLE that a critical dependency is often shared. In our case study, results show that this hypothesis is right. But we should do more experiments to confirm it.

Smalltalk Software. We analyzed Smalltalk source code which contains class extension. Class extension is a feature available in multiple languages. It makes easier to modularize software, but it makes also easier to make cycles. A work is in progress to analyze Java software.

6.6.4 Conclusion of the Study

We create this visualization to assess cycles at package level. To analyze the usefulness of the visualization, we do not have other visualization tools to compare. We build a node-link visualization which shows shared dependencies. The benefit of node-link visualization is that there is no learning time.

The study shows that CYCLETABLE is efficient to detect and to help reengineers break cycles between packages. There are still some limits that we would like to overcome, with the goal to make CYCLETABLE more effective for reengineers.

The order of cycles in the matrix is based on the similarity they share with each other, based on their common shared dependencies. The order of packages follows cycle sequences as soon as cycles are inserted in the matrix. This heuristic gives good result in the

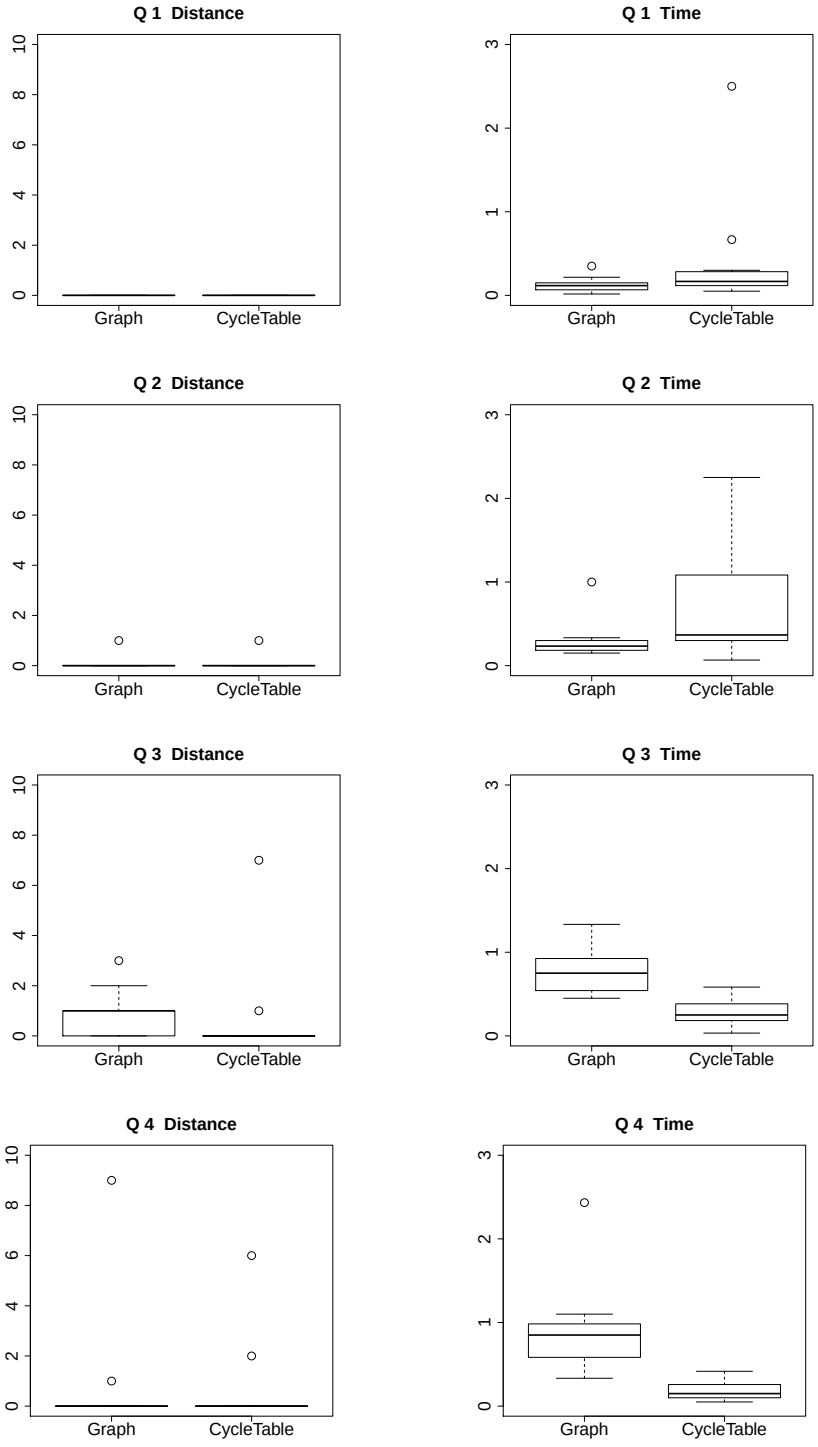


Figure 6.10: Boxplots showing distance to expected answer in absolute and time in minutes for questions 1 to 4. Graph shown on left and CYCLETABLE on right for each question.

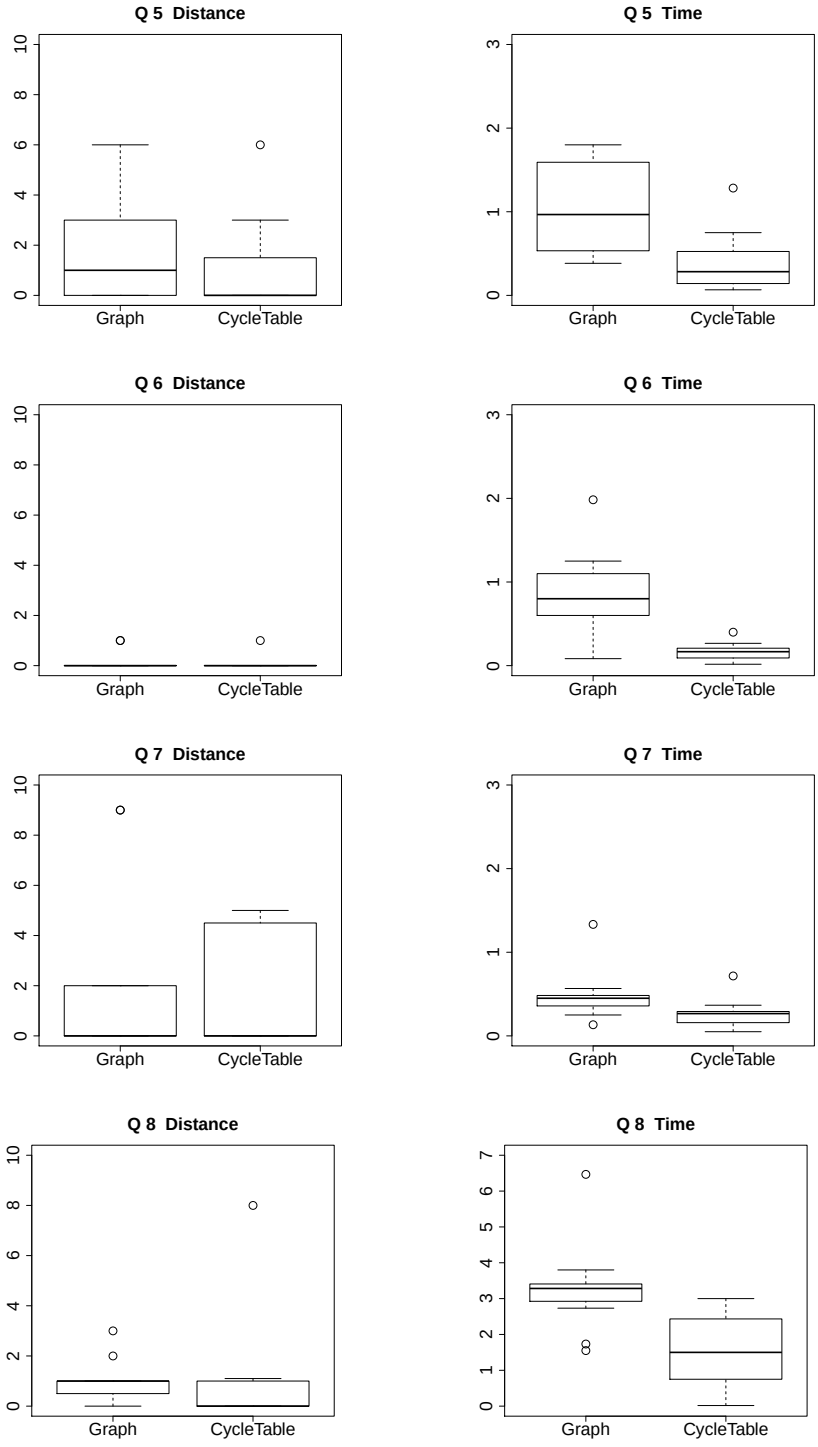


Figure 6.11: Boxplots showing distance to expected answer in absolute and time in minutes for questions 5 to 8. Graph shown on left and CYCLETABLE on right for each question.

first rows and columns of the visualization. Then it becomes difficult to arrange cycles and packages so that a shared dependency forms a unique line of color in its row.

When there are cycles without shared dependencies, CYCLETABLE shows cycles separately but without colors. Such systems are actually simple to understand. In this case the use of other visualization such as node-link or DSM could be better.

6.7 Related work

Node-link visualization. Often node-link Visualizations are used to show dependencies among software entities. Several tools such as dotty/GraphViz, Walrus or Guess can be used. Using node-link visualization is intuitive and has a short learning curve. One problem with node-link visualization 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, cycle identification is not trivial.

Package Blueprint. It shows how one package uses and is used by other packages in an application [Ducasse 2007]. It provides a fine-grained view. However, package blueprint lacks (1) the identification of cycles at system level and (2) the detailed focus on classes actually involved in the cycles.

Dependency Structural Matrix. Contrary to node-link, a DSM visualization preserves the same structure whatever the data size is. This enables the user to dive fast into the representation using the normal process. SCCs can be identified by colored cells. Moreover, EDSM (Chapter 5, p.55) displays 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.

Dependence Clusters. Brinkley and Harman proposed two visualizations for assessing program dependencies, both from a qualitative and quantitative point of view [Binkley 2004]. They identify global variables and formal parameters in software source-code. Subsequently, they visualize the dependencies. Additionally, the MSG visualization [Binkley 2005] helps finding *dependence clusters* and locating avoidable dependencies. Some aspects of their work are similar to ours. Granularity and the methodology employed differ: they operate on source-code and use slicing method, while we focused on coarse-grained entities and use model analysis.

6.8 Summary

This chapter presented CYCLETABLE, a visualization showing cycles between packages in order to break cyclic dependencies. A fundamental heuristic of CYCLETABLE is the focus on *shared dependencies*, which can impact multiple cycles at once by their removal. The visualization is completed with ECELL, which has already been integrated in EDSM (Chapter 5). We validated the heuristic of *shared dependencies* in a case study and the efficiency of CYCLETABLE over a node-link visualization in a comparative study.

The next chapter introduces OZONE. It is an approach which provides proposition of dependencies to remove with the goal to have a layered system (*i.e.,* without package cycle). To propose dependencies to remove, OZONE bases its analysis on two heuristics highlighted in EDSM and CYCLETABLE: remove direct cycles and remove shared dependencies.

CHAPTER 7

oZone, Recovering Package Layer Structure

Contents

I feel very strongly that change is good because it stirs up the system.

[Ann Richards]

At a Glance

In this chapter, we propose an approach which provides (i) a strategy that supports the understanding of dependencies which violate the Acyclic Dependency Principle, (ii) their consequent removal, and (iii) an organization of packages (even in presence of cycles) in multiple layers. While our approach can be run automatically, it also supports human inputs and constraints. We validate our approach with two studies: (i) The first one validates the relevance of the strategy's propositions, and (ii) the second one is a comparison of our results with MFAS.

Keywords: Layered organization, Acyclic Dependency Principle, breaking cycle.

7.1 Introduction

Having a layered organization allows for a simpler system maintenance because side effects are limited to layers. Therefore, it is beneficial to organize packages of an object-oriented system into layers. Such layered organization can help ascertain package dependencies and therefore, facilitate the assessment of changes on different packages. However, identifying such layered structure is difficult. The difficulty arises from the fact that packages often form cyclic dependencies amongst each other. The presence of cycles amongst packages not only breaks the ADP but it also hampers the computation of a layered organization for the packages.

In this chapter, we propose an approach, OZONE, which provides (i) a strategy to highlight and remove the dependencies which break the ADP; (ii) an organization of packages (even in presence of cycles) in multiple layers; and (iii) a simple visualization to allow for developer inputs. The benefits of our approach are two-fold: First, it helps removing unwanted cyclic dependencies. Second, it helps grouping packages into layers.

The approach automatically groups software packages into a layered organization by ignoring *shared dependencies i.e.,* it does not take into account dependencies that are present in multiple cycles. In addition, we believe that the reengineer knows the systems best and knows which dependencies are pertinent to be removed from the system. Therefore, while our approach can be run automatically, it also supports human inputs and constraints so that system knowledge can be imparted in the creation of layers. We validate our approach with a study on the structure of two large open-source software applications: the Moose and Pharo projects. Engineers of these two projects validate the results reported by our approach.

Structure of the Chapter

Section 7.2 (p.104) details the importance of layered architecture and Section 7.3 (p.106) presents the problem of layered organization computation in current approaches. Section 7.4 (p.109) explains our intuition and Section 7.5 (p.111) explains our approach. Section 7.6 (p.114) proposes a simple user to interact with a layered organization. Section 7.7 (p.116) presents the validation of the approach. Related work is presented in Section 7.8 (p.123). Section 7.9 (p.126) discusses and concludes the chapter.

7.2 Layer Identification

Using a layered view of the software architecture is a common approach to understand it [Ducasse 2009c]. In addition, when the structure of a large system is layered, it simplifies its evolution since changes are limited to layers. To build a layered organization at the package-level, there are two important principles: the ADP which is a pre-condition to the computation of the second principle and the computation of the layered architecture that we want to build.

Layered architecture. For Bachmann *et al.* [Bachmann 2000], there are two properties in a layered architecture: (i) a layer B is below a layer A if elements (*i.e.,* packages) in layer A can use elements in layer B and (ii) layer A can use only packages below it (in Figure 7.1, layer B). In Figure 7.1 the dotted dependency from *Kernel* to *pA* should not exist. Szyperski [Szyperski 1998] and Bachmann [Bachmann 2000] make a distinction between *Closed* layering and *Open* layering. In closed layering, the implementation of one layer should only depend on the layer directly below it. In this case, in Figure 7.1 the dependency from *pE* to *Kernel* should not exist. In open layering, any lower layer may be used.

Our layered organization is based on the Open layering because the Closed one is not adapted to software architecture. The Closed layering is useful for a clean and well encapsulated system, which is not adapted to our purpose because we do not want to identify a well-encapsulated system, but to identify dependencies which break the layered organization.

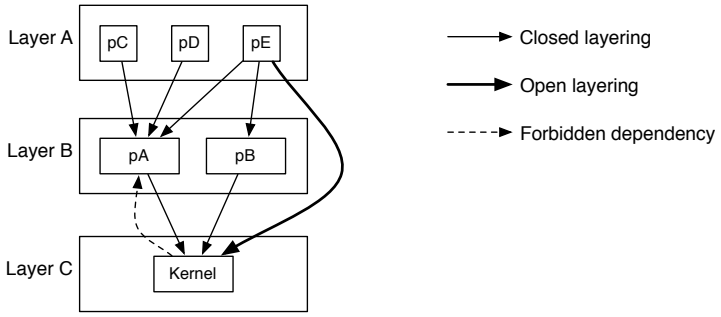


Figure 7.1: Layer Description - dashed arrow represents an *unwanted* dependency, thick arrow represents a dependency allowed in *Opened* layering.

Acyclic Dependency Principle (ADP). A layered system offers good properties of modifiability and portability [Bachmann 2000]. It means that there are no cycles between packages and that software changes do not propagate to all the layers. Martin defines the Acyclic Dependencies Principle (ADP) [Martin 2000]. It proposes that the dependency graph between packages should be a directed acyclic graph: there should not be any cyclic dependency between packages. Package structures with cycles are in general more difficult to understand, maintain and deploy than those that conform to the ADP. Legacy and large systems often present structures which do not respect this principle. Like an organic system and following the entropy principle, the more software grow, the harder it is to keep this property.

Cycles: a problem for layer identification. Identifying layers in a package system is not trivial. In particular, when some packages are in cycle, layers cannot be computed without considering cycles as a special artifact. Different strategies may be used: for example, a

cycle may be considered outside the layered architecture as NDepend[1] does. It may be integrated in a single layer as Lattix or MudPie [Vainsencher 2004] do (Section 7.3.1, p.107). The same strategy is used at the class level [Lutz 2001, Mancoridis 1999, Mitchell 2006], all classes in a cycle are assigned to the same layer.

This way to compute layers is not well suited for packages. First, most of the time, packages should not be in cycles, as the ADP states. Second, if two packages in a cycle use each other, these cannot be placed in two different layers. An alternative is to place them in the same layer. However, in a large software system with numerous packages in cycle, the granularity of a layer can become so large that the layered structure would be useless and totally artificial. For example, working on the remodularization of Pharo[2], an open-source Smalltalk environment, we found 70 packages in cycles. These packages belong to the core (package Kernel), to the UI (package Morphic), to the protocol (package Network) and to other subsystems. In this case, grouping cyclic packages would lead to giant layers. Another approach, Minimum Feedback Arc Set provides a good approach to reduce cycles present in a graph. However, it does not allow for user constraints [Eades 1993].

To better understand the different approaches to build package layered view in the presence of cycles, we illustrate common approaches used in the two commercial tools already cited: Lattix and Structure101. We now present these approaches and their limits.

7.3 Limitation of Existing Approaches

In this section, we demonstrate the problem of layer identification in existing tools. For illustration purposes, we use an example graph to clearly present the problem that we address. In Figure 7.2, we propose a simple graph with 4 nodes (*i.e.,* packages). All the nodes in the graph are in cycle. All edges in the figure are labeled with the number of dependencies between classes, the *weight* of the package dependency. In the following, we describe the limitation of the layer creation in the existing tools.

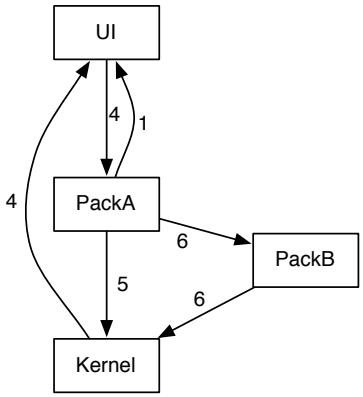


Figure 7.2: An example of cycles between packages.

[1] http://www.ndepend.com
[2] http://www.pharo-project.org

We know two main approaches that extract layered structure from package dependencies. These two approaches are based on Dependency Structure Matrix (DSM) [Sangal 2005] and Minimum Feedback Arc Set (MFAS) [Eades 1993]. These two approaches are incarnated in two tools: Lattix[3] and Structure101[4]. The problem with the first approach is that it does not deal adequately with cycle consideration and identification of effective layered structure in presence of cycles. The problem of the second approach is that the algorithm used (MFAS) does not allow for user constraints.

Other approaches exist in Regression and Integration Testing domain. Particularly, Le Traon et al. [Le Traon 2000] find SCCs and search to minimize the number of stub creation. The goal is not to minimize the number of edges but to minimize the number of vertices impacted. Brian et al. [Briand 2001] also find SCCs and try to remove only association dependencies, because it is the weakest type of dependency [Kung 1996]. More details about these approaches are presented in Section 2.4 (p.17) and in related work of this chapter (Section 7.8.3, p.124).

7.3.1 Dependency Structural Matrix

Dependencies structural matrix is the underlying approach used by tools such as Lattix LDM [5]. It allows engineers to create a dependency model of a software system. These dependencies can be managed with help of visualizations based on DSM. The tool provides features to make propositions for reengineering cyclic dependencies and provides a what-if approach to work on the structure.

The approach is the concrete implementation of the work of Sangal et al. [Sangal 2005]. In the paper, Sangal et al. propose to populate Dependency Structural Matrix (DSM) based on the information of package dependencies. Further, the dependency information is used to work out a layered organization for the system under analysis. This work considers a cycle as a feature, not as a modularization problem. It proposes to group each cycle in a container, named “module”. A module contains a group of packages in cycle. When computing layered organization of packages, modules are considered as a separate layer. Non-cyclic packages are computed with the algorithm listed below:

```
1:   Model::computeLayers(): void {
2:       for( Package package: allPackages() ) {
3:           if (package.isInCycle() or package.useAPackageInCycle())
4:               package.layer := notAttributed
5:           elseif (package.providers() = nil)
6:               package.layer := 0
7:           elseif (package.providers().layer() = 0)
8:               package.layer := 1
9:           else
10:              package.layer := 1 + Max(package.providers().layer())
```

[3] http://www.lattix.com

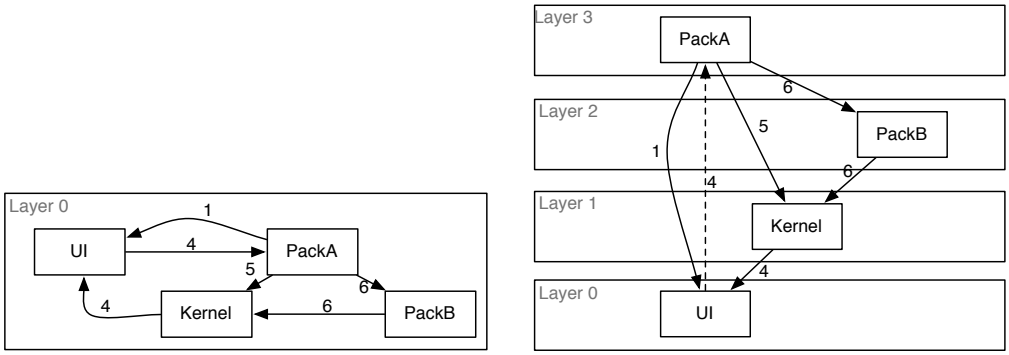
[4] http://www.headwaysoftware.com/products/structure101/

[5] http://www.lattix.com/

```
11:     }
12: }
```

Figure 7.3(a) shows an example of the package layers discovered with Lattix considering the packages in Figure 7.2. As the tool creates a single layer for all the packages, the packages present in the system are placed in a single cycle.

Problems in this approach are multiple: (i) if cycles exist between packages that should be in different layers, they will be grouped in the same layer; (ii) it is not possible to differentiate a layer built from a cycle and one built from dependencies to a lower layer; (iii) when there are dependencies creating cycles between multiple packages, it is not possible to identify them because the layered view does not provide this kind of information (does not show package dependencies). The tool does not target to resolve cyclic dependency problem and does not include cycles for computing a layered organization.



(a) Result of applying Lattix's Strategy.

(b) Result of applying MFAS Strategy.

Figure 7.3: Layered organization obtained by different strategies when applied to the system described in Figure 7.2.

7.3.2 Feedback Arc Set

In graph theory, a feedback arc set computes an acyclic graph. The *minimum* feedback arc set (MFAS) is the minimal collection of edges in the graph to remove to break a cycle. In a weighted graph, it removes the lightest of the dependencies to remove cycles This approach can produce good results working on package dependencies because it requires to make minimum modification to the software structure to break cycles. A limitation of using MFAS is that it does not take into account the semantic of the software structure. Optimizing a graph is not equivalent to identify the layered architecture of an existing software system.

MFAS principle does not take into account the semantic of the structure. It removes edges from a graph without taking into account the importance or the kind of dependencies. Moreover, it does not take into account interaction with the user. In addition, MFAS

algorithm acts as a black box without possibility of user interaction.

Figure 7.3(b) presents the layered organizations obtained by applying the MFAS algorithm. This organization produced by MFAS does not adequately handle cycles between packages: it removes edges to break cycles without taking into account the importance of the removed dependency. The resulting organization places user interface (UI) at the bottom of the layer organization, which is not correct in this case. The UI should be independent of the underlying implementation. Hence, breaking the minimum number of cycles can create incorrect layers. Cycles should be broken such that the resulting organization should capture the reengineer's understanding of the system.

A tool, named Structure101[6], proposes to build layer organization and to analyze dependencies which violate the layered organization. The tool employs feedback arc set to break cycles amongst packages. This tool is well integrated with the source-code environment and it provides services to modify directly source code. However, the interaction with the developer to remove cyclic dependency is minimal.

7.4 The Intuition behind OZONE: Direct Cycles and Shared Dependencies

As a general principle, cycles between packages should not exist. We consider that package cycles stem from design issues, architecture violations, or programming errors. A dependency which creates a cycle should be highlighted when an analysis of the system is performed for creating package layers. This provides software engineers an opportunity to focus their attention on the dependency that should probably be eradicated from the system. Hence, the cornerstone of our approach is to find *unwanted* dependencies *i.e.,* the dependencies that can be removed from the system to break cycles.

Intuition. Our approach, OZONE, is based on two intuitions for finding *unwanted* dependencies. The first intuition is that in a cycle, all package dependencies don't have the same strength. The strength of a package dependency is the number of relations amongst classes involved in the dependency. In a cycle between two packages (defined as *direct cycle* in Section 3.3, p.28), we remove the lightest of the dependencies to break the cycle. We hypothesize that the lightest dependency is inadvertently introduced in the system and it is the result of a design defect or a programming error. Moreover, our hypothesis is that lightest dependencies often require the least amount of work. Based on this hypothesis, our approach analyzes only *direct* cycles and ignore for each direct cycle, the lightest dependency before analyzing the whole graph. This is a strong difference with MFAS, because MFAS analyzes the whole graph and remove the dependencies that are involved in the most cycles.

Our second intuition is based on the occurrence of dependency in cycles involving three or more packages (defined as *indirect cycle* in Section 3.3, p.28): A frequent dependency in indirect package cycles is a suitable candidate to be ignored to break cycles between

[6]http://www.headwaysoftware.com/products/structure101/

packages. We refer to it as a *shared dependency*. We hypothesize that *shared dependencies* have a strong impact on the system structure. The hypothesis follows logically from the fact that dependencies shared the most, are the ones that have a strong and bad impact on the system structure.

Principle. To determine *unwanted* dependencies in a package structure, we decompose package cycles into small ones such that no cycle appears twice. Amongst these cycles, first we select the direct cycle and in each direct cycle we ignore the weakest dependency. Then, amongst the rest of the cycles (*i.e.,* indirect cycles), we ignore *shared dependencies*. In this way, layer identification is possible while circumventing the problem of layer creation in the presence of cycles mentioned in the previous sections.

Figure 7.4 presents the layered organization that we would like to see for the package structure presented in Figure 7.2. We would like to have a decomposition of the structure in multiple layers while highlighting the dependencies which can be removed to break cycles.

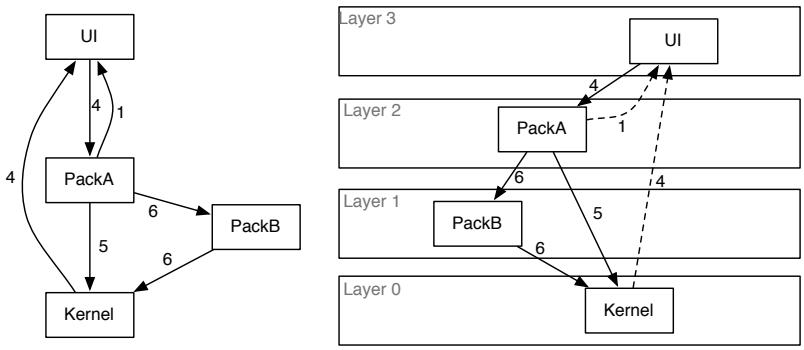


Figure 7.4: The expected layered organization.

To compute the layered organization for the structure in Figure 7.4-left based on the removal of *unwanted* dependencies:

- First we decompose cycles amongst packages into smaller ones such that no cycles get repeated. The cycles amongst the packages (PackA, PackB, Kernel, UI) are decomposed into three cycles: (UI -> PackA -> UI), (UI -> PackA -> Kernel-> UI), and (UI -> PackA -> PackB -> Kernel-> UI). The cycle (UI -> PackA -> UI) is a direct cycle so we start with this to break cyclic dependencies. Applying our first intuition, we ignore the lightest dependency *i.e.,* (PackA -> UI).

- Then there are two indirect cycles and we can notice that the dependency (Kernel -> UI) appears in both indirect cycles: It is a *shared dependency*. Therefore, we ignore this dependency to break the cycle amongst the packages. Once these *unwanted* dependencies are ignored (dotted in Figure 7.4), all the package cycles are addressed and we can correctly compute the package organization as illustrated in Figure 7.4-right.

7.5 Our Solution: Detecting Dependencies Hampering Layer Creation

In this section, we present a strategy to build a layered organization of system packages even if there are cycles amongst these packages. The approach considers all cycles and finds the "adequate" dependencies to ignore to create distinct layers. We named them *dependencies breaking or hampering layer creation*. These dependencies are defined based on the intuition explained in previous section. It focuses only on dependencies in cycles. The strategy is not concerned with other dependencies because the primary goal is to break cycles to build layered organization.

This section is organized as follows: First, we describe the algorithm for finding *layer-breaking dependencies*. Second, we explain how to compute a layered organization. Finally, we explain a feature to define constraints on dependencies manually, for adapting layered organization to the reengineer vision.

7.5.1 Highlighting Layer-breaking Dependencies

As we showed in the previous section, the problem to build a layered architecture is how to take into account cycles between packages. We propose a heuristic that is based on the observations and experiments from EDSM (Chapter 5, p.55) and CYCLETABLE (Chapter 6, p.83). We apply the strategy on a graph where each node represents a package and each edge represents a dependency between two packages. A dependency between packages depends on relations between classes and methods inside these packages. Edges are weighted with the number of dependencies between elements in the package (class inheritance, class extension, class reference, method invocation, and variable access).

The algorithm runs in two steps and ignores unwanted edges from a copy of a system model. We ignore ("remove logically") these *unwanted* dependencies when assigning packages to layers. Once these dependencies are ignored, we compute layer organization for the packages present in the system:

1. one of the two dependencies of each *direct cycles* is ignored. To eliminate a direct cycle, the algorithm considers the weight of each dependency in the cycle and marks the lightest as unwanted.

2. when all direct cycles are removed, the algorithm considers the new version of the graph and computes SCC and minimal cycles to retrieve *shared dependencies*. When there are *shared dependencies*, it removes the one shared the most, because this is the one with the highest impact on minimal cycles. We repeat this action as long as there are cycles. If two dependencies are shared by the same number of cycles, the algorithm selects the lightest of them.

The algorithm computes first direct cycles. There are two reasons to begin with them: (i) when a cycle is addressed, it could have an impact on other larger cycles that include

it[7], (ii) addressing a direct cycle is somehow simpler because there are only two solutions: cutting one or the other of the two package dependencies.

Based on the definitions and strategy previously explained, we propose an algorithm to ignore edges in a package dependency graph. In this algorithm, the term "ignore" is used to (i) remove an edge from the graph to make it acyclic and (ii) to store the dependency to highlight it with the goal to later understand it in the source code.

```
1:  Model::getRemovedEdges(Graph graph): Collection {
2:      for (Cycle cycle: computeDirectCycles()) {
3:          if (cycle.edgeOne.weight() > cycle.edgeTwo.weight() * 3)
4:              graph.remove(cycle.edgeTwo)
5:          elseif (cycle.edgeTwo.weight() > cycle.edgeOne.weight() * 3)
6:              graph.remove(cycle.edgeOne)
7:          else
8:              graph.removeTheMostSharedEdge(cycle.edgeOne, cycle.edgeTwo)
9:              or (graph.removeTheLightestEdge(cycle.edgeOne, cycle.edgeTwo))
10:             or (graph.remove(cycle.edgeOne))
11:     }
12:     while (computeSCC().notEmpty()) {
13:         graph.computeMinimalCycles()
14:         graph.removeTheMostSharedEdge()
15:     }
16:     return graph.removedEdges
17: }
```

The presented algorithm works as follows: from line 2 to line 11, direct cycles are removed from the graph. It checks for large differences between the two edges of the direct cycle (it uses a ratio of 1/3) and removes the lightest (l.3 to l.6). If the difference is not important enough, the algorithm checks *shared dependencies* and removes the most shared (l. 8). If the two edges have the same number of *shared dependencies*, it removes the lightest edge (l.9). If none of these conditions are satisfied, the algorithm removes the first edge (l.10). This last line is necessary to remove all cycles to make a layered architecture. It is akin to removing a random dependency in the cycle, but the engineer can specify constraints to guide the tool by explicitly marking a dependency as valid or not (explained in Section 7.5.3, p.113). Then from line 12 to 15, the algorithm removes other cycles if other SCC are detected. It computes minimal cycles (l.13) and removes the most shared (l.14). If there are multiple dependencies with the same shared number, it selects the less weighted dependency. The algorithm returns a collection of *layer-breaking dependencies*.

7.5.2 Building Layers

When the previous algorithm returns *layer-breaking dependencies*, we can convert cyclic package dependencies graph into an acyclic graph by ignoring the dependencies creating

[7] For example in Figure 7.2 breaking the direct cycle PackA-PackB, can also break the cycle PackA-PackB-PackC.

cycles. With this acyclic graph, we can easily build a layered organization. These specific dependencies are however, later presented to the reengineer for further analysis.

The algorithm is same as used in other approaches. The first layer is built with the packages which do not use any other packages. Then each layer is built with packages which use only packages on lower layers.

```
1:   Model::buildLayers(Graph aCyclicGraph): Collection {
2:       L := Collection
3:       N := aCyclicGraph.nodes
4:       while (N.notEmpty()) {
5:           currentL = L.addNewLayer()
6:           concernedNodes = N.selectNodesWithoutOutgoing(N)
7:           currentL.add(concernedNodes)
8:           N remove(concernedNodes)
9:       }
10:      return L
11:  }
```

The previous algorithm builds the layered architecture. The lines 2 and 3 initialize variables: L is a collection of layers. Each layer is a collection of packages. N is the collection of all packages of the system. Lines 4 to 9 build the layered architecture. Each layer (l.5) is filled by putting into it packages without any outgoing dependencies to packages contained in N (l.6 and 7). Finally, it removes the selected nodes from N (l.8).

7.5.3 Manually Defining Constraints

The algorithm may provide a result that does not completely match the vision of the reengineer. The automatic computation based on our strategy provides a first step to understand the shape of the system. The reengineer should be able to impart his/her knowledge of the system and impose constraints on cyclic dependencies removal. He should evaluate each dependency in the system and introduce constraints. Then the system should recompute the layer organization according to the provided constraints.

For this purpose, we design four possible evaluations: (i) *flaggedByAlgo* for dependencies detected by the algorithm as unwanted (the *layer-breaking dependencies*), (ii) *unwanted* for dependencies that the software engineer would like to remove, (iii) *notFlagged* for the dependencies for which the software engineer does not know whether they are expected or not and the algorithm left untouched, (iv) *expected* for the dependencies which should not be removed in the software engineer opinion.

These constraints add a new dimension to the algorithm:

- By default the algorithm flags with *flaggedByAlgo* dependencies which it considers breaking layers. Then the engineer should confirm with the flag *unwanted* or invalidate with the flag *expected*.
- When a dependency is flagged *expected*, it is not removed by the algorithm, and it checks another dependency to remove the cycle. If all dependencies of a cycle are

flagged as *expected*, the cycle cannot be removed. In this case, all involved packages are together in a single layer.

- When a dependency is flagged *unwanted*, it is automatically ignored. So the algorithm does not take into account the dependency. When it is a part of a cycle, it is not computed in SCC and minimal cycle search.
- When a dependency is *not flagged*, the algorithm considers it as removable if necessary.

In the following section, we show a simple prototype that illustrates how results could be presented to the engineer.

7.6 A Prototype of an Interactive Browser to Build Layers

We built a user interface to present features provided by the approach: (i) highlighting dependency strategy, (ii) user defined constraints, and (iii) layered architecture building. This interface is a prototype, it has been created for our study and should probably be improved. We do not consider it as a contribution of this article.

The prototype has been implemented on top of the Moose software analysis platform [Ducasse 2005a] and it is based on the FAMIX language independent source code metamodel [Demeyer 2001]. It can work on Java, C#, and C++ as well. Therefore while implemented in Smalltalk, it can be applied to mainstream object-oriented languages.

It is composed of three main panels: a layers visualization with polymetric view (on the left of the UI) [Lanza 2003b], a list of unwanted dependencies (on the top center of the UI), and a list of all dependencies of the system (on the top right of the UI)[8]. The list of unwanted dependencies contains dependencies selected by the algorithm and dependencies removed manually by the engineer.

7.6.1 The Layer Visualization

As its user interface shows, the layer visualization is an experimental tool. We build it to show our main concerns: relations between packages, packages forming SCC, and some convenient metrics about the size of packages to help us to develop a basic understanding of the system.

Layer. It is a rectangle with boxes inside representing packages. Bottom packages are in Layer 0 (*i.e.,* core layer). Each new rectangle represents a new Layer (Figure 7.5, left part).

Package. Packages are represented with polymetric views [Lanza 2003b, Demeyer 1999, Gîrba 2005b]. Each package is represented by its name and a box. The height, the width, the fill color and the border color have a meaning (Figure 7.6):

[8]The lower right part of the figure is for more advanced analysis. It represents an ECELL visualization (Chapter 4, p.39)

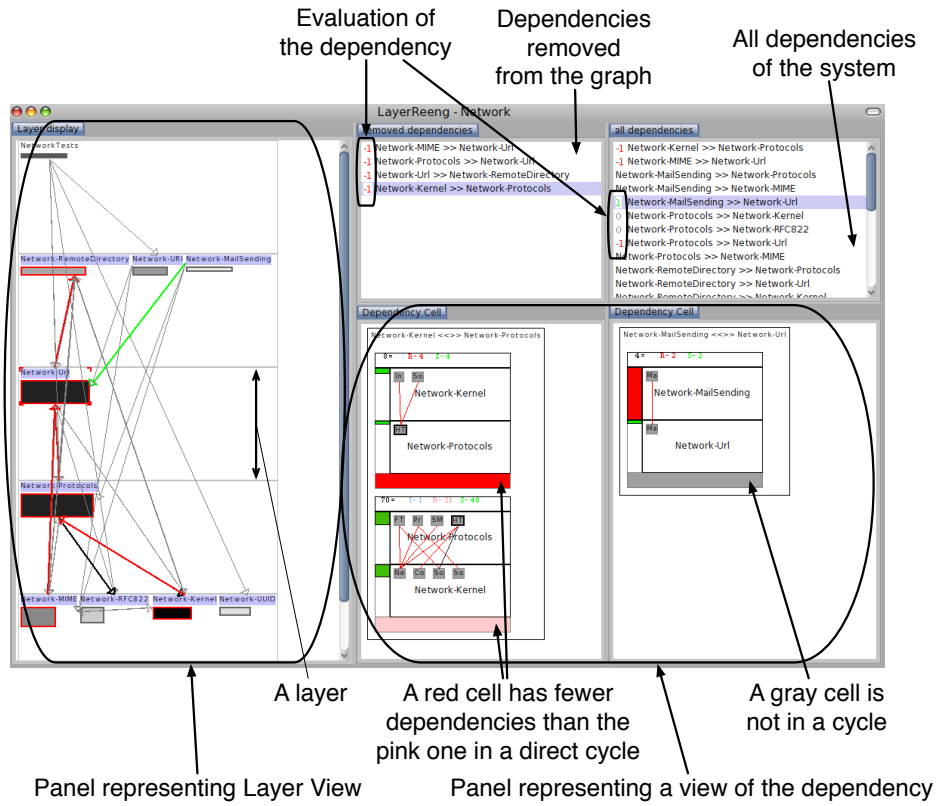


Figure 7.5: UI to build layered view based on constraints.

- height: it represents the number of client packages of the package. It means that tall packages should be located in lower layers.
- width: it represents the number of used packages of the package. It means that a package in a low layer should be narrow.
- fill color: it represents the number of classes in the package. The darker a package is, the more classes it contains.
- border color: it represents an SCC. If the color is gray, the package is not in an SCC. If it is another color, the package is in an SCC with all other packages with the same border color.

This kind of information allows the engineer to understand at a glance the dependencies of the packages present in the system, but again this is not the focus of this article.

Dependency. A dependency is represented by a line with and arrow: the arrow is in the direction of the dependency. Dependencies can have three colors. The red color represents a dependency flagged *unwanted*, a dependency that the engineer wants to remove. The

green color is a dependency flagged *expected*, a normal dependency. The black color is a dependency not evaluated. Finally the color gray represents dependencies not yet evaluated.

This system allows the reengineer to understand the dependencies between packages in more detail.

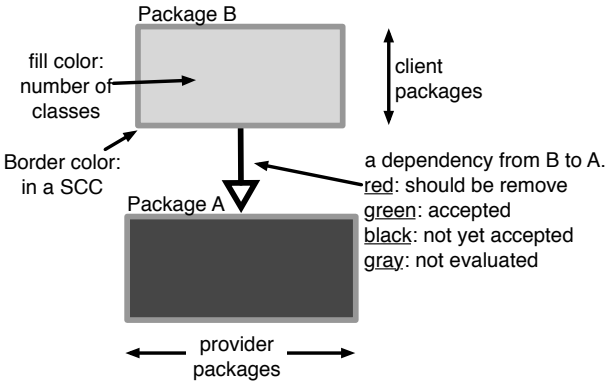


Figure 7.6: The polymetric view used in Layer View

7.6.2 Interactions

Interactions on lists. On the two lists on the top right of the UI (one as unwanted dependencies and one with all dependencies of the system), reengineers can define constraints explained in Section 7.5.3 (p.113). With a simple right click, a popup menu appears and proposes to select one of the three available values: 1 (for *expected*), 0 (for *notFlagged*) or -1 (for *unwanted*).

When a new value is given to a dependency, the algorithm recomputes the layered organization and the UI is updated. One can see the result automatically.

Interactions on layer view. On the layered view, it is possible to select a package and put it in another layer.

A package (fixed on a layer) influences the algorithm to consider as *layer-breaking dependencies* all dependencies which do not respect the layered architecture definition *i.e.,* all dependencies breaking layer are ignored. Then the reengineer should confirm that all these dependencies are unwanted by flagging them with *unwanted*.

7.7 Validation

We performed two distinct experiences and validation studies. *Layer relevance:* The first study validates that the dependencies selected by our heuristic are relevant and build a layer organization corresponding to the structure of the software. *Algorithms comparative study:*

The second study validates our heuristic on a big system (Pharo 1.1 has 115 packages) and results are compared with the MFAS algorithm.

We performed the experiments on two systems: the Moose software analysis platform [Nierstrasz 2005] and the Pharo Smalltalk development environment [Black 2009]. We selected these two systems because they are large, contain realistic package dependencies, are open-source and available to other researchers, and also because we were able to get feedback on our results by engineers working on the two projects. In particular, Pharo and Moose engineers (independent of the authors of this article) checked and qualified all the dependencies of the Moose and Pharo system and this allows us to compute recall and precision between our algorithm and MFSA.

7.7.1 Layer Relevance on Moose Software Analysis Platform

We performed a study to validate our approach. The goal of this study is to validate two important features of the approach: (i) Are *layer-breaking dependencies* relevant? (ii) Does the layer organization fit the programmers understanding of the system?

7.7.1.1 Protocol

The case study was realized on the beta version 4 of Moose 4.0. The program contains 33 packages and 106 dependencies amongst these packages. This version was chosen because it contains a lot of cyclic dependencies, since it is known to not be well modularized. A developer from the Moose team (independent from this research) evaluated all the 106 package dependencies of the system. He flagged the dependencies as already explained: *expected*, *not flagged*, and *unwanted*. Some metrics for Moose 4.0 beta 4 are presented in Table 7.1.

Moose 4.0 beta4 characteristics	
Number of packages	33
Number of packages in cycles	14
Number of package SCC	1
Number of direct cycles	10
Number of package dependencies	106
Number of package dependencies involved in SCC	56

Table 7.1: Moose 4.0 beta4 characteristics.

Then, we ran our algorithms on the system and compared *layer-breaking dependencies* found by the algorithm and *unwanted* dependencies given by the engineer.

7.7.1.2 Results and Discussion

Phase 1: Algorithm results. The algorithm proposed to remove 15 dependencies (Table 7.2) that we analyzed and compared with the values given by the Moose engineer. Table 7.2 shows that 10 out of 15 *layer-breaking dependencies* (66%) found by our tool are considered *unwanted* by the engineer. Without manual changes, results validate that the

approach has a good strategy. The 5 false-positive dependencies are due to two kinds of issues in the algorithm.

- First, the two couples *Moose-Finder – Moose-Wizard* and *Moose-SmalltalkImporter – Famix-Implementation* are in direct cycle. It means that there are also two dependencies *Moose-Wizard to Moose-Finder* and *Famix-Implementation to Moose-SmalltalkImporter*.

  In these two cases, both dependencies in the two direct cycles have similar weights. Without *shared dependencies* available, the algorithm had no clue which dependency to ignore and had to choose "randomly" the first of the two.

  This point should be improved in future work with the possibility for the maintainer to choose the *unwanted* dependencies avoiding this kind of false-positive.

- Second, the three false-positive results going to Moose-Core (*Famix-Core » Moose-Core*, *Moose-SmalltalkImporter » MooseCore* and *Moose-GenericImporter » Moose-Core*) are related to a choice of the algorithm to remove the dependency *Famix-Core » Moose-Core* instead of *Moose-Core » Famix-Core*.

  In this case, the problem comes from two dependencies from a direct cycle which have the same weight but *Famix-Core » Moose-Core* is shared one more time than *Moose-Core » Famix-Core*. If we constrain to remove *Moose-Core » Famix-Core*, these three false-positive results disappear.

Layer-breaking dependencies	Value given by engineer
Famix-Extensions » Moose-Finder	unwanted
Moose-Core » Famix-Implementation	unwanted
Fame » Moose-Core	unwanted
Famix-Extensions » DSMCore	unwanted
Famix-Core » Famix-Implementation	unwanted
DSMCore » DSMCycleTable	unwanted
Glamour-Helpers » Glamour-Core	unwanted
Glamour-Browsers » Glamour-Scripting	unwanted
Moose-Core » Famix-Extensions	unwanted
Famix-Smalltalk » Famix-Extensions	unwanted
Moose-SmalltalkImporter » MooseCore	expected
Moose-Finder » Moose-Wizard	expected
Moose-GenericImporter » Moose-Core	expected
Famix-Core » Moose-Core	expected
Moose-SmalltalkImporter » Famix-Implementation	expected

Table 7.2: *Layer-breaking dependencies* returned by the algorithm.

The first version of the layer organization proposes 11 layers. Figure 7.7 shows a simple view of layers organization. It shows particularly the main problem revealed in the first result of the algorithm: Moose-Core is too high in the layer organization, due to a false-positive result (*Famix-Core » Moose-Core*).

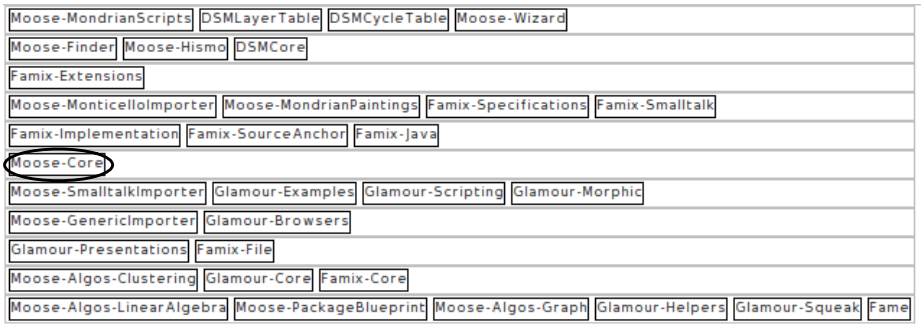


Figure 7.7: Moose packages layer organization proposed by the algorithm.

Phase 2: Manual flagging. Then, based on manual evaluation of dependencies by the Moose developer, we manually flagged as *unwanted* the 10 correct propositions of our algorithm and flagged as *expected* the 5 false-positive propositions.

Phase 3: Algorithm results. Finally, after the manual flagging, the algorithm retrieved 4 new *layer-breaking dependencies*, shown in Table 7.3. All of them are considered *unwanted* dependencies by the engineer. In total, the algorithm proposes 14 dependencies considered *unwanted*.

Layer-breaking dependency	Value given by engineer
MooseCore » Moose-SmalltalkImporter	unwanted
Moose-Core » Famix-Core	unwanted
Moose-Wizard » Moose-Finder	unwanted
Moose-Core » Moose-GenericImporter	unwanted

Table 7.3: New *layer-breaking dependencies* after manual constraints.

After manually evaluating results and providing new information to remove the false-positive results, the algorithm recomputes the layers organization and provides a 8 layers organization (Figure 7.8). In this view, we see that the problem of Moose-Core is resolved because Moose-Core appears on the second layer, which is the correct place for this package.

7.7.2 Comparative Study with MFAS on Pharo

In this section, we present the validation of our approach performed on Pharo. Pharo is a new open-source Smalltalk-inspired environment (1558 classes in 115 packages in version 1.1). We applied our approach to find the cyclic dependencies present in Pharo. The purpose of the study is to validate the results obtained with our approach and compare them with the results obtained by MFAS. Some metrics roughly characterizing Pharo are presented in Table 7.4.

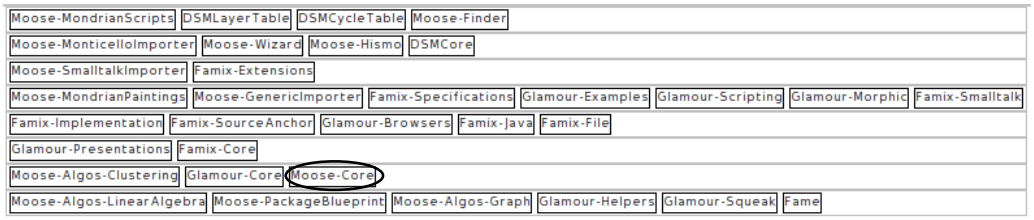


Figure 7.8: Moose packages layer organization proposed after providing manual constraints.

Pharo 1.1 characteristics	
Number of packages	115
Number of packages in cycles	68
Number of classes	1558
Number of package SCC	1
Number of direct cycles	98
Number of package dependencies	1328
Number of package dependencies involved in SCC	922

Table 7.4: Pharo 1.1 characteristics.

7.7.2.1 Protocol

We asked engineers from the Pharo community to evaluate all the 1328 dependencies between packages present in Pharo: the dependencies were systematically evaluated by the engineers over a period of a couple of days. They flagged them using the ranking we presented before. It helps comparing our results with the package dependencies extracted by the developers of the system. Hence, we present a comparison of the values manually computed by the developers.

Using our approach independently to this evaluation, we computed edges removed by the JooJ MFAS tools [Melton 2007b] and unwanted dependencies found by our own algorithm.

Then, we compute the precision and recall of our results: tp $(true\ positives)$ is the number of dependencies to be ignored computed by the algorithm and identified as unwanted by the developer. fp $(false\ positives)$ indicates the number of dependencies which have been detected by the algorithm but which are not identified as unwanted by the developer (note that they can be not flagged), while fn $(false\ negatives)$ indicates the number of unwanted dependencies for the developer which are not detected by the algorithm. The precision of a solution is computed as:

$$Precision = \frac{tp}{tp + fp}$$

and indicates how much of the detected dependencies are unwanted. The closer to 1 P is,

the more precise the tool. The recall is defined as:

$$Recall = \frac{tp}{tp + fn}$$

and indicates how many of the unwanted dependencies the algorithm is able to recover.

7.7.2.2 Results

Manual evaluation results. We provide in this section results of the study. Table 7.5 describes the evaluation of various dependencies discovered in Pharo: 20% of the total dependencies were marked as unwanted.

Evaluation	Value
unwanted	0.20
not flagged	0.02
wanted	0.78

Table 7.5: Manual evaluation of dependencies in Pharo.

SCC manual evaluation results. Since both of the approaches that we evaluate consider ignoring dependencies that constitute SCCs, we compute the percentage of the dependencies in SCCs that are marked as unwanted. Table 7.6 illustrates the results. Of all the dependencies that exist in SCCs, only 25% are considered unwanted by the developers.

Evaluation	Value
unwanted	0.25
not flagged	0.02
wanted	0.73

Table 7.6: *Unwanted* Dependencies in SCCs.

Comparison of MFAS and OZONE. We compared the precision and the recall of MFAS and OZONE. Table 7.7 presents the results.

Technique	Precision	Recall
MFAS	0.61	0.39
OZONE	0.64	0.42

Table 7.7: Comparison of the Accuracy: OZONE and MFAS

The validation of the approach demonstrates that the accuracy of our approach is better in finding cyclic dependencies (Table 7.7). Whereas, 61% of the dependencies removed by MFAS were accurate, our approach removed 64% dependencies. Also, we could identify 42% of the total *unwanted* dependencies, whereas MFAS computed only 40% of those. Hence, our approach performs better than MFAS on this example. Moreover, our approach

allows the reengineer to review the dependencies marked as *unwanted* by the approach and applies appropriate constraints. These constraints are then used in layer computation. We also provide a visualization user interface to support the evaluation of the dependencies that we consider should be removed from the system.

Although our approach produced better results, we believe that it can be improved by validating it over more systems. Also, the approach could be extended to take into account design patterns such as Model-View-Controller (MVC) that could provide semantic information. In this case, if a cycle is discovered between the view and the model, we should remove the dependency from model to view. It can also be interesting to compute the number of cycles that are considered as *normal* by the developer by checking all dependencies that are considered *expected* and are in a cycle.

7.7.2.3 Threats to Validity

The threats to the validity of our results may arise due to four different reasons.

(i) Only two systems were tested. We validate our approach based on manual validation by developers. We think that this is the best validation that we can have. The problem of this kind of validation is that it takes a long period of time to evaluate each dependency manually. Here, for Moose it took a couple of hours and for Pharo a couple of days. We plan to apply our approach on other systems, developed in other languages *e.g.,* Java to further evaluate results, however one difficulty is to find engineers ready to spend time performing an evaluation.

(ii) The developers may have made mistakes. This threat to validity has a small impact because packages are coarse-grained entities. Knowing that a package should depend or not on another one is easy for developers for most of the packages. It happens that some packages are in direct cycles without to be conceptually wrong, in such a case they belong to the same layer. When such situation occurs engineers may flag differently the dependencies depending on whether they took this dimension in perspective.

(iii) There are different heuristics for MFAS. We use the heuristic proposed by JooJ [Melton 2007b] because it was already developed and readily available. A different heuristic may provide better results.

(iv) In our algorithm, we ignore dependencies to obtain an acyclic graph. After ignoring *direct cycles* and *shared dependencies*, the algorithm ignore light dependencies. The number of this kind of dependencies was not evaluated and the impact of this heuristics was not evaluated. In practice we did not get different results because this case was marginal. However, we plan to analyze these dependencies and improve our algorithm.

(v) The two systems are Smalltalk systems. Since Smalltalk (as well as other languages such as Objective-C, C#, Ruby, Python) supports class extensions (a method can be packaged in a different package than its class), packages tend to rely less on inheritance but class extensions to extend existing behavior. This produces better layered applications. We should study the influence of such criteria on our algorithm. We believe that since we only take into account dependencies class extensions are just one dependencies and since we took case studies exhibiting a realistic numbers of cycles and dependencies, this thread should not modify our results.

7.8 Related Work

Research has been done on the problem of package cycle identification and removal. We present these approaches in this section.

7.8.1 Heuristics

Some research work has been done on package dependencies and package layer computation. PASTA [Hautus 2002] is a tool for analyzing the dependency graph of Java packages. It focuses on detecting layers in the graph and consequently provides two heuristics to deal with cycles. The first heuristic is to consider packages in the same strongly connected component as a single package. The other heuristic selectively ignores some *undesirable* dependencies until no more cycles are detected. Thus, PASTA reports the *undesirable* dependencies which should be removed to break cycles. The *undesirable* dependencies are selected by computing their weights and selecting the minimal ones. Our approach takes one more parameter into account: we introduce *shared dependencies* and use it as a heuristic to remove package cycles.

In graph theory, a feedback arc set is a collection of edges which should be removed to obtain an acyclic graph. The *minimum* feedback arc set is the minimal collection of edges to remove to obtain an acyclic graph. This theoretical approach cannot be used for three particular reasons: (i) It is a NP-complete problem (optimized by Kann [Kann 1992] to become APX-hard). Some approaches proposes heuristic to compute the Feedback Arc Set Problem in reasonable time [Eades 1993]; (ii) It does not take into account the semantic of the software structure. Optimizing a graph is not equivalent to a possible solution at the software level; (iii) The goal of breaking cycles in software applications is not to break a minimal set of links, but the more pertinent ones.

JooJ [Melton 2007b] is an eclipse plugin (not released) to detect and remove as early as possible cyclic dependencies between classes. The principle of JooJ is to highlight statements creating cyclic dependencies directly in the code editor. It computes the strongly connected components to detect cycles among classes. It also computes an approximation of the minimal set of edges to remove to make the dependency graph totally acyclic based on feedback arc set. However, no study is made to validate this approach for cycle removal. It is possible that the selected dependencies are not to be removed because they are valid in the domain of the program. In another study, Melton et al. [Melton 2007a] propose an empirical study of cycles among classes. They employ minimum feedback arc set to resolve cycles present in the software system. They particularly indicate that it is crucial to take into account the semantic of the software architecture to not break dependencies that should not be broken. For this purpose, the authors propose to add constraints to minimum feedback arc set such as not removing inheritance relationship while breaking cycles. We also consider user input an essential feature to remove cycles. In our work, we include user validation to take into account the semantics of the program.

Mudpie [Vainsencher 2004] is a tool to help the maintenance of software system by bringing out SCCs and focusing on the dependencies in SCC. However, no strategy is presented to break the cycles present and the tool relies on the developer's intuition to

remove cycles.

Multiple works [Abdeen 2009] exist to decompose a system by using genetic heuristics. Lutz [Lutz 2001] proposes a hierarchical decomposition of a software system. It uses a genetic algorithm to find the best way to group components of the system into coarse-grained components. Our work is not in this domain. Our goal is to discover dependencies which break the system, in particular, the layered organization of the system.

7.8.2 Software Clustering

Software clustering is another domain in relation to our work. The goal of clustering is to order elements into modules based on some criteria defined by the engineer [Andriyevska 2005, Praditwong 2010]. This kind of approaches can be useful to manipulate fine-grained information. In our work we manipulate packages and we consider that a package has a meaning for engineers, as it should not be broken automatically.

Bunch [Mitchell 2006] is a tool which remodularizes automatically a software. It proposes to decompose and to show an architectural-view of the system structure based on classes and function calls. It helps maintainer to understand relations between classes. This tool breaks the package concerns and does not provide the information we need to make a layered organization of a package system. Our work is based on package architecture.

The Kleinberg algorithm [Kleinberg 1999] defines authority and hub values for each class in a system. A high authority means the class is used by a big part of the system, and the hub value means the class uses multiple other classes in the system. Scanniello et al. [Scanniello 2010] propose an approach to build layers of classes based on this algorithm. They identify relations between classes and use the Kleinberg algorithm to group them into layers. They propose a semi-automatic approach which allows the maintainer to manipulate the architecture and add its proper meaning of the system.

7.8.3 Regression and Integration Testing

This domain is in relation to our work because researchers work on finding the better solution to remove cycles between entities. Le Traon et al. [Le Traon 2000] propose a model to break Strongly Connected Components (SCC). The algorithm computes a weight for each vertex in a SCC based on incoming and outgoing dependencies. Then, in each SCC, it selects the vertex with maximal weight and remove incoming dependencies. Briand et al. [Briand 2001] combines this algorithm with Tai and Daniels strategy [Tai 1997] to compute a weight for each dependency. The goal of this kind of algorithm is to minimize stub creation. Le Hanh et al. [Hanh 2001] propose an experimental comparison of four approaches to break SCC for stub minimization. The goal is to find the best candidate that can remove cycles to build an order of integration. The difference with our work is about the goal. The goal of this kind of algorithm is to find vertices to create stub. Our goal is to find edges that should be removed. Consequently, algorithms are different: our work computes a weight on each edge, whereas this kind of algorithm computes a weight on each vertex. A future work is to analyze how we can use these algorithms. An interesting idea is about the differentiation of the kind of dependencies to avoid removing

inheritances [Kung 1996, Tai 1997]. A future work is to use this idea in OZONE to select unwanted dependencies.

7.8.4 Visualization

Several approaches propose to recover software structure, visualize classes and files organizations [Vainsencher 2004]. Only few approaches provide layered organization for packages and in particular take cycles into account. A few approaches help determine information on packages and their relationships, with visualizations and metrics [Ducasse 2007]. In these approaches, it is not easy to understand the place of a package in a system, particularly when large systems are analyzed. Some other approaches propose to recover software structure and visualize the organization of classes and files [Mitchell 2006]. To understand the complexity of large object-oriented software systems and particularly the package structure, there are some visualization tools [Ducasse 2006, Ducasse 2005b, Balzer 2005, Langelier 2005]. Package Blueprint [Ducasse 2007] shows the communications between packages; eDSM (Chapter 5, p.55) and CycleTable (Chapter 6, p.83) highlight the cycle problems in a system. However, these approaches do not identify layers for packages present in a software system.

Dong and Godfrey [Dong 2007a] propose an approach to study dependencies between packages and to give a new meaning to packages with (i) characterization of external properties of components, (ii) usage of resource and (iii) connectors. It helps the maintainers to understand the nature of package dependencies. This kind of tool is useful to understand a global system. It could be used in the view of a dependency to replace eCell. We can also replace eCell by node-link visualization which does not need learning time.

Lungu et al. [Lungu 2006] propose a collection of package patterns to help reengineers to understand large software system. They propose to recover architecture based on package information and an automatic process to recover defined patterns. Then they propose an user interface to interact with the package structure. This approach is useful to understand the behavior of a package in the system. It can provide information about the position of a package in a layered organization. This kind of patterns could be used to add more informations on a package and to propose more information about the breaking of a dependency, for example knowing that a package is autonomous is a valuable information.

7.8.5 Tools

There are other tools like JDepend[9] or Classycles[10] which allow software engineers to see package dependencies and cycles. But these tools do not provide a layered organization of the package structure. JDepend is a tool which computes design metrics on Java class directories to generate a quality view of packages. Classycle is a tool to see cycles between classes and shows package dependencies and cycles based on class information. It uses the same algorithm as JDepend. NDepend[11] is a tool to help engineers to maintain software

[9] http://www.clarkware.com/software/JDepend.html
[10] http://classycle.sourceforge.net/
[11] http://www.ndepend.com

with the help of visualization and metrics. It provides a UI to manage large software maintenance. However, the tool does not support the removal of package cycles.

7.9 Summary

In this chapter we show that building a package layer organization is not trivial in the presence of cycles. Two existing approaches are presented that use different strategies to take cycles into account. One puts cycles outside the layer structure, the other searches to remove edges that would allow minimal changes to a software system without taking into account the semantics of the system. We believe that neither of the two solutions is appropriate. Based on our experience, we propose a strategy to organize a system containing cycles between packages in layers. We consider *layer-breaking dependencies* defined in the strategy and provide a user interface to add manual constraints. As the weight of the dependencies is not enough to break cycles, our approach selects *unwanted* dependencies based on two characteristics: *light* dependencies and *shared* dependencies. These two characteristics have already been proposed visually by EDSM (Chapter 5, p.55) and CYCLETABLE (Chapter 6, p.83).

The study shows that the strategy provides good results but the strategy should be improved to be more flexible. First, we can see that a manual verification is needed to validate the *layer-breaking dependencies* computed by our algorithm. By extension, the user interface and the visualization provided in Section 7.6 (p.114) should be improved to be more usable and to highlight some feature of the system. The idea is that the algorithm should not remove absolutely all cycles, but ask the reengineer to validate the computed dependencies and introduce dependency constraints.

Finally, the strategy is based on *shared dependencies*, it depends on the analysis of the complete system to have all *shared dependencies*. In the case of computing the algorithm on only a part of a system, *shared dependencies* are lower and the algorithm should return more false-positive values. Here again, the algorithm should be more flexible and ask input from the reengineer.

The next chapter introduces ORION. It provides a system for change impact analysis. After having analyzed the system with EDSM and CYCLETABLE, analyzed propositions made by OZONE, reengineers can analyze the impact of changes in the structure of the system before applying real changes.

CHAPTER 8

ORION, Simultaneous Versions for Change Analysis

Contents

The future belongs to those who prepare for it today.
[Malcolm X]

At a Glance

In this chapter, we propose ORION, an interactive prototyping tool for reengineering, to simulate changes and compare their impact on multiple versions of software source code models. Our approach offers an interactive simulation of changes. We devise an infrastructure which optimizes memory usage of multiple versions for large models. We validate our approach by running benchmarks on memory usage and computation time of model queries on large models. They show that the ORION approach scales up well in terms of memory usage, while the current implementation could be optimized to lower its computation time. We also report on two large case studies on which we applied ORION.

Keywords: impact analysis, change simulation, simultaneous version.

8.1 Introduction

While software reengineers would greatly benefit from the possibility to *assess different choices*, in practice they mostly rely on experience or intuition because of the lack of approaches providing comparison between possible variations of a change. Software reengineers do not have the possibility to easily *apply analyses on different version branches of a system and compare them* to pick up the most adequate changes.

Usually reengineering tools use an internal representation of the source code (AST for refactoring engine, simpler source code meta-models for others) [Chikofsky 1990]. Similarly our approach is based on a source code model on which source code changes are applied interactively. We present an approach which allows reengineers (1) to create several futures by performing changes and (2) compare them. Each future is a full-fledged model which can be assessed through usual software metrics, quality models, visualization... Of course versioning systems have supported branching for decades. We propose to be able to navigate and apply changes to possible futures or branches without actually committing them in a code repository. Specifically, we propose an infrastructure which optimizes memory usage of multiple versions for large models, enabling to work interactively on multiple models. Moreover, the concepts supporting our infrastructure are generic enough to blend in many meta-models. Existing tools can be reused on top of such versioned models without adaptation.

In this chapter we raise the problem of the scalability of such multiple futures and branching versions. First, what do reengineers expect during the workflow, what tools do they need and what kind of feedback should tools provide? Second, what is the infrastructure to put in place to support it efficiently? How to support model manipulations (edition, analyses) of large source code models with many small modifications (class changes, method changes)? A naive implementation is to make a copy of the original model for each future version and to modify the copies. However, with this naive approach a lot of memory is wasted by copying unchanged model entities. For example, modifying one package in a system with 100 packages would imply 99 useless copies.

Structure of the Chapter

This chapter presents in Section 8.2 (p.129) our vision for reengineering. Section 8.3 (p.132) details the principles and challenges of the model-based infrastructure supporting our approach and Section 8.4 (p.139) gives code samples of the critical parts. Section 8.5 (p.143) provides benchmarks about the scalability of the model compared to a naive full copy approach as well as brief reports about two large case studies. In Section 8.6 (p.149), we discuss how our vision could be implemented (less efficiently) with revision control systems. Section 8.7 (p.149) presents related work and Section 8.8 (p.153) concludes this chapter.

8.2 ORION Vision for Reengineering Support

In this section we present the vision behind this work and we draw requirements for the implementation of such a vision.

8.2.1 Efficiency in Reengineering

A reengineer has basically four forces driving his work. He should: (1) identify issues, (2) solve issues, (3) avoid degradation of the system, and (4) minimize costs of change [Bohner 1996]. While the first three items are checked externally (bug report, review, tests), the reengineer has larger latitude to assess changes and their cost. Often there are multiple solutions to solve an issue, and assessing the most adequate one is a challenge of its own. The reengineer usually relies on his experience and intuition to select the most promising candidate.

8.2.2 Motivating Scenario

We describe now a scenario dealing with reengineering package dependencies, especially the removal of cycles between packages. From the scenario, we extract general requirements for the ORION approach *i.e.,* the simultaneous analyses and comparison of multiple versions of the system and illustrate them with examples.

A Relevant Scenario. Our experience with identification and removal of cycles in large software systems shows us that one of the key challenges is to eliminate a cycle without creating a new one. Let us take an example from Moose, a platform for software analyses and reverse engineering [Nierstrasz 2005] (See Figure 8.1.a). In the original model, we are interested in three packages, two classes, and the two methods Model::inferNamespaceParents and Model::allNamespaces[1]. The black arrow from inferNamespaceParents indicates a reference to class Namespace. The gray arrow from inferNamespaceParents indicates an invocation of method allNamespaces. They create a dependency from package Moose-Core to respectively package Famix-Core and package Famix-Extensions. The dotted arrows from Famix-Core to Moose-Core and from Famix-Extensions to Famix-Core indicate dependencies of the same kinds seen at package level (coming from classes not shown in the figure). Altogether, the three packages make a strongly connected component. This component can be decomposed into two circuits: Moose-Core depends on Famix-Core and reciprocally, but Moose-Core also depends on Famix-Extensions, which depends on Famix-Core, which comes back to Moose-Core.

Possible Changes. In Figure 8.1.b, inferNamespaceParents, which is directly involved in one cycle, is changed into a class extension in package Famix-Core. As a consequence both previous cycles are broken since there is no dependency coming out of Moose-Core. The reference to Namespace is now internal to Famix-Core. However, the invocation escapes

[1]Notice that *allNamespaces* is defined in a different package than its parent class. This feature called *class extension* (or *partial class*) is especially useful to make packages more modular.

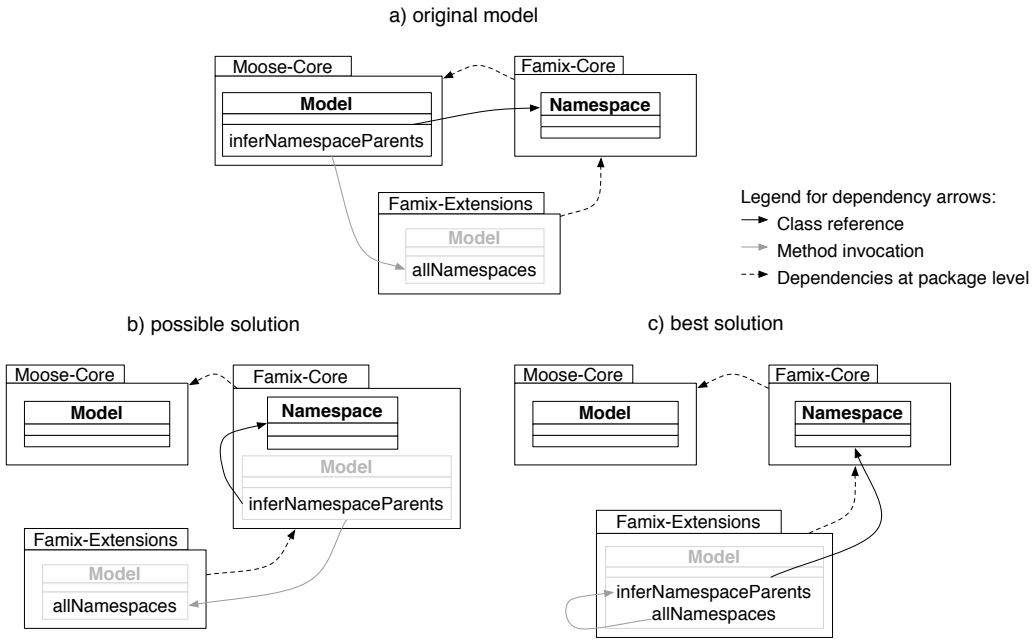


Figure 8.1: a) two dependency cycles between three packages; b) a change that removes the two cycles but creates a new one; c) a change that effectively removes the two cycles.

the package and a new cycle is actually created between Famix-Core and Famix-Extensions. Overall, this solution is possible but not good, because the new cycle is a degradation.

In Figure 8.1.c, inferNamespaceParents is changed into a class extension in package Famix-Extensions. Now the invocation is internal to Famix-Extensions, while the reference escapes the package but "blends" into the existing dependency from Famix-Extensions to Famix-Core. No new dependency is created at the package level, while the two previous cycles are effectively removed. In this case, a *single* cheap change cuts two cycles, which is a very positive outcome.

8.2.3 Requirements

In general, removing cycles is hard because predicting the full impact of a change is difficult, be it positive or negative as illustrated in the above example. From this experience, we see that having the possibility to compare two solutions applied to the same original code model would help reach a decision. Taking such scenario as an illustration, we extract the following requirements for an infrastructure supporting this vision.

- The reengineer needs access to different tools to assess the current situation: system structure (as a diagram or other visualizations), algorithms, metrics, queries to compute relationships between entities of the system. For our example, one needs graph algorithms such as Tarjan [Tarjan 1972] to compute cycles between packages.

This also implies running queries over the entities of the model to build the graph of dependencies between packages. We develop a dedicated visualization using Dependency Structural Matrix to analyze in details cycles between packages (Chapter 5, p.55). Metrics such as the number of cycles (as strongly connected components in the graph) are also useful to provide a quick assessment of system status.

- The reengineer needs change actions to derive new versions of the system. Such actions have to be at the appropriate level of granularity for the task at hand. For example, changing package dependencies can involve many actions at different levels: moving classes around, moving methods between classes, merging/splitting packages.

- The reengineer needs to run the same set of tools on the original model and on derived versions to analyze the new systems and assess whether he reached his goals. He then can decide to stop and select this version, continue working on this version, mark it as a "landmark" and derive from it a new version to work on, or come back to another version and starts in a new direction. For example, an often unforeseen yet common consequence of cycle removal is the creation of a new cycle in another place (see Figure 8.1.b). At this point, the developer has two possibilities: he continues to work on this version to also remove the new cycle; or, he considers this new cycle too costly to fix and comes back to a previous version to work out a different solution.

- The reengineer needs to assess what changed between two versions, to follow the impact of a change on the system and the progression towards a goal. This involves assessments focused on changes: changed entities directly impacted by the actions, but also changed properties of the system, or difference between two measures of a metric. He may eventually design custom tools, such as dedicated visualizations, to look at changes from the point of view of his task. For example, after performing a change, the reengineer should be informed of the destruction and creation of cycles. He can follow his overall progression by looking at the total number of cycles for each version.

- Finally, when the reengineer settles on a version and wants to create this version starting from the original model, he needs the sequence of actions to apply, derived from the branch of the selected version.

First, this discussion stresses that the reengineer needs specific tools appropriate for the task at hand. Developing tools is costly, thus being able to *reuse existing tools* is an important asset for any reengineering infrastructure. On the other hand, these tools have to work on a generic reengineering infrastructure. In the following subsection, we present how the ORION approach embodies the above requirements. This presentation shows how specific requirements for the task of cycle removal and generic requirements for reengineering interplay in the front-end user interface. The remainder of the chapter is dedicated to a more in-depth review of how ORION manages the generic requirements.

8.3 ORION Design and Dynamics

This section presents the design and challenges for the realization of ORION requirements. We first present the meta-model of ORION as well as its efficient implementation using shared entities: to save memory space and creation time, entities which do not change are shared between different versions of the model. We explain the creation of a new version and the dynamics of actions on a version. Finally, we detail how queries are resolved in the context of multiple versions and shared entities. In particular, we show that a model should be *navigated from one specific* version even if a query may navigate to shared elements that are reused from older versions. Note that the infrastructure we present is not specific to source code meta-model but can be applied to any meta-model.

8.3.1 ORION Meta-Model: Core and FAMIX Integration

ORION Core Meta-Model. The ORION approach is built around the three main elements shown in bold in Figure 8.2 (OrionModel, OrionEntity, and OrionAction). One instance of OrionModel stands for one version of the system. Each version points to its parentVersion, building a tree-like history of the system. The tree root represents the original model and contains all entities from the current source code. Hence, a version derives from a single parent but can have multiple children as concurrent versions are explored. Each OrionModel owns its OrionEntities. The system also contains a single OrionContext, which points to the current version on which the reengineer is working. Thus, navigating between versions is as easy as changing the OrionContext to point to the wanted version.

An OrionEntity represents a structural entity or a reified association between entities in the model. ORION entities represent the level of abstraction upon which reengineering actions are performed. For the task reported in this chapter, we support four kinds of entities: OrionClass, OrionMethod, OrionPackage, OrionNamespace, and four kinds of association: OrionReference (from one class to another), OrionInvocation (of method), OrionInheritance, and OrionAccess (from a method to a variable).

Each OrionEntity has an orionID which is unique across all versions. A newly created entity receives a new, unique orionID. A changed entity keeps the same orionID as its ancestor. This identifier allows ORION to keep track of changed entities between different versions of the system.

OrionAction is the superclass for different kinds of actions. We distinguish between AtomicActions such as "remove a method", "move a class", or "create a package", and CompositeActions such as "merge two packages" or "split a class", built using a composite pattern. An instance of OrionAction runs to modify the current version but also stores information about the execution of an action (current version, target entity, specific parameters of the action) to keep track of changes. When executed, an action runs on the current model in OrionContext and modifies the entities in place.

FAMIX Meta-Model Integration. ORION is an extension of FAMIX, a family of meta-models which are customized for various aspects of code representation (static, dynamic, history). FAMIX-core describes the static structure of software systems, particularly object-

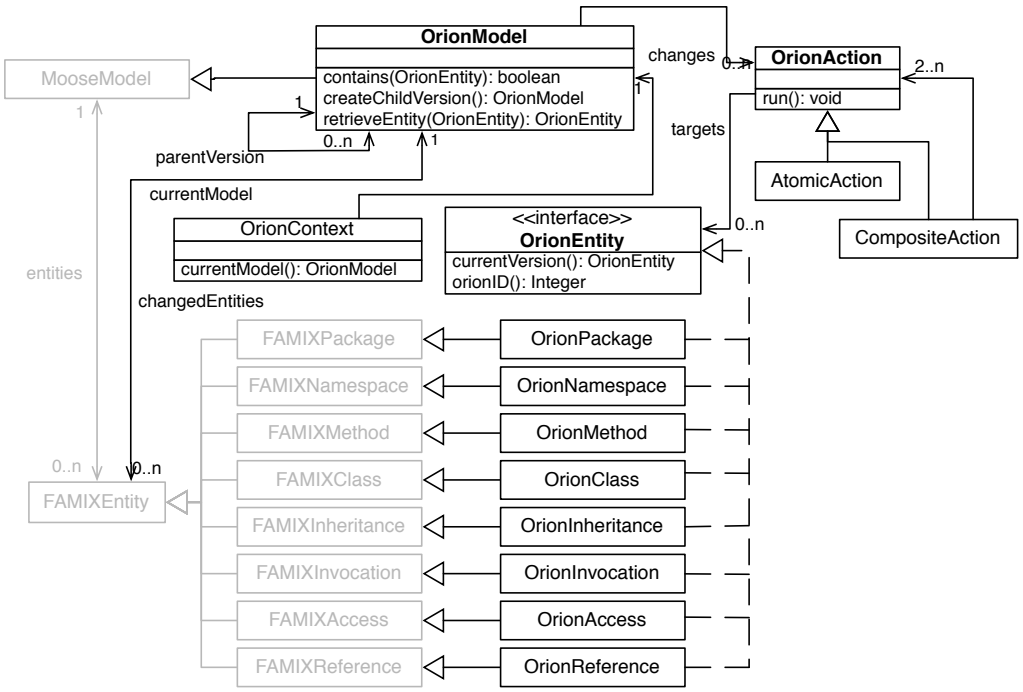


Figure 8.2: ORION meta-model.

oriented software systems[2]. Extending FAMIX is a major asset of ORION as it allows us to reuse tools and analyses developed on top of FAMIX [Ducasse 2009b]. Especially, it fulfills requirementwhich states that *tools should run indifferently on the derived versions as on the original model*.

In practice, the original model is created as a regular FAMIX model before being imported into ORION. During the import, FAMIX entities upon which actions can be applied are converted to their corresponding ORION entities. Other FAMIX entities which do not support OrionAction are directly included in the ORION model (for example, FamixVariable and FamixParameter). An ORION model deals seamlessly with both FAMIX entities and ORION entities.

8.3.2 The Need for Sharing Entities Between Versions

Models for reengineering are typically large because they reify a lot of information to perform meaningful analyses. For example, one system under study with ORION is Pharo[3], an open-source Smalltalk platform comprising 1800 classes in 150 packages. Its FAMIX representation has more than $800,000$ entities, because it includes entities for variables, accesses, invocations... It becomes a major concern for an approach such as ORION,

[2] see [Demeyer 2001] and http://www.moosetechnology.org/docs/famix

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

because we need several such models in memory to enable an interactive experience for the reengineer. In the following, we briefly review some of the strategies we analyzed in [Laval 2009b] and explain the dynamics of our model with shared entities.

The most straightforward strategy is the full copy, where a version is created by copying all entities from its parent version. Then two versions are two independent models in memory and tools run as is on each model. However, this approach has a prohibitive cost both in term of memory space and creation time. In early experiments analyzing Pharo, each model took 350Mb in memory and, more annoyingly, the copy took more than one hour to allocate and create the $800,000$ instances for each version. This was useless in the context of our approach.

Another common strategy is the partial copy approach. The principle is to copy only the entity changed as well as the entities connected to it, so that they still make a consistent graph in the current version. Unfortunately, this view does not hold in the FAMIX meta-model where all entities are transitively connected together through their relationships (each class representation points to its methods while each method representation points to its parent class). Thus, copying an entity and its linked partners comes back to copying the full model.

Our solution is a variation of the partial copy approach, but requires an adaptation of the access of entities through links. The trade-off is between the memory cost of large models and the time cost of running queries on such models. Only entities which are directly changed are copied (then modified) in our approach. Other entities are left unchanged in their version, making the copy "sparse" and efficient. Changed and unchanged entities are reachable from the current version through a reference table, which is copied from the parent when creating the new version and modified by actions. Thus entities are effectively shared across different versions. However, dynamics are more complex than a simple Copy-on-Write standard approach as explained below.

Figure 8.3 illustrates how an ORION system manages the three kinds of change for an entity: creation, change, and deletion. Figure 8.3.a shows the original model with four OrionEntities with orionId **1** to **4**; the light gray area on the left represents the reference table of the ORION model, which holds pointers to each OrionEntity. Figure 8.3.b shows a child version where entity **5** has been created; the reference table holds a new pointer to entity **5** at the end (gray rectangle). Entities **1** through **4** are still accessible from the reference table. Figure 8.3.c shows another version where entity **4** has changed. Consequently, a new entity **4** appears in the version and the reference in the table is replaced with the new pointer. Figure 8.3.d shows a version where entity **3** has been deleted. Only the pointer to entity **3** is really removed from the reference table, making the entity unreachable and effectively deleted from this version.

Table 8.1 summarizes the pros and cons of the above approach for our constraints. Numbers in parenthesis refer to the Pharo case study, which is our largest case to date with $800,000$ entities per model.

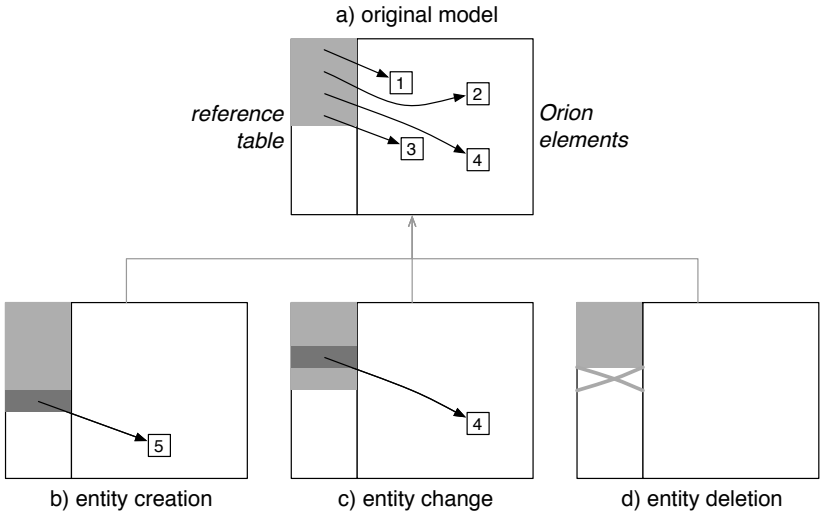


Figure 8.3: Illustration of ORION dynamics through different changes (creation, change, deletion).

Approach	**Creation cost for new version**	**memory cost for *x* versions**	**access cost to an entity**
Full copy in FAMIX	Copying all entities (70 minutes)	*x* * original size (350 Mo per version)	direct access
Partial copy + table look-up in ORION	Copying the reference table (30 seconds)	original model size + *x* times reference tables + size of each changed entity (350Mo + around 10Mo per version)	table look-up

Table 8.1: Comparison between copy model and shared entity model in the case of Pharo.

8.3.3 Running Queries in the Presence of Shared Entities

Queries are the foundations for tools as they enable navigation between entities of the model. Basic queries represent direct relationships between entities: a class can be queried for its methods, a method can be queried for its outgoing invocations (*i.e.,* method calls within the method), a package for its classes, ... More complex queries made by tools are composed from such queries.

Sharing entities across different models has an important impact on the way queries are run in a version. In particular, starting from a given version, a query may run on shared entities from older versions: results returned by such shared entities must always be interpreted in the context of the starting version, as older entities may link to entities which have changed since. This specific aspect makes our solution more subtle to implement than a simple Copy-on-Write. The challenge of running queries over shared entities is summarized as follows:

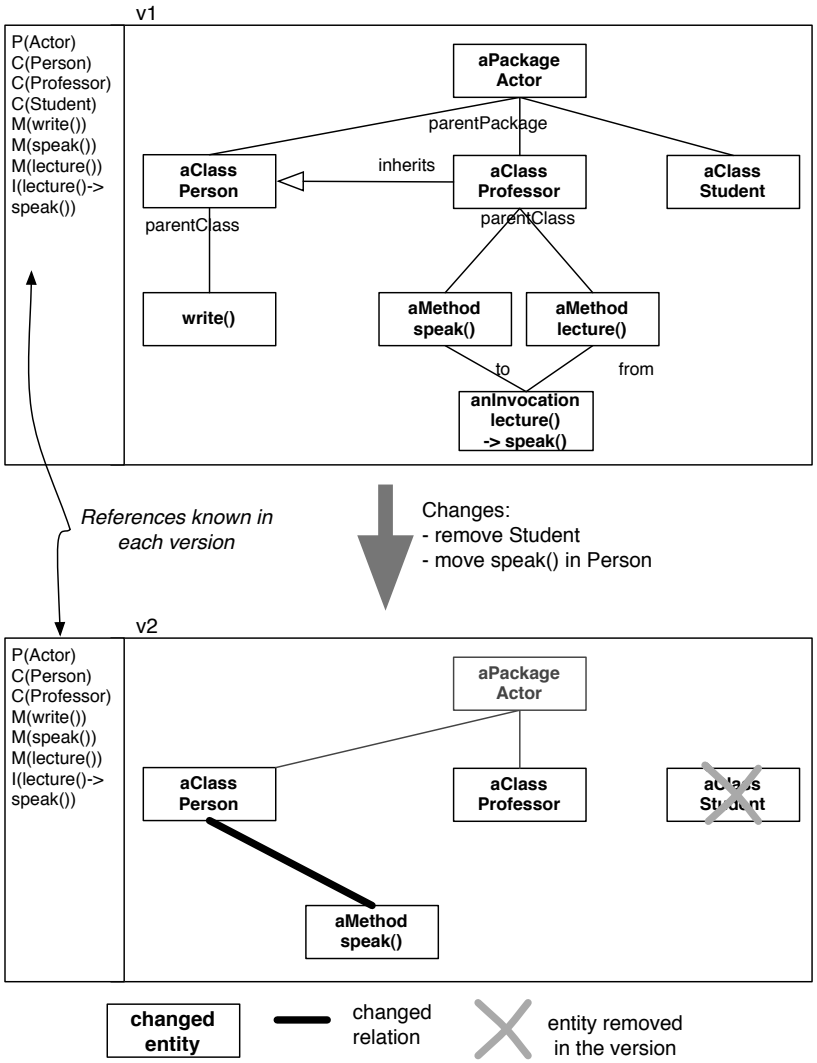


Figure 8.4: Sample model with one derived version: classC is deleted from Actor and method mB1() moved from classB to classA.

1. basic queries retrieve entities which may or may not reside in a parent version;
2. then ORION should resolve each retrieved entity to its most recent entity (sharing the same orionId) reachable from the current version.

The challenge is akin to late binding in object-oriented languages. An entity residing in a parent version is always interpreted in the context of the current version where the query is run, just as a method invocation is always resolved against the dynamic class of this, even when the call comes from a method in a superclass. In our solution, there is no look-up through parent versions to resolve the most recent entity, but a direct access through the reference table of the current version.

Example. Let us illustrate this challenge with the following case. In Figure 8.4, two changes are applied on the original model v1 (top diagram): a class deletion (class Student is removed in version v2) and a method move between classes (method speak() moves from Professor to Person).

The deletion action directly impacts the parent package Actor. In the new version v2, class Student is removed and package Actor should be updated so that it does not reference Student. First, v2.Actor is created as a new OrionEntity (with the same orionID as v1.Actor) as it only knows about Person and Professor; second, Student is not in the reference table of v2 (left sidebar) so it is unreachable.

The second change involves three OrionEntities for which new versions are created to mirror changes: v2.Professor does not contain method speak() anymore while v2.Person now contains it, and v2.speak() itself now refers to v2.Person as its parent class. Notice that the invocation lecture()→speak() is not touched by this change as it is still considered as an invocation on method speak(). Methods lecture() and write() are not updated in v2 because they are not directly impacted by the changes from v1 to v2.

Notice how we use the dotted notation version.element to refer unambiguously to an OrionEntity residing in a version. For example in Figure 8.4, v1.Person refers to Person in the original model. v2.Person is a new element which shares the same orionID. v1.write() and v2.write() represent the method write() in their respective version, but the entity is actually shared.

Queries. The following queries illustrate, from basic to more challenging cases, how navigation across shared entities is resolved in ORION. A query takes two parameters: a target entity and the current version as a context (v1 or v2). The general algorithm for processing basic queries takes two steps. First, the query is actually run against the target entity and returns entities which possibly originate from different versions. Second, for such entities, orionIDs are matched against the reference table of the current version to retrieve the latest entities corresponding to each orionID.

In Figure 8.5, several queries are represented, from basic to more complex, which we explain below. qV and qVI especially illustrate the challenge of shared entities.

qI – v1.Professor.getAllMethods() → {v1.speak(), v1.lecture()}. This query returns all methods of the class Professor in the context v1.

qII – v2.Professor.getAllMethods() → {v1.lecture()}. lecture() exists in v2 but resides in v1, because this entity has not been modified. This is the standard case of shared entities between versions. Since lecture() is in the reference table of v2, it is reachable. Since a specific v2.lecture() does not exist, the reference actually points to the most recent entity with respect to v2, which is v1.lecture().

qIII – v1.speak().getParentClass() → v1.Professor. This query runs in the context of v1, giving the original view of the model.

qIV – v2.speak().getParentClass() → v2.Person. This query runs in the context of v2. It returns a different result than qIII respecting changes applied in v2.

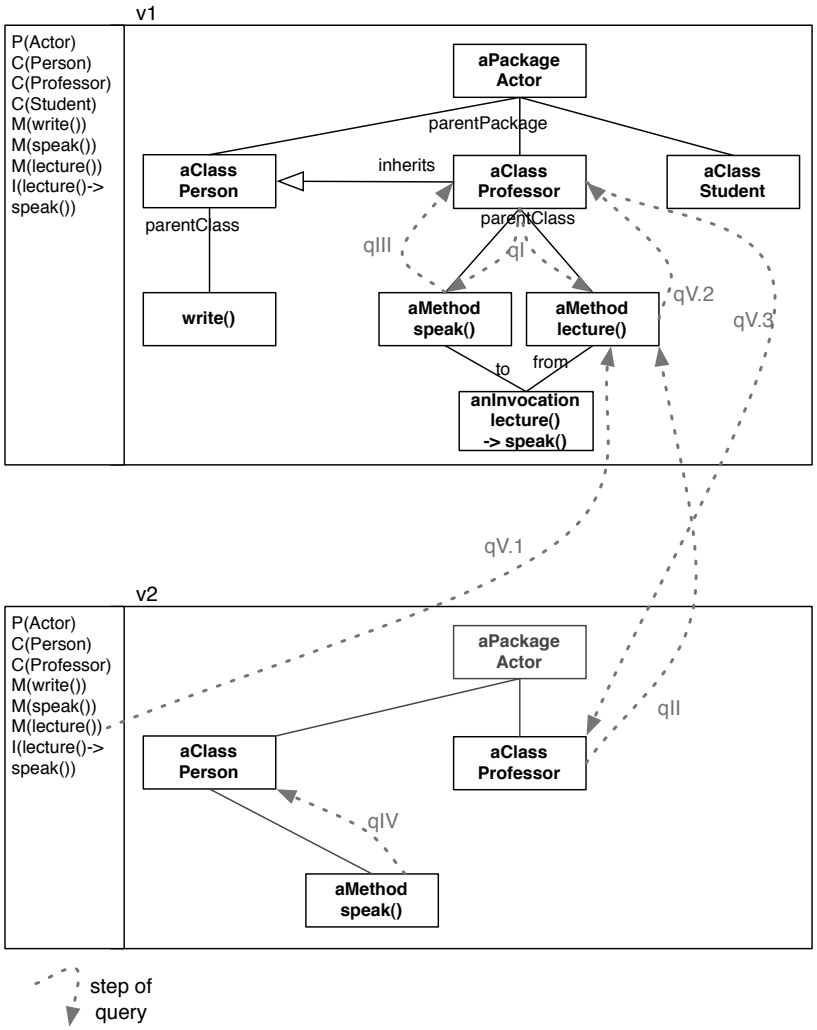


Figure 8.5: Sample model of Figure 8.4 representing query mechanism.

qV – v2.lecture().getParentClass() → v2.Professor. As noted for qII, v2.lecture() is actually v1.lecture() and the query is run against the later (represented by step qV.1 in Figure 8.5). Then the query resolves in two steps: first message parentClass sent on v1.lecture() retrieves v1.Professor; second, since the query runs in the context of v2, ORION retrieves the correct v2.Professor from the reference table of v2 using the orionID as the common denominator.

qVI – v2.lecture().getParentClass().getAllMethods() → {v2.lecture()}. This is a composed query. Here we get the same scenario than with qV (v2.Professor) but in addition we query all its methods as in qII, which returns its sole method v1.lecture() in the version v2.

Queries qV and qVI show the subtlety of running queries on shared entities. First, query qV is launched on v2.lecture() but actually runs on v1.lecture() - since it is shared between

v1 and v2. Second, when a query is run in a version, it must follow changes related to this version: in query qVI, v2.lecture() is v1.lecture(), but querying element Professor selects entity v2.Professor, not v1.Professor, so that the correct set of methods is retrieved. The last query stresses that even if an entity from a parent version is returned, traversing this entity implies a resolution against the current version to retrieve changed entities. In presence of shared entities between versions, a composed query may reach entities of a parent version, not residing in the current version, yet it should always return entities as seen from the current version. A version acts as a view on a graph and from such a view the graph and its navigation should be consistent.

8.4 Implementation

8.4.1 Core Implementation

We give some details of the implementation which illustrate the dynamics of ORION. Our goal is to give enough information so that the approach can be reproduced in other meta-modeling environments such as the ones supporting EMF. We give code samples in pseudo-code for the following cases: creation of a new version, action execution, basic query. It shows ORION internals, which create changed entities and resolve an entity in the current version.

Version Creation. A new version is created from a parent version by copying the full list of references (reference table) from the parent model (only references are copied, not entities). The version model also stores a reference to its parent model and the parent adds the version as a new child:

```
OrionModel::createChildVersion(): OrionModel {
    OrionModel newVersion = new OrionModel();
    childrenVersions.add(newVersion);
    newVersion.setParentVersion(this);
    for(OrionEntity entity: entities()){
        newVersion.addEntity(entity); }
    return newVersion; }
```

Action Execution (Move Method). An instance of OrionAction runs to modify the current version but also stores information about the execution (current version, target entity, specific parameters of the action) to keep track of changes. To move a method from its current class to another class, ActionMoveMethod needs three parameters: the current version as orionContext, the method as targetEntity, and the target class as targetClass. These parameters are stored in instance variables of the action, set at initialization.

The method run of the action retrieves the entities concerned by the change from the orionContext then directly update these entities. The orionContext (the current version) takes

care of copying entities from the parent version and updating its reference table. ActionMoveMethod touches three entities: the method and the two classes. Its method run updates the links between those three entities to apply the change.

```
ActionMoveMethod::run(): void {
    OrionMethod method = orionContext().retrieveEntity( targetEntity() );
    OrionClass oldClass = orionContext().retrieveEntity( method().parentClass() );
    OrionClass newClass = orionContext().retrieveEntity( targetClass() );

    oldClass.methods().remove(method);
    newClass.methods().add(method);
    method.setParentClass(newClass); }
```

The method retrieveEntity from OrionModel first checks whether the entity resides in the model. In this case it means that the entity has already been changed and can be directly modified. Otherwise, it makes a shallowCopy of the entity since it comes from a parent version. shallowCopy copies only references to other entities as well as the orionId.

```
OrionModel::retrieveEntity(OrionEntity anEntity): OrionEntity {
    if( contains(anEntity) ) {
        return anEntity;
    } else {
        OrionEntity changedEntity = anEntity.shallowCopy();
        changedEntity.setModel(this);
        return entities().add(changedEntity);
    }}

OrionModel::contains(OrionEntity anEntity): boolean {
    return this == anEntity.model(); }
```

Query Execution. The main concern of queries in ORION is that they always return entities as seen through the current version. Basic queries, which directly access entities in a model, needs to be adapted in ORION to resolve direct access in the context of the current version.

```
OrionMethod::parentClass(): OrionClass {
    return parentClass.currentVersion(); }
```

The naive implementation of currentVersion below looks for the entity with the same orionId in the current model. However, it involves the traversal of the reference table (entities()) each time an entity needs to be resolved. Care is needed to optimize this method as well as the traversal. A straightforward optimization is to first test whether the entity belongs to the current version using OrionModel::contains (see above), otherwise to launch the traversal. A more general optimization is to use an efficient data structure such as a search tree to implement the reference table.

```
OrionEntity::currentVersion(): OrionEntity {
    for(OrionEntity entity: OrionContext.currentModel().entities()){
        if( entity.orionId == orionId ) {
            return entity; } }
    return null; // should never happen
  }
```

Complex queries are built on basic queries to compute more information. They always get the most recent entities from basic queries and thus do not require adaptation in ORION. This is especially interesting as most analyses are built with complex queries, enabling reuse of existing tools, while basic queries (which require adaptation) form a limited set fixed by the meta-model.

8.4.2 Experimental ORION Browser: Removal of Cycles Between Packages

We designed and implemented ORION, an infrastructure for reengineering, as a realization of the requirements of Section 8.2 (p.129) and proof of concept of our vision. We developed an experimental version browser dedicated to the analysis of cycles between packages. It serves as the central place for running version and cycle analyses (shown in Figure 8.6). This section describes the core functions of the browser with respect to our vision, illustrated in the context of cycle removal. We do not claim that this browser is the sole solution to manage simultaneous versions. It is an illustration of our vision and of possibilities offered by ORION.

We distinguish two parts in the browser layout: top row for navigation in the system, bottom row for task analyses and change assessment. The reengineer interacts with the browser by selecting elements in the different panels and opening a contextual menu.

The navigation row at the top is built from two main panels: the left-most panel shows the tree of model versions. The second panel is a sliding navigation browser of the selected version, embedded in the ORION browser (middle and right panels in Figure 8.6). It first displays the list of entities by group (all packages, all classes...). It is possible to browse entities by selecting a group. Then by selecting an entity, a new panel opens on the right of the current panel with all linked entities (in Figure 8.6, selected method in middle panel displays groups of linked entities in right panel (accesses, invocations...), which can be browsed in turn). In the first panel (left), one can create a new child version from the selected version, or delete an existing one (from a contextual menu). Another action accessible on the selected version is to display the sequence of actions leading from the original model to the version. Each entity in the second panel (middle) defines the list of reengineering actions enabled on itself, like moving a method to a class, or moving a class to a package. A dedicated contextual menu is accessible, listing possible actions (see the pop-up menu in Figure 8.6). Applying an action produces a change which is recorded in the currently selected version. By default, the reengineer can browse and filter the full set of entities in the model. But the reengineer can also switch to changed entities in the second panel to only show entities which have already changed in the selected version.

The bottom row is for analyses and assessment. It contains one large visualization panel

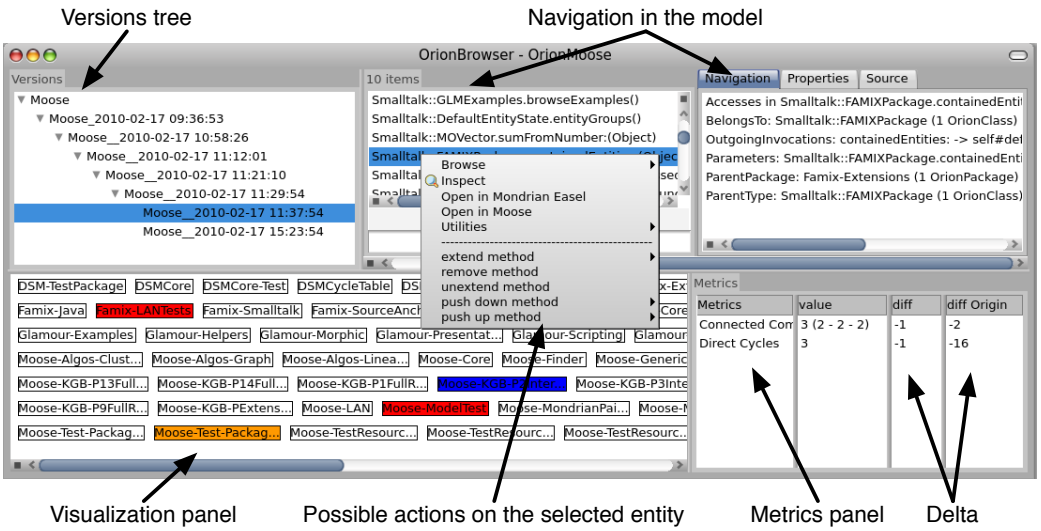


Figure 8.6: An ORION browser dedicated to cycle removal.

and another panel for displaying metric measures. This part of the browser is customized with specific visualizations and metrics for the current task. It may not necessarily display all information relevant for the task at hand, but rather be a starting point to launch more complex and intensive analyses, depending on the assessment of the reengineer for the current situation .

Visualization. Figure 8.6 shows a very simple visualization in the bottom left panel dedicated to highlight strongly connected components (SCC) involving packages. This visualization has been chosen because it is both space-savvy (not requiring a complex graph layout nor a large matrix) and time efficient (using Tarjan algorithm for detecting SCCs in linear time [Tarjan 1972]). Each boxed label represents one package. Colorized boxes indicate packages which belong to the SCC involving all packages of the same color. Hence, the reengineer gets a fast overview of cyclic dependencies in the system and can focus on each problematic subset (that is, each SCC) separately. From the visualization, he can select a single SCC and launch from a contextual menu a more sophisticated visualization such as eDSM (Chapter 5, p.55) to perform a detailed analysis and devise the plan of actions (not shown).

Change Impact Metrics. The bottom right panel displays properties of the system dedicated to the task at hand. Such properties are chosen to assess the current version with respect to the global goal and to follow the progression from the original model. In line with the visualization, we only use simple metrics in the browser. More sophisticated metrics are only useful in specific analyses. In this case, following the number and size of strongly connected components is a good indicator of progression toward the objective (no cycle

implies no SCC). We also include the number of direct cycles, which are a primary target for reengineering. Figure 8.6 shows these two metrics with their values and two change indicators: difference with the parent version and difference with the original version. More specifically, the value of "Connected Components" is *3 (2 - 2 - 2)*, which indicates three strongly connected components, each composed of only two packages. This is of good omen for the next versions. The last column of "Direct Cycles" shows -16, which means that 16 direct cycles have been removed in this version, compared to the original model.

Again this browser is just an illustration of the ORION infrastructure.

8.5 Benchmarks and Case Studies

In this section we run three benchmarks showing that ORION scales up well in terms of memory usage without slowing too much the time necessary to run queries. We compare our approach with the full model copy, which defines the baseline for computation time (no overhead at all) but does not scale up well in memory. The first benchmark shows that ORION saves a lot of memory compared to the full copy approach. The second benchmark shows that the time overhead induced by ORION on queries is acceptable for an interactive experience. The third benchmark shows that the creation time of a new version is insignificant in ORION, while it is fairly slow for the full copy approach, making it impractical in an interactive scenario. We also report on two case studies undertaken on large projects, with insights on the initial changes performed with ORION.

8.5.1 Test Data

We ran our benchmarks against a model of Moose imported in FAMIX [Nierstrasz 2005]. We will call this model *Famix-Moose* from now on. Famix-Moose is a typically large model with more than $150,000$ entities, including 721 classes in 69 packages, $8,574$ methods, $65,378$ method invocations, and other entities.

8.5.2 Memory Benchmark

Goal. This benchmark shows the difference in memory usage between ORION and the regular Moose. We devise two settings to assess this benchmark: one with a few changes per version (low impact setting), and one with many changes per version (high impact setting). The goal of the two settings is to check how the memory usage of ORION is impacted by small and large changes.

Experimental Settings.

Regular Moose. In this experiment, we first create the Famix-Moose model, and we copy it 20 times. Between each copy, we measure the memory used by the different models.

ORION with Low Impact Setting (ORION Low). In this experiment, we first create the Famix-Moose model. Using ORION, we create 20 successive versions (resulting in 21 models in memory in total). For each version, we do the following operation 5 times: We randomly select two methods and add an invocation between them. This modification impacts only the two concerned methods, so at most 10 entities are changed in each version (provided each method is selected once per version). Between each version creation, we measure the memory used by the different models.

ORION with High Impact Setting (ORION High). In this experiment, we first create the Famix-Moose model. Using ORION, we create 20 successive versions. For each version, we do the following operation 10 times We randomly select a method and delete it. The deletion of a method has a high impact because it changes multiple entities: linked invocation entities are deleted, methods invoking or invoked are changed, as well as its parent class and parent package. For instance in the run of this experiment, methods such as **=** or **do:** have been removed, forcing the copy of respectively $2,917$ and 813 elements. Between each version creation, we measure the memory used by the different models.

Results. Figure 8.7 shows benchmark results. The first point represents memory usage of the infrastructure and of the original model, which takes up to 100 mega-bytes. It is clear that copying the full Moose model induces a huge memory usage. For instance, 10 such models require almost 600 mega-bytes of memory, which is a lot even for a recent computer. On the other hand, ORION behaves very well from this point of view. To store the 20 new versions with ORION, only 220 mega-bytes are sufficient for the low setting, and 230 mega-bytes for the high setting, which is a huge improvement. The difference between low and high settings is due to the change of many entities in high setting. It makes versions of multiple entities stored in the reference table. We can also see that ORION is robust even when the changes are complex.

8.5.3 Query Time Benchmark

Goal. This benchmark shows the difference in query running time between ORION and the regular Moose. Using a full copy of a Moose model actually boils down to using the regular Moose system in the day-to-day usage, which defines the baseline for timing queries. We devise two settings to assess this benchmark: one with a few changes per version (low impact setting), and one with lots of changes per version (high impact setting). The goal of the two settings is to check how query performance of ORION is impacted by small and large changes.

Experimental Settings. We perform 4 queries:

- invokedMethods, on each class of the model. This query returns all methods invoked by the methods of a class. It is executed for all classes. We chose this query because it runs over all methods, browsing both changed and unchanged entities. This query

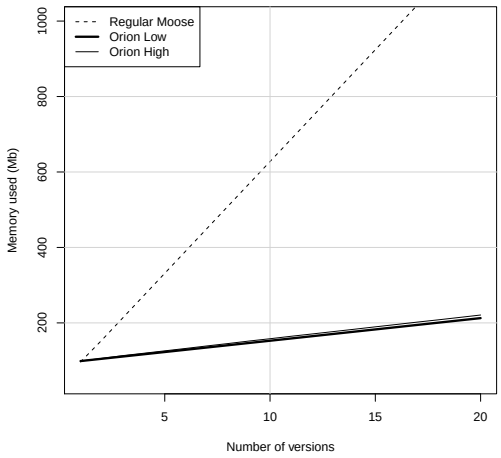


Figure 8.7: Memory usage in ORION vs full copy. The x-axis shows the number of versions and the y-axis shows memory usage (in mega-bytes).

is simple but retrieves a large result, while most analyses are performed on subsets with complex queries.

- allMethods, on each package of the model. This query returns all methods contained in the classes of a package, it is executed for all packages.
- allSubclasses on each class. This query is simpler than the two preceding, but it represents a good indicator for usual queries. It returns all subclasses of a class, it is executed for all classes.
- superclass on each class. This query returns the superclass of a class, it is executed for all classes.

We perform the four queries on each class (or package, depending of the scope) of the model. We run these queries 10 times and take the mean time.

Regular Moose. In this experiment, we run the queries on a standard Famix-Moose model. Since copying the model does not affect the time spent in queries, we run the experiment on a single version.

Low Impact Setting (ORION Low). In this experiment, we take the same setting as *Low impact setting* in Section 8.5.2 (p.143). For each version, we measure the mean time spent to run one query.

High Impact Setting (ORION High). In this experiment, we take the same setting as *High impact setting* in Section 8.5.2 (p.143). For each version, we measure the mean time spent to run one query.

Results. Figure 8.8 shows the result of the query invokedMethods, which is the worst result of the four queries. It shows also that results are constant and up to three times slower than regular Moose model. Table 8.2 shows benchmark result averages. ORION induces an overhead on the time spent in queries which is acceptable in view of query time. This overhead does not depend on the number of versions nor on changes but on the structure used for the reference table. Nevertheless, the time overhead is not so important (see column Factor in Table 8.2) as to disturb the interactive experience. The High impact setting provides approximately the same results as Low impact setting.

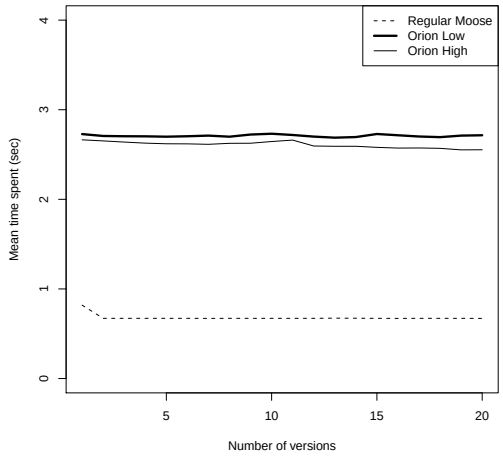


Figure 8.8: Comparison of time spent for the query invokedMethods (worst case) in ORION vs Regular Moose. The x-axis shows the number of versions and the y-axis shows the time spent (in seconds).

	Regular Moose	ORION Low	ORION High	Factor
Invoked methods	735.25	2708.4	2653.45	3.64
All methods of each package	9.9	19.05	20.7	2
All subclasses of each class	3.45	6.65	6.6	1.92
Superclass of each class	4.55	5.4	5.45	1.20

Table 8.2: Average of query time (in milliseconds) .

8.5.4 Creation Time Benchmark

Goal. This benchmark shows the difference in creation time for a new version between ORION and the regular Moose. Creation of a new version should not hamper the interactive experience in the infrastructure, supporting the workflow of the reengineer. With the regular Moose, the creation of a new version is a deep copy of the model. With ORION, it is only a copy of the reference table.

Experimental Design.

Regular Moose. In this experiment, we first create the Famix-Moose model, and we copy it 10 times. We measure the time spent in copying each version and then compute the mean time of copy for version creation.

ORION. In this experiment, we first create the Famix-Moose model. We perform 10 successive version creations (each version being the parent of the next one). We measure the time spent in creating each version. Finally, we compute the mean time of version creation. This experiment does not need change execution because ORION just copies the reference table during creation.

Results.

- *Regular Moose:* $67,45$ seconds
- ORION*:* $3,15$ seconds

We can clearly see that the copy time of a Moose model (more than one minute) is impractical for an interactive tool. On the other hand, the version creation time required by ORION (around three seconds) is acceptable and should not hamper the user experience.

8.5.5 Threats to Validity

The aim of benchmarks is to compare our solution with a naive duplication of models. Two threats to validity are highlighted: one external validity, one construct validity. External validity claims that the generality of the results is justified [Easterbrook 2008, Perry 2000]. The threat is the meta-model used by ORION. Construct validity claims that the construction and the measures are correct. The threat in this part is the randomized changes selected for benchmarks.

Meta-Model. The meta-model provides a reification of dependencies between elements. So, a model based on Famix has a lot of elements due to this reification. A meta-model without this kind of behavior is smaller and could have different results. However, there would still be a memory gain because sharing entities is better than copying all elements in the model.

Randomized Changes A change should impact more or less elements in the model, due to their relations with other elements in the model. For example, the deletion of the method Collection»at: impacts a lot of entities in the model. Some other methods can be deleted with no impact because they are not used (*i.e.,* dead code). We paid attention that in each benchmark, methods with a lot of relationships are removed such as add:.

8.5.6 Case Studies

We now report on two case studies performed on large projects with ORION: Moose and Pharo. Pharo is an open-source Smalltalk environment comprising 1800 classes in 150 packages. For this case study, we ran a smaller model containing $685,000$ entities (only packages in cycles with their embedded entities) instead of $800,000$.

The goal of the case studies was the removal of cycles between packages as shown in Section 8.2 (p.129). To customize the ORION browser for this task, we developed the dedicated visualization as well as goal metrics tracking cycle evolution (Figure 8.6). It allows us to follow the evolution of changes, and focus on packages in cycles, for which we use the EDSM tool (enhanced Dependency Structural Matrix), an advanced visualization for detailed analysis of cyclic dependencies (Chapter 5, p.55). Each case study involved two experts using ORION. One was a reverse engineering expert who assessed cycles based on the report from the browser and eDSM, while the other was an expert of the system under study who suggested changes based on the previous assessment. Once a change is applied in ORION, visualization and metrics are computed again on the changed version to follow the evolution of the system, and a new assessment can begin. The process can be repeated until the goal is met.

Moose. Initial assessment of Moose showed 17 packages (among 69) in 5 strongly connected components, one involving 7 packages. 19 direct cycles between packages were detected. Seven versions were created and assessed to achieve the final objective, which removed 15 direct cycles. The 4 remaining cycles actually relate to test resources, that is deliberate cycles created to test Moose tools. The proposed plan of action touches 52 entities in the model with 22 actions. All were basic actions (3 class moves, 1 reference removal, 1 method delete, 17 method extensions). Among those 22 actions, 13 were readily integrated into Moose source code. The remaining 9 changes, participating in the same single direct cycle, are pending, as the involved concern might be completely refactored.

Pharo. The complexity of the Pharo case study is much larger as initial assessment showed 68 packages in a single strongly connected component, creating 360 direct cycles. We only report on a single 2-hour session, as the case is still ongoing. Eleven versions were created during this session, impacting 110 entities and removing 36 direct cycles. Twenty nine actions were executed, 3 of which were composite actions (merge packages, remove class). Such actions effectively extracted 5 packages from the original SCC. All actions have been integrated in the development release of Pharo.

8.5.7 Threats to Validity of Case Studies

The case study made on the Moose project is possibly biased because we are maintainers of the project and we therefore know the system. But we noticed that the infrastructure still helped us to detect some unpredictable impact. We think that our own experience in this matter is useful.

The case study on Pharo has one threat to validity: it is not possible to know all characteristics of the system. Consequently, it is difficult to generalize the result. However, this study involved two people: one knew how to manipulate ORION, the other had a deep knowledge of Pharo as a maintainer. Ideally, a single person should be able to work with the tool on his model.

8.6 Discussion: Reengineering Using Revision Control Systems

Revision control systems (like CVS and SVN) have managed version branching for decades. They offer compact ways of storing delta between versions but reengineering environments or IDEs like Eclipse do not take advantage of such incremental storage. For Eclipse, there is only one version of the source code in memory when we perform a given refactoring.

Using code versioning to support our scenario boils down to (1) create one branch in the code repository for each possible solution, (2) effectively apply changes in the code, then (3) run the analysis tools on each branch. The developer would eventually need some tools to compare versions. Note that we are not concerned by textual differences between versions but software quality assessment based on visualization and metrics of the different versions.

Such a process is possible but costly, as one has to check out one working copy, set up the reengineering tools for each version, apply effective changes in the code. In addition it is cumbersome to navigate and compare the versions. It makes it impractical to test numerous small solutions. In practice, developers often cannot support such costs: they give their best guess at what would be the adequate solution, apply it, and rollback if it reveals too problematic.

Overall, this process would have two drawbacks:

- it consumes time and resources, as the developer has to switch between analyses and reengineering environments, work directly with the code, version its code so as to move forward and rollback between changes. Moreover, it breaks the flow of work while one has to deal with these multiple concerns.
- due to these costs, it is impractical to compare multiple alternative solutions as one should produce the code for each solution.

These drawbacks are not present in ORION due to a single environment for analyses and reengineering. Moreover, as ORION works on models, it does not change source code, and make available comparison of multiple alternative solutions. It allows us to import only one time the source code as model and to work on version without changing the original model.

8.7 Related Work

There are three domains in relation with this work that one may want to compare to: versioning mechanism, change management, and change impact analysis. To the best of our

knowledge, there is no approach which supports the navigation and manipulation of multiple versions simultaneously in memory.

8.7.1 Software Configuration Management and Revision Control

Software Configuration Management (SCM) is the discipline of managing the evolution of a software system. It integrates Revision control which is the management of changes. It is the predominant approach to save software evolution. It allows one to manage high-level abstraction evolution.

The majority of revision control systems uses a diff-based approach. They only store changes so they are efficient in memory. In our approach, we need a compromise between memory efficiency and a permanent graph access. So, the domain of revision control does not provide a model which allows us to navigate between multiple versions of a model. In fact, this is not a real goal of the revision control domain.

Smalltalk basic mechanisms to record changes dynamically is called a changeset. A changeset captures a list of elementary changes that can be manipulated, saved to files and reapplied if necessary. However, in Smalltalk change set system, only one single version of a system can be refactored at a time, even if changeset containing several versions can be manipulated. The same happens with Cheops and change-oriented methodology and IDES [Ebraert 2009]. In [Robbes 2008], the author argues that managing changes as first-class entities is better than traditional approaches. The implementation of this approach records fine-grained changes and provides a better comprehension of changes history. This approach is applied on a single version of source code. ORION integrates first-class changes as each action is represented by an object. It allows the developer to have fine-grained information.

In [Buckley 2005], three tools are compared: Refactoring Browser, CVS and eLiza. A refactoring browser transforms source code. It has basic undo mechanism but does not manage versions. So, it is really useful for refactoring source code but it works on a current model of source code. It is not really adapted for the application of various analyses on different versions. CVS (Concurrent Versions System) works on file system and supports parallel changes. However since CVS does not include a domain model of the information contained in the files they manipulate, it is difficult to use a CVS model to perform multiple analyses on various versions. It is possible but limited. The third element compared is eLiza. This system from IBM has been created to provide systems that would adapt to changes in their operational environment. This system provides a sequential versioning system because only one configuration can be active. This system is not adapted to our subject because it is based on an automatic change system in relation with the environment.

Molhado [Nguyen 2005a] is a SCM framework and infrastructure which provides the possibility to build version and SCM services for objects, as main SCM systems provide only versioning for files. As it is flexible, the authors work on several specific SCM built on Molhado: web-based application [Nguyen 2005b], refactoring aware [Dig 2007] to manage changes and merge branches. The main topic of Molhado is to provide a SCM system based on logical abstraction, without the concrete level of files management. This approach is

orthogonal to ORION because it controls changes while ORION simulates changes. There are some similarities between the two approaches and it is probable that ORION could integrate a SCM, in the future.

8.7.2 Change Impact Analysis

Compared to Software Configuration Management (SCM) and Revision Control System, which supports change persistence and comparison, the domain of change impact analysis deals with computing (and often predicting) the effect of changes on a system. Our approach is orthogonal to change impact analysis. Tools performing change impact analysis can be used in the ORION infrastructure to perform change assessment on a version and guide the reengineer when creating new versions and testing new changes. We structure the domain in two parts: change model, which could inspire future work on the ORION meta-model; change assessment, which provides tools and analyses (metrics, visualization, change prediction).

Change Model. Han [Han 1997] considers system components (variable, method, class) that will be impacted by a change. The approach is focussed on how the system reacts to a change. Links between these components are association (S), aggregation (G), inheritance (H), invocation (I). Change impact is computed based on the value of a boolean expression. For example a change is considered as S~H+G. This work has been reused in [Abdi 2006]. The class-based change impact model [Chaumun 2002] is based on the same semantics, with a more general model. It analyses history and identifies classes which are likely to change often. These approaches use the same type of link between elements as ORION. This type of analysis is not included in ORION, our approach provides metrics and visualization based on the direct analysis of the model. In the future, it will be possible to integrate a similar approach, as ORION knows which entities are modified.

A history-based approach is Hismo [Gîrba 2005a], a meta-model for software evolution. This approach is based on the notion of history as a sequence of versions. A version is a snapshot taken at a particular moment. It makes version from the past based on a copy approach: each version is a FAMIX model. It has some similarity with our idea, however, it is a copy-based approach, so it is impractical to perform interactive modifications. Another difference is that our idea is based on analysis of the future, Hismo is a study of the past. In [Gîrba 2005a], the author proposes some metrics to compare which elements have changed. In our approach the goal is to have a notion of impact of a change.

Other models exist as [Abdi 2009] which proposes a technique based on a probabilistic model, a Bayesian network is used to analyze the impact of an entry scenario. ORION is not concerned by this type of model because it provides the real modification of models.

Change Assessment. [Li 1996] and [Lee 1998] propose an algorithm for analyzing change impact with the detection of inheritance, encapsulation, and polymorphism. The algorithm proposes an order of changes based on the repercussion on self, children and clients. This method is the first one applied to an object model. It is also restricted to classes. Some

approaches try to predict changeability, they assess the impact of a change to a code location by looking at previous change impact upon this location. This domain could be used in ORION to better reflect the impact of changes. [Cai 2007] presents a decision-tree-based framework to assess design modularization for changeability. It formalizes design evolution problem as decision problems, model designs and potential changes using augmented constraint networks (ACN). [Antoniol 1999] uses metrics comparison to try to predict the size of evolving object-oriented systems based on the analysis of classes impacted by a change. A change is computed as a number of lines of code added or modified), but they do not provide the possibility to compare some versions and to choose one.

Other approaches propose change impact analysis based on test regression [Ryder 2001]. [Kung 1994] proposes a change impact analysis tool for regression tests. Authors define a classification of changes based on inheritance, association and aggregation. They also define formal algorithms to compute impacted classes and ripple effects. The Chianti tool [Ren 2004] is able to identify tests which run over the changed source code. They can be run in priority to test regression in the system. For each affected test, Chianti reports the set of related changes. ORION cannot do this kind of analysis because it does not work on source code.

8.7.3 Change-based Environment

Some models exist to support changes as a part of development. The Prism Model [Madhavji 1992] proposes a software development environment to support change impact analysis. This work introduces a model of change based on deltas that supports incremental changes. As it is based on deltas, it is not really possible to analyze different models in parallel.

Another work in change management system is Worlds [Warth 2008]. It is a language construct which reifies the notion of program state. The author notes that when a part of a program modifies an object, all elements which reference this object are affected. Worlds has been created to control the scope of side effects. It manages several parallel universes. The limitation of Worlds is that it only captures the in-memory side effects. Compared to ORION, this work is a source-code-based approach, ORION is model based. In the future, it could be possible to use this type of approach to expand ORION and populate changes on source code with mastering side effects.

With the same idea to control side-effect but restricted to source code, ChangeBoxes [Denker 2007] proposes a mechanism to make changes as first-class entities. ChangeBoxes support concurrent views of software artifacts in the same running system. We can manipulate ChangeBoxes to control the scope of a change at runtime. Compared to ORION which is structure oriented, ChangeBoxes are used to integrate changes in a runtime environment.

The Model Driven Engineering domain and particularly Model transformation could have similar ideas with ORION approach. In [Mens 2006], authors propose a taxonomy of Model Transformation for helping developers who want to choose a model transformation approach. In the enumeration of characteristics of a model transformation, there is no information about multiple models management. One more time, in the future, ORION could integrate a link to source code and have a bidirectional transformations mechanism,

as Model transformation provides.

8.8 Summary

In this chapter, we present a novel approach to reengineering through simulation of changes in source code model. In particular, we claim that software maintainers should be able to assess and compare multiple change scenarios. We define user and technical requirements for an infrastructure supporting our vision, ranging from reuse of existing tools to handling simultaneous versions in memory. We implemented the requirements in ORION, our current infrastructure.

The main concern detailed in this chapter is the efficient manipulation of simultaneous versions in memory of large source code models. Copy approach does not scale up in memory for such models. In ORION, only changed entities are copied between versions, unchanged entities being effectively shared. Then, basic queries take care of retrieving a consistent view of entities in the analyzed version. Our benchmarks report the large gain in memory for our approach with an acceptable overhead in query running time. Overall, it allows ORION to scale up and be usable.

This chapter is the last part of the ECOO approach. The next chapter concludes the thesis by resuming ideas proposed in ECOO. Then we open discussions about problems in software remodularization.

CHAPTER 9

Conclusion

Contents

The important thing is not to stop questioning.

[Albert Einstein]

At a Glance

This chapter concludes the dissertation. We summarize the ECOO approach and our proposal. We discuss the global approach by explaining lessons learned and future work. We finish the dissertation by detailing open issues.

9.1 As a Conclusion

Software systems evolving during years are difficult to understand and change. They lost modularity with their evolution. Software remodularization is a problem now and for the future.

We have reviewed various approaches related to software remodularization. We organized them in three parts: (i) software visualization for structure assessment; (ii) remodularization approaches; and (iii) software change impact analysis. They focus on understanding global structure and automatic or semi-automatic approaches to solve problems. Change impact analysis avoid degradation problems. No one provides completely what a reengineer needs to remodularize a system.

In this dissertation, we argue for the need of remodularize architecture to ensure evolution. On the structure at package level, we argue that a reengineer has three needs: (i) understand the structure and assess its quality; (ii) identify and understand the structural problems; and (iii) take decisions and verify the impact of these decisions.

Our solution provides a complete approach to help reengineers resolving cyclic dependencies at package level. ECELL provides a view of a package dependency showing fine-grained information. EDSM provides a view of coarse-grained information. Then EDSM and CYCLETABLE highlight strongly connected components as structural problems. OZONE proposes dependencies to remove with the goal of obtaining an acyclic package structure. ORION helps reengineers analyzing the impact of possible changes.

Our validations of each step shows that the approach works and can be used with object-oriented systems. We validate each part of the approach separately, which provides finally a complete validation.

9.2 Discussion: How ECOO Answer the Reengineer Needs

The ECOO approach provides architecture analysis with ECELL and EDSM; cycle understanding with EDSM and CYCLETABLE; package layers understanding with EDSM and OZONE; change impact analysis with ORION; and help for decision with OZONE and ORION.

We argue that reengineers need to (i) understand package structure and dependencies; (ii) identify package dependency problems; and (iii) analyze the impact of a change on the package structure. The ECOO approach provides an answer to these three points:

Understanding Fine-Grained Package Structure and Dependencies. ECELL provides a fine-grained view of package dependencies. It builds a view of dependency from a package to another one. ECELL shows only information needed to understand the dependency. It is based on four kinds of dependencies: inheritance, class reference, method invocation, and class extension.

Identifying Package Dependency Problems. EDSM allows one to see, in a matrix, the structure of a software application. Using colors, it provides a preattentive visualization

to spot cycle issues. With the integration of ECELL it helps reengineers understand cycles at a fine-grained level. CYCLETABLE decomposes each strongly connected component already detected in EDSM in minimal cycles and highlights dependencies implied in several of them. ECELL included in CYCLETABLE provides fine-grained information for each dependency. OZONE is a semi-automatic algorithm which provides a list of *unwanted* dependencies based on direct cycles shown in EDSM and shared dependencies already highlighted in CYCLETABLE. It builds a layered architecture based on this list. The reengineer can add his own evaluation of dependencies to avoid false-positive results of the algorithm.

Analyzing the Impact of a Change on the Package Structure. ORION is an approach to simulate changes in multiple versions of a model without impacting the real system. It allows reengineers to try changes without breaking the system and having feedback from reengineering tools (*i.e.,* visualizations, metrics).

9.3 Open Issues

There are multiple open issues that were not discussed in this thesis, but should be explored in future work.

9.3.1 Cycle Identification

In the user study of EDSM (Chapter 5, p.73), 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 [Martin 2000] and probably that lots of 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 have similar characteristics and it would be interesting to analyze them and discover particular patterns that can help in detection of unwanted dependencies.

Second, there are some cycles that seem not to be breakable because of the semantic. For convenient reasons, engineers organize classes in multiple small packages instead of one big. This organization creates package cycles, which 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 cycles because of a semantic division that make sense for the engineer and is not critical for the modularity of the application.

- What is the definition of a package? What are the definitions/roles of packages? Related to the first question, we do not know how an engineer builds a package: is it for structural architecture or other convenience?

- Finally we could ask: Is Acyclic Dependency Principle (ADP) valid in all cases? Is it not too costly to apply ADP at package level?

Third, some algorithms exist in the regression and integration testing domain that could be useful to improve our strategy. The goal of these algorithms is to find vertices to create stub instead of finding edges that should be removed. Consequently, a future work is to analyze how we can use this different vision and these algorithms. Another idea from this domain is the differentiation of the kind of dependencies to avoid removing inheritances [Kung 1996, Tai 1997]. A future work is to use this idea in OZONE to select unwanted dependencies.

9.3.2 Dependency Analysis

Lightest Dependencies. Our work focuses on the syntactic analysis of dependencies and cycles, without exploring the semantics. There is a hypothesis that the "lightest" dependencies should be broken to break a cycle. This was validated, but there was no further exploration of the nature of dependencies and cycles.

A next step would be to try to classify the kinds of dependencies and cycles that occur in practice. This information could be used to improve the algorithms to provide more accurate indications of *unwanted dependencies*.

Moreover, our algorithms do not take into account the different kinds of dependencies (*i.e.,* inheritance, class references, method invocation, class extension) that we already detect in ECELL. Future work will follow two directions: (i) analyzing which dependency patterns arise most commonly in cycles; and (ii) analyzing which kind of strength we can give to the different kinds of dependencies to minimize false-positive results in the algorithms.

Shared Dependencies. The strategy based on the *shared dependencies* in CYCLETABLE and OZONE depends on the analysis of the complete system to have all shared dependencies. In the case of computing the algorithm on only a part of a system, shared dependencies are smaller and the algorithm should return more false-positive values.

We should investigate the *shared dependencies* strategy and compare it with other approaches. It could be used as a metric to compute the quality of a dependency. Aggregated to other metrics, we can remove false-positive detected dependencies.

Cost analysis. Another issue is to estimate the cost of breaking cycles. If there is a cycle with 8 dependencies from a package A to a package B, and 20 from B to A, it could be that the 20 dependencies all go to a single class in A, but the 8 dependencies go to different locations in B. If this single class in A has no further dependencies within A, then moving this class to package B will be easier to break the cycle than to try breaking the 8 dependencies. We need to investigate a cost analysis.

9.3.3 ORION Change Impact Analysis

We plan to optimize query running time by using optimized search structure for retrieving entities between versions. We envision dedicated visualizations for change assessment. We also need to assess how our current infrastructure handles new reengineering tasks: then

new tools and models need to be developed on top of ORION and possibly adapted. In particular, we believe ORION could be a useful approach to assess automatic reengineering tools such as [Abdeen 2009]. Such tools usually provide a refactored model without rationale for their decisions, which makes reengineers wary of the result. With ORION, we could "watch" automatic algorithms iterate over the model, creating new versions at each important step, and giving better insights on the inner working of the tool. Overall, ORION aims to provide better decision support for software reengineers.

APPENDIX

APPENDIX A

How to use Enriched Dependency Structural Matrix

Contents

A.1 DSM Presentation

Dependency Structural Matrix (DSM) is an adjacency matrix: a cell represents a link between two packages. The rule for reading the matrix is: an element in column header references an element in row header at the crossing of the column and the row. For example, in Figure A.1, A references B and C, B references A, C references D and D references C.

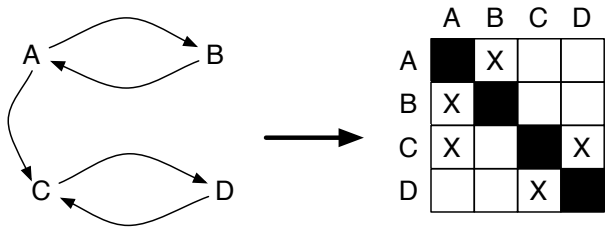


Figure A.1: A simple dependency graph and its matching DSM

In this visualization, the studied elements are packages. Previously in this tutorial, we talk about package rather than element.

A.2 DSM Cell color

A DSM is represented with numbers and several colors Figure A.2. Numbers represents the sum of all dependencies between the two involved packages. Color is an help system to understand the structure:

- gray cells: represent a dependency between a client package and a provider package
- blue squares (covering several cells): represent a **strongly connected component (SCC)**, all packages involved in a SCC are in cycle
- yellow cell: represent a single dependency involved in a SCC.
- red/pink cell: represent a cycle between two packages ($A \rightarrow B$ and $B \rightarrow A$), we name it *direct cycle*. If a cell is red, it means that the number of dependencies from column to row is a third (or less) than the number of dependencies from row to column (note: the symetric cell will be in pink). It means that this dependency is probably easier to remove than its counterpart.

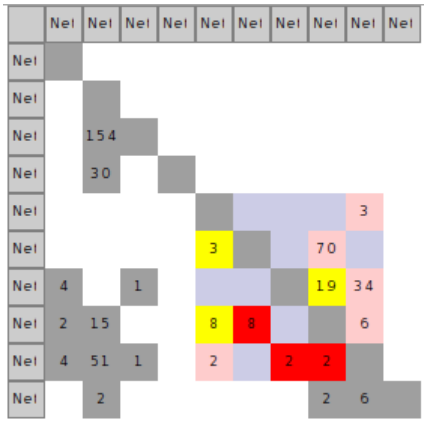


Figure A.2: Color in DSM: red/pink → direct cycle, yellow → a link composing a SCC.

A.3 Enriched contextual cell information

A cell can be *opened* to give more details on the dependencies. We call it an *eCell* It is presented as Figure A.3. An eCell is written from top to bottom:

- on top, there is the number of dependencies. It shows the total number of dependencies, and the number or four different types of dependencies (inheritance, reference, invocation, extension).
- In the core of the eCell, we show the classes involved in the dependency and the links between these classes. A class can have three colors and one specific border. Specific colors are: pale orange means the class is involved in other dependencies, strong orange means a method of this class is involved in other dependencies, gray is for other classes. A black thick border means that the class is implied in both dependencies of a direct cycle. A dotted border represents a class extension: the class is not defined in the targeted package

- at the bottom, there is the state of the dependency, which is the color of the cell presented in Section A.2.

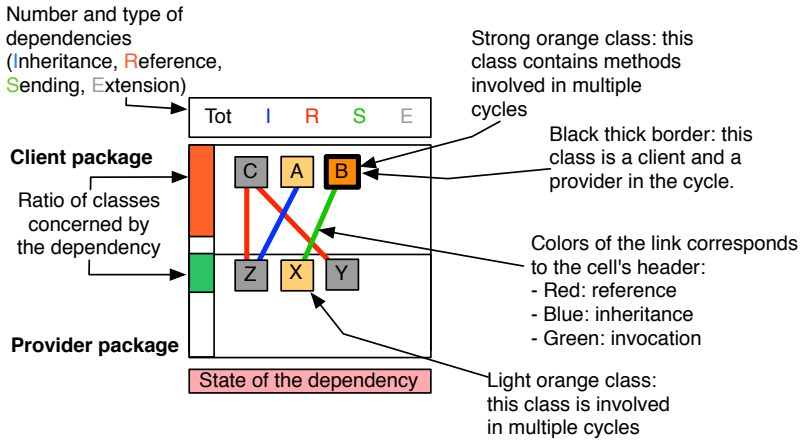


Figure A.3: Enriched cell structural information.

A.3.1 Tooltip

When the cursor is on a non-empty cell: a tooltip appears with an eCell inside. If it is on a direct cycle, it shows the two concerned dependencies, as in Figure A.4

A.3.2 Contextual menu and double click

By right clicking on colored cells (blue, red or yellow), it is possible to open a DSM or a DSM with eCell focusing the specific SCC. In Figure A.5, we can see the contextual menu, which provide, for example a DSM as in Figure A.6.

The use of double-click is possible on header and on SCC. For the header, it opens a DSM with all packages on which the selected package depends. And for the SCC, it opens a window with an eDSM of the SCC, as in Figure A.6.

A.3.3 Show source code

To see the source-code, it is possible to right-click on a class and to select “Browse” on the class or on a specific method as in Figure A.7.

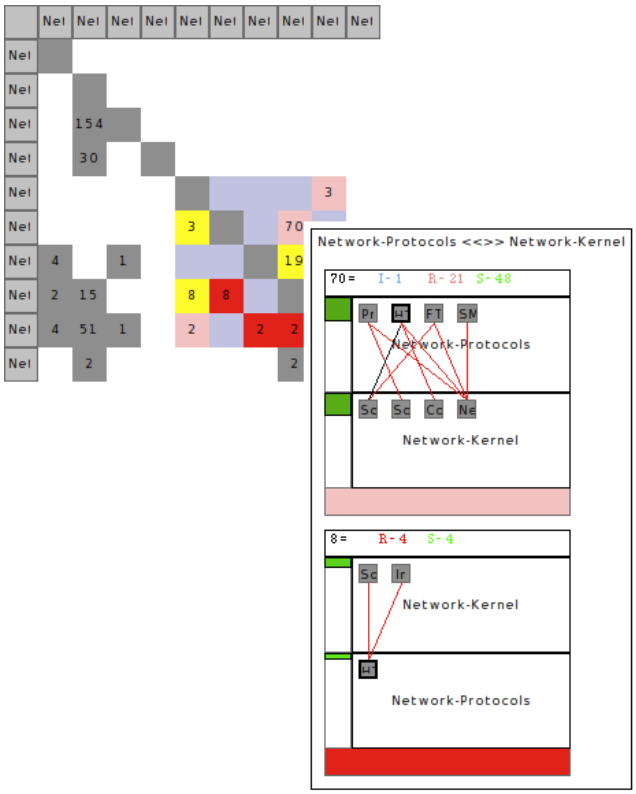


Figure A.4: Fly by help in action.

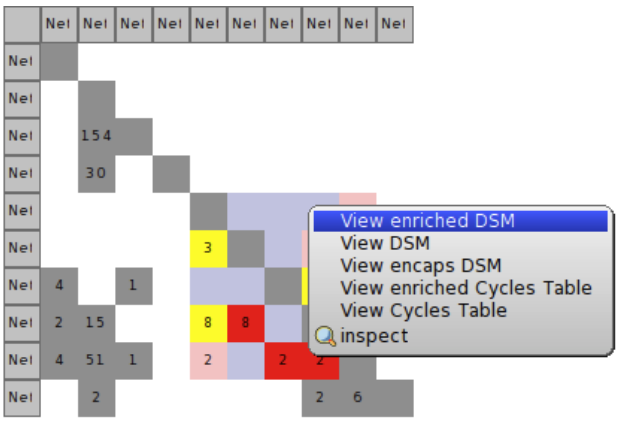


Figure A.5: Contextual menu.

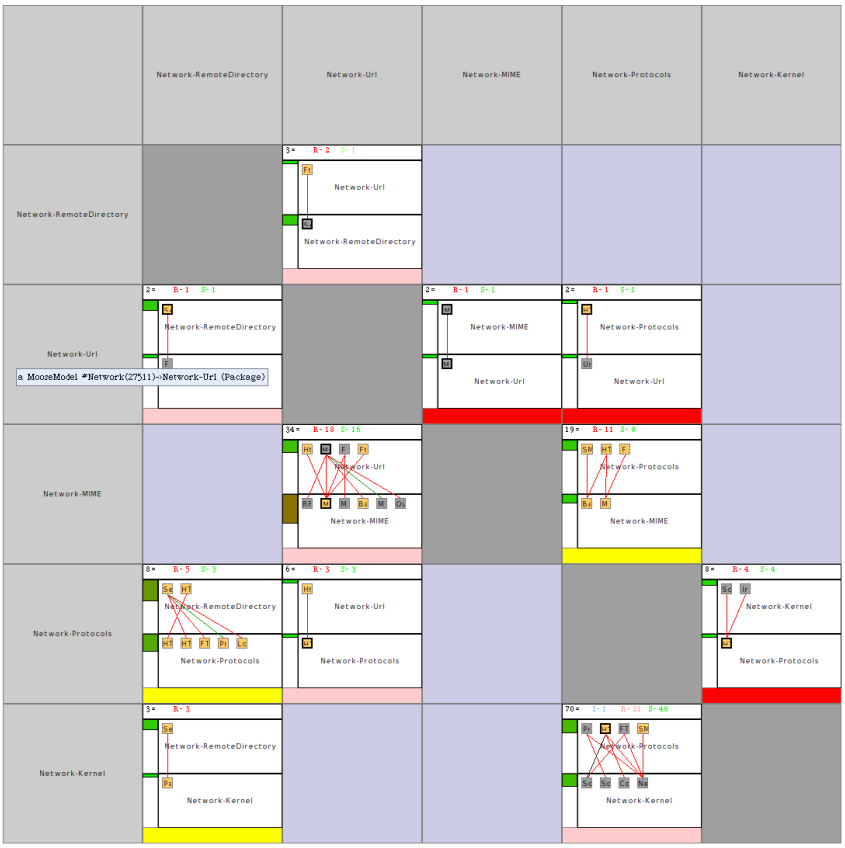


Figure A.6: one SCC centric eDSM.

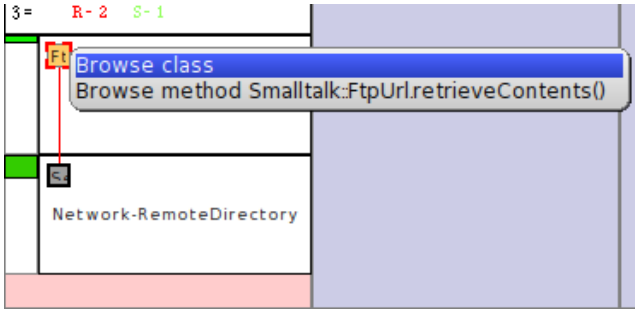


Figure A.7: We can browse source code.

Bibliography

[Abdeen 2008] Hani Abdeen, Ilham Alloui, Stéphane Ducasse, Damien Pollet and Mathieu Suen. *Package Reference Fingerprint: a Rich and Compact Visualization to Understand Package Relationships*. In European Conference on Software Maintenance and Reengineering (CSMR), pages 213–222. IEEE Computer Society Press, 2008. 3

[Abdeen 2009] Hani Abdeen, Stéphane Ducasse, Houari A. Sahraoui and Ilham Alloui. *Automatic Package Coupling and Cycle Minimization*. In International Working Conference on Reverse Engineering (WCRE), pages 103–112, Washington, DC, USA, 2009. IEEE Computer Society Press. 16, 17, 124, 159

[Abdi 2006] M. K. Abdi, H. Lounis and H. Sahraoui. *Analyzing Change Impact in Object-Oriented Systems*. In EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, pages 310–319, Washington, DC, USA, 2006. IEEE Computer Society. 21, 151

[Abdi 2009] Mustapha Abdi, Hakim Lounis and Houari Sahraoui. *Analyse et prédiction de l'impact de changements dans un système à objets : Approche probabiliste*. In Proceedings of LMO'09, 2009. 21, 151

[Abello 2004] James Abello and Frank van Ham. *Matrix Zoom: A Visual Interface to Semi-External Graphs*. In 10th IEEE Symposium on Information Visualization (InfoVis 2004), 10-12 October 2004, Austin, TX, USA, pages 183–190. IEEE Computer Society, 2004. 14

[Abreu 2001] Fernando Brito e Abreu and Miguel Goulão. *Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded?* In Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, CSMR '01, pages 47–, Washington, DC, USA, 2001. IEEE Computer Society. 3

[Adar 2006] Eytan Adar. *GUESS: a language and interface for graph exploration*. In Proceedings of the SIGCHI conference on Human Factors in computing systems, CHI '06, pages 791–800, New York, NY, USA, 2006. ACM. 11

[Anderson 2001] Paul Anderson and Tim Teitelbaum. *Software Inspection Using CodeSurfer*. In Proceedings of WISE'01 (International Workshop on Inspection in Software Engineering), 2001. 20

[Anderson 2005] Paul Anderson and Mark Zarins. *The CodeSurfer Software Understanding Platform*. International Conference on Program Comprehension, vol. 0, pages 147–148, 2005. 20

[Andriyevska 2005] Olena Andriyevska, Natalia Dragan, Bonita Simoes and Jonathan I. Maletic. *Evaluating UML Class Diagram Layout based on Architectural Importance*. VISSOFT 2005. 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis, vol. 0, page 9, 2005. 124

[Antoniol 1999] G. Antoniol, G. Canfora and A. de Lucia. *Estimating the Size of Changes for Evolving Object Oriented Systems: A Case Study*. In METRICS '99: Proceedings of the 6th International Symposium on Software Metrics, page 250, Washington, DC, USA, 1999. IEEE Computer Society. 22, 152

[Bachmann 2000] Felix Bachmann, Len Bass, Jeromy Carriere, Paul Clements, David Garlan, James Ivers, Robert Nord, Reed Little, Norton L. Compton and Lt Col. *Software Architecture Documentation in Practice: Documenting Architectural Layers*, 2000. 4, 29, 105

[Balmas 2009a] Françoise Balmas, Fabrice Bellingard, Simon Denier, Stéphane Ducasse, Jannik Laval and Karine Mordal-Manet. *Practices in the Squale Quality Model (Squale Deliverable 1.3)*. Rapport technique, INRIA, 2009. 10

[Balmas 2009b] Françoise Balmas, Alexandre Bergel, Simon Denier, Stéphane Ducasse, Jannik Laval, Karine Mordal-Manet, Hani Abdeen and Fabrice Bellingard. *Software metric for Java and C++ practices*. Rapport technique, INRIA Lille Nord Europe, 2009. 10

[Balzer 2005] Michael Balzer, Oliver Deussen and Claus Lewerentz. *Voronoi treemaps for the visualization of software metrics*. In SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, pages 165–172, New York, NY, USA, 2005. ACM. 125

[Bansal 2009] P. Bansal, S. Sabharwal and P. Sidhu. *An investigation of strategies for finding test order during Integration testing of object Oriented applications*. In Methods and Models in Computer Science, 2009. ICM2CS 2009. Proceeding of International Conference on, pages 1 –8, dec. 2009. 19

[Bergel 2005] Alexandre Bergel, Stéphane Ducasse and Oscar Nierstrasz. *Analyzing Module Diversity*. Journal of Universal Computer Science, vol. 11, no. 10, pages 1613–1644, November 2005. 28

[Binkley 2004] David Binkley and Mark Harman. *Analysis and Visualization of Predicate Dependence on Formal Parameters and Global Variables*. IEEE Trans. Softw. Eng., vol. 30, no. 11, pages 715–735, 2004. 81, 100

[Binkley 2005] David Binkley and Mark Harman. *Locating Dependence Clusters and Dependence Pollution*. In ICSM '05, pages 177–186, Washington, DC, USA, 2005. IEEE Computer Society. 81, 100

[Bischofberger 2004] Walter R. Bischofberger, Jan Kühl and Silvio Löffler. *Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking*. In Flávio Oquendo, Brian Warboys and Ronald Morrison, editeurs, EWSA, volume 3047 of *Lecture Notes in Computer Science*, pages 1–9. Springer, 2004. 14

[Black 2009] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou and Marcus Denker. Pharo by example. Square Bracket Associates, 2009. 117

[Bohner 1996] Shawn A. Bohner and Robert S. Arnold. Software change impact analysis. IEEE Computer Society Press, 1996. 20, 129

[Bourdoncle 1993] Francois Bourdoncle. *Efficient Chaotic Iteration Strategies With Widenings*. In In Proceedings of the International Conference on Formal Methods in Programming and their Applications, pages 128–141. Springer-Verlag, 1993. 19

[Briand 2001] L.C. Briand, Y. Labiche and Yihong Wang. *Revisiting strategies for ordering class integration testing in the presence of dependency cycles*. In Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on, pages 287 – 296, nov. 2001. 18, 19, 107, 124

[Briand 2002] Lionel C. Briand, Jie Feng and Yvan Labiche. *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, pages 43–50, New York, NY, USA, 2002. ACM. 19

[Briand 2003] L.C. Briand, Y. Labiche and Yihong Wang. *An investigation of graph-based class integration test order strategies*. Software Engineering, IEEE Transactions on, vol. 29, no. 7, pages 594 – 607, july 2003. 18, 19

[Browning 2001] T.R. Browning. *Applying the design structure matrix to system decomposition and integration problems: a review and new directions*. IEEE Transactions on Engineering Management, vol. 48, no. 3, pages 292–306, 2001. 57

[Buckley 2005] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid and Günter Kniesel. *Towards a Taxonomy of Software Change*. Journal on Software Maintenance and Evolution: Research and Practice, pages 309–332, 2005. 22, 150

[Cai 2007] Yuangfang Cai and Sunny Huynh. *An Evolution Model for Software Modularity Assessment*. In WoSQ '07: Proceedings of the 5th International Workshop on Software Quality, page 3, Washington, DC, USA, 2007. IEEE Computer Society. 21, 57, 152

[Chan 2002] W. K. Chan, T. Y. Chen and T. H. Tse. *An Overview of Integration Testing Techniques for Object-Oriented Programs*. In Proceedings of the 2nd ACIS Annual International Conference on Computer and Information Science (ICIS 2002), pages 696–701. International Association for Computer and Information Science, 2002. 17

[Chaumun 2002] M. Ajmal Chaumun, Hind Kabaili, Rudolf K. Keller and Francois Lustman. *A change impact model for changeability assessment in object-oriented software systems*. Science of Computer Programming, vol. 45, no. 2-3, pages 155 – 174, 2002. 21, 151

[Chen 2001] Huo Yan Chen, T. H. Tse and T. Y. Chen. *TACCLE: a methodology for object-oriented software Testing At the Class and Cluster LEvels*. ACM Transactions on Software Engineering and Methodology, vol. 10, no. 1, pages 56–109, 2001. 17

[Chikofsky 1990] Elliot Chikofsky and James Cross II. *Reverse Engineering and Design Recovery: A Taxonomy*. IEEE Software, vol. 7, no. 1, pages 13–17, January 1990. 128

[Corbi 1989] Thomas A. Corbi. *Program Understanding: Challenge for the 1990's*. IBM Systems Journal, vol. 28, no. 2, pages 294–306, 1989. 2

[Da Veiga Cabral 2010] Rafael Da Veiga Cabral, Aurora Pozo and Silvia Regina Vergilio. *A Pareto ant colony algorithm applied to the class integration and test order problem*. In Proceedings of the 22nd IFIP WG 6.1 international conference on Testing software and systems, ICTSS'10, pages 16–29, Berlin, Heidelberg, 2010. Springer-Verlag. 19

[D'Ambros 2006] Marco D'Ambros and Michele Lanza. *Software Bugs and Evolution: A Visual Approach to Uncover Their Relationship*. In Proceedings of CSMR 2006 (10th IEEE European Conference on Software Maintenance and Reengineering), pages 227 – 236. IEEE Computer Society Press, 2006. 10

[D'Ambros 2007] Marco D'Ambros and Michele Lanza. *BugCrawler: Visualizing Evolving Software Systems*. In Proceedings of CSMR 2007 (11th IEEE European Conference on Software Maintenance and Reengineering), page to be published, 2007. 10

[Davis 1995] Alan Mark Davis. 201 principles of software development. McGraw-Hill, 1995. 2

[Demeyer 1999] Serge Demeyer, Stéphane Ducasse and Michele Lanza. *A Hybrid Reverse Engineering Platform Combining Metrics and Program Visualization*. In Francoise Balmas, Mike Blaha and Spencer Rugaber, editeurs, Proceedings of 6th Working Conference on Reverse Engineering (WCRE '99). IEEE Computer Society, October 1999. 114

[Demeyer 2001] Serge Demeyer, Sander Tichelaar and Stéphane Ducasse. *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Rapport technique, University of Bern, 2001. 36, 59, 114, 133

[Demeyer 2002] Serge Demeyer, Stéphane Ducasse and Oscar Nierstrasz. Object-oriented reengineering patterns. Morgan Kaufmann, 2002. 4

[Denier 2010a] Simon Denier, Jannik Laval, Stéphane Ducasse and Fabrice Bellingard. *Technical and Economical Model (Squale Deliverable 2.1)*. Rapport technique, INRIA, 2010. 10

[Denier 2010b] Simon Denier, Jannik Laval, Stéphane Ducasse and Fabrice Bellingard. *Technical Model for Remediation (Squale Deliverable 2.2)*. Rapport technique, INRIA, 2010. 10

[Denker 2007] Marcus Denker, Tudor Gîrba, Adrian Lienhard, Oscar Nierstrasz, Lukas Renggli and Pascal Zumkehr. *Encapsulating and Exploiting Change with Changeboxes*. In Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), pages 25–49. ACM Digital Library, 2007. 21, 152

[DeRemer 1976] Frank DeRemer and Hans H. Kron. *Programming in the Large Versus Programming in the Small*. IEEE Transactions on Software Engineering, vol. 2, no. 2, pages 80–86, June 1976. 3, 5

[Dig 2007] Danny Dig, Kashif Manzoor, Ralph Johnson and Tien Nguyen. *Refactoring-aware Configuration Management for Object-Oriented Programs*. In International Conference on Software Engineering (ICSE 2007), pages 427–436, 2007. 22, 150

[Dong 2007a] Xinyi Dong and M.W. Godfrey. *System-level Usage Dependency Analysis of Object-Oriented Systems*. In ICSM 2007. IEEE Comp. Society, 2007. 4, 15, 56, 125

[Dong 2007b] Xinyi Dong and M.W. Godfrey. *System-level Usage Dependency Analysis of Object-Oriented Systems*. In ICSM 2007: IEEE International Conference on Software Maintenance, pages 375–384, October 2007. 10

[Ducasse 2005a] Stéphane Ducasse, Tudor Gîrba, Michele Lanza and Serge Demeyer. *Moose: a Collaborative and Extensible Reengineering Environment*. In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005. 36, 59, 114

[Ducasse 2005b] Stéphane Ducasse and Michele Lanza. *The Class Blueprint: Visually Supporting the Understanding of Classes*. Transactions on Software Engineering (TSE), vol. 31, no. 1, pages 75–90, January 2005. 10, 56, 125

[Ducasse 2005c] Stéphane Ducasse, Michele Lanza and Laura Ponisio. *Butterflies: A Visual Approach to Characterize Packages*. In Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), pages 70–77. IEEE Computer Society, 2005. 4, 10

[Ducasse 2006] 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. 10, 56, 125

[Ducasse 2007] 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. 3, 4, 11, 56, 80, 100, 125

[Ducasse 2009a] Stéphane Ducasse, Simon Denier, Françoise Balmas, Alexandre Bergel, Jannik Laval, Karine Mordal-Manet and Fabrice Bellingard. *Visualization of Practices and Metrics (Squale Deliverable 1.2)*. Rapport technique, INRIA, 2009. 10

[Ducasse 2009b] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn and Lukas Renggli. *Meta-Environment and Executable Meta-Language using Smalltalk: an Experience Report*. Journal of Software and Systems Modeling (SOSYM), vol. 8, no. 1, pages 5–19, February 2009. 37, 133

[Ducasse 2009c] Stéphane Ducasse and Damien Pollet. *Software Architecture Reconstruction: A Process-Oriented Taxonomy*. IEEE Transactions on Software Engineering, vol. 35, no. 4, pages 573–591, July 2009. 104

[Eades 1993] Peter Eades, Xuemin Lin and W. F. Smyth. *A Fast Effective Heuristic For The Feedback Arc Set Problem*. Information Processing Letters, vol. 47, pages 319–323, 1993. 16, 106, 107, 123

[Easterbrook 2008] Steve Easterbrook, Janice Singer, Margaret anne Storey and Daniela Damian. *Selecting Empirical Methods for Software Engineering Research*, 2008. 147

[Ebraert 2009] Peter Ebraert. *A bottom-up approach to program variation*. PhD thesis, Vrije Universiteit Brussels, 2009. 20, 150

[Eick 2001] Stephen Eick, Todd Graves, Alan Karr, J. Marron and Audris Mockus. *Does Code Decay? Assessing the Evidence from Change Management Data*. IEEE Transactions on Software Engineering, vol. 27, no. 1, pages 1–12, 2001. 2

[Erlikh 2000] Len Erlikh. *Leveraging Legacy System Dollars for E-Business*. IT Professional, vol. 2, no. 3, pages 17–23, 2000. 2

[Falleri 2008] Jean-Rémi Falleri, Marianne Huchard and Clémentine Nebut. *A generic approach for class model normalization (short paper)*. In ASE 2008: 23th IEEE/ACM International Conference on Automated Software Engineering, 2008. 17

[Falleri 2010] Jean Rémi Falleri, Simon Denier, Jannik Laval, Philipe Vismara and Stéphane Ducasse. *Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems*. Rapport technique, INRIA, November 2010. Computer Science Technical Report. 31

[Frankl 1994] Phyllis G. Frankl. *The ASTOOT approach to testing object-oriented programs*. ACM Transactions on Software Engineering, page 453, 1994. 17

[Gallagher 1991] Keith Brian Gallagher and James R. Lyle. *Using Program Slicing in Software Maintenance*. Transactions on Software Engineering, vol. 17, no. 18, pages 751–761, August 1991. 20

[Gansner 2000] Gansner and North. *An Open Graph Visualization System and its Applications to Software Engineering*. Software Practice Experience., vol. 30, no. 11, pages 1203–1233, 2000. 11

[Ghoniem 2005] Mohammad Ghoniem, Jean-Daniel Fekete and Philippe Castagliola. *On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis*. Information Visualization, vol. 4, no. 2, pages 114–135, 2005. 12

[Gîrba 2005a] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Bern, Bern, November 2005. 21, 151

[Gîrba 2005b] Tudor Gîrba, Michele Lanza and Stéphane Ducasse. *Characterizing the Evolution of Class Hierarchies*. In Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society. 114

[Griswold 1993] William G. Griswold and David Notkin. *Automated assistance for program restructuring*. ACM Trans. Softw. Eng. Methodol., vol. 2, no. 3, pages 228–269, 1993. 2

[Han 1997] Jun Han. *Supporting Impact Analysis and Change Propagation in Software Engineering Environments*. In STEP '97: Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP '97) (including CASE '97), page 172, Washington, DC, USA, 1997. IEEE Computer Society. 21, 151

[Hanh 2001] Vu Le Hanh, Kamel Akif, Yves Le Traon and Jean-Marc Jézéquel. *Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies*. In Proceedings of the 15th European Conference on Object-Oriented Programming, ECOOP '01, pages 381–401, London, UK, UK, 2001. Springer-Verlag. 19, 124

[Harman 2002] Mark Harman, Robert Hierons and Mark Proctor. *A new representation and crossover operator for search-based optimization of software modularization*. In In GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 1351–1358. Morgan Kaufmann Publishers, 2002. 17

[Hautus 2002] Edwin Hautus. *Improving Java Software Through Package Structure Analysis*. In IASTED, International Conference on Software Engineering and Applications, 2002. 10, 15, 123

[Healey 1992] C. G. Healey. Visualization of multivariate data using preattentive processing. Master's thesis, Department of Computer Science, University of Bristish Columbia, 1992. 10, 42, 61

[Healey 1993] C. G. Healey, K. S. Booth and Enns J. T. *Harnessing Preattentive Processes for Multivariate Data Visualization*. In GI '93: Proceedings of Graphics Interface, 1993. 42, 61

[Heer 2010] Jeffrey Heer, Michael Bostock and Vadim Ogievetsky. *A Tour through the Visualization Zoo*. Queue, vol. 8, no. 5, pages 20–30, 2010. 56, 86

[Henry 2007] Nathalie Henry, Jean-Daniel Fekete and Michael J. McGuffin. *NodeTrix: a Hybrid Visualization of Social Networks*. IEEE Trans. Vis. Comput. Graph., vol. 13, no. 6, pages 1302–1309, 2007. 12

[Holt 2006] Holt, Schürr, Sim and Winter. *GXL: A graph-based standard exchange format for reengineering*. Science of Computer Programming, vol. 60, no. 2, pages 149–170, April 2006. 11

[Holten 2009] Danny Holten. *Visualization of Graphs and Trees for Software Analysis*. PhD thesis, Computer science department, 2009. ISBN 978-90-386-1882-1. 11

[Jain 1988] Anil Kumar Jain and Richard C. Dubes. Algorithms for clustering data. Prentice Hall, Englewood Cliffs, 1988. 16

[Jain 1999] Anil Kumar Jain, M. Narasimha Murty and Patrick Joseph Flynn. *Data Clustering: a Review*. ACM Computing Surveys, vol. 31, no. 3, pages 264–323, 1999. 16

[Johnson 1988] Ralph Johnson. *TS: An Optimizing Compiler for Smalltalk*. In Proceedings OOPSLA '88, ACM SIGPLAN Notices, volume 23, pages 18–26, November 1988. 21

[Kann 1992] Viggo Kann. *On the Approximability of NP-complete Optimization Problems*. PhD thesis, Royal Institute of Technology Stockholm, 1992. 16, 84, 123

[Kleinberg 1999] Jon M. Kleinberg. *Authoritative Sources in a Hyperlinked Environment*. JOURNAL OF THE ACM, vol. 46, no. 5, pages 604–632, 1999. 15, 124

[Koschke 2000] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000. 15

[Kung 1994] D. Kung, J. Gao, P. Hsia, F. Wen, Y. Y. Toyoshima and C. Chen. *Change impact identification in object oriented software maintenance*. In Proceedings of the International Conference on Software Maintenance, pages 202–211, 1994. 22, 152

[Kung 1995] David Chenho Kung, Jerry Gao, Pei Hsia, Jeremy Lin and Yasufumi Toyoshima. *Class Firewall, Test Order, and Regression Testing of Object-Oriented Programs*. JOOP, vol. 8, no. 2, pages 51–65, 1995. 18

[Kung 1996] David C. Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima and Cris Chen. *On regression testing of object-oriented programs*. J. Syst. Softw., vol. 32, pages 21–40, January 1996. 18, 19, 107, 125, 158

[Labiche 2000] Y. Labiche, P. Thevenod-Fosse, H. Waeselynck and M.-H. Durand. *Testing levels for object-oriented software*. In Software Engineering, 2000. Proceedings of the 2000 International Conference on, pages 136 –145, 2000. 18

[Langelier 2005] Guillaume Langelier, Houari A. Sahraoui and Pierre Poulin. *Visualization-based analysis of quality for large-scale software systems*. In ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 214–223, New York, NY, USA, 2005. ACM. 10, 125

[Lanza 2001] Michele Lanza. *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*. In Proceedings of IWPSE 2001 (International Workshop on Principles of Software Evolution), pages 37–42, 2001. 10

[Lanza 2002] Michele Lanza and Stéphane Ducasse. *Understanding Software Evolution Using a Combination of Software Visualization and Software Metrics*. In Proceedings of Langages et Modèles à Objets (LMO'02), pages 135–149, Paris, 2002. Lavoisier. 10

[Lanza 2003a] Michele Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Bern, May 2003. 10

[Lanza 2003b] Michele Lanza and Stéphane Ducasse. *Polymetric Views—A Lightweight Visual Approach to Reverse Engineering*. Transactions on Software Engineering (TSE), vol. 29, no. 9, pages 782–795, September 2003. 10, 68, 114

[Laval 2009a] Jannik Laval, Simon Denier, Stéphane Ducasse and Alexandre Bergel. *Identifying cycle causes with Enriched Dependency Structural Matrix*. In WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering, Lille, France, 2009. 5

[Laval 2009b] Jannik Laval, Simon Denier, Stéphane Ducasse and Andy Kellens. *Supporting Incremental Changes in Large Models*. In Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2009), Brest, France, 2009. 134

[Laval 2010a] Jannik Laval, Nicolas Anquetil and Stéphane Ducasse. *OZONE: Package Layered Structure Identification in presence of Cycles*. In Proceedings of the 9th edition of the Workshop BElgian-NEtherlands software eVOLution seminar, BENEVOL 2010, Lille, France, 2010. 6

[Laval 2010b] Jannik Laval, Simon Denier and Stéphane Ducasse. *Cycles Assessment with CycleTable*. Rapport technique, INRIA, 2010. 6

[Laval 2010c] Jannik Laval, Simon Denier, Stéphane Ducasse and Jean-Rémy Falleri. *Supporting Simultaneous Versions for Software Evolution Assessment*. Journal of Science of Computer Programming (SCP), May 2010. 6

[Le Traon 2000] Y. Le Traon, T. Jeron, J.-M. Jezequel and P. Morel. *Efficient object-oriented integration and regression testing*. Reliability, IEEE Transactions on, vol. 49, no. 1, pages 12 –25, March 2000. 18, 19, 107, 124

[Lee 1998] Michelle Li Lee. *Change impact analysis of object-oriented software*. PhD thesis, George Mason University, Fairfax, VA, USA, 1998. Director-Jeff Offutt. 21, 151

[Lehman 1985] Manny Lehman and Les Belady. Program evolution: Processes of software change. London Academic Press, London, 1985. 2

[Lehman 1996] Manny Lehman. *Laws of Software Evolution Revisited*. In European Workshop on Software Process Technology, pages 108–124, Berlin, 1996. Springer. 2, 3

[Li 1996] Li Li and A. Jefferson Offutt. *Algorithmic Analysis of the Impact of Changes to Object-Oriented Software*. In ICSM '96: Proceedings of the 1996 International Conference on Software Maintenance, pages 171–184, Washington, DC, USA, 1996. IEEE Computer Society. 21, 151

[Lopes 2005] Antónia Lopes and José Luiz Fiadeiro. *Context-Awareness in Software Architectures*. In Proceeding of the 2nd European Workshop on Software Architecture (EWSA), volume 3527 of *Lecture Notes in Computer Science*, pages 146–161. Springer, 2005. 56, 85

[Lungu 2006] Mircea Lungu, Michele Lanza and Tudor Gîrba. *Package Patterns for Visual Architecture Recovery*. In Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering), pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press. 4, 10, 15, 125

[Lutz 2001] Rudi Lutz. *Evolving good hierarchical decompositions of complex systems*. Journal of Systems Architecture, vol. 47, no. 7, pages 613–634, 2001. 16, 106, 124

[MacCormack 2006] Alan MacCormack, John Rusnak and Carliss Y. Baldwin. *Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code*. Management Science, vol. 52, no. 7, pages 1015–1030, 2006. 56

[Madhavji 1992] Nazim H. Madhavji. *Environment Evolution: The Prism Model of Changes*. IEEE Transaction in Software Engineering, vol. 18, no. 5, pages 380–392, 1992. 20, 152

[Maletic 2001] Jonathan I. Maletic and Andrian Marcus. *Supporting Program Comprehension Using Semantic and Structural Information*. In Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pages 103–112, May 2001. 10

[Mancoridis 1998] Spiros Mancoridis and Brian S. Mitchell. *Using Automatic Clustering to produce High-Level System Organizations of Source Code*. In Proceedings of IWPC '98 (International Workshop on Program Comprehension). IEEE Computer Society Press, 1998. 16

[Mancoridis 1999] Spiros Mancoridis, Brian S. Mitchell, Y. Chen and E. R. Gansner. *Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures*. In Proceedings of ICSM '99 (International Conference on Software Maintenance), Oxford, England, 1999. IEEE Computer Society Press. 15, 16, 106

[Marcus 2003] Andrian Marcus, Louis Feng and Jonathan I. Maletic. *3D Representations for Software Visualization*. In Proceedings of the ACM Symposium on Software Visualization, pages 27–ff. IEEE, 2003. 10

[Martin 2000] Robert C. Martin. *Design Principles and Design Patterns*, 2000. www.objectmentor.com. 3, 4, 29, 80, 105, 157

[Martin 2002] Robert Cecil Martin. Agile software development. principles, patterns, and practices. Prentice-Hall, 2002. 10, 14

[Melton 2007a] Hayden Melton and Ewan Tempero. *An empirical study of cycles among classes in Java*. Empirical Software Engineering, vol. 12, no. 4, pages 389–415, 2007. 15, 123

[Melton 2007b] Hayden Melton and Ewan D. Tempero. *Jooj: Real-Time Support For Avoiding Cyclic Dependencies*. In APSEC 2007 - 14th Asia-Pacific Software Engineering Conference, pages 87–95. IEEE Computer Society, 2007. 4, 16, 120, 122, 123

[Mens 2006] Tom Mens and Pieter Van Gorp. *A Taxonomy of Model Transformation*. Electr. Notes Theor. Comput. Sci., vol. 152, pages 125–142, 2006. 21, 152

[Mitchell 2002] Brian S. Mitchell and Spiros Mancoridis. *Using Heuristic Search Techniques To Extract Design Abstractions From Source Code*. In Proceedings of GECCO'02, pages 1375–1382, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 16

[Mitchell 2006] Brian S. Mitchell and Spiros Mancoridis. *On the Automatic Modularization of Software Systems Using the Bunch Tool*. IEEE Transactions on Software Engineering, vol. 32, no. 3, pages 193–208, 2006. 15, 16, 106, 124, 125

[Mitchell 2008] Brian S. Mitchell and Spiros Mancoridis. *On the evaluation of the Bunch search-based software modularization algorithm*. Soft Computing - A Fusion of Foundations, Methodologies and Applications, vol. 12, no. 1, pages 77–93, 2008. 16

[Mordal-Manet 2011] Karine Mordal-Manet, Jannik Laval, Stéphane Ducasse, Nicolas Anquetil, Franǫise Balmas, Fabrice Bellingard, Laurent bouhier, Philippe Vaillergues and Thomas McCabe. *An empirical model for continuous and weighted metric aggregation*. In CSMR 2011: Proceedings of the 15th European Conference on Software Maintenance and Reengineering, Oldenburg, Germany, 2011. 10

[MORmSETT 1993] J. G. MORmSETT. *Generalizing first-class stores*. In ACM SIGPLAN Workshop on State of Programing Language, pages 73–87, New York, 1993. ACM. 21

[Munzner 2000] Tamara Macushla Munzner. *Interactive visualization of large graphs and networks*. PhD thesis, Stanford University, Stanford, CA, USA, 2000. AAI9995264. 11

[Myers 1978] G. J. Myers. Composite/structured design. Van Nostrand Reinhold, 1978. 3, 5

[Nguyen 2005a] Tien Nguyen, Ethan Munson and John Boyland. *An Infrastructure for Development of Object-Oriented, Multi-level Configuration Management Services*. In Internationl Conference on Software Engineering (ICSE 2005), pages 215–224. ACM Press, 2005. 22, 150

[Nguyen 2005b] Tien N. Nguyen, Ethan V. Munson and Cheng Thao. *Managing the Evolution of Web-Based Applications with WebSCM*. In ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance, pages 577–586, Washington, DC, USA, 2005. IEEE Computer Society. 22, 150

[Nierstrasz 2005] Oscar Nierstrasz, Stéphane Ducasse and Tudor Gîrba. *The Story of Moose: an Agile Reengineering Environment*. In Proceedings of the European Software Engineering Conference (ESEC/FSE'05), pages 1–10, New York NY, 2005. ACM Press. Invited paper. 117, 129, 143

[Perry 2000] Dewayne E. Perry, Adam A. Porter and Lawrence G. Votta. *Empirical studies of software engineering: a roadmap*. In ICSE '00: Proceedings of the Conference on The Future of Software Engineering, pages 345–355, New York, NY, USA, 2000. ACM. 147

[Pinzger 2005] Martin Pinzger, Harald Gall, Michael Fischer and Michele Lanza. *Visualizing Multiple Evolution Metrics*. In Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization), pages 67–75, St. Louis, Missouri, USA, May 2005. 10

[Ponisio 2006a] Laura Ponisio and Oscar Nierstrasz. *Using Context Information to Re-architect a System*. In Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06), pages 91–103, 2006. 3, 4

[Ponisio 2006b] María Laura Ponisio. *Exploiting Client Usage to Manage Program Modularity*. PhD thesis, University of Bern, Bern, June 2006. 4, 10

[Praditwong 2010] Kata Praditwong, Mark Harman and Xin Yao. *Software Module Clustering as a Multi-Objective Search Problem*. IEEE Transactions on Software Engineering, vol. 99, no. PrePrints, 2010. 124

[Pressman 1994] Roger S. Pressman. Software engineering: A practitioner's approach. McGraw-Hill, 1994. 3, 5

[Purushothaman 2005] R. Purushothaman and D.E. Perry. *Toward understanding the rhetoric of small source code changes*. Software Engineering, IEEE Transactions on, vol. 31, no. 6, pages 511 – 526, June 2005. 2

[Ren 2004] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara Ryder and Ophelia Chesley. *Chianti: A tool for change impact analysis of Java programs*. In Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004), pages 432–448, Vancouver, BC, Canada, oct 2004. 22, 152

[Robbes 2008] Romain Robbes. *Of Change and Software*. PhD thesis, University of Lugano (Switzerland), December 2008. 20, 150

[Ryder 2001] Barbara G. Ryder and Frank Tip. *Change impact analysis for object-oriented programs*. In Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pages 46–53. ACM Press, 2001. 22, 152

[Sangal 2005] 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. 10, 56, 57, 58, 76, 85, 86, 107

[Scanniello 2010] Giuseppe Scanniello, Anna D'Amico, Carmela D'Amico and Teodora D'Amico. *Architectural layer recovery for software system understanding and evolution*. Softw. Pract. Exper., vol. 40, no. 10, pages 897–916, 2010. 15, 124

[Shashank 2010] S.P. Shashank, P. Chakka and D.V. Kumar. *A systematic literature survey of integration testing in component-based software engineering*. In Computer and Communication Technology (ICCCT), 2010 International Conference on, pages 562 –568, sept. 2010. 17

[Snelting 1998] Gregor Snelting and Frank Tip. *Reengineering Class Hierarchies using Concept Analysis*. In ACM Trans. Programming Languages and Systems, 1998. 16

[Sommerville 1996] Ian Sommerville. Software engineering. Addison Wesley, fifth édition, 1996. 2

[Steward 1981] D. Steward. *The design structure matrix: A method for managing the design of complex systems*. IEEE Transactions on Engineering Management, vol. 28, no. 3, pages 71–74, 1981. 56, 85

[Storey 1997] Margaret-Anne D. Storey, Kenny Wong, F. D. Fracchia and Hausi A. Müller. *On integrating visualization techniques for effective software exploration*. In Proceedings of IEEE Symposium on Information Visualization (InfoVis '97), pages 38–48. IEEE Computer Society, 1997. 10

[Sullivan 2001] 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. 56, 85

[Sullivan 2005] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari and Hridesh Rajan. *Information hiding interfaces for aspect-oriented design*. In Proceedings of the ESEC/SIGSOFT FSE 2005, pages 166–175, 2005. 56, 85

[Szyperski 1998] Clemens A. Szyperski. Component software. Addison Wesley, 1998. 29, 105

[Tai 1997] Kuo-Chung Tai and F.J. Daniels. *Test order for inter-class integration testing of object-oriented software*. In Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings., The Twenty-First Annual International, pages 602 –607, aug 1997. 18, 19, 124, 125, 158

[Tarjan 1972] Robert Endre Tarjan. *Depth-First Search and Linear Graph Algorithms*. SIAM J. Comput., vol. 1, no. 2, pages 146–160, 1972. 18, 29, 59, 130, 142

[Tiernan 1970] James C. Tiernan. *An efficient search algorithm to find the elementary circuits of a graph*. Commun. ACM, vol. 13, pages 722–726, December 1970. 31

[Tip 1995] Frank Tip. *A survey of program slicing techniques*. Journal of Programming Languages, vol. 3, pages 121–189, 1995. 20

[Treisman 1985] Anne Treisman. *Preattentive processing in vision*. Computer Vision, Graphics, and Image Processing, vol. 31, no. 2, pages 156–177, 1985. 42, 61

[Tufte 1997] Edward R. Tufte. Visual explanations: images and quantities, evidence and narrative. Graphics Press, Cheshire, CT, USA, 1997. 10, 60, 66, 81, 87

[Tufte 2001] Edward R. Tufte. The visual display of quantitative information. Graphics Press, 2nd édition, 2001. 10

[Vainsencher 2004] Daniel Vainsencher. *MudPie: layers in the ball of mud.* Computer Languages, Systems & Structures, vol. 30, no. 1-2, pages 5–19, 2004. 14, 16, 106, 123, 125

[Ware 2000] Colin Ware. Information visualization: perception for design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. 10, 42, 61

[Warfield 1973] J.N. Warfield. *Binary Matrices in System Modeling*. IEEE Transactions on Systems, Man, and Cybernetics, vol. 3, no. 5, pages 441–449, 1973. 57

[Warth 2008] Alessandro Warth and Alan Kay. *Worlds: Controlling the Scope of Side Effects*. Rapport technique RN-2008-001, Viewpoints Research, 2008. 21, 152

[Weinblatt 1972] Herbert Weinblatt. *A New Search Algorithm for Finding the Simple Cycles of a Finite Directed Graph*. J. ACM, vol. 19, pages 43–56, January 1972. 31

[Welser 1984] M. Welser. *Program Slicing*. IEEE Transactions on Software Engineering, pages 352–357, 1984. 20

[Wettel 2007a] Richard Wettel and Michele Lanza. *Program Comprehension through Software Habitability*. In Proceedings of ICPC 2007 (15th International Conference on Program Comprehension), pages 231–240. IEEE CS Press, 2007. 10

[Wettel 2007b] Richard Wettel and Michele Lanza. *Visualizing Software Systems as Cities*. In Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis), pages 92–99, 2007. 10

[Wilde 1992] Norman Wilde and Ross Huitt. *Maintenance Support for Object-Oriented Programs*. IEEE Transactions on Software Engineering, vol. SE-18, no. 12, pages 1038–1044, December 1992. 15

[Wu 2000] Jingwei Wu and Margaret anne D. Storey. *A multi-perspective software visualization environment*. In In Proc. of CASCON'00, pages 41–50. IBM Press, 2000. 56

[Wysseier 2005] Christoph Wysseier. Interactive 3-D visualization of feature-traces. Master's thesis, University of Bern, Switzerland, November 2005. 10

[Yassine 1999] A. Yassine, D. Falkenburg and K. Chelst. *Engineering design management: an information structure approach*. International Journal of Production Research, vol. 37, no. 13, pages 2957–2975, 1999. 57

[Yourdon 1979] Edward Yourdon. Classics in software engineering. Yourdon Press, 1979. 3, 5

[Zaidman 2006] Andy Zaidman. *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, Universiteit Antwerpen, 2006. 16

Object-Oriented Architecture Analysis and Remediation

Software evolves over time with the modification, addition and removal of new classes, methods, functions, dependencies. A consequence is that behavior may not be placed in the right packages and the software modularization is broken. A good organization of classes into identifiable and collaborating packages eases the understanding, maintenance, test and evolution of software systems. We argue that maintainers lack tool support for understanding the concrete organization and for structuring packages within their context.

Our claim is that the maintenance of large software modularizations needs approaches that help (i) understanding the structure at package level and assessing its quality; (ii) identifying modularity problems; and (iii) take decisions and verify the impact of these decisions.

In this thesis, we propose ECOO, an approach to help reengineers identify and understand structural problems in software architectures and to support the remodularization activity. It concerns the three following research fields:

- Understanding package dependency problems. We propose visualizations to highlight cyclic dependency problems at package level.
- Proposing dependencies to be changed for remodularization. The approach proposes dependencies to break to make the system more modular.
- Analyzing impact of change. The approach proposes a change impact analysis to try modifications before applying them on the real system.

The approaches presented in this thesis have been qualitatively and quantitatively validated and results have been taken into account in the reengineering of analyzed systems. The results we obtained demonstrate the usefulness of our approach.

Keywords: remodularization; dependency analysis; visualization; change impact analysis; package dependency