Université des Sciences et Technologies de Lille – Lille 1

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

Visualizing, Assessing and Re-Modularizing Object-Oriented Architectural Elements

THÈSE

présentée et soutenue publiquement le 24 Novembre 2009

pour l'obtention du

Doctorat de l'Université des Sciences et Technologies de Lille
(spécialité informatique)

par

Hani Abdeen

Composition du jury

Président :	Laurence Duchien	
Rapporteurs :	Marianne Huchard	(Professeur – Université de Montpellier)
	Françoise Balmas	(Maitre de Conference – Université Paris 8)
Examinateurs :	Manuel Oriol	(Senior Lecturer – Université de York)
	Laurence Duchien	(Professeur – Université Lille I)
Directeur de thèse :	Stéphane Ducasse	(Professeur – Université Lille I)
Co-Encadreur de thèse :	Ilham Alloui	(Maitre de Conference – Université de Savoie)

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

Numéro d'ordre: 40109

Mis en page avec la classe thloria.

Contents

List of Tables ix

Chapter 1 Introduction 7

1.1 Context: Object-Oriented Software Modularization 7

1.2 Problem: Software Re-Modularization Challenges 8

1.2.1 The Problem of Understanding Packages 9

1.2.2 The Problem of Modularization Optimization 10

1.3 Our Claim 12

1.4 Contributions 12

1.5 Structure of the Dissertation 13

Chapter 2 Software Re-Modularisation:
Challenges and Approaches 15

2.1 Introduction 15

2.2 Background and Terminology 16

2.3 Package Understanding 21

2.3.1 Quantitive Information 23

2.3.2 Qualitative Information (cohesion vs. coupling) 23

2.3.3 Role and Contextual Information (central vs. peripheral) 25

2.3.4 Organizational Information (developers vs. team) 26

2.4 Challenges in Optimizing Modularization 26

2.4.1 Modularization Complexity 26

2.4.2 Class Distribution over Packages 26

2.4.3 Package Optimization Trade-Offs 27

2.5 Existing Approaches to Understand Packages 29

2.6 Existing Approaches to Assess Package quality 32

2.7 Existing Approaches to Optimize Modularizations 34
2.8 A Combined Approach for The Maintenance of Software Modularization 37

Chapter 3 Package Blueprint:
Visually Understanding Package Structure and Interactions 39
3.1 Introduction 39
3.2 Visualization Challenges 40
3.3 Package Blueprint basic principles 41
3.4 Package Blueprint Detailed Visualizations 42
3.4.1 Outgoing Reference Blueprints 42
3.4.2 Incoming Reference Blueprints 44
3.4.3 The Case of Inheritance 46
3.5 An Example: The Network::Kernel Package 47
3.6 Packages Within Their Software System 49
3.6.1 Outgoing Reference Package Blueprint Analysis 49
3.6.2 Incoming Reference Package Blueprint Analysis 52
3.6.3 Inheritance Package Blueprint Overview 55
3.6.4 The views together 56
3.7 Striking Shapes 56
3.7.1 Shapes of Packages and Surfaces 56
3.7.2 Shapes of Classes 59
3.8 User Case Study on Squeak Compiler 60
3.8.1 Experimental Setup 61
3.8.2 Results 62
3.9 Evaluation and Discussion 63
3.9.1 Evaluation 63
3.9.2 Discussion 64
3.10 Related Work 66
3.11 Conclusion 66

Chapter 4 Package Fingerprints:
Visually Summarizing Package Interface Usage 69
4.1 Introduction 69
4.2 Package Fingerprint Principles 70
4.2.1 Terminology 71
4.2.2 Fingerprint Intention 72
4.2.3 Fingerprint Skeleton 73

4.2.4 Enriching the Fingerprint Skeleton Layout 74
4.3 Decorticating a Fingerprint 76
4.4 Reading the Fingerprint From Far Away 79
4.5 Outgoing Fingerprint 82
4.6 Relevant Visual Patterns 85
4.6.1 Black Fill Pattern 85
4.6.2 Arrow Pattern 89
4.6.3 Mosaic Pattern 92
4.6.4 Diverse Patterns 96
4.7 Discussion and Evaluation 98
4.7.1 Graphical concerns 98
4.7.2 About Coupling and Hints at Improvements 99
4.7.3 Related Work 101
4.8 Conclusion 102

Chapter 5 Automatically Measuring and Optimizing Modularization Quality 103
5.1 Introduction 103
5.2 Modularization Quality 105
5.2.1 Measuring Modularization Quality 105
5.2.2 Measuring Package Quality 106
5.3 Optimization Technique (Methodology) 107
5.3.1 Technique Overview 107
5.3.2 Evaluating Modularization Quality (Fitness) 108
5.3.3 Modularization Constraints 109
5.3.4 Deriving New Modularization (Neighbor) 110
5.4 Experiments and Validation 112
5.5 Related Works 119
5.6 Conclusion and Future Work 121

Chapter 6 Conclusions and Future Work 123
6.1 Open Issues 126

Bibliography 127

List of Figures

2.1 Example of two modularizations: different decompositions of the set of classes $\{c_1..c_9\}$ into 3 packages $\{p_1,p_2,p_3\}$. 16
2.2 Explanation of Package Cycles within *Modularization*$_1$ (Figure 2.1 (p. 16)). 19
2.3 An Example of Subsystem Notation: the subsystem $subsystem_1$ contains two packages (pkg_1 and pkg_2) and one subsystem ($subsystem_2$). 21
2.4 Different package configurations over the same number of classes. . . 22
2.5 An example illustrating the propagations of package change impact. . 27
2.6 An example illustrating distinct impacts when optimizing package structure. 28
2.7 An Example Illustrating the Kiviat Diagram [*Pinzger et al.*, 2005]. 31

3.1 Consider P1 that references four classes in three other packages (a). A blueprint shows the surfaces of the observed package as stacked subdivisions (b). Small boxes represent classes, either in the observed package (right white part) or in referenced packages (left gray part) (c). 42
3.2 Surface package blueprint detailed view (Outgoing Reference view for P1). 43
3.3 To distinguish it from the outgoing reference blueprint (left), we rotate the incoming reference blueprint (right) by 90°, so that the important details are still read first; in the incoming view, the references are made by the external classes, at the top, to the internal classes below them. . 45
3.4 Package blueprint detailed view (Incoming Reference view for P3 Figure 3.2 (p. 43)) 45
3.5 Inheritance package blueprint. Orange bordered classes inherit directly from external classes. 46
3.6 Analyzing the Network::Kernel Package. 47
3.7 Interacting with package blueprint: using the mouse and pointing at the box shows, through a fly-by-help, the class and package names. In this view, the mouse is pointing to the box representing HTTPSocket and the fly-by-help shows, in addition to the class name, the name of Network::Kernel classes that refer to HTTPSocket. 50

3.8 Outgoing reference blueprints of some packages of the Network system. In this view, the Kernel package was selected in orange, surfaces with Protocols package are highlighted in yellow, class HTTPSocket in red, class SocksSocket in blue, class InternetConfiguration in green, and class Password in fuchsia. 51
3.9 Incoming Reference global view in Network system. In this view, the TelNetWordNet package was selected in orange, surfaces with RemoteDirectory package are highlighted in green. 53
3.10 Inheritance global view in Network system 53
3.11 A Sumo Blueprint: the Critics package in ArgoUML. The view is in the context of the subsystem argoUML::uml. 57
3.12 A Loner Blueprint: Url::Tests and MailSending packages in Network. 58
3.13 A Tower Blueprint: the Peer package in Azureus. The view is in the context of the subsystem azureus::ui::swt::views. 58
3.14 Global view in Compiler system. In this view, the class Parser is highlighted in green, the class ParseNode in red and the class DecompilerConstructor in blue. 60

4.1 Terminology – An example of references between packages 71
4.2 Grouping incoming and outgoing references into In- and Out- interfaces. 71
4.3 The Incoming Fingerprint skeleton with P_1 (Figure 4.2(a) (p. 71)). 73
4.4 Showing the Incoming Fingerprint of P_1 (Figure 4.3 (p. 73)) with the classes involved in the relations inside each cell. 75
4.5 The Incoming Fingerprint of the package render::renderer, from the theme subsystem of Jboss. 77
4.6 The Incoming Fingerprint of renderer package (Figure 4.5 (p. 77)) zoomed-out twice. 79
4.7 Interacting with the Fingerprint. 80
4.8 The Incoming Fingerprint of utils package, from plugins subsystem (Azureus Application). 81
4.9 P_1 Outgoing Fingerprint skeleton (Figure 4.2(b) (p. 71)). 82
4.10 Showing the Outgoing Fingerprint of P_1 (Figure 4.2(b) (p. 71)) with the classes involved in the relations inside each cell. 83
4.11 The Outgoing Fingerprint of impl::api::user package, from the subsystem Jboss.portal.core. 84
4.12 Examples of *Black Fill* Fingerprints. 85
4.13 An example of the *Black-White* pattern: the Incoming Fingerprint of invocation package, from Jboss system. 87
4.14 Arrow Pattern: the Incoming Fingerprint of UI package of the Squeak38::Monticello subsystem. 88
4.15 Variations of *Arrow* pattern. 89
4.16 An example of the *Mosaic* pattern: the Incoming Fingerprint of Morphic::Basic package, from Squeak38 system. 92
4.17 An example of the *Mosaic* pattern: the Incoming Fingerprint of model package, from Argouml system. 93

5.1 Package size and cohesion into *ArgoUML original* (dark gray) and *resulting* (light gray) modularizations. Packages have the same order in diagrams. 113
5.2 Package size, cohesion quality (*CohesionQ*) and cyclic dependency quality (*CyclicDQ*) into *ArgoUML original* (dark gray) and *resulting* (light gray) modularizations. Packages have the same order in diagrams. The constraints are: (1) the size of packages that entail more than 35 should not increase ($size_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*). 117

List of Tables

5.1 Information about used software applications. 113
5.2 Package Quality in original modularizations 115
5.3 Optimizations on Inter-Package Connectivity. The top table shows the percent of reduction of IPD, .. , IPCC (Table 5.1 (p. 113)) into resulting modularizations. The biggest negative value is, the best optimization is. The bottom table shows these information when $distance_{max}$ is specified and limited to 5%. 115
5.4 Modifications on Package Size. The top table shows the percent of empty packages (Table 5.1 (p. 113)), the biggest and the average package size into resulting modularizations. The bottom table shows these information when $distance_{max}$ is specified and limited to 5%. . . 116
5.5 Optimizations on Package quality. The top table shows the average optimizations on package quality into resulting modularizations. Values are based on Table 5.2 (p. 115). The biggest positive value is, the best optimization is. The bottom table shows these information when $distance_{max}$ is specified and limited to 5%. 116
5.6 Resulting Modularization Consistency. Table shows the average distance between ten resulting modularizations for each application. 116
5.7 Modifications on Package Size for *ArgoUML*. It shows the percentage of empty packages (Table 5.1 (p. 113)), the biggest and the average package size into resulting modularizations. The constraints are: (1) the size of packages that entail more than 35 should not increase (*size*$_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*). 118
5.8 Optimizations on Package Quality for *ArgoUML*. Values are based on Table 5.2 (p. 115). The biggest positive value is, the best optimization is. The constraints are: (1) the size of packages that entail more than 35 should not increase (*size*$_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*). 118
5.9 Optimizations on Inter-Package Connectivity for *ArgoUML*. It shows the percentage of reduction of IPD, .. , IPCC (Table 5.1 (p. 113)) into resulting modularizations. The biggest negative value is, the best optimization is. The constraints are: (1) the size of packages that entail more than 35 should not increase (*size*$_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*). 118

Abstract

To cope with the complexity of large object-oriented software systems, developers organize classes into subsystems using the concepts of module or package. Such modular structure helps software systems to evolve when facing new requirements. The organization of classes into packages and/or subsystems represents the software modularization. the software modularization usually follows interrelationships between classes. Ideally, packages should to be loosely coupled and cohesive to a certain extent. However, Studies show that as software evolves to meet requirements and environment changes, the software modularization gradually drifts and looses quality. As a consequence, the software modularization must be maintained. It is thus important to *understand*, to *assess* and to *optimize* the organization of packages and their relationships.

Our claim is that the maintenance of large and complex software modularizations needs approaches that help in: (1) understanding package shapes and relationships; (2) assessing the quality of a modularization, as well as the quality of a single package within a given modularization; (3) optimizing the quality of an existing modularization.

In this thesis, we concentrate on three research fields: software visualizations, metrics and algorithms. At first, we define two visualizations that help maintainers: (1) to understand packages structure, usage and relationships; (2) to spot patterns; and (3) to identify misplaced classes and structural anomalies. In addition to visualizations, we define a suite of metrics that help in assessing the package design quality (*i.e.*, package cohesion and coupling). We also define metrics that assess the quality of a collection of inter-dependent packages from different view points, such as the degree of package coupling and cycles. Finally, we define a search-based algorithm that automatically reduces package coupling and cycles only by moving classes over existing packages. Our optimization approach takes explicitly into account the original class organization and package structure. It also allows maintainers to control the optimization process by specifying: (1) the maximal number of classes that may change their packages; (2) the classes that are candidate for moving and the classes that should not; (3) the packages that are candidate for restructuring and the packages that should not; and (4) the maximal number of classes that a given package can entail.

The approaches presented in this thesis have been applied to real large object-oriented software systems. The results we obtained demonstrate the usefulness of our visualizations and metrics; and the effectiveness of our optimization algorithm.

Résumé

Pour faire face à la complexité des grands systèmes logiciels orientés objets, les programmeurs organisent les classes en sous-systèmes en utilisant les concepts de module ou de package. Une telle structure modulaire permet aux systèmes logiciels d'évoluer face aux nouvelles exigences. L'organisation des classes dans des packages et / ou sous-systèmes, que nous appelons la modularisation du logiciel, suit habituellement les relations entre les classes. Il est de usage de vouloir les packages faiblement couplés et assez cohésifs. Cependant, les études montrent que quand les systèmes logiciels s'adaptent aux exigences et aux modifications de l'environnement, leurs modularisations dérivent et perdent progressivement leur qualité. En conséquence, la modularisation des systèmes logiciels doit être maintenue. Il est donc important de *comprendre*, d'*évaluer* et d'*optimiser* l'organisation des packages et de leurs relations.

Le point défendu dans la thèse est que le maintien des modularisations logiciels de grande taille et complexes requiert des approches qui contribuent à: (1) la compréhension des packages et de leurs relations; (2) l'évaluation de la qualité d'une modularisation, ainsi que la qualité d'un package dans le contexte d'une modularisation donnée; (3) l'optimisation de la qualité d'une modularisation existante.

Dans cette thèse, nous nous concentrons sur trois domaines de recherche: visualisations de programmes, métriques et algorithmes. Dans un premier temps, nous définissons deux visualisations qui aident les mainteneurs à: (1) la compréhension de la structure des packages, et de leurs utilisations et leurs relations; (2) l'identification des modèles; et (3) l'identification des anomalies structurelles. En plus de visualisations, nous définissons un ensemble de métriques qui aident à évaluer la qualité d'un package (*i.e.*, la cohésion et le couplage). Nous définissons également des métriques qui permettent d'évaluer la qualité d'une collection des packages inter-dépendants. Ceci en prenant en compte le degré de couplage et de cycles entre les packages. Enfin, nous définissons une algorithme de recherche qui réduit automatiquement le couplage et les cycles entre les packages, en déplaçant seulement les classes sur les packages existants. Notre approche d'optimisation prend explicitement en compte l'organisation des classes et la structure originale des packages. Il permet également aux mainteneurs de contrôler le processus d'optimisation en spécifiant: (1) le nombre maximal des classes qui peuvent changer leurs packages; (2) les classes qui sont candidates pour se déplacer et celles qui ne doivent pas changer leurs packages; (3) les packages qui sont candidates pour la restructuration et ceux qui ne doivent pas se changer; et (4) le nombre maximal des classes qu'un package donné peut contenir.

Les approches présentées dans cette thèse ont été appliquées à des systèmes logiciels orienté objets, réels et de grand taille. Les résultats obtenus démontrent l'utilité de nos visualisations et métriques, et l'efficacité de notre algorithme d'optimisation.

To my parents...

Acknowledgments

Je vous remercie tous.

Chapter 1

Introduction

1.1 Context: Object-Oriented Software Modularization

> *"Modularity is the single attribute of software that allows a program to be intellectually manageable.* Myers *1978"*

To cope with the complexity of large object-oriented software systems, programmers organize classes into subsystems using the concepts of module or package. The organization of classes into packages and/or subsystems represents the software modularization. The software modularization usually follows interrelationships between classes, that the developers would like to maintain over the ineluctable software system evolution.

Already, from more than thirteen years, researchers, such as Stevens, Myers and Constantine during their studies on the design of software structure [*Stevens et al.*, 1974] and Parnas during his studies on the criteria to decompose software systems [*Parnas*, 1972], have observed that the programs which are composed of simple and independent modules are easier to implement and maintain. In that respect, researchers in object-oriented programming, such as Brian, Fowler and Martin, have found that: *ideally, packages should keep as less coupling and as much cohesion as possible* [*Briand et al.*, 1999a; *Fowler*, 2001; *Martin*, 2002a; *Ponisio* and *Nierstrasz*, 2006; *Ponisio*, 2006], claiming that a good organization of classes into identifiable and collaborating subsystems eases the understanding, maintenance, test and evolution of software systems [*Abreu* and *Goulao*, 2001; *DeRemer* and *Kron*, 1976; *Myers*, 1978; *Ponisio* and *Nierstrasz*, 2006; *Pressman*, 1994; *Yourdon*, 1979].

In the context of this thesis, we define the concepts *Package* and *Modularization* as follows:

Definition **1 (Package)** *Package is the name we use for a class container in object-oriented software systems (e.g., Package in Smalltalk VW*[1]*, Package/Namespace in Java* [Flanagan, *1999]).*

Definition **2 (Modularization)** *Modularization is the name we use to refer to class organization into packages, in the context of a given object-oriented software system.*

[1] For more information see: http://www.cincomsmalltalk.com/userblogs/cincom/blogView

> *"A central feature of the evolution of large software systems is that change – which is necessary to add new functionality, accommodate new hardware and repair faults – becomes increasingly difficult over time.* Eick et al. *2001"*

The studies of Eick [*Eick et al.*, 2001] have proved 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 inevitably become more complex and their modularization drifts and looses quality [*Lehman* and *Belady*, 1985]. As a consequence, the software modularization must be maintained. In that respect, it is then inevitably important to *understand*, to *assess* and to *optimize* the concrete organization of packages and their relationships.

It is worth to note that the maintenance of a software modularization should not change the concerned software behavior. Such a task should be done through *perfective changes* which are intended to improve the software adaptability and changeability without altering its functionality [*Eick et al.*, 2001; *Feathers*, 2005]. In our context (software modularization), the optimization of a software modularization is widely known as software *re-modularization*: *i.e.*, the *re*-organization of the software classes into packages.

Over the last fifteen years of research effort, researchers in the software engineering area have proposed several approaches for automatically re-modularizing software systems [*Abreu* and *Goulao*, 2001; *Bauer* and *Trifu*, 2004; *Doval et al.*, 1999; *Harman* and *Hierons*, 2002; *Liu et al.*, 2001; *Lung et al.*, 2006; *Lutz*, 2001; *Maini et al.*, 1994; *Mancoridis* and *Mitchell*, 1998; *Mancoridis et al.*, 1999; *Mitchell* and *Mancoridis*, 2006, 2008; *Mitchell et al.*, 2004; *Seng et al.*, 2005; *Serban* and *l. G. Czibula*, 2007]. These approaches, while valuable, are not really satisfactory. We believe that the main problem of these approaches is that they often produce new modularizations that are completely different for the original ones. In such a case, it can be difficult for a software engineer to understand the resulting structure and to map it back to the situation he knows.

Another problem that those re-modularization approaches face, is that the existing approaches for understanding the structure of packages and software modularization, [*Ducasse et al.*, 2005b; *Laval et al.*, 2009; *Lungu et al.*, 2006; *Martin*, 2002a; *Ponisio* and *Nierstrasz*, 2006; *Ponisio*, 2006], fall short of providing a fine-grained view of packages that would help maintainers understand the re-modularization efforts: *i.e.*, (1) understand package structure and interrelationships; (2) identify package roles within a system; and (3) mapping back alternative modularizations to original ones. Another problem that maintainers face: the lack of a real investigation in the evaluation of the software modularization quality. Computing the software modularization quality is particularly important when re-modularizing software systems.

1.2 Problem: Software Re-Modularization Challenges

The maintenance of legacy software systems is a well-known challenge that the software industry faces [*Demeyer et al.*, 2002]. Most industrial object-oriented software

systems are large and complex. A reason of their complexity is that the source code is composed of a large collection of inter-dependent classes that mutually co-operate to achieve some desired services. Classes and their relationships represent the static structure of the software system and maintainers need understanding this structure for maintaining and upgrading software systems. For large and complex object-oriented software systems, maintainers are overwhelmed by the large number of classes and the high degree of inter-class dependencies.

In this thesis, we focus on object-oriented software systems since: many current software systems are being (or already) implemented in object-oriented languages (*e.g.*, Java, C++ and Smalltalk); as a consequence, those systems will represent future legacy software systems to maintain.

Object oriented languages, such as Java, Smalltalk and C++, provide the notion of packages to support the decomposition of software systems into subsystems [*Martin*, 1996, 2000]. The ultimate goal of packages is supporting the *information-hiding* criterion for the decomposition of software systems into modules (*i.e.*, packages and subsystems) [*Parnas*, 1972; *Sullivan et al.*, 2001]. The idea is to improve the quality of software, *e.g.*, adaptability and changeability, by decoupling design elements that are likely to change so that they can be changed independently: the organization of classes into identifiable, collaborating and loose-coupled subsystems is the way for easily understanding, maintaining, testing and evolving large object-oriented software systems [*DeRemer* and *Kron*, 1976; *Myers*, 1978; *Pressman*, 1994; *Yourdon*, 1979].

However, software evolves over time with the modification, addition and removal of new classes, methods, functions, dependencies. A consequence is that some classes may not be placed in suitable packages and the software modularization is broken [*Eick et al.*, 2001; *Griswold* and *Notkin*, 1993]. To improve the quality of software modularization, optimizing the class organization into packages is required. There, a well known problem is that, often there is no ideal modularization: often, there are a variety of possible modularizations that are based on a variety of decomposition criteria. In addition, the decomposition criteria are some times conflicting.

The following subsections introduce the software re-modularization challenges.

1.2.1 The Problem of Understanding Packages

Software re-modularization, as a software maintenance task, is done by maintainers. Maintainers who need understanding the software structure and the class organization into packages: (1) to take decisions –before changing the software modularization– and also (2) to understand the software modularization and assessing its quality –after the software re-modularization process. Otherwise, maintainers may introduce anomalies and breakdown the software modularization, they thus may loose the client's trust [*Feathers*, 2005].

Packages, however, are not mere class containers. Packages are complex entities that represent code ownership, feature containment, team organization, deployment entities [*Abreu* and *Goulao*, 2001]. Packages provide or require services and they can play different roles, some central to the system, others peripheral [*Ducasse et al.*, 2007]: some packages act as reference hubs, others as authorities. Packages have different usage patterns, often depending on the clients that use them [*Abdeen et al.*, 2008].

These multiple facets of packages do not ease the understanding and the maintenance of inter-package relationships nor even quick identification of a package clients or providers.

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

Many approaches, mainly based on visualization techniques [*Healey*, 1992; *Healey et al.*, 1995; *Tufte*, 1997; *Ware*, 2000], have flourished to facilitate the construction of a mental picture of the software structure [*Dong* and *Godfrey*, 2007; *Ducasse et al.*, 2005b; *Langelier et al.*, 2005; *Lanza*, 2003; *Laval et al.*, 2009; *Lungu et al.*, 2006; *Sangal et al.*, 2005; *Storey et al.*, 1997; *Wettel* and *Lanza*, 2007a,b; *Wysseier*, 2005]; to show how properties are spread in a population of packages [*Ducasse et al.*, 2006a,b]; to identify software bugs and to understand software evolution [*D'Ambros* and *Lanza*, 2006a,b,c, 2007; *D'Ambros et al.*, 2006; *Lanza*, 2001]. Other approaches have also used metrics to assess software design quality [*Lanza* and *Ducasse*, 2002; *Laval et al.*, 2008; *Martin*, 2002a; *Pinzger et al.*, 2005; *Ponisio* and *Nierstrasz*, 2006; *Ponisio*, 2006]; to assess the effort of maintaining package structure [*Hautus*, 2002].
Few of these approaches are for addressing the problem of package understanding [*Ducasse et al.*, 2005b; *Laval et al.*, 2009; *Lungu et al.*, 2006; *Martin*, 2002a; *Ponisio* and *Nierstrasz*, 2006; *Ponisio*, 2006].
These approaches, while valuable, fall short of providing a fine-grained view of packages that would help maintainers understand the re-modularization efforts: (1) understanding package structure and size; (2) understanding package interrelationships; (3) identifying misplaced classes and structural anomalies; (4) identifying the functionalities that a package provides and/or requires; (5) identifying package roles within a system, etc.
In that respect, we reveal the following question that we want to address in this dissertation:

Research Question:

What do the maintainers need to better understand package structure, and to better identify structural anomalies?

1.2.2 The Problem of Modularization Optimization

Another problem that maintainers face when maintaining packages and looking for a good software modularization is that, *often, there is no ideal modularization*. The organization of software classes is usually based on a variety of (sometimes conflicting) criteria. Mitchell's studies show that software modularization is a graph partitioning problem [*Mitchell*, 2002; *Mitchell* and *Mancoridis*, 2008; *Mitchell et al.*, 2004], which is known to be a NP-hard problem [*Farrugia*, 2004]. As a consequence, searching for good modularization using deterministic procedures or exhaustive exploration of the search space is not feasible without additional heuristics [*Chapman et al.*, 2001; *Mitchell* and *Mancoridis*, 2008]. This is also the case of optimizing package structure in large and complex object-oriented software systems because of the following reasons:

1. Large software systems contain thousands of heavily inter-dependent classes. Many of the dependencies are between classes belonging to different packages, which increases package coupling and, as a consequence, negatively impacts the propagation of package changes.

2. Optimizing some modularization criteria may degrade others. For example, increasing package cohesion by putting a very large number of classes in one package effectively degrades the modularization quality: a software modularization that consists of one package has no usefulness in software understanding and maintenance.

3. In large software systems, classes are usually not well distributed over packages and most software packages depend on some packages that contain a large set of software classes.

4. Furthermore, it is difficult to determine an ideal number of classes that a package may entail, since that number may depend on external factors such as the team structure, domain or coding practice.

Over the last fifteen years of research effort, many approaches have been proposed for automatically re-modularizing software systems. They are mainly based on clustering [*Abreu* and *Goulao*, 2001; *Anquetil* and *Lethbridge*, 1999; *Bauer* and *Trifu*, 2004; *Lung et al.*, 2006; *Mancoridis* and *Mitchell*, 1998; *Mancoridis et al.*, 1999; *Mitchell*, 2002; *Mitchell et al.*, 2004; *Serban* and *l. G. Czibula*, 2007; *Tzerpo* and *Holt*, 1997] and evolutionary or search-based algorithms [*Doval et al.*, 1999; *Harman* and *Hierons*, 2002; *Harman* and *Tratt*, 2007; *Liu et al.*, 2001; *Lutz*, 2001; *Maini et al.*, 1994; *Mitchell* and *Mancoridis*, 2006, 2008; *Mitchell et al.*, 2004; *Seng et al.*, 2005]. These approaches, while valuable, are not really satisfactory. We think that the main reasons behind their failure to satisfy maintainers are:

1. Those approaches do not take into account the original class organization and package structure. As a consequence, they often produce new modularizations that are completely different for the original ones. In such a case, it can be difficult for a software engineer to understand the resulting structure and to map it back to the situation he knows.

2. Another problem is that most of these approaches have as unique goal to maximize dependencies among classes belonging to the same package. As a consequence, they produce modularizations that satisfy this goal regardless other criteria for the concerned software modularization.

In that respect, we reveal the following question that we want address in this dissertation:

Research Question:

What do maintainers need to better search for alternative modularizations that optimize inter-package coupling and satisfy distinct criteria?

1.3 Our Claim

The research questions, raised above, led to our claim:

> **Thesis:** *To maintain a large and complex software modularization, we need visualizations that help* ***to understand****, the package structure, as well as the structural anomalies. We also need an adapted approach that* ***automatically proposes*** *alternative modularizations that optimize inter-package coupling.*

Understanding package structure. To maintain a large and complex software modularization, maintainers need visualizations that help in understanding the static structure of packages. Since often there is no ideal modularization, but there is a variety of good alternative modularizations, based on a variety of criteria, the visualizations are important to assess the re-modularization results. Such visualizations should help maintainers: (1) understand the relationships among classes that are packaged together or belonging to different packages; (2) spot package use/usage patterns; (3) identify package roles within the concerned system; (4) identify misplaced classes and structural anomalies; (5) identify the functionalities/services that a package provides and/or requires, *etc.*

Optimizing software modularization. The maintenance of a large and complex software modularization needs an approach that automatically proposes alternative modularizations. Such an approach should: (1) optimize the modularization quality, *i.e.*, reduce package coupling and cycles and optimize package cohesion; (2) search good alternative modularization by doing near minimal modification in the original organization of classes/packages; (3) take into account maintainer constraints. The maintainer constraints may concern: the classes that should or should not change their packages; the packages that should not be changed; the maximal modification that the optimization process may do in the concerned modularization.

Assessing software modularization quality. To automatically search for alternative modularizations we need metrics that compute the quality of the modularization, from different perspectives: *e.g.*, (1) inter-package coupling; (2) package cycles; (3) package cohesion. We also need metrics that compute the quality of a single package within the concerned modularization. Such metrics are important to: (1) compute package design quality; (2) compute the impact of changes within the modularization; (3) automatically identify candidate packages for restructuring.

1.4 Contributions

In this dissertation we propose two visualizations, named *Package Blueprint* and *Package Fingerprint* , that help in understanding package features in fine and coarse grained level. To address the problem of assessing software modularization quality, we also propose a suite of metrics that compute the quality of software modularization. In addition, we propose a methodology, using our metrics, to automatically

optimize package structure. All our tools (visualizations, metrics and the optimization algorithm) are implemented over the Moose[2] open-source reengineering environment [*Ducasse et al.*, 2005a], using the programming platform VW Smalltalk. Since Moose software models are based on the FAMIX independent-language meta-model [*Demeyer et al.*, 2001], our tools work for mainstream object oriented programming languages.

We summarize the main contribution of this thesis as follows:

1. *Package Blueprint* [*Ducasse et al.*, 2009, 2007]: a compact visualization revealing, in detail, package structure and dependencies. A package blueprint is structured around the concept of a *surface*, which represents and details the dependencies between the observed package and its provider and client packages. Package Blueprint reveals the overall size and complexity of a package, as well as its relations with other packages: it shows the distribution of dependencies to classes within and outside the observed package.

2. *Package Fingerprint* [*Abdeen et al.*, 2008, 2009a]: a compact, rich and zoomable visualization to better support the understanding of package interfaces, relationships and the co- usage/use of package classes (*i.e.*, class conceptual coupling). The goal of this visualization is to help maintainers during their early contacts with unknown packages. We propose two complementary variants of the Package Fingerprint, structured around the distribution of *use* dependencies from or to the classes of the analyzed package: (1) the *incoming fingerprint* shows how the system uses the package classes, and highlights the cohesion of the analyzed package, as defined by Ponisio [*Ponisio* and *Nierstrasz*, 2006]; (2) the *outgoing fingerprint* shows how the package classes use the system.

3. *Package-Modularization Metric Suite* [*Abdeen et al.*, 2009b]: we define a suite of metrics, based on the principles of package design as described by Martin [*Martin*, 2002a]: Common Closure Principle (CCP), Common Reuse Principle (CRP) and Acyclic Dependencies Principle (ADP). The aim of those metrics is: (1) to automatically assess the quality of a modularization; (2) to automatically assess the quality of a package within a given modularization; (3) to identify candidate packages for restructuring.

4. *Automatically Reducing Package Coupling and Cycles* [*Abdeen et al.*, 2009b]: we present an approach for automatically reducing package coupling and cycles only by moving classes over packages. Our approach takes into account the existing class organization and package structure. In our approach, maintainers can define: (1) the maximal number of classes that can change their packages; (2) the maximal number of classes that a package can contain; (3) the classes that should not change their packages; (4) the packages that should not be changed.

1.5 Structure of the Dissertation

Chapter 2 analyses the problem of the maintenance of large and complex object oriented software modularization. It also sets our terminology, and evaluates

[2]For more information about Moose see: http://moose.unibe.ch/

existing approaches that tried to solve a substantial body of the underlined problem. The result of the survey is summarized in a list of requirements for the maintenance of a large and complex software modularization: (1) understanding package structure; (2) understanding package roles and usage; (3) assessing the quality of a software modularization and packages; (4) automatically optimizing existing software modularization, with respect to distinct principles of package design quality.

Chapter 3 proposes a compact visualization, named *Package Blueprint*. The proposal is to explicitly show the distribution of dependencies to classes within and outside the observed package. The aim of the Package Blueprint is to help maintainers in understanding the overall size and complexity of a package, as well as its relations with other packages/classes. The Package Blueprint visualization is structured around the concept of a *surface*. A *surface* represents and details the dependencies between the observed package and its provider and client packages. In this chapter: (1) we describe and show the utility of the Package Blueprint visualization to analyze a package and its relationships; (2) we identify a set of visual patterns that help to quickly spot the package shape and the class organization.

Chapter 4 proposes a compact, rich and zoomable visualization, named *Package Fingerprint*. The proposal is to help maintainers, during their early contacts with unknown packages, in understanding package interfaces and the co- usage/use of package classes (*i.e.*, class conceptual coupling). We propose two complementary variants of the Package Fingerprint, structured around the distribution of *use* dependencies from or to the classes of the analyzed package: (1) the *incoming fingerprint* shows how the system uses the package classes, and highlights the cohesion of the analyzed package, as defined by Ponisio [*Ponisio* and *Nierstrasz*, 2006]; (2) the *outgoing fingerprint* shows how the package classes use the system. In this chapter, we describe and show the utility of the Package Fingerprint visualization to analyze package coupling and cohesion (from the point of view of its client and provider packages).

Chapter 5 proposes a suite of metrics for automatically assessing package and modularization quality. In this chapter, we also present an approach for automatically optimizing package structure: reducing package cycles and coupling. The optimization approach explicitly takes into account the original organization of classes, and allows maintainers to control the optimization process by defining constraints on possible alternative modularizations.

In **Chapter 6** **(p. 123)** we summarize how our proposals satisfy the requirements identified in Chapter 2 (p. 15) (2.8), for the maintenance of large and complex software modularizations. The chapter ends with an outlook on the opened future work.

Chapter 2

Software Re-Modularisation: Challenges and Approaches

2.1 Introduction

> *"Complexity arises, then, when we have a large system and when the system divides into a number of components that interact with each other in ways that amount to something more than uniform, frequent elastic collisions.* Callebaut *and* Rasskin-Gutman *2005"*

Although packages, as pointed-out in Chapter 1 (p. 7), are important to cope with the complexity of large and complex software, it is frequent to have large object-oriented software systems structured over large number of heavily inter-dependent packages, where the rational behind packages is broken. We mean that, the modularization of such software systems does not respect the *information-hiding* criterion for decomposing software classes into packages [*Parnas*, 1972], where packages should keep as less coupling and as much cohesion as possible [*Briand et al.*, 1999a; *Martin*, 2002a; *Ponisio* and *Nierstrasz*, 2006].

However, even if software systems were originally well modularized, Griswold *et al.* [*Griswold* and *Notkin*, 1993] and Eick *et al.* [*Eick et al.*, 2001] show that, as software systems evolve, to meet requirements and environment changes, their modularization drifts and loose quality and the software systems inevitably become more complex; as a consequence, the software system modularization must be maintained and optimized.

On the other hand, Denker and Ducasse, during their experience while being responsible of the release of Squeak, a large open-source Smalltalk [*Denker* and *Ducasse*, 2007], and Abreu *et al.* in their work on software re-modularization [*Abreu* and *Goulao*, 2001], both point out that *packages are important, but understanding and optimizing package structure present hard challenges*. They underline the fact that, (1) packages play different development roles: *e.g.*, code ownership, feature containment, team organization, deployment entities; (2) packages have also different usage patterns: *e.g.*, packages act as reference hubs, others as authorities [*Abdeen et al.*, 2008; *Ducasse et al.*, 2007]. In addition, optimizing a criterion of package structure may decrease another

criterion: *e.g.*, maximizing package cohesion may break the system deployment by introducing package cycles. Also, since packages are inter-dependent, optimizing the structure of a given package may degrade the structure quality of another one: *e.g.*, optimizing the cohesion of a package p_1 by moving a class c from another package p_2 to p_1 may degrade the cohesion/coupling of p_2.

Mitchell *et al.* consider that software re-modularization problem is a graph partitioning problem [*Mitchell* and *Mancoridis*, 2008; *Mitchell et al.*, 2004], which is known to be a NP-hard problem [*Chapman et al.*, 2001; *Farrugia*, 2004]. As a consequence, searching for good modularization by using deterministic procedures or exhaustive exploration of the search space is not feasible without additional heuristics [*Chapman et al.*, 2001; *Mitchell* and *Mancoridis*, 2008; *Mitchell et al.*, 2004].

In this chapter we define our terminology and explain the software remodularization problem. We then detail and discuss the most relevant existing approaches that have been proposed to address this problem.

2.2 Background and Terminology

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

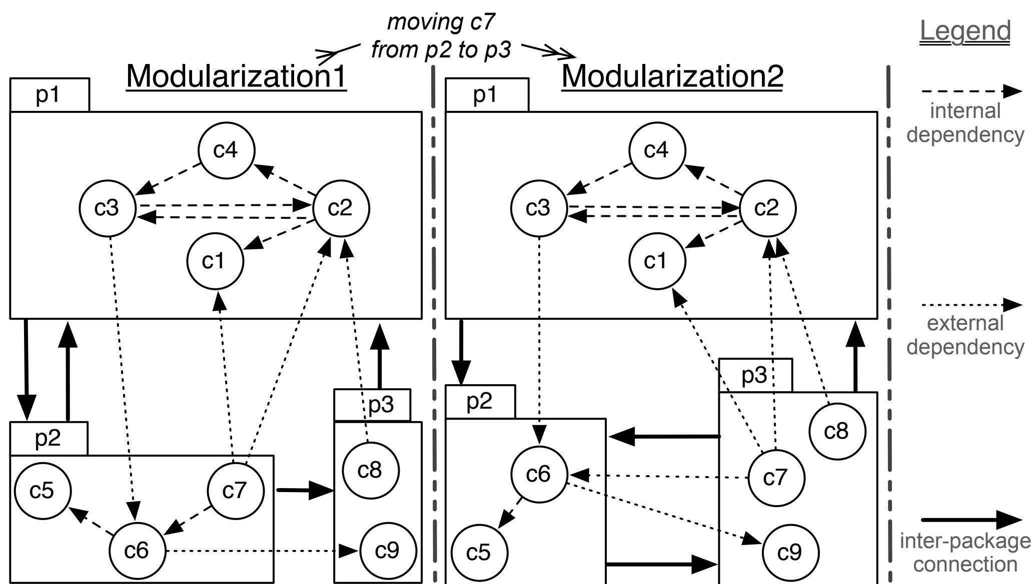


Figure 2.1: Example of two modularizations: different decompositions of the set of classes $\{c_1..c_9\}$ into 3 packages $\{p_1,p_2,p_3\}$.

Modularization. Let C be the set of classes of a given object-oriented software system. We define, then, the software *modularization* $\mathcal{M}$ as a decomposition of C into a set of packages P. In such a context, C represents the modularization classes (denoted $\mathcal{M}_C$) and P represents the modularization packages (denoted $\mathcal{M}_P$).

Figure 2.1 (p. 16) shows two modularizations, both consists of 9 classes ($|\mathcal{M}_C| = 9$) distributed over 3 packages ($|\mathcal{M}_P| = 3$).

Package. We define *Package* as an entity that only contains classes. We denote the set of classes that a given package p contains by p_C. We denote the package of a given class c by c_p. We define *package size* (denoted p_{size}) by the cardinality of the set of its classes: *e.g.*, in Figure 2.1 (p. 16), the size of p_1 is equal to 4.

Inter-class dependencies. Every class c can be related to other classes through a set of *Dependencies* (denoted c_D). This set consists of two subsets: *Outgoing dependencies* (denoted $c_{Out.D}$) and *Incoming dependencies* (denoted $c_{Inc.D}$). We denote a dependency d that goes from a class c_i to another one c_j by the pair (c_i,c_j). In that context, we say that the c incoming dependencies relate c to its client classes (denoted $c_{Cli.C}$); and the c outgoing dependencies relate c to its provider classes (denoted $c_{Pro.C}$).

Examples. In *Modularization*$_1$ (Figure 2.1 (p. 16)), the class c_6 has 4 dependencies: c_{6_D} = {(c_7,c_6) , (c_6,c_5) , (c_3,c_6) , (c_6,c_9)}. Two dependencies are incoming dependencies: $c_{6_{Inc.D}}$ = {(c_7,c_6) , (c_3,c_6)}; and two are outgoing dependency: $c_{6_{Out.D}}$ = {(c_6,c_9) , (c_6,c_5)}. As well, c_6 has two client classes: $c_{6_{Cli.C}}$ = {c_7 , c_3}; and two provider classes: $c_{6_{Pro.C}}$ = {c_9 , c_5}.

Dependency kinds. Of all the kinds of dependencies that we can find in object-oriented code we concentrate on the following kinds of dependencies: *method call*, *class access*, *class inheritance*, or *class extension*. We describe these kinds as follows:

1. *method call:* we say that there is a *method call* dependency that goes from a given class $class_1$ to another one $class_2$ if there is at least one method within $class_1$ that invokes at least one method of $class_2$.
   Note that, in dynamic typed languages (*e.g.*, Smalltalk), we often can not statically determine the class of the invoked method (*i.e.*, the class of the target object in the run-time). Our strategy consists in creating dependencies for every potential candidate class (*i.e.*, every class within it there is a method that has the invoked method signature).
2. *class access:* we say that there is a *class access* dependency that goes from a class $class_1$ to another one $class_2$ if $class_2$ (*i.e.*, the name of $class_2$) is explicitly used in $class_1$ code: *e.g.*, $class_2$ is used within $class_1$ as a type of an instance and/or a class variable; $class_2$ is used within a method of $class_1$ as a variable/parameter type).
3. *class inheritance:* we say that there is a *class inheritance* dependency that exits a class $class_1$ and points to $class_2$ if $class_1$ is a subclass of $class_2$.
4. *class extension:* a class extension is a method defined in a package, for which the class is defined in a different package [*Bergel et al.*, 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.

We say that there is a *class extension* dependency that goes from a package p_i to a class c, which is defined in another package p_j, if a method m of the c methods is defined in p_i.

In the context of this thesis , we use *Reference* as a dependency kind to cover the kinds *method call* and *class access*. As well, we say that a given class $class_1$ *refers to* another class $class_2$ if there is a *method call* or *class access* dependency that goes from $class_1$ to $class_2$. In the same vein, we say that $class_2$ is referenced by $class_1$.

Dependency scope. Every dependency is *internal* if it is related to two classes belonging to the same package. Otherwise, it is *external*.

Examples. In *Modularization*$_1$ (Figure 2.1 (p. 16)) the dependency (c_7,c_6) is internal, while the dependencies (c_7,c_1) and (c_7,c_2) are external.

For a given class c, we denote the set of c internal dependencies by $c_{Int.D}$; and denote the set of c external dependencies by $c_{Ext.D}$.

Package dependencies. The set of dependencies related to a package p (p_D) is the union of the dependency sets of p classes: $p_D = \cup_{c \in p_C} c_D$. In this context, the union of the internal dependency sets of p classes represents p internal dependencies ($p_{Int.D}$): $p_{Int.D} = \cup_{c \in p_C} c_{Int.D}$; and the union of the external dependency sets of p classes represents p external dependencies ($p_{Ext.D}$): $p_{Ext.D} = \cup_{c \in p_C} c_{Ext.D}$.

The set $p_{Ext.D}$ consists of two subsets: dependencies that are either exiting p ($p_{Ext.Out.D}$) or that are pointing to p ($p_{Ext.Inc.D}$). The set $p_{Ext.Out.D}$ relates p to its provider packages, $p_{Pro.P}$; while the set $p_{Ext.Inc.D}$ relates p to its client packages, $p_{Cli.P}$.

Examples. Modularization$_1$ in Figure 2.1 (p. 16) shows that there is one dependency exiting p_1 and three dependencies pointing to p_1: ${p_1}_{Ext.Out.D}$ = {(c_3,c_6)}; ${p_1}_{Ext.Inc.D}$ = {(c_7,c_1) , (c_7,c_2) , (c_8,c_2)}. It also shows that p_1 has one provider package and two client packages: ${p_1}_{Pro.P}$ = {p_2}; ${p_1}_{Cli.P}$ = {p_2 , p_3}.

In the context of this thesis , when we say that a package p_i depends on a class c_j, or c_j is imported by p_i, we mean that classes contained in p_i depend on c_j: *e.g.*, in Figure 2.1 (p. 16), *Modularization*$_1$ shows that p_3 depends on c_2 and c_2 is imported by p_3 and p_2; while *Modularization*$_2$ shows that p_3 depends on c_1, c_2 and c_6, where this last, c_6, depends on p_3.
In the same vein, we say that the p_3 provider classes (${p_3}_{Pro.C}$) are c_1, c_2 and c_6, where p_3 is also a provider package for c_6.

Package interfaces. In the context of a given package, we mean by *Package Interfaces* the set of the package classes that are visible to the rest of the system: *i.e.*, the package classes that communicate with the rest of the system via external incoming and/or outgoing dependencies. In that respect, we define package *In-Interface* and *Out-Interface* by the following:

Definition **3 (In-Interface)** *The In-Interface of a package p is the set of classes of p that have external incoming dependencies.*

$$p_{In-Interface} = \{c \mid c \in p_C \land c_{Ext.Inc.D} \neq \emptyset\}$$

Definition **4 (Out-Interface)** *The Out-Interface of a package p is the set of classes of p that have external outgoing dependencies.*

$$p_{Out-Interface} = \{c \mid c \in p_C \land c_{Ext.Out.D} \neq \emptyset\}$$

As well, when we talk about *reference* dependencies, we then talk about package reference interfaces. Also, we mean by In-Interface (Out-Interface) size, the number of classes that are involved in that In-Interface (Out-Interface).

Connectivity in package level. As mentioned by Martin [*Martin*, 2000], importing a class implies importing the complete package. Therefore importing two classes from the same package is quite different from importing them from two different packages –since in the latter case we import all the classes of the two packages.

To determine connectivity at the package level (*i.e.*, package interactions), we say that there is a *Connection* that exiting a package p_i and points to another one p_j if there is n ($n > 0$) dependencies of the set $p_{i_{Ext.Out.D}}$ pointing to p_j. In other words, if p_i is a client package of p_j then there is a *connection* from p_i to p_j. We denote such a connection by the pair (p_i,p_j). In this context, two packages, p_i and p_j, can have at maximum two connections among them: (p_i,p_j) and (p_j,p_i).

We denote the set of connections related to a package p by p_{Con}. This set consists of two subsets: connections that are either exiting p, *Outgoing connections*, ($p_{Out.Con}$) or that are pointing to p, *Incoming connections*, ($p_{Inc.Con}$).

Examples. Modularization$_1$ in Figure 2.1 (p. 16) shows that there is one connection exiting p_1: $p1_{Out.Con} = \{(p_1,p_2)\}$; and there are two connections pointing to p_1: $p1_{Inc.Con} = \{(p_2,p_1)\ ,\ (p_3,p_1)\}$; *Modularization*$_2$ shows that the connection (p_2,p_1) does not exist any more.

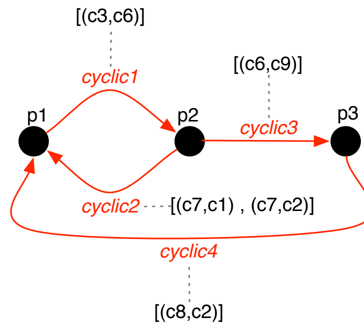


Figure 2.2: Explanation of Package Cycles within *Modularization*$_1$ (Figure 2.1 (p. 16)).

Package cycles. Dependencies can form *cyclic connections* between packages. Figure 2.2 (p. 19) shows the connection graph of p_1, p_2 and p_3 within *Modularization*$_1$ (Figure 2.1 (p. 16)). We note that within this graph, all the connections represent cyclic-connections. We distinguish two categories of cyclic connections:

- *Indirect cyclic-connections:* the connections (p_1,p_2), (p_2,p_3) and (p_3,p_1) represent indirect cyclic connections between p_1 and p_3 –since p_1 depends, *indirectly* (via p_2), on p_3 and p_3 depends on p_1.
- *Direct cyclic-connections:* the connections (p_1,p_2), (p_2,p_1) represent direct cyclic connections between p_1 and p_2 –since p_1 and p_2, both depend, *directly*, on each other.

In that context, we say that a dependency d represent a *cyclic dependency* between two packages $package_i$ and $package_j$ if d takes a part of a cyclic connection between $package_i$ and $package_j$.

Example 1, in *Modularization*$_1$ (Figure 2.2 (p. 19)), since the connection (p_1,p_2) represents an indirect cyclic connection between p_1 and p_3, thus all the dependencies that exiting p_1 and pointing to p_2, which are {(c_3,c_6)}, represent indirect cyclic dependencies between p_1 and p_3; similarly, we find that the dependencies (c_6,c_9) and (c_8,c_2) represent indirect cyclic dependencies between p_1 and p_3.

Example 2, in *Modularization*$_1$ (Figure 2.2 (p. 19)), since the connection (p_2,p_1) is a direct cyclic connection between p_2 and p_1, thus all the dependencies that exiting p_2 and pointing to p_1, which are {(c_7,c_1) , (c_7,c_2)}, represent direct cyclic dependencies between p_2 and p_1; similarly, we find that the dependency (c_3,c_6) is also a direct cyclic dependency between p_2 and p_1.

In the context of this thesis , when we say *cyclic* connection/dependency we mean that it is a *direct cyclic* connection/dependency.

We denote the set of cyclic dependencies related to a package p by $p_{Cyc.D}$. The set $p_{Cyc.D}$ consists also of two subsets: cyclic-dependencies that are either exiting p ($p_{Out.Cyc.D}$) or pointing to p ($p_{Inc.Cyc.D}$).
Note that, $p_{Out.Cyc.D}$ and $p_{Inc.Cyc.D}$ are dependencies causing cycles between packages (and not classes) in the context of the client-provider relation.
Similarly, we denote the set of cyclic connections related to p by $p_{Cyc.Con}$: $p_{Cyc.Con} = p_{Out.Cyc.Con} \cup p_{Inc.Cyc.Con}$.

Example, Modularization$_1$ in Figure 2.1 (p. 16) shows that p_1 has one outgoing cyclic-dependency {(c_3,c_6)} and two incoming cyclic-dependencies {(c_7,c_1) , (c_7,c_2)}. Those cyclic-dependencies produce two cyclic-connections: ${p_1}_{Cyc.Con}$ = ${p_2}_{Cyc.Con}$ = {(p_1,p_2) , (p_2,p_1)}.

Subsystem definition. We mainly define *Subsystem* as a collection of packages. To cover the *nesting* notation, a subsystem can also contains other subsystems, but, unlike the case of *Java Packaging notation* [*Flanagan*, 1999], a subsystem does not contain classes: classes are defined only within packages.

For example, Figure 2.3 (p. 21) shows that the subsystem $subsystem_1$ contains two packages (pkg_1 and pkg_2) and one subsystem ($subsystem_2$). It shows that the subsystem $subsystem_2$ contains also two packages (pkg_3 and pkg_4).

In the context of this thesis , when we talk about *subsystem packages*, we then talk about all the packages that are in the scope of that subsystem: *e.g.*, in Figure 2.3 (p. 21), the packages of $subsystem_1$ are pkg_1, pkg_2, pkg_3 and pkg_4.

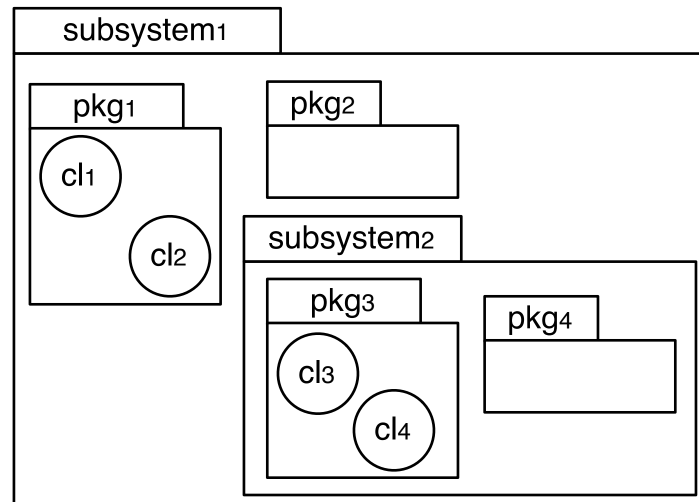


Figure 2.3: An Example of Subsystem Notation: the subsystem $subsystem_1$ contains two packages (pkg_1 and pkg_2) and one subsystem ($subsystem_2$).

Similarly, when we talk about *subsystem classes*, we then talk about all the classes of its packages.
By convention, to print *nested package (subsystem) name* we use the symbol (::) for representing the nesting path: *e.g.*, in Figure 2.3 (p. 21), the complete name of pkg_4 is $subsystem_1$::$subsystem_2$::pkg_4.

In the next section we illustrate the challenges of understanding packages.

2.3 Package Understanding

Actually, packages reflect several organizations: they are units of code deployment or units of code ownership; they can also encode team structure, architecture and stratification. A package can interact with other ones in several ways: either as a provider, or as a consumer or both. In addition some packages may have either a lot of references to other packages or only a couple of them. If a package defines subclasses, those can form either a flat or deep subclass hierarchy. A package may define domain abstraction (*i.e.*, containing the top super classes, abstract classes, of the concerned domain), or domain implementer (*i.e.*, containing classes that inherit and specialize domain definer classes), ... All those multiple facets of packages led us to say that understanding and maintaining packages represent crucial and hard challenges.

Note that, understanding packages is also important in the context of automatic remodularization approaches, *e.g.*, [*Anquetil* and *Lethbridge*, 1999; *Mancoridis et al.*, 1999; *Mitchell* and *Mancoridis*, 2006; *Wiggerts*, 1997]. There it is important to understand how the proposed remodularisation compares with the existing code (*i.e.*, the original

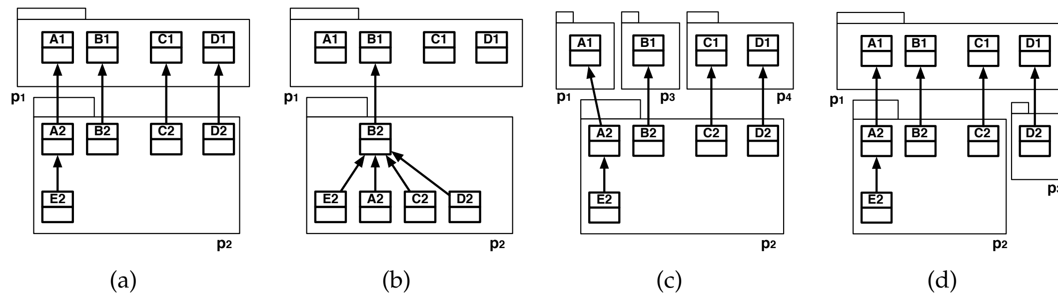


Figure 2.4: Different package configurations over the same number of classes.

modularization). This problem is particularly stressed in presence of large legacy software systems that consist of thousands of classes and hundreds of packages.

Figure 2.4 (p. 22) shows different situations involving the same group of classes. For illustrating purpose, Figure 2.4 (p. 22) only shows inter-class references; the same idea holds for inheritance between classes contained in different packages. In both cases Figure 2.4(a) (p. 22) and Figure 2.4(b) (p. 22), there are only two packages but in case Figure 2.4(a) (p. 22) most of the classes of p_2 reference a class in p_1, while in case Figure 2.4(b) (p. 22) most classes of p_2 reference internally the class B2. Revealing this difference is important to the maintainer who wants to understand if s/he can change the relationships between p_1 and p_2 during a refactoring process. In cases Figure 2.4(a) (p. 22) and Figure 2.4(c) (p. 22), we have exactly the same relationships between classes but the package structure is different. As mentioned by Martin, [*Martin*, 2000, 2002a], importing a class equals importing the complete package, therefore importing two classes from the same package is quite different from importing them from two different packages since in the latter case we import all the classes of the two packages and we increase the number of inter-connected packages within the concerned modularization.

Although languages such as Java offer a mechanism for modelling the dependencies between packages (*i.e.*, via the import statement), this mechanism does not really help to understand the structure of a given package: *e.g.*, package internal and external dependencies; package interfaces; *etc.*

On the other hand, up to now, there are few works that illustrate and tackle the problem of understanding package structure and relationships. Unfortunately, all what we can find about package design in the literature is that good packages should be self-contained, or only have a few clear dependencies to other packages [*Arisholm et al.*, 2004; *Briand et al.*, 1999a; *Lanza* and *Marinescu*, 2006]. In addition to that, Martin [*Martin*, 2000, 2002a], has recently provided and generally discussed other package design principles related to package abstraction, stability and cyclic dependencies (connections).

In the following subsections, we present a coarse list of useful pieces of information that maintainer may need to understand packages. Our goal here is to identify the challenges that maintainers are facing and not to define an exhaustive list of the problems that a particular solution should tackle.

2.3.1 Quantitive Information

To understand the packages of a system and their relevance in the general picture, gathering quantitative information is a good way to offer a mental picture to the maintainers [*Lanza* and *Ducasse*, 2003; *Petre*, 1995]. Here is a list of relevant questions:

- What is the general size of a package in terms of classes?
- How many *internal dependencies* that a given package has?
- How many *external dependencies* that a given package has?
- What is the density of the *internal dependencies* of a given class?
- What is the density of the *external dependencies* of a given class?
- How many packages depend upon a given package (*client* packages)?
- How many packages does a given package depend upon (*provider* packages)?
- How many classes, of a given package, are visible to the rest of the system (package *interfaces*)?

These quantitive information are usually used to assess package design quality. The next section explains the principles of package design.

2.3.2 Qualitative Information (cohesion vs. coupling)

Transforming or evolving an application follows natural boundaries defined by coupling and cohesion [*Arisholm et al.*, 2004; *Briand et al.*, 1999a]. The question is then to see the impact (if any) of the transformation on package cohesion and coupling. Assessing these properties is then important.

In that respect, Martin, [*Martin*, 2000, 2002a], introduced some principles for the design of software architecture and package, addressing by those principles the different perspectives of package cohesion and coupling. The package cohesion principles are:

Reuse-Release Equivalence Principle (REP). *The granule of reuse is the granule of release.*
Since packages are the unit of release, they are also the unit of reuse. Therefore a good package should only contain a group of classes that are reusable together.

Common-Closure Principle (CCP). *The classes in a package should be closed together against the same kinds of changes.*
To minimize the number of packages that are changed in any given release cycle, it is better to group classes that change together into the same package.

Common-Reuse Principle (CRP). *If you reuse one of the classes in a package, you reuse them all.*
Since a dependency upon a package is a dependency upon everything within the package, classes that are reused together should be grouped together. This

way, in any given release, changing any class within a considered package will have the same impact-propagation if maintainers change another class within the same package. Thus the impact-propagation of the package changes is always constrained to one graph.

Coupling is always used with cohesion to determine package quality and it is generally defined as: *if changing one package in a program requires changing another package, then coupling between these two packages exists* [*Briand et al.*, 1998; *Fowler*, 2001]. Martin defines two types of coupling: *Efferent Coupling* and *Afferent Coupling* [*Martin*, 2000, 2002a].

Definition **5 (Efferent Coupling)** *A package p_x has an efferent coupling to another package p_y if p_x depends upon p_y.*

Definition **6 (Afferent Coupling)** *A package p_y has an afferent coupling to another package p_x if p_x depends upon p_y.*

In addition to the package cohesion principles cited above, Martin's package coupling principles are:

Acyclic-Dependencies Principle (ADP). *Allow no cycles in the package dependency graph.*
To achieve the package quality which is desired by the package cohesion principles, especially the CRP and CCP, the dependencies between packages must not form cycles.

Stable-Dependencies Principle (SDP). *Depend in the direction of stability.*
A package's stability is related to the amount of work required to make a change on it, it is thus related, in addition to the package internal size and complexity, to the number of other packages which depend on it.

This way, a package with lots of incoming dependencies from other packages is stable —since it is seen as responsible to those packages. In another hand, a package that has not any incoming dependency is considered as independent and very unstable.

Stable-Abstractions Principle (SAP). *A package should be as abstract as it is stable.*
It is clear that the more packages are hard to change, the less flexible the overall design will be. Thus, to improve the flexibility of systems, architects should compose those systems from unstable packages that are easy to change, and stable packages that are easy to extend. In this context, the stable packages should be highly abstract.

Cohesion and coupling metrics are among the most used metrics during perfective maintenance, because they help identify which packages should be restructured [*Abreu* and *Goulao*, 2001; *Arisholm et al.*, 2004; *Briand et al.*, 1999a; *Lanza* and *Marinescu*, 2006; *Melton* and *Tempero*, 2007; *Rising* and *Calliss*, 1992]. In general, good packages should group classes that are needed for the same task [*Ponisio* and *Nierstrasz*, 2006], and they should have a few clear dependencies to other packages: they should

be highly cohesive and lightly coupled. However, cohesion and coupling metrics alone do not help maintainers understand the structure, roles, or relationships of packages. In particular, they do *not* indicate whether, why and how a package respects Martin's cohesion and coupling principles, nor do they help decide what to do if such principles are not respected.

For this, maintainers need more detailed information. For example, it is important to know if some classes in a package are *always used together* or not, and conversely the proportion of package classes that uses the same set of classes/packages. Knowing about the usage relations between a package and its clients and providers offers another perspective on package cohesion, since that gives the maintainer information on the package role and cohesion according to Common Reuse Principle [*Ponisio* and *Nierstrasz*, 2006].

In addition to the qualitative information, maintainers need also to determine package roles in the context of a given software system. The next section reveals the information that maintainers need to determine package roles.

2.3.3 Role and Contextual Information (central vs. peripheral)

To determine package role within the system, two correlated pieces of information are important:

1. does a package belong to the core of an application or is it more peripheral?

2. does a package provide or use functionality?

Answering these questions is mainly based on the package quantitive information, in addition to the package relationships with the rest of the system. To answer the first question (1), we usually need answers for the following questions:

- How many packages use the concerned package?

- Those client packages depend upon how many classes within the concerned package?

- Do those client packages belong to one subsystem? or is its usage dispersed over multiple subsystems and how many ones?

On the other hand, to answer the second question (2), we need also the answers for the questions cited above but from both points of view: package outgoing and incoming dependencies, and connections. In other words, we need measuring the portion between: the package clients (client classes, packages and subsystems) and the package providers (provider classes, packages and subsystems); the package In-Interface size and Out-Interface size.

However, packages are not simply class containers and structural entities. Packages represent also team organization, and maintainers need to understand package organizational information.

2.3.4 Organizational Information (developers vs. team)

Knowing who are the developers and maintainers of the application and packages helps to: (1) understand the architecture of the application; (2) qualify package roles; and, (3) know who among maintainers and developers should communicate during the maintenance tasks [*Gîrba et al.*, 2005; *Pollet et al.*, 2007]. In that respect, Ducasse provided a generic visualization, named Distribution Map [*Ducasse et al.*, 2006a]. The Distribution Map visualization is useful to show how properties are distributed over design elements: *e.g.*, how maintainers and developers are distributed over software classes and packages.

2.4 Challenges in Optimizing Modularization

In addition to the challenges of understanding packages and their interrelationships, the maintenance of software modularizations raises other challenges.

It is actually frequent to have large legacy object-oriented software systems that entail some thousands of heavily inter-dependent classes, distributed over some hundreds packages. In these systems, almost inter-class dependencies are among classes belonging to different packages. In such cases, the rational idea behind software modularization is mostly broken: package interfaces are not even clean, abstraction advantages (*i.e.*, minimizing the coupling between packages and maximizing packages internal cohesion) are not even present. Therefore, software modularization must be optimized.

In the following subsections, we present a coarse list of problems that maintainers face when maintaining the modularization of large legacy object-oriented software systems:

2.4.1 Modularization Complexity

Large software systems are usually very complex, they contain thousands of heavily inter-connected classes. Many of the dependencies are between classes belonging to different packages, which increases the inter-package connectivity. In such situation, the optimization problem is more difficult.

Figure 2.5 (p. 27) illustrates the problem of the maintenance of heavily inter-connected packages. Figure 2.5 (p. 27) (b) shows that modifying the package p_5, which means modifying the subsystem $subsystem_2$ in which p_5 is nested, may directly impact the packages p_3 and p_6 –since p_3 and p_6 depend directly on the modified package p_5. As a consequence, the packages p_1, p_2 and p_7 may, in turn, be also impacted –since these packages depend on the impacted package p_3. Figure 2.5 (p. 27) (b) shows also that the impact propagation to p_6 may be direct and indirect, since p_6 depend directly and also indirectly (via p_7 and p_3) on the modified package p_5.

2.4.2 Class Distribution over Packages

In real software systems, classes usually are not well distributed over packages: some packages contain few number of classes while others contain large sets of classes. As

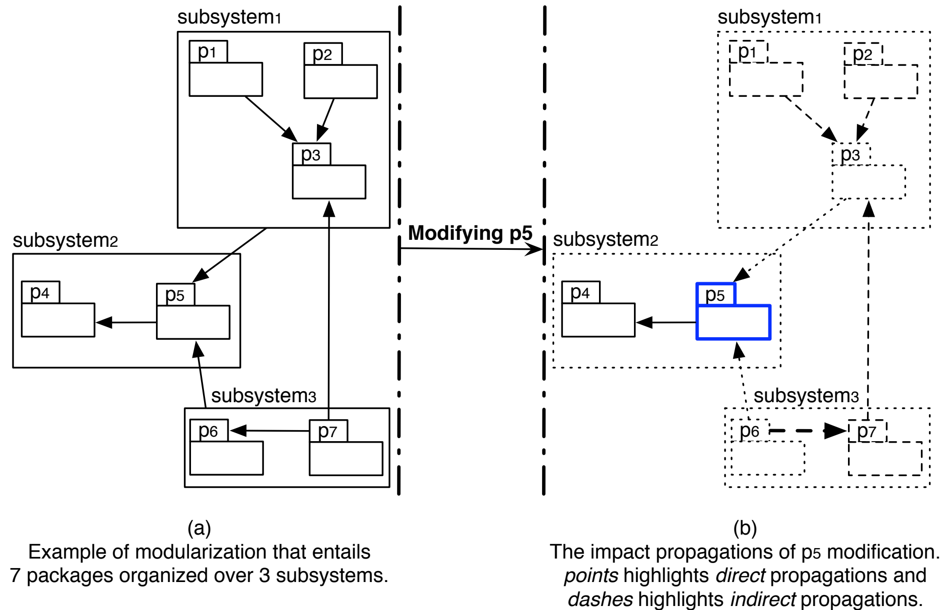


Figure 2.5: An example illustrating the propagations of package change impact.

consequence, most application packages depend on large packages (*i.e.*, dominant packages). In such cases, the reduction of coupling to dominant packages, should not be done by increasing the size of those dominant packages: *e.g.*, removing smaller packages and moving their classes to dominant packages. In that respect, every optimization process should take into account the modification of package size and balances package size with package coupling and cohesion [*Abreu* and *Goulao*, 2001; *Harman* and *Tratt*, 2007].

Furthermore, although researchers like Meyer suggest that the range of package size is 5..40 [*Meyer*, 1989], we believe that it is difficult to determine an ideal package size since it may depend on external factors such as the team structure, domain, or coding practice, *etc.*

2.4.3 Package Optimization Trade-Offs

The multiple facets and roles of packages imply that the optimization process should consider the trade-offs between the distinct features of package design, where optimizing some modularization criteria may degrade other ones. For example, minimizing inter-packages dependencies/connections may increase the number of cyclic ones.

Figure 2.6 (p. 28) shows four modularizations that entail the same set of classes $\{c_1..c_9\}$: the modularizations $Modularization_2$, $Modularization_3$ and $Modularization_4$ are resulting from simple modifications in $Modularization_1$ (moving classes over existing packages):

Example 1. The difference between $Modularization_1$ and $Modularization_2$ is c_7 that is moved from p_2 to p_3 in $Modularization_2$. In $Modularization_2$ there are 2 inter-package cyclic dependencies $\{(c_7,c_6)\ ,(c_6,c_9)\}$ compared to 3 in $Modularization_1$ $\{(c_3,c_6)\ ,(c_7,c_1)$

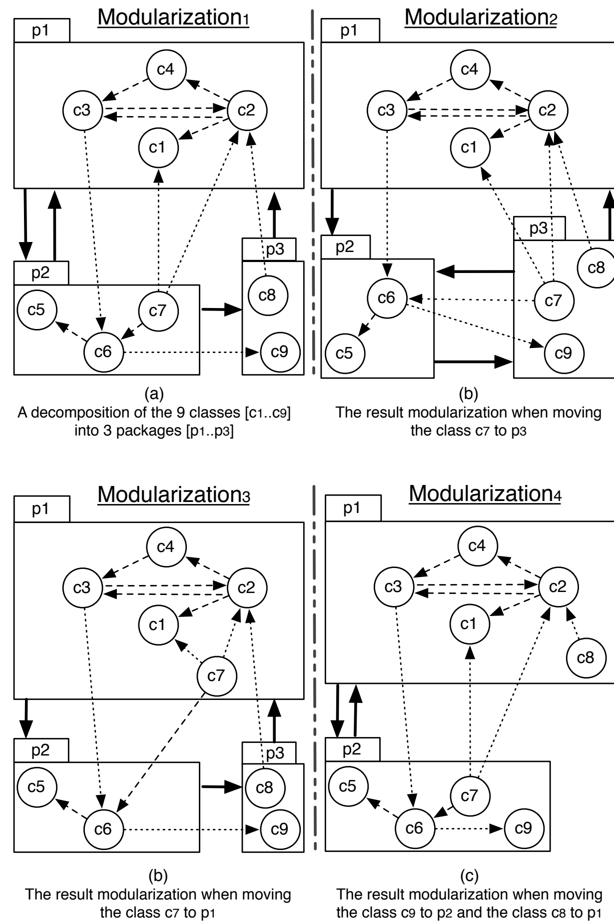


Figure 2.6: An example illustrating distinct impacts when optimizing package structure.

, $(c_7,c_2)\}$. Thus moving c_7 has reduced the number of inter-package cyclic dependencies. On the other hand, moving c_7 has increased the number of inter-package dependencies. In *Modularization*$_2$, there are 6 inter-package dependencies compared to 5 for *Modularization*$_1$.

Example 2. The difference between *Modularization*$_1$ and *Modularization*$_3$ is c_7 that is moved from p_2 to p_1 in *Modularization*$_3$. In *Modularization*$_3$ there is no cyclic dependency and there are 4 inter-package dependencies $\{(c_7,c_6)$, (c_3,c_6) , (c_6,c_9) , $(c_8,c_2)\}$ compared to 5 for *Modularization*$_1$. Thus moving c_7 to p_1 has eliminated all inter-package cyclic dependencies and reduced the number of inter-package dependencies. On the other hand, moving c_7 increases the size of p_1, where it becomes more than 2 times bigger than p_2 and p_3.
Similarly, in *Modularization*$_4$ c_9 is moved from p_3 to p_2 and c_8 is moved from p_3 to p_1. In *Modularization*$_4$ there are only 3 inter-package dependencies compared to 5 in *Modularization*$_1$, but in *Modularization*$_4$ there are only 2 packages (p_1 and p_2) compared to 3 in *Modularization*$_1$, *i.e.*, p_3 is empty in *Modularization*$_4$, which may impact the organization of developers/teams.

2.5 Existing Approaches to Understand Packages

Since object-oriented languages do not really support all the information that is important to understand packages; and since understanding packages is important to understand the results of software re-modularization process, several approaches, mainly based on visualization techniques and/or metrics, have been developed for that respect. In this section we present the most relevant existing approaches to understand packages and a software system modularization.

Understanding software system design and architecture

Already, in 1981, Steward *et al.* [*Steward*, 1981] have used the Dependency Structure Matrix (DSM) technique to identify software component dependencies and manage system design. Similarly, Sullivan *et al.* [*Sullivan et al.*, 2001] used the DSM technique to model software design. In Sullivan's approach, the DSM of a given software system represents the design space in terms of system design parameters, the design parameter values and the dependencies among them; where they map a design parameter as a choice to be made about some aspect (*e.g.*, data structures, algorithms, security aspect...) of the concerned design.

Originally the DSM technique has been developed for process optimization to identify dependencies between tasks, but it is also used in the context of software architecture analysis [*MacCormack et al.*, 2006]. Recently, the DSM technique is used to visualize and manage the static structure of software systems. Sangal *et al.* [*Sangal et al.*, 2005] used the DSM to provide the tool Lattix Inc's Dependency Manager (LDM), in which the inter-package dependencies are displayed using either binary values or numbers presenting dependency strength. The positive side of the LDM tool is that: (1) it shows well whether a given software system has a layered architecture or not, and where are the dependencies that break the layering, if any; (2) it identifies and spots the direct cyclic dependencies among packages. While the LDM limitations are that: (1) it does not identify and show the cause of inter-package cyclic dependencies –the classes and methods that cause those cyclic dependencies and the kind of those dependencies; as a consequence it does not help maintainers to take decisions solving package cycles; (2) it does not identify and show package classes, the number of classes that have external dependencies, the spread of dependencies over classes, package size, package internal dependencies...

To identify and show the cause of inter-package cyclic dependencies, Laval *et al.* [*Laval et al.*, 2009] proposed the enriched DSM (eDSM). In their proposal, they used the principle of small multiples[3] to show within the eDSM cells, where a cell represents the intersection between two packages, additional contextual information such as: (1) the dependency kind (*e.g.*, class inheritance, reference); (2) the classes that are related to cyclic dependencies. The main limitations of the eDSM are that: (1) the contextual information shown within the eDSM cells is limited to cover package cycle causes, which means that they do not include information about noncyclic dependencies, nor about classes which do not cause package cycles; (2) another problem of the eDSM is screen space limitation, where showing the contextual information within the cells

[3] 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].

requires, for all the eDSM cells (even cells that do not involve information), more space size; in addition to the fact that DSM visualizations usually use a lot of useless space when there are empty cells.

Exploring package relationships

Lungu *et al.* [*Lungu et al.*, 2006] provide a visual approach to guide and augment the exploration process of hierarchical package decomposition (*i.e.*, nested packages) with information about the worthiness of the various exploration paths. In their approach, they propose a set of package patterns based on the package nesting and on the number of package internal/external dependencies. Their patterns cover the following relationship roles that a package can have: (1) *silent package* if the package has not external dependencies; (2) *consumer package* if the package has outgoing external dependencies; (3) *provider package* if the package has incoming external dependencies; and (4) *hybrid package* if the package has outgoing and incoming external dependencies. They then use those basic patterns to define patterns on subsystems.

Storey *et al.* [*Storey et al.*, 1997] offer a visualization technique with multiple top-down views for exploring software system structure. Their technique presents the software system structure as a nested graph consisting in composite nodes (*e.g.*, packages or subsystems). The relationships among the composite nodes are presented by arcs, where an arc may present any number of dependencies. The main limitation of their proposed views is that they do not scale well with the number of packages and package relationships. The views are hard to understand in cases of large and complex software systems.

Lanza *et al.* [*Lanza* and *Ducasse*, 2003] offer the Polymetric views as a technique to visualize summary about software entities, such as classes or packages, and the connectivity among those entities. Their views are represented as graphs consisting in rectangles and links connecting them. In their approach, the developer can associate software entity attributes to the rectangle shape. For example, if the visualized entities are packages, the developer can map the package size to the height of the rectangle representing that package. However, those views are limited to show summary about software entities.

Ducasse *et al.* [*Ducasse et al.*, 2005b] present Butterfly, a radar-based visualization that summarizes incoming and outcoming relationships for a package, but only gives a high-level client/provider trend.

X. Dong *et al.* [*Dong* and *Godfrey*, 2007] present the High-level Object Dependency Graph (HODG) that helps capturing, from a high-level point of view, possible usage dependencies among coarse-grained software entities, namely packages. In their approach, they interpret the usage dependencies between classes in the context of their inheritance hierarchy and present a new graph of the system under analysis. The given graph is helpful only for understanding the considered system from a high-level point of view. Also, their graph visualization still difficult to be interpreted by human eyes because within it, the visualized nodes have different sizes but the node size does not map any information. The HODG has not visual semantics and it uses numbers to visualize almost all information.

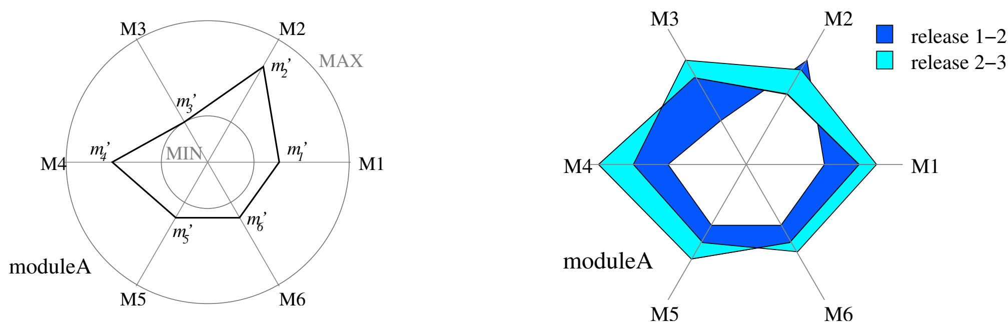
Understanding class organization

Langelier *et al.* [*Langelier et al.*, 2005] proposed a technique based on visualization for quality analysis of object-oriented software systems. In their approach, the visualization provides a graphic view of the relevant packages, where the visualization looks to be a city observed from the sky: the classes represent the buildings and the packages represent the layers under the buildings. Wettel *et al.* [*Wettel* and *Lanza*, 2007b] used very similar aspects to visualize the hierarchy of package nesting, package classes and class characteristics. These visualizations characterize packages by the classes defined in them, for example, they show the number of classes defined within a package, whether those classes are abstract or not, and the amount of methods that are defined in each class. However, they do not visualize inter-package dependencies, nor dependencies among classes.

Kuhn *et al.* [*Kuhn et al.*, 2007, 2008] used information retrieval to exploit linguistic information. They introduced semantic clustering to group source artifacts that use similar vocabulary. They use vocabulary topics to reveal the intention of the code and the similarity between its artifacts, then they provide a consistent visualization. While the approach is valuable to identify similar classes (*i.e.*, classes that use similar vocabulary topics), it does not help visualizing and understanding dependencies among classes and/or among packages.

Understanding package evolution

Pzinger *et al.* use Kiviat diagrams to present the evolution of package metrics [*Pinzger et al.*, 2005]. Kiviat diagram is *a graphical method of displaying multivariate data in the form of a two-dimensional chart of three or more quantitative variables*. Figure 2.7 (p. 31) shows an example of the Kiviat diagram that displays the values of 6 metrics $M1, M2, ..., M6$ for a given package ModuleA.



(a) Basic Kiviat diagram with measures of 6 metrics $M1, M2, ..., M6$ of the given package ModuleA.

(b) Kiviat diagram with 6 metrics $M1, M2, ..., M6$ of 3 releases of the given package ModuleA.

Figure 2.7: An Example Illustrating the Kiviat Diagram [*Pinzger et al.*, 2005].

Chuah and Eick use rich glyphs to characterize software artifacts and their evolution (number of bugs, number of deleted lines, kind of language...) [*Chuah* and *Eick*,

1998]. In particular, the time wheel exploits preattentive processing, and the infobug presents many different data sources in a compact way.

D'Ambros *et al.* propose an evolution radar to understand the package coupling based on their evolution [*D'Ambros* and *Lanza*, 2006b]. The radar view is effective at identifying outliers but does not detail the structure.

Summary:

> *Those existing approaches, while valuable, they fall short of: (1) providing a fine-grained view of packages that would help understanding the package shapes and dependencies; (2) supporting the identification of package roles within a software system: e.g., central or peripheral package, abstract or implementer package.*

2.6 Existing Approaches to Assess Package quality

To cope with software system complexity, Parnas *et al.* [*Parnas*, 1972] and Stevens *et al.* [*Stevens et al.*, 1974] have introduced the idea of decomposing software systems with the intention of increasing module cohesion and minimizing inter-module coupling. Since then, many metrics have been defined to compute the cohesion and coupling of a module, where module concept is usually used to represent a composite software entity (*e.g.*, in object-oriented programming, a module may be a class or a package).

Cohesion. There exist many metrics on cohesion in structured programming and for classes in object-oriented software systems. However, there are few cohesion metrics devoted to packages [*Allen* and *Khoshgoftaar*, 2001; *Fenton* and *Pfleeger*, 1996; *Martin*, 2002a; *Morris*, 1989]. Emerson presents a metric to compute cohesion applicable to modules in the sense of Pascal procedures [*Emerson*, 1984]. His metric is based on a graph theoretic property that quantifies the relationship between control flow paths and references to variables. Bieman *et al.* [*Bieman* and *L.M.Ott*, 1994] compute cohesion using slice abstraction of a program based on data slices.

In object-oriented programming, there is a plethora of software metrics on class cohesion. Chidamber *et al.* proposed a metric for class cohesion, named LCOM (Lack of Cohesion in Methods) [*Chidamber* and *Kemerer*, 1994], criticized and improved by Henderson-Sellers's LCOM* [*Henderson-Sellers*, 1996]. Bieman *et al.* [*Bieman* and *Kang*, 1995, 1998] proposed the metric TCC (Tight Class Cohesion) that measures class cohesion using the number of pairs of methods in a class that access common instance variables.

Hautus [*Hautus*, 2002] proposes a tool to analyze the structure of Java programs and a metric to determine the quality of the package architecture. He defines a new metric that indicates the percentage of changes to make a package structure acyclic. Hautus's metric does not indicate package cohesion. Patel *et al.* [*Patel et al.*, 1992] compute the cohesion of Ada packages based on the similarity of its members (programs). The idea is to measure cohesion based on the

similarity of the subprograms. They use the keywords shared between the subprograms. They consider only the specification of the package, not the keywords present in the body, which are invisible from outside the package. Similarly, Allen *et al.* define information theory-based (as opposed to counting) coupling and cohesion measures for subsystems [*Allen* and *Khoshgoftaar*, 2001]. Their measures are applied to modules, which are represented as graphs. They define cohesion in terms of intramodule coupling (*i.e.*, the similarity between the objects of the concerned modules). However these approaches do not take into account classes and their relationships.

Misic adopts a different perspective and measures the cohesion of a package as an external property [*Mišić*, 2001]. He claims that the internal organization of a package is not enough to determine its cohesion. Morris follows this line by computing module cohesion considering the usage of its contained objects [*Morris*, 1989]. Similarly, Ponisio *et al.* introduced the notion of use cohesion (or conceptual cohesion) [*Ponisio* and *Nierstrasz*, 2006]. They measure the cohesion of a package considering the usage of the package classes from the client packages. Their cohesion metric does not take into account the explicit dependencies among the package classes (*e.g.*, *method call, class access* or *class inheritance* –Section 2.2 (p. 16)).

Recently, the most known metrics that are used to assess packages and measure package cohesion, are those proposed by Martin [*Martin*, 2002a]. Martin proposed the Rational Cohesion metric that is supposed to measure one of the package cohesion aspects. He defined his metric as the average number of package internal dependencies per class. The rational cohesion (H) for a package p that entails N classes ($N = p_{size}$) and R internal dependencies ($R = |p_{Int.D}|$) is:

$$H = \frac{R+1}{N}$$

Where the extra *1* in the formula prevents $H = 0$ when p entails only one class ($N = 1$): *i.e.*, in such a case $R = 0$.
However, Martin's cohesion metric measures the connectivity among the internal classes of a given package, regardless the amount of dependencies that the package classes have with external classes.

Coupling. As earlier mentioned in Section 2.3.2 (p. 23), coupling is always used with cohesion to determine package quality, since analysis of coupling reveals change impact propagation [*Briand et al.*, 1999b]: *i.e.*, changes in a package p may impact all the packages that are coupled to p. On the other hand, coupling between packages is necessary to delegate responsibility [*Berard*, 1993; *Martin*, 2002a]. As a consequence, the package design should balance between the package cohesion and coupling. In that respect, Callebaut [*Callebaut* and *Rasskin-Gutman*, 2005] suppose that: *"the frequencies of interaction among elements in any particular subsystem of a system are (should be) two times greater than the frequencies of interaction between the subsystems"*.

However, there are few metrics that are devoted to assess package coupling. As we underlined in Section 2.3.2 (p. 23), there are two kinds of package coupling:

efferent coupling (C_e) and *afferent coupling* (C_a). The C_e is to assess the coupling degree between a package p and its *provider* packages. While the C_a is to assess the coupling degree between p and its *client* packages.

Martin [*Martin*, 2002a] defines the C_e metric for a package p as the number of p's provider classes, and defines the C_a metric as the number of p's client classes. Recently, in 2005 [*Martin*, 2005], he redefines these metrics: p's C_e is the number of p's provider packages, while p's C_e is the number of p's client packages.

Summary:

Existing metrics for measuring package cohesion and coupling, while valuable, they fall short of assessing modularization quality: i.e., assessing the quality of the whole organization of classes and packages. This way, maintainers should manually assess the modification impact each time they modify a package within a given modularization.

2.7 Existing Approaches to Optimize Modularizations

Since software modularization is a graph partitioning problem [*Mitchell*, 2002; *Mitchell* and *Mancoridis*, 2008; *Mitchell et al.*, 2004] (Section 2.4 (p. 26)) and since this last is known as a NP-hard problem [*Farrugia*, 2004], searching for good modularization by using deterministic procedures or exhaustive exploration of the search space is not feasible without additional heuristics [*Chapman et al.*, 2001; *Mitchell* and *Mancoridis*, 2008]. Therefore, researchers have adapted heuristic search methods to the software modularization problem [*Clarke et al.*, 2003; *Harman*, 2007].

The existing approaches to re-modularize software systems or to optimize existing modularizations are mainly based on clustering [*Abreu* and *Goulao*, 2001; *Bauer* and *Trifu*, 2004; *Lung et al.*, 2006; *Mancoridis* and *Mitchell*, 1998; *Mancoridis et al.*, 1999; *Mitchell et al.*, 2004; *Serban* and *l. G. Czibula*, 2007] and evolutionary or search based algorithms [*Doval et al.*, 1999; *Harman* and *Hierons*, 2002; *Liu et al.*, 2001; *Lutz*, 2001; *Maini et al.*, 1994; *Mitchell* and *Mancoridis*, 2006, 2008; *Mitchell et al.*, 2004; *Seng et al.*, 2005; *Tzerpo* and *Holt*, 1997].

We distinguish two categories of these existing approaches: (1) approaches that re-modularize software systems without taking into account the original organization of classes (*i.e.*, they do not take into account the existing packages); (2) approaches that optimize the existing organization of classes by taking into account the existing packages.

Remodularizing software systems

The most known tool in the software remodularization literature is Bunch tool, where Mancoridis and Mitchell [*Mancoridis* and *Mitchell*, 1998; *Mancoridis et al.*, 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 population of random modularizations or with an initial random partition of the

module dependency graph (MDG) which represents the graph of the software classes (*i.e.*, the software classes are nodes and the inter-class dependencies are edges). This way, their approach can start with the system original modularization. The clustering algorithm clusters each of the random modularization and selects the result with the largest quality as the suboptimal solution. Recently, they used Simulated Annealing technique to optimize resulting clusters [*Mitchell* and *Mancoridis*, 2002, 2006, 2008]. Their optimization approach creates new modularizations by moving randomly some classes (a block of classes) to new clusters. The goal of their approach is increasing cluster (package in our context) internal dependencies (*i.e.*, decreasing inter-package dependencies). They calculate the modularization quality (MQ) by summing the cluster factor (CF) for each cluster of the partitioned MDG: the CF of a given cluster is defined as a normalized ratio between the total weight of the internal dependencies and half of the total weight of the external edges.

MQ. Let k be the number of packages within a given modularization $\mathcal{M}$ and CF_i be the cluster factor of the package p_i ($p_i \in \mathcal{M}_C$); and let μ_i be the sum of p_i internal dependencies ($\mu_i = |\, p_{i_{Int.D}} \,|$) and ω_i be the sum of p_i external dependencies ($\omega_i = |\, p_{i_{Ext.D}} \,|$), then the MQ of $\mathcal{M}$ is defined as follows:

$$MQ = \sum_{1 \leq i \leq k} CF_i \qquad CF_i = \begin{cases} 0 & \mu_i = 0 \\ \frac{2*\mu_i}{2*\mu_i + \omega_i} & otherwise \end{cases}$$

Similarly, Seng *et al.* [*Seng et al.*, 2005] and Harman *et al.* [*Harman* and *Hierons*, 2002] proposed genetic algorithms to partition software classes into subsystems (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 works is increasing package internal dependencies. Seng *et al.* [*Seng et al.*, 2005] consider also cyclic dependencies between packages as anti-pattern for package design quality. However, their definition of the modularization/package quality (*i.e.*, the fitness function that computes the quality of resulting modularizations) is completely ambiguous. For example, they define the fitness value for package cycles as "the summing-up of the size of strongly connected component within the considered modularization" without explaining what does that mean.

Abreu *et al.* [*Abreu* and *Goulao*, 2001] adapted an heuristic clustering approach to (re-)modularize software systems. They used hierarchical agglomerative clustering methods to decompose software classes into packages. Their clustering methods start with a set of classes considering that each class is placed within a singleton cluster. The goal of their approach is also increasing package internal dependencies (*i.e.*, package cohesion). In their approach, they define package cohesion as the *Intra-modular Coupling Density* (ICD is the ratio of the package internal dependencies to the package dependencies), where ICD should be maximized and ideally takes *1* as value:

$$ICD = \frac{|p_{Int.D}|}{|p_D|}$$

In addition to the package cohesion, Abreu *et al.* [*Abreu* and *Goulao*, 2001] used the dispersion of classes over packages (*i.e.*, package size dispersion) as a factor to measure the modularization quality. They claim that dispersion on package size

should be somehow constrained to avoid extremely skewed distribution of classes (*e.g.*, a modularization that entails only one package nesting all the modularization classes, or a modularization that entails N packages, where every package contains only one class, *i.e.*, $N = |\mathcal{M}_C|$). Therefore, they firstly suppose that a modularization must contain at less 2 packages and at maximum $N-1$ packages, where N represents the number of classes. In this context, they define the *Relative Module Dispersion* (RMD) metric, as a factor to measure the modularization quality, as follows:

$$RMD = \frac{LargestPackageSize - SmallestPackageSize}{AveragePackageSize}$$

Optimizing existing software modularization

In the software (re-)modularization literature, there are few works addressing the problem of optimizing existing software modularization without creating new packages/subsystems.

Harman *et al.* [*Harman* and *Tratt*, 2007] introduce a non-exhaustive hill climbing approach to optimize and determine a sequence of class refactorings. They restricted their approach to only move methods (classes in our context) over existing classes (packages in our context). The goal of their approach is reducing the class coupling, based on the Coupling Between Objects (CBO) metric [*Briand et al.*, 1998]. To avoid having a very large classes, they, similarly to Abreu *et al.* [*Abreu* and *Goulao*, 2001] approach, used the dispersion of methods over classes (the standard deviation of methods per class metric) as a factor to measure the quality of resulting class refactoring sequences.

Differently from all those approaches to (re-)modularize software systems, Tzerpo *et al.* [*Tzerpo* and *Holt*, 1997] introduced a *deterministic* algorithm that assigns a newly introduced resource (class in our context) to a subsystem (package in our context). Their algorithm accommodates structural changes by considering the existing resources as newly added. Their approach suffers from several limitations: (1) to accommodate structural changes, the maintainers should specify the resources which are considered as newly introduced; (2) in their approach, the criteria of assigning a resource to a subsystem depends only on the relationships between the considered resource and subsystem, without any consideration of the whole modularization quality: the algorithm assign a resource r to a subsystem s if s is the most related subsystem to r; (3) in addition, in their approach, assigning a resource to a subsystem is consistent: which means that the algorithm definitively assigns a newly introduced resource r to a subsystem s, without any consideration of newly introduced resources that are not assigned yet to subsystems. Those resources may radically change the overall subsystem structure. As a consequence, the assignment of the resource r to s may finally appear inappropriate.

Summary:

Existing approaches to re-modularize software systems often change (to various degrees) the existing package structure of the concerned software

system. These approaches do not take into account the original organization of classes and the original package shape (i.e., package size); nor the distance between the alternative modularizations and the original one. In such a case, it can be difficult for a software engineer to understand the resulting structure and to map it back to the situation s/he knows.
In addition, they do not allow maintainers to define any constraint on alternative modularizations: e.g., (1) constraints on the maximal size of a given package; (2) constraints on classes that should not change their packages; (3) constraints on packages that should not be modified.
Finally, all those approaches do not take into account the connections among packages. They limit their approaches to inter-class dependencies without verifying the number of inter-dependent packages (i.e., the degree of package coupling within the considered modularization); and similarly for package cycles.

2.8 A Combined Approach for The Maintenance of Software Modularization

A summary of this chapter is that the problem of (re-)modularizing software systems is an old problem. During the last three decades, researchers tried to address this problem from different perspectives: visualization, metrics, heuristic and search based algorithms. In this chapter, after our analysis of the problems of the software modularization maintenance and our analysis of the state of the art, we found that the existing approaches suffer from several limitations. In that respect, we have identified a set of open problems that we address in this thesis:

Understanding package structure. *The maintenance of software modularization needs detailed information about the concrete organization of classes for understanding: package size, complexity and internal vs. external dependencies, regarding the different kinds of dependencies (class inheritance and/or reference).*

Understanding package role and the usage of their interfaces. *The maintenance of software modularization needs to capture the information about package interfaces, their cohesion and their roles. For example, what are the cardinalities of the interfaces of a given package? What are the functions that a package provide and to which packages it provides them? What are the functionalities that a package require and from which packages it requires them? Are the functionalities, that a given package provides, used together or not?...*

Assessing the quality of software modularization and packages. *The maintenance of software modularization needs to assess the quality of the concerned software modularization. It also needs to assess the quality of a single package, with respect to the concerned modularization. Therefore, we need to define metrics that compute the quality of a modularization/package from different perspectives (the principles of package cohesion and coupling).*

Automatically optimizing existing software modularization. *In the case of a large and complex software modularization, searching for good modularization using deterministic procedures or exhaustive exploration of the search space is not feasible without additional heuristics. As a consequence, the approach of the maintenance of a software modularization should provide an automatic, or semi-automatique, methodology that: (1) detects candidate package fro restructuring; (2) proposes suitable changes in the class organization; (3) optimizes the existing modularization, with respect to the principles of package cohesion and coupling. In addition, the automatic optimization process should: (1) take into account the original class organization; (2) search for good alternative modularizations by doing a near minimal modification; (3) allow maintainers to specify constraints on the alternative modularizations, such as the number of classes that are allowed to change their packages, the allowed changes on package size...*

Chapter 3

Package Blueprint: Visually Understanding Package Structure and Interactions

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

3.1 Introduction

Maintainers of large software systems face the problem of understanding how packages are structured in general and how they are in relation with each others in their provider/client roles. This problem was experienced first-hand during two years by Stéphane Ducasse, the supervisor of this thesis, while being responsible of the release of Squeak, a large open-source Smalltalk [*Denker* and *Ducasse*, 2007]. In addition, approaches that support software system remodularization succeed in producing alternative views for system refactorings, but proposed changes remain difficult to understand and assess [*Anquetil* and *Lethbridge*, 1999; *Wiggerts*, 1997]. Hence even if there is a good support for the algorithmic parts, much work remains to help users understand, compare and assess proposed solutions.

As we explained in Section 2.5 (p. 29), Several previous works, mainly based on visualization techniques, provide information on packages and their dependencies, by visualizing software artifacts, metrics, their structure or their evolution. However, while those approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the package characteristics: the number of classes it defines; the inheritance dependencies among its classes; how the package classes inherit from and interact with classes packaged into other packages...

Contribution of the chapter

In this chapter, we propose the Package Blueprint, *a compact visualization revealing, in detail, package structure and dependencies*. A package blueprint is structured around

the concept of a *surface*. A *surface* represents and details the connections between the observed package and its provider and client packages. *Package Blueprint reveals the overall size and complexity of a package, as well as its relations with other packages, by showing the distribution of dependencies to classes within and outside the observed package*. Package complexity is defined by the quantity of dependencies among the package classes (*i.e.*, internal complexity) and the quantity of dependencies between the package classes and other classes (*i.e.*, external complexity).

The content of this chapter is the object of our paper submitted to the *IEEE Transactions on Software Engineering (TSE) journal* [*Ducasse et al.*, 2009].

Structure of the chapter

Section 3.2 (p. 40) summarizes the properties that we expect from effective visualizations. Section 3.3 presents the structuring principles of a package blueprint, which are then declined to support an outgoing reference view, an incoming reference view and an inheritance view in Section 3.4 (p. 42). Sections 3.5 & 3.6 present the different views of package blueprint at work. The next section presents the results of a limited case study with advanced developers. In sections 3.9 and 3.10, we discuss our visualizations and position them w.r.t. related work before concluding.

3.2 Visualization Challenges

In addition to the challenges related to the difficulties of understanding packages (Section 2.3 (p. 21)), the visualization itself raised challenges. Several work identified the characteristics that an efficient visualization should hold [*Bertin*, 1983; *Tufte*, 2001; *Ware*, 2000]. As our focus is on providing a first impression of a package and its context, we would like to exploit the gestalt principles of visualization and preattentive processing[4] as much as possible to help spotting important information [*Healey*, 1992; *Healey et al.*, 1993, 1995; *Treisman*, 1985; *Ware*, 2000].

We stress that our visualization should take into account the following properties:

Good mapping to reality. The visualization should offer a good representation of the situation that the maintainer can trust and from which s/he can draw and validate hypotheses.

We expect from the visualization to highlight the general tendency of a package in terms of its internal size, internal and external references. In particular we want to spot classes or dependencies that stand out in a given package.

[4] Researchers in psychology and vision have discovered a number of visual properties that are preattentively processed. 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 power (in a context of filled squares and empty circles, a filled circle is usually not detected preattentively). Some of the features are not adapted to our needs. For example, we do not consider motion as applicable.

Scalability and simple navigation. The maintainer should easily access the information. The visualization should scale *i.e.*, we should be able to have system overview as well as focusing on a particular package. We target a visualization that scales well with the number of packages and of dependencies, unlike those using graph representations [*Dong* and *Godfrey*, 2007].

Low visual complexity. By being regular and well structured, *i.e.*, reusing the same conventions for color and position, the visualization should help the maintainer learn it and understand it. In addition, while the visualization should offer a lot of information, it should not be complex to analyze.

3.3 Package Blueprint basic principles

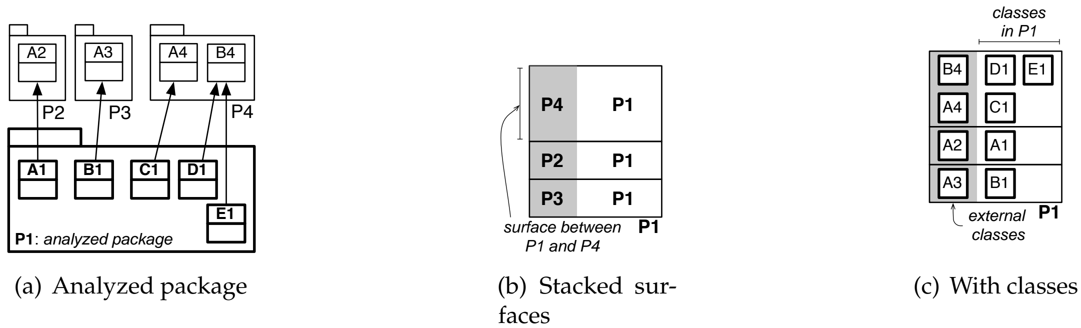
To meet most the requirements cited so far, we propose our compact visualization: package blueprint. A package blueprint represents either how the package under analysis references other packages, or how it is referenced by them. Figure 3.1 (p. 42) presents the key principles of a package blueprint; these principles are realized slightly differently according to the kind (references or inheritance) and the orientation of the considered dependencies (incoming or outgoing class references).

The package blueprint visualization is structured around the "contact areas" between packages, that we name *surfaces*. A *surface* represents conceptually the dependencies between the observed package and another package. In Figure 3.1(a) (p. 42) the package P1 is in relation with three packages P2, P3, and P4, via different dependencies between its own classes and the classes present in the other packages; so P1 has three surfaces.

A package blueprint shows the observed package as a rectangle, vertically subdivided into parts representing its surfaces. Each surface between the observed package and a referenced package is more or less tall, according to the strength of the dependencies between them. In Figure 3.1(b) (p. 42), as P1 references three other packages, its blueprint is formed from three stacked boxes. The box of the surface between P1 and P4 is taller than the others because P1 references more classes in P4 than in P2 and in P3.

In each subdivision, we also show the classes involved in the corresponding surface. By convention, we *always* show the classes in the referenced packages on the leftmost gray-colored column of each surface, and the classes of the observed package on the right. In Figure 3.1(c) (p. 42), the topmost surface shows that classes D1 and E1 reference class B4, and that C1 references A4. If many classes reference the same external class, we show them all on a horizontal row; we can thus assess the importance of a class by looking at the number of classes on the corresponding row: in Figure 3.1 (p. 42) (c), the row of B4 stands out because the two referencing classes D1 and E1 make it wider.

To display incoming references and inheritance, we define variants of the layout: to distinguish incoming from outgoing references we rotate the layout (see Figure 3.3, Section 3.4.2 (p. 44)), and to display inheritance we arrange hierarchies as trees instead of rows (see Figure 3.5, Section 3.4.3 (p. 46)).



(a) Analyzed package (b) Stacked surfaces (c) With classes

Figure 3.1: Consider P1 that references four classes in three other packages (a). A blueprint shows the surfaces of the observed package as stacked subdivisions (b). Small boxes represent classes, either in the observed package (right white part) or in referenced packages (left gray part) (c).

3.4 Package Blueprint Detailed Visualizations

To convey more information, we refine the basic layout previously described as illustrated in Figure 3.2 (p. 43).

3.4.1 Outgoing Reference Blueprints

Internal References. To support the understanding of references between classes inside the observed package, we add a particular surface with a thick border at the top of the blueprint. This surface is the head of the blueprint, and the rest its body. In the head, the first column represents the internal classes of the package under consideration. Thus among these classes we see those that are referenced from within the package itself: for the package P1 in Figure 3.2 (p. 43), the class A1 is referenced by B1 and C1; C1 is referenced by E1; D1 by C1 and E1; E1 by D1 and G1 is referenced by H1 and I1. The height of the head surface indicates the number of classes defined within the package.

Position. Internal referencing classes are arranged by columns: each column (after the leftmost one) is reserved to the same internal class for all the surfaces. The width of the blueprint indicates the number of referencing classes of the package. Figure 3.2 (p. 43) shows that class E1 internally references C1 and D1, and externally references B3, C3, A3 and A4.

We order classes of the concerned package in both horizontal and vertical directions to present important elements according to the (occidental) reading direction. Horizontally, we sort classes from left to right according to the number of classes they reference. Hence classes referencing the most occupy the nearest columns from the left gray column. Figure 3.2 (p. 43) shows that class E1 occupies the nearest column from the left gray column since it references six classes (D1, C1, A3, B3, C3 and A4) while D1 references three classes and C1 two classes; each of the remaining internal classes references only one class.

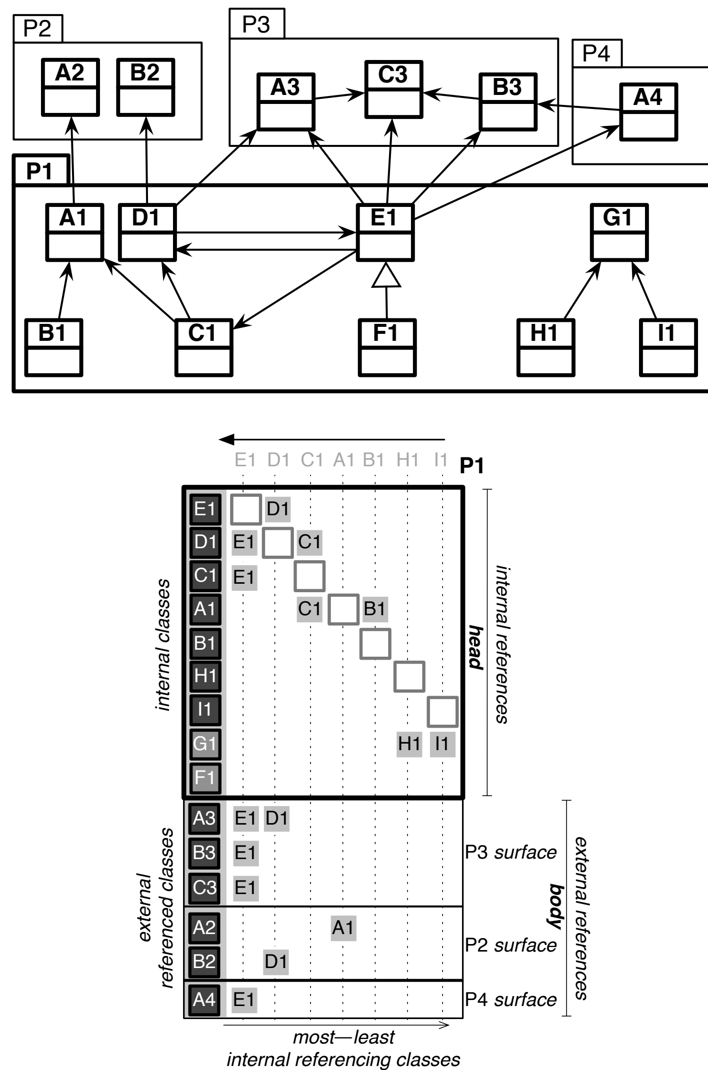


Figure 3.2: Surface package blueprint detailed view (Outgoing Reference view for P1).

We apply the same principle to the vertical ordering, for both surfaces within a blueprint and rows (*i.e.*, external classes) within a body surface. Within a package, we position surfaces that present the most external classes the highest. Within a body surface, we order external classes from most referenced at the top, to the least referenced at the bottom. This is why in Figure 3.2 (p. 43) the surface in relation with P3 is the highest and why the surface with P2 is above P4: there are more referenced classes into P2 than into P4.
Within the head surface, the vertical ordering of the internal classes is identical to their horizontal ordering. Figure 3.2 (p. 43) shows that internal classes (E1, D1, ... and I1) are ordered the same way vertically and horizontally. Bordered squares, that are diagonally placed from the top-left to the bottom-right within the head, help the users to clearly see the symmetry between the horizontal and

vertical orderings. This diagonal helps also to detect direct cyclic references within the concerned package: within the head of P1 blueprint (Figure 3.2 (p. 43)), we see that there is a direct cyclic-reference between D1 and E1 – since there are a node of D1 and a node of E1 that have symmetrical positions relatively to the head diagonal; which means that E1 refers to D1 and D1 refers to E1. Internal classes with no reference to others are placed at the bottom of the left most column in the head (*e.g.*, G1 and F1 do no reference).

The head surface therefore conveys the package size as well as the ratio between defined classes and their internal dependencies. Both referencing classes and unreferencing ones together with internal references among them are shown (*e.g.*, the unreferencing G1 is referenced by H1 and I1).

Color. Color intensity assigned to a node representing a class conveys the number of references it is doing: the darker the more references it does. Both intensity and horizontal position represent the number of references, but position is computed relatively to the whole package blueprint, while intensity is relative to each surface. Thus, while classes on the left of surfaces will generally tend to be dark, a class that has many references but few in a particular surface will stand up in this surface since it will be light gray. For example, in a blueprint of P1 (Figure 3.2 (p. 43)), within the head surface, the nodes of C1 and E1 should have the same color intensity and both should be darker than the nodes of D1 – since C1 and E1 each references two internal classes (*i.e.*, within the head surface), while D1 references only one class in the same surface. Within P3 surface, E1 should be slightly darker than D1 since the former references three classes within P3 while the latter references only one class.

In the first column and within the head surface, we distinguish unreferencing internal classes from others by making their fill lighter than the one of internal referencing classes (*e.g.*, G1 and F1 fill color is lighter than the fill color of I1..E1).

Finally, we want to distinguish referenced classes depending on whether they belong to a framework or the base system, or are within the scope of the system under study. When a referenced class (in the first column and in the external referenced part - see Figure 3.2 (p. 43)) is not part of the system we are currently analyzing, the fill of its node is cyan.

3.4.2 Incoming Reference Blueprints

We use a view similar to the outgoing reference blueprint for exploring incoming references. However we visually distinguish the incoming reference blueprint from the former as follows: as shown in Figure 3.3 (p. 45), the global view is rotated counter-clockwise by 90°; the external referencing classes are placed at the top while the package internal classes are placed below them. The blueprint surfaces are ordered from right to left: the head is at the most right and surfaces of client packages are ordered by the number of referencing classes they enclose. The referencing classes are thus displayed on the top row, and we sort internal referenced classes from the most referenced on the second row, to the least referenced on the bottom row.

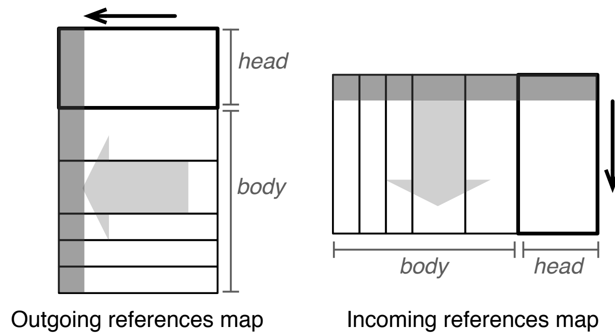


Figure 3.3: To distinguish it from the outgoing reference blueprint (left), we rotate the incoming reference blueprint (right) by 90°, so that the important details are still read first; in the incoming view, the references are made by the external classes, at the top, to the internal classes below them.

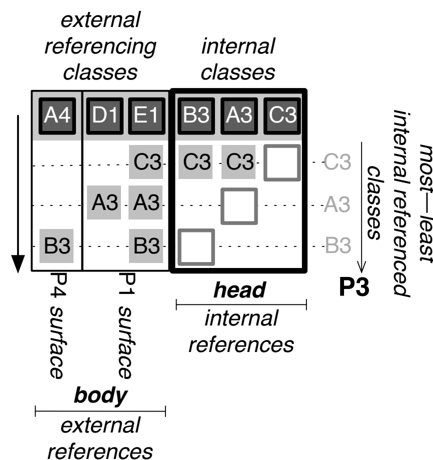


Figure 3.4: Package blueprint detailed view (Incoming Reference view for P3 Figure 3.2 (p. 43))

Figure 3.4 (p. 45) shows the incoming reference blueprint of P3 (Figure 3.2 (p. 43)). The blueprint body has two surfaces: P1 surface and P4 surface – since P3 clients are P1 and P4. P1 surface is at the right of P4 surface, since the former involves more referencing classes (two classes: E1 and D1) than the latter (one class: A4). C3 is the most referenced class within P3: C3 is referenced from three classes (A3, B3 and E1), while A3 is referenced from two classes (E1 and D1) and B3 is also referenced from two classes E1 and A4.

In a first version of the package blueprint, views were visually too close and it was difficult to distinguish them in a glance. Finding a view that was really distinct from the previous while sharing the same visual effect was important to avoid confusion.

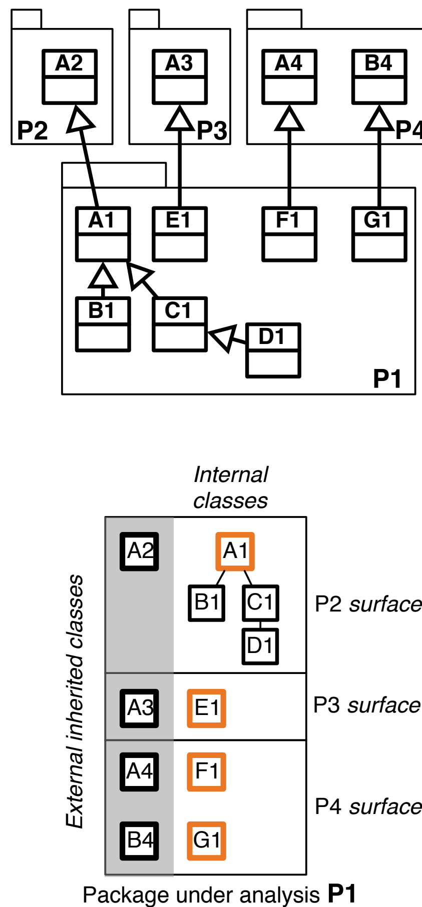


Figure 3.5: Inheritance package blueprint. Orange bordered classes inherit directly from external classes.

3.4.3 The Case of Inheritance

Up to now, we only discussed references, but inheritance is a really important structural relationship in object-oriented programming. We then offer a specific view which structures the inheritance relationship within the package according to the client packages, as shown in Figure 3.5 (p. 46).

We consider only single inheritance so we can display all classes and subclasses transitively inheriting from external classes on the same row. We distinguish the direct subclasses of *external* classes by showing them with an orange border; indirect subclasses are black-bordered and arranged in trees under their superclass. Figure 3.5 (p. 46) shows the inheritance blueprint of P1. P1 classes inherit from classes packaged in distinct packages (P2, P3 and P4): A1 inherits from A2 defined in package P2, while B1, C1, and D1 inherit from A1; E1 inherits from A3 defined in package P3; F1 and G1 inherit respectively from A4 and B4, both defined in package P4. This view highlights internal inheritance roots as well as external inheritance usage.

As explained subsequently, to distinguish root classes such as Object and classes

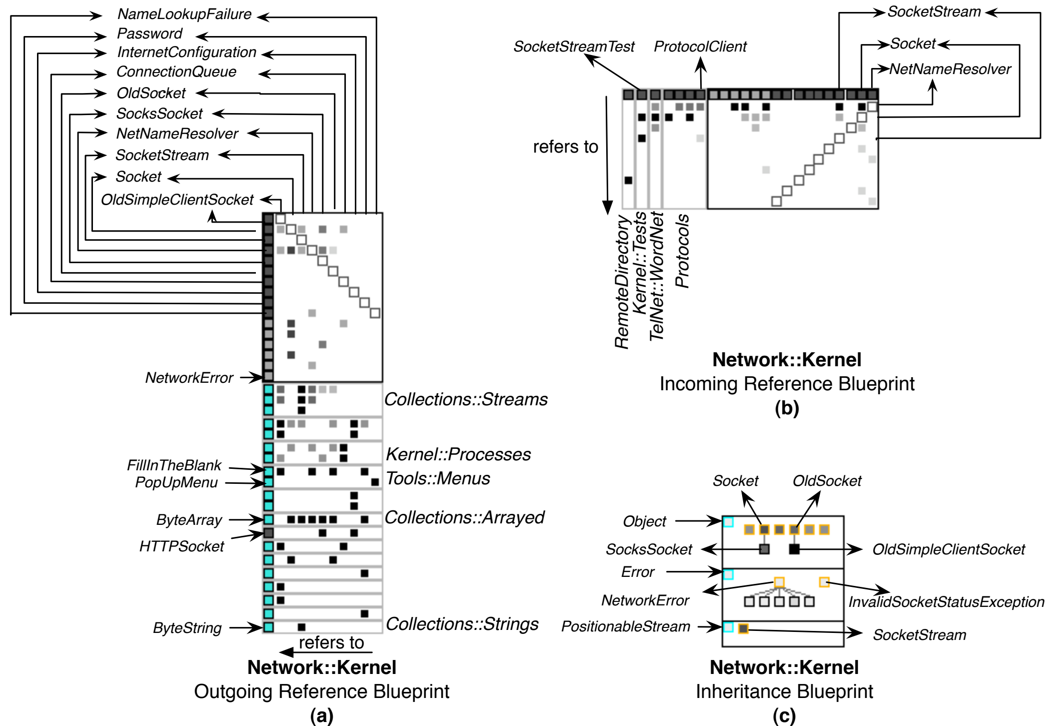


Figure 3.6: Analyzing the Network::Kernel Package.

that do not belong to the system under-analysis we use cyan as fill color. Similarly, we use blue as a fill color to distinguish abstract classes. The fill color of other classes in the inheritance view still represent the number of *references* made by the class, but relatively to the *package* and not to the surface like in the reference views. This enables maintainers to correlate inheritance and reference views. For instance, in Figure 3.6 (p. 47) (c), the inheritance blueprint of the package Network::Kernel shows that most references come from the classes OldSimpleClientSocket and SocksSocket – since they have the darker fill color – which are respectively subclasses of the abstract classes OldSocket and Socket. Blueprint width represents the maximum number of subclasses at one level and its height the depth of inheritance.

3.5 An Example: The Network::Kernel Package

We are now ready to have a deeper look at an example. The Squeak Network subsystem contains 178 classes and 26 packages – making up a library and a set of applications such as a complete mail reader. Figure 3.6 (p. 47) shows the blueprints (outgoing reference, incoming reference and inheritance) of the Network::Kernel package in Squeak.

Glancing at the package outgoing reference, Figure 3.6 (p. 47) (a), we see that it has a lot of gray squares inside its head surface, which indicates that there is a lot of interactions among the internal classes of the analyzed package Network::Kernel. This

blueprint shows also that there are more gray squares inside the body surfaces than inside the head, indicating that the package classes have more interaction with classes of other packages than internally in the package. This conveys a first impression of the package cohesion even if it is not really precise [*Briand et al.*, 1999a].

The number of the body surfaces indicates that Network::Kernel is in relation with 14 other packages. Most of the referenced classes are cyan, which means that they are not part of the Network subsystem. Indeed they belong to the core libraries (*e.g.*, Collections::Streams, Collections::Arrayed and Collections::Strings) on top of which Network::Kernel is defined. What is striking is that all except one of the referenced classes are outside the system (HTTPSocket in Figure 3.6 (p. 47) (a)). Since the package is named Network::Kernel, it is strange that it refers to other classes from the same system, and especially to only one. This is clearly a layering bug.

The outgoing reference blueprint shows clearly which provider packages are important for the analyzed one Network::Kernel and which are less important: some of the referenced packages, such as Collections::Streams and Kernel::Processes, are strongly referenced by Network::Kernel – since there are a lot of gray squares inside the corresponding surfaces; other referenced packages, such as Collections::Strings, are referenced by only one class or a couple of referencing classes.

Analyzing the blueprints and inspecting the class and package names, we found via the outgoing reference blueprint, Figure 3.6 (p. 47) (a), another improper layering: the Tools::Menus surface shows that Network::Kernel is referencing UI classes (FillInTheBlank and PopUpMenu) via the package Tools::Menus which seems inappropriate.

On another hand, we learn that the class making most internal references is named Socket: this class is represented in the outgoing reference blueprint by the second column; this latter includes, within the blueprint head, the biggest number of gray squares (4) that are darker than the remaining ones within the head. This means that Socket is referencing 4 classes within the analyzed package Network::Kernel and it is the class which does the biggest number of references within Network::Kernel.

We also learn that the class OldSimpleClientSocket, represented by the first column in the outgoing reference blueprint, makes most external references – the class column includes, within the blueprint body, nine gray squares that are distributed over seven distinct surfaces, which means that this class refers to nine classes into seven packages. However OldSimpleClientSocket refers to only 2 classes within Network::Kernel as shown in the head.

The incoming reference blueprint (Figure 3.6 (p. 47) (b)) shows that most internally referenced class is NetNameResolver, since the class row includes the biggest number of squares within the blueprint head and those squares are darker than the other ones. Similarly, we see that the second most referenced is Socket. So this is a sign of good design since important domain classes, namely NetNameResolver and Socket are well used within the package.

On the another hand, the incoming reference blueprint (Figure 3.6 (p. 47) (b)) shows that Network::Kernel is referenced by only 4 client packages and all belong to the Network system. One of those referencing package is Kernel::Tests which includes test classes. The corresponding surface of Kernel::Tests shows that there is one referencing class (SocketStreamTest) that refers to two classes within the analyzed package Network::Kernel (Socket and SocketStream). We learn that most Network::Kernel classes are not tested and this is a bad sign about the quality of the Network system.

The inheritance package blueprint (Figure 3.6 (p. 47) (c)) shows that the Network::Kernel package is bound to three external packages containing the three superclasses Object, Error, and PositionableStream. In addition the package, while inheriting a lot from external packages, is inheriting mostly from the same class, here Object. The difference between the two main surfaces is interesting to discuss: the topmost surface shows that most of the classes are directly inheriting from one external superclass (Object), while the second one shows that errors are specialized internally to the package. All in all, this makes sense and provides a good characterization of the package.

3.6 Packages Within Their Software System

Understanding a package in isolation is interesting but lacks information about the overall context of classes and packages in relation with it. As shown in the following subsections, our approach also supports the understanding of the usage of a class/package within the context of a complete software system. Relevant questions are for instance: Which other packages use a given class? Are two classes always co-used by others? Are two packages used with a same importance by others? and so on.

To help maintainers navigate among packages and classes and to quickly collect information, we introduced a fly-by-help mechanism to the Package Blueprint. In addition to the fly-by-help (see Figure 3.7 (p. 50)), maintainers can select a class (*i.e.*, any box representing the concerned class), or a package (*i.e.*, any surface representing the concerned package), and mark it with a particular color: the fill of boxes/surfaces representing the selected class/package will all have the selected color. Similarly, maintainers can mark several classes and packages with distinct colors (see Figure 3.8 (p. 51)).

3.6.1 Outgoing Reference Package Blueprint Analysis

Figure 3.8 (p. 51) shows the blueprints of all the packages referencing and defining the class HTTPSocket of the Network system.

Hub classes. It is striking to see that HTTPSocket, highlighted in red, is a central class of the package Protocols as it refers to most of the classes referenced by that package. We can deduce the same thing for the class ServerDirectory in RemoteDirectory package. In addition, we can easily see that almost all referencing packages to the package Protocols, whose surface is highlighted in yellow, use the class HTTPSocket of Protocols. Only RemoteDirectory refers, in addition to HTTPSocket, to two classes in Protocols. Note that HTTPSocket has no incoming references nor outgoing references inside its package Protocols – since in the head of Protocols blueprint the column and the row of HTTPSocket contain no gray squares.

Figure 3.8 (p. 51) also shows how the package Kernel, whose surface is highlighted in orange, is used within the Network system.

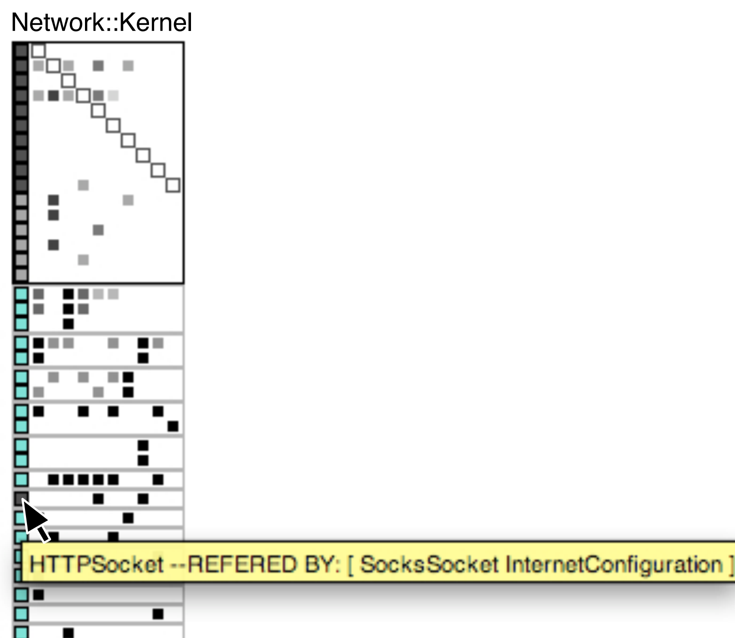


Figure 3.7: Interacting with package blueprint: using the mouse and pointing at the box shows, through a fly-by-help, the class and package names. In this view, the mouse is pointing to the box representing HTTPSocket and the fly-by-help shows, in addition to the class name, the name of Network::Kernel classes that refer to HTTPSocket.

Core Packages. Apparently, Kernel is less important than Protocols, *i.e.*, Kernel is referenced by 3 packages (Protocols, RemoteDirectory and TelNetWordNet), while Protocols is referenced by all other packages in Figure 3.8 (p. 51) (5 packages including Kernel). However, looking to the orange surface in the body of Protocols package blueprint, we find that 4 classes of Protocols reference 3 classes of Kernel, while the yellow surface in the body of Kernel package blueprint shows that only 2 classes of the latter reference a single class (HTTPSocket) of Protocols. In addition, Kernel does not refer to any other package in Network system (classes colored in cyan do not belong to the subsystem under analysis). Looking more closely at Protocols referenced packages, we can see that Kernel (the orange surface in the body of Protocols package blueprint) is placed above Url and RFC822, the three only packages referencing classes belonging to the Network system. This reinforces the idea that Kernel is the basic package of the Network system core.

Cyclic References. On another side, the cyclic reference between Kernel and Protocols raises the known problem about the order of deploying or loading the Network system. One possible way to remove this cyclic reference consists in moving class HTTPSocket to Kernel package. However, HTTPSocket also refers to URL package. Therefore moving HTTPSocket to Kernel will result in adding one referenced package to the latter, thus disturbing its status as a core package. To keep Kernel without references to any other Network package, a better solution is to move the referencing classes SocksSocket, colored in blue, and InternetConfiguration, colored

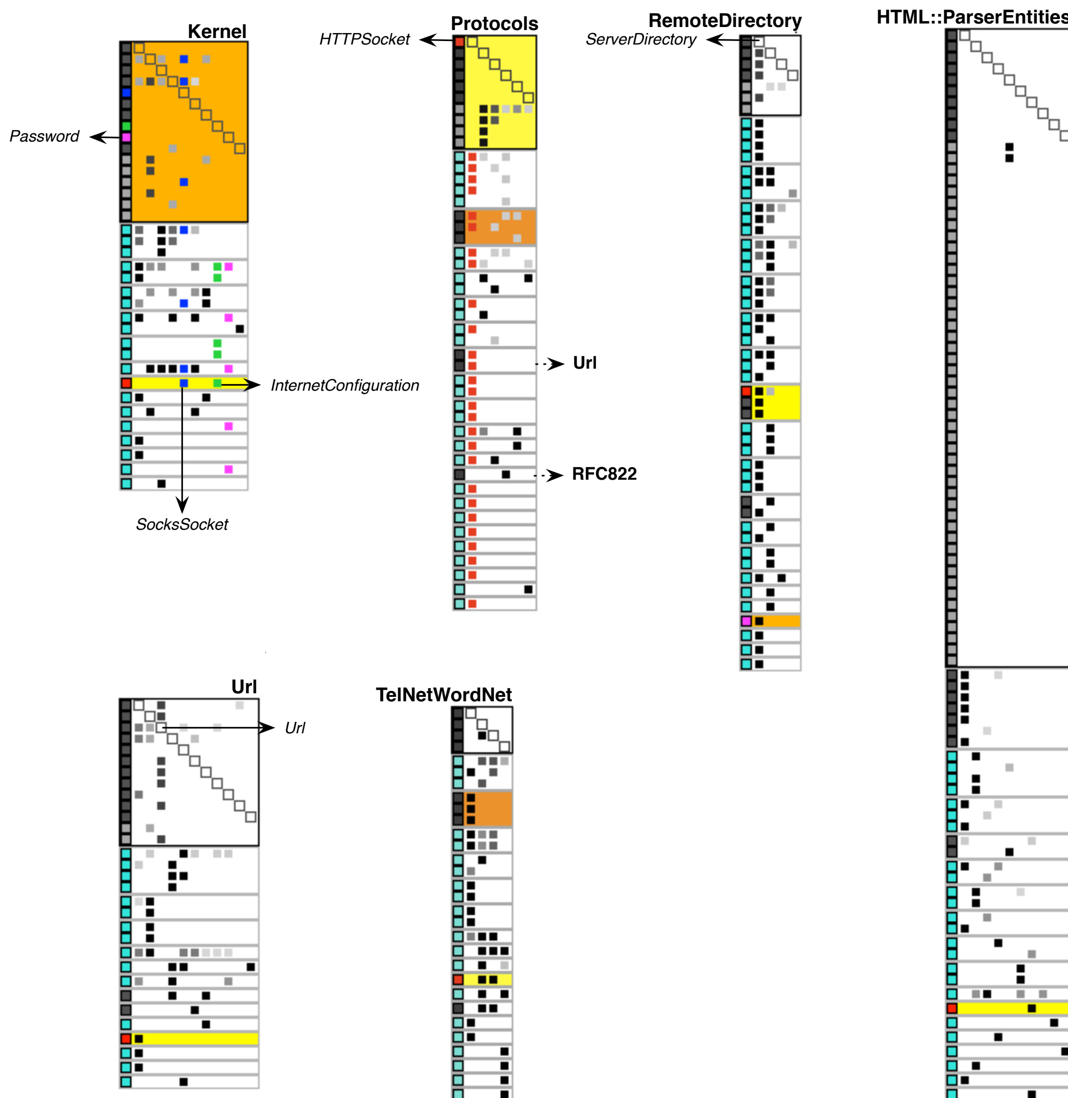


Figure 3.8: Outgoing reference blueprints of some packages of the Network system. In this view, the Kernel package was selected in orange, surfaces with Protocols package are highlighted in yellow, class HTTPSocket in red, class SocksSocket in blue, class InternetConfiguration in green, and class Password in fuchsia.

in green, to Protocols package. InternetConfiguration has no incoming nor outgoing references inside Kernel package (see in the head of the blueprint, the column and the row for this class), but InternetConfiguration references the HTTPSocket class in Protocols package. So, moving InternetConfiguration to Protocols package will increase the cohesion of both packages; SocksSocket refers to 3 classes inside Kernel but is not referenced inside it. So moving SocksSocket to Protocols will increase a bit the coupling between Protocols and Kernel but will increase the Protocols package cohesion –since SocksSocket refers to HTTPSocket in Protocols package. This way Kernel becomes a proper core package for Network system.

Potentially misplaced classes. Again in Kernel package, we found that the class Password, colored in fuchsia, has no outgoing nor incoming references inside Kernel package – see in the head of Kernel package blueprint the column and the row of this class. Looking closely at Password, we see that it is referenced by only one package: RemoteDirectory refers to Password class in Kernel – see the orange surface and the fuchsia referenced class in the body of RemoteDirectory package blueprint. Thus we think that moving Password class to this last package will increase the cohesion of both packages, Kernel and RemoteDirectory.

Internally loosely connected packages. Figure 3.8 (p. 51) shows that HTML::ParserEntities and TelNetWordNet packages are not cohesive from the point of view of inter-class references – since the heads of HTML::ParserEntities and TelNetWordNet blueprints contain respectively only two and one gray boxes.

Internal Interconnected Packages. In addition we see in Figure 3.8 (p. 51) that Kernel and Url packages contain classes that are tightly inter-referenced – since there are a lot of gray boxes within the head of Kernel and Url package blueprints. For example, within Url package, the class Url refers to almost all classes of its package and it does not refer to any class outside the Url package (see the column of Url class in the blueprint body).

Similarly, we can see that RemoteDirectory and Protocols packages are less cohesive than Kernel and Url packages but more cohesive than HTML::ParserEntities and TelNetWordNet packages. It is worth to note that in the Protocols package, all internally referenced classes are classes that do not reference other classes – since all gray boxes within the head of the Protocols blueprint are under the head diagonal.

3.6.2 Incoming Reference Package Blueprint Analysis

Incoming reference Package Blueprint is similar to outgoing reference Package Blueprint we described in previous sections. The difference between them is that the incoming reference Package Blueprint shows the package dependencies with its users while outgoing reference Package Blueprint shows the package dependencies with the packages it uses.

Figure 3.9 (p. 53) shows the incoming reference blueprints for Network packages where only references within the Network system are taken into account (*i.e.*, references from packages outside Network are not shown). In this figure, surfaces of RemoteDirectory package are highlighted in green and those of TelNetWordNet in orange.

Figure 3.9 (p. 53) shows that most referenced packages within Network system are Protocols and Url – since they have the biggest number of surfaces within the body of their blueprints: both are referenced from 7 packages within Network. Thus we deduce that these packages are the core of Network.

The package Kernel is referenced by only four packages within Network: Protocols, TelNetWordNet, Kernel::Test and RemoteDirectory. Protocols package heavily refers to Kernel package: the Kernel package incoming reference blueprint shows that the surface

denoted by Protocols represents 4 external referencing classes. This means that there are 4 classes of Protocols package that refer to Kernel classes.

Since Kernel package is heavily referenced by the core package Protocols (and that as already explained in Section 3.6.1 (p. 49) Kernel does not refer to packages within Network), Kernel represents the basic package within Network.

Most referenced package class. Within this package, Kernel, most referenced classes (*i.e.*, dominant referenced classes) are NetNameResolver and Socket (see the number of gray boxes within the class rows of Kernel blueprint). Since the nodes of Socket class, within the body surfaces of Kernel blueprint, are darker than those of

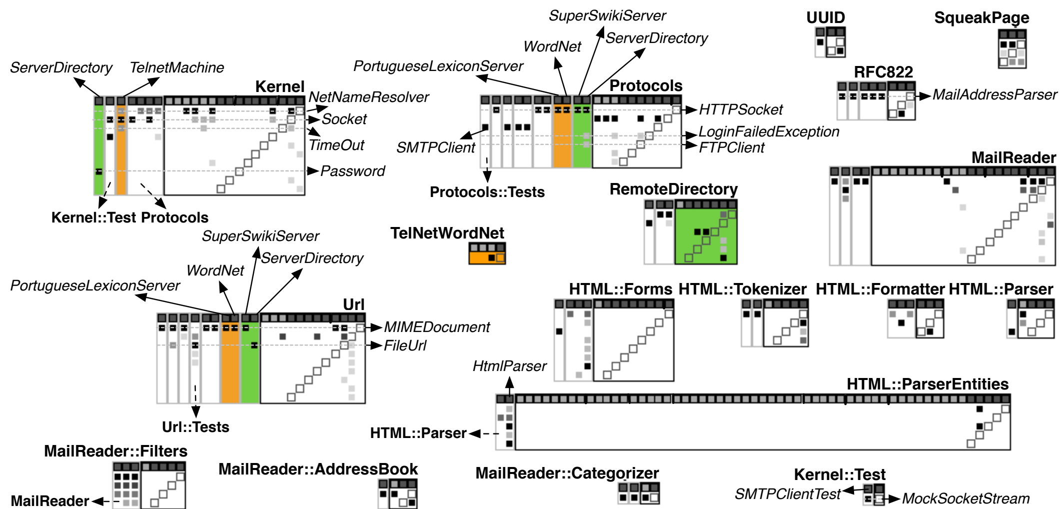


Figure 3.9: Incoming Reference global view in Network system. In this view, the TelNetWordNet package was selected in orange, surfaces with RemoteDirectory package are highlighted in green.

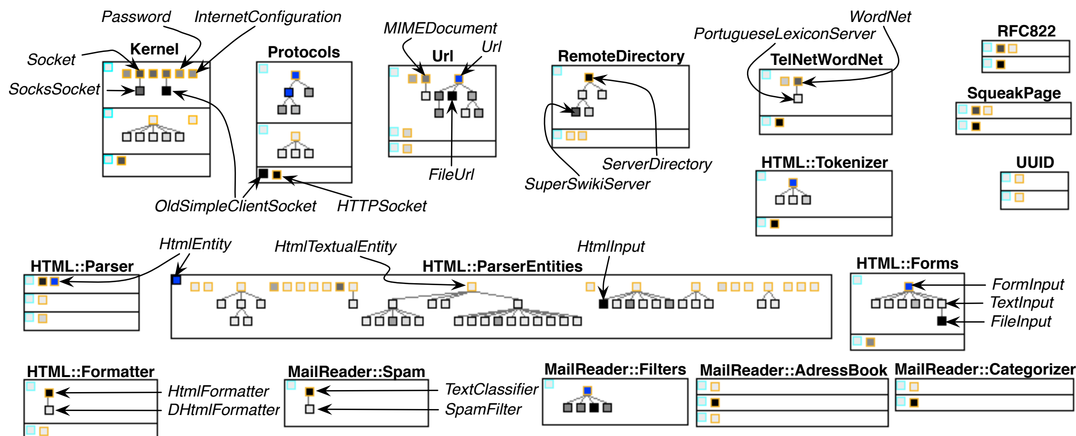


Figure 3.10: Inheritance global view in Network system

NetNameResolver, thus the former has a bigger number of incoming references than the latter.

Similarly we detect dominant referenced classes into other packages: the dominant referenced class in Url package is MIMEDocument; in Protocols package, it is HTTPSocket; in RFC822 package, it is MailAddressParser.

Leaf packages. Figure 3.9 (p. 53) clearly shows Network leaf packages (*i.e.*, packages which are referenced by only one package) such as MailReader::Filters which is only referenced by MailReader, or HTML::ParserEntities by HTML::Parser. We also identify packages which are completely isolated (*i.e.*, are not referenced by any package), since their blueprints contain only one surface (the head surface), such as SqueakPage and TelNetWordNet packages.

Low test coverage. During our analysis we found that almost no class is tested. For Protocols package, only the class SMTPClient has incoming references from a test class within Protocols::Tests, while most referenced class HTTPSocket has none (see the surface denoted by Protocols::Tests within the package blueprint of Protocols). Also for Url package, only four classes from twelve have incoming references from a test class within Url::Tests. Similarly for Kernel package, only two classes have incoming references from Kernel::Test and the class NetNameResolver has none. We thus deduce that the core packages of Network are not well tested, particularly their most important classes (classes which are heavily referenced from other packages) are not tested.

Internal/No Internal use. Package Blueprint stresses the different nature of packages. At first, we was surprised to see that some packages contain classes without any reference among them, while they are heavily referenced by external classes. For example, the reference package blueprint of MailReader::Filters (Figure 3.9 (p. 53)) shows that it contains classes without any package internal reference – since there is no gray boxes within the head surface. On the other hand, almost all classes of MailReader::Filters (four classes from five) are referenced by classes into the package MailReader (see the surface denoted by MailReader within the package blueprint of MailReader::Filter). The package HTML::Forms also presents such characteristics.

During our inspection, we found that these packages are defined around class inheritance instead of inter-class references: HTML::Forms is defined around the inheritance hierarchy of the class FormInput; similarly the package MailReader::Filters does (see inheritance package blueprint in Figure 3.10 (p. 53)).

Co-Referencers. Figure 3.9 (p. 53) shows that the packages RemoteDirectory and TelNetWordNet are referencing together the same set of packages within Network: both refer to classes into Kernel, Protocols and Url packages (see the green and orange surfaces within the body of blueprints). This gives us an idea about the similarity between RemoteDirectory and TelNetWordNet packages in terms of package co-referencing within Network.

Looking more closely at the referenced packages (in Kernel, Protocols and Url blueprints), we see that the similarity between RemoteDirectory and TelNetWordNet

is improper at the class granularity level: the referencing classes of RemoteDirectory and TelNetWordNet packages do not reference the same classes within the cited referenced packages.

Except for Protocols package, the classes ServerDirectory and SuperSwikiServer of RemoteDirectory package, PortugueseLexiconServer and WordNet of TelNetWordNet package, all refer to the class HTTPSocket.

For Url package, the classes SuperSwikiServer, PortugueseLexiconServer and WordNet refer to the class MIMEDocument class, while ServerDirectory refers to FileUrl.

However, we see that the classes PortugueseLexiconServer and WordNet of TelNetWordNet package, both refer to the same set of classes (HTTPSocket within Protocols and MIMEDocument within Url). It is worth to note that WordNet is a superclass of PortugueseLexiconServer (we can see that in the inheritance blueprint of TelNetWordNet, Figure 3.10 (p. 53)). Thus PortugueseLexiconServer inherits the behavior of WordNet. We think that it is a design defect that a subclass references the same set of classes as its superclass.

3.6.3 Inheritance Package Blueprint Overview

Finally during our case studies, thanks to the inheritance package blueprint, we identified a few remarkable usage patterns: a package can mainly contain big inheritance hierarchies (potentially a single one); classes in a package may inherit from superclasses within the system itself or from frameworks or the base system; or a package can specialize functionality and have few internal inheritance dependencies.

Mispackaged inheritance root. Figure 3.10 (p. 53) shows all the package inheritance blueprints of the Network subsystem in Squeak. It shows that there are only two places where classes inherit from classes within the Network subsystem scope: HtmlEntity and OldSimpleClientSocket. HtmlEntity is defined in HTML::Parser package and directly inherited by a lot of classes within HTML::ParserEntities package; OldSimpleClientSocket is defined in Kernel package and inherited by the class HTTPSocket within Protocols package. Note however that HtmlEntity class has blue fill color; this indicates that it is an abstract class.

Clicking on the HtmlEntity box, we can see that it is defined in the HTML::Parser package, away of all its subclasses defined in HTML::ParserEntities. We consider that it is defined in the wrong package.

Heavy inheritance structured packages. We can immediately spot that some packages are heavily structured around inheritance. Examples are: the package HTML::ParserEntities where the main class, in terms of inheritance, is HtmlTextualEntity; the package Url is structured around Url class which is internally inherited by almost all Url package classes; HTML::Forms or MailReader::Filters where both define a single hierarchy.

Heavy referencing classes. The overview also shows classes doing a lot of references (indicated as black boxes) such as HTTPSocket in Protocols package, HtmlFormatter in HTML::Formatter package, HtmlInput in HTML::ParserEntities package, FileInput in HTML::Forms package and TextClassifier in MailReader::Spam package.

However, in the context of inheritance, we should pay attention to the fact that all the subclasses of a class inherit its behavior and references. The case of HtmlInput in HTML::ParserEntities package and FileInput in HTML::Forms package is interesting: while they are inheritance leaves, they are darker than other classes, in particular than their superclasses, which means that they make direct references radically more than their superclasses; this indicates that such classes are complex and may heavily specialize their superclass inherited behavior.

This case is the reverse of that of DHtmlFormatter in HTML::Formatter package and SpamFilter in MailReader::Spam package – since these classes do direct references radically less than their superclasses.

3.6.4 The views together

While the views are simple, they convey powerful information. For Inheritance Package Blueprint, we can see that the percentage of black-bordered boxes reveals the amount of internal reuse. Orange-bordered classes that inherit from a cyan class indicate reuse of functionality from outside the system. Note that this is different from many orange-bordered classes inheriting from a black-bordered one (like with HtmlEntity in HTML::ParserEntities), since a lot of classes inherit from Object and indeed do not share the same domain. In contrast, inheriting from HtmlEntity clearly reuses its domain.

In addition to that, Inheritance Package Blueprint is an interesting complementary view to Reference Package Blueprint.

For example, in Section 3.6.1 (p. 49) we proposed to move the classes InternetConfiguration and SocksSocket from Kernel package to Protocols package – since these classes are not referenced within Kernel and reference the class HTTPSocket defined in Protocols package. The inheritance package blueprint of Kernel package (Figure 3.10 (p. 53)) shows that InternetConfiguration class has no inheritance relationship within Kernel, thus moving InternetConfiguration to Protocols package will not break any inheritance hierarchy within the package. package blueprint also shows that SocksSocket class inherits from Socket class within Kernel. Indeed the package Protocols has an inheritance relationship with Kernel package (HTTPSocket in Protocols inherits from OldSimpleClientSocket in Kernel) that does not affect SocksSocket. Moving SocksSocket to Protocols package will then not change the inheritance layering within Network system.

3.7 Striking Shapes

While applying blueprints to large software systems we identified some striking shapes that the blueprint, a surface or a class within a blueprint would produce. We present here most frequent ones.

3.7.1 Shapes of Packages and Surfaces

Sumo Package. A very large and tall reference blueprint denotes a package that makes a lot of references from many classes. Figure 3.11 (p. 57) shows an

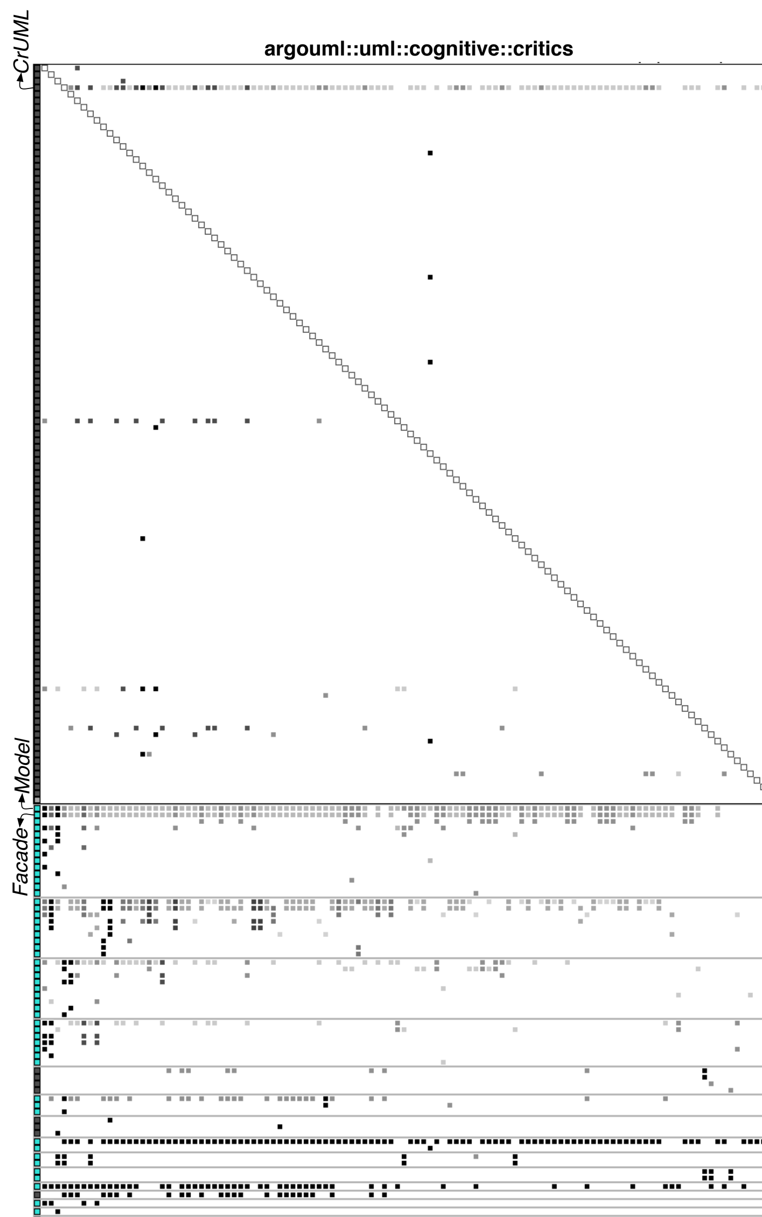


Figure 3.11: A Sumo Blueprint: the Critics package in ArgoUML. The view is in the context of the subsystem argoUML::uml.

example: the package Critics of ArgoUML that defines all the rules for assessing the quality of models.

Small House Package. A small inheritance blueprint with only a couple of surfaces and few inheritance hierarchies often denotes a package that offers a well packaged functionality, like Protocols (Figure 3.10 (p. 53)). Such blueprints are usually taller than larger.

Flat Head Package. A reference blueprint with a wide but flat head indicates limited internal references. HTML::ParserEntities in Figure 3.8 (p. 51) is flat head blueprint.

Exclusive External Referencer Package. When the first column in a outgoing reference blueprint is almost or completely cyan, the package makes most or all of its external references to classes outside the scope of the analyzed system. These packages typically extend a framework or a core library; Kernel in Figure 3.8 (p. 51) is an example.

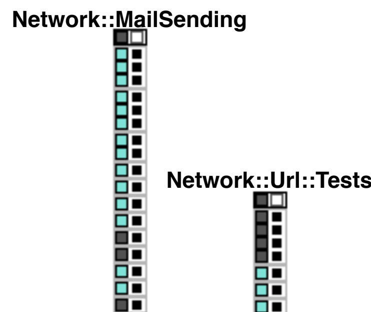


Figure 3.12: A Loner Blueprint: Url::Tests and MailSending packages in Network.

Loner Package. A loner is a package that contains only a couple of classes. It is often containing a single test case class. The blueprints of Url::Tests and MailSending packages in Figure 3.12 (p. 58) or MailReader::Categorizer, UUID, MailReader::Spam of Figure 3.10 (p. 53) are loners.

Tower Package. A reference blueprint with a small head and a thin body denotes

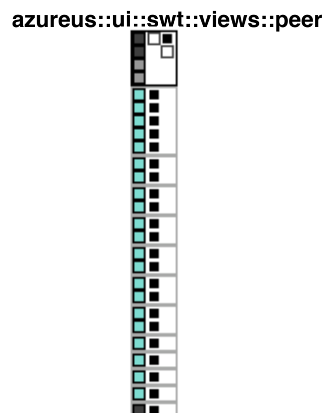


Figure 3.13: A Tower Blueprint: the Peer package in Azureus. The view is in the context of the subsystem azureus::ui::swt::views.

a package with few internal references but that makes many external references. This package may not be cohesive but highly coupled with the external packages. The package peer in Azureus is an extreme of this shape, as shown in Figure 3.13 (p. 58). In Figure 3.8 (p. 51), RemoteDirectory package has a more cohesive head and three classes intensively referencing external packages.

Large External Surface. When the topmost external surfaces are really large, like the four surfaces below the head in Figure 3.11 (p. 57), they identify packages that we must pay attention to, because changes in these external packages will very probably impact the package under analysis. In Figure 3.9 (p. 53), most right external surface of Kernel package blueprint, denoted by Protocols, and of MailReader::Filters, denoted by MailReader are large external surfaces. They indicate that those referencing packages, Protocols and MailReader, heavily depend respectively on Kernel and MailReader::Filters.

Dark Head Package. A package whose almost, if not all, classes have references among them will have a reference blueprint with a dark head surface, since there are a lot of gray boxes within the head surface; this denotes a package that is quite cohesive from the point of view of inter-class references. In Figure 3.9 (p. 53), SqueakPage has clearly a dark head.

White Head Package. This is the reverse case of *Dark Head Package*. In this pattern, the package contains classes that are loosely coupled (*i.e.*, the density of internal references among its classes is very small), since there are a very small number of gray boxes within the head surface. Such as the case of MailReader::Filters and HTML::Forms in Figure 3.9 (p. 53), which have clearly a white head.

3.7.2 Shapes of Classes

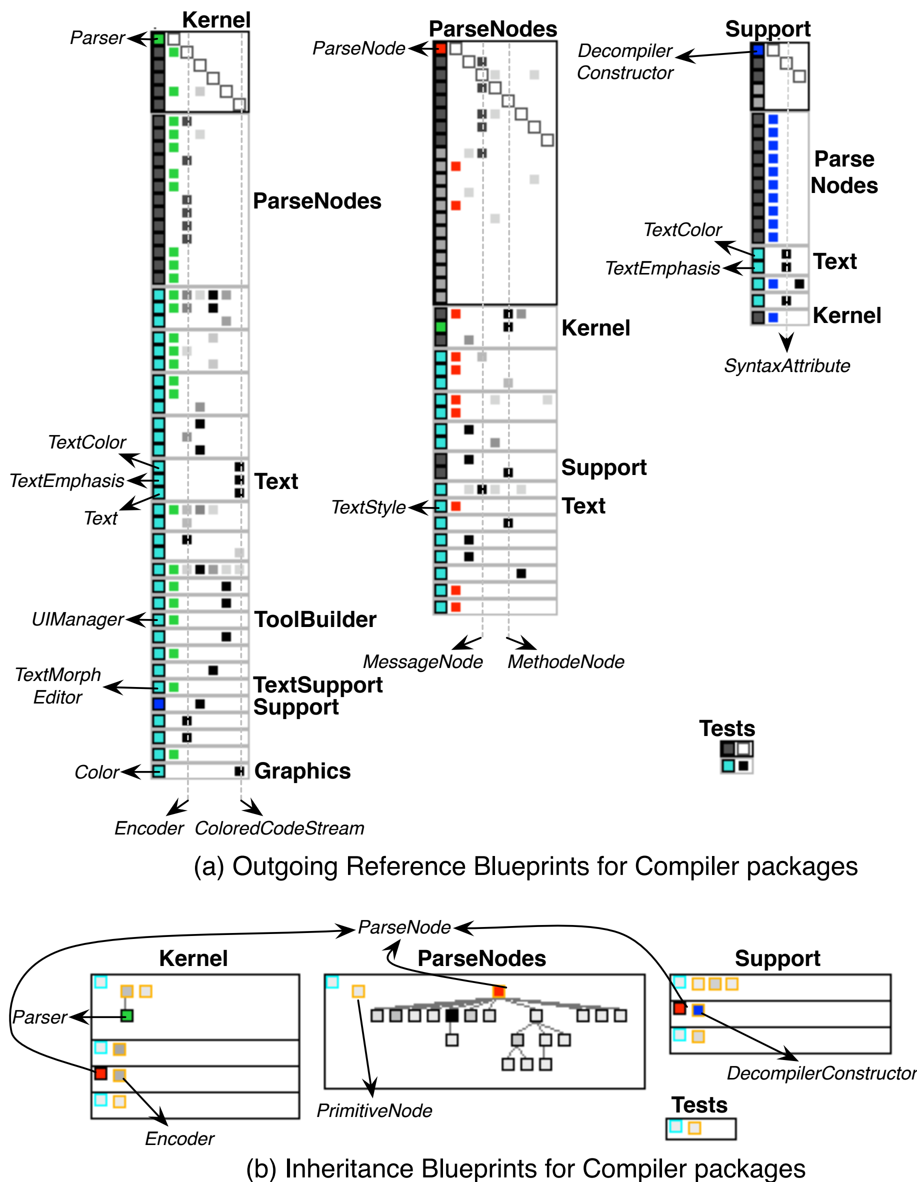
Main Referencer Class. A vertical alignment of dark squares in the body of a reference blueprint denotes a class that is responsible for many references to classes in other packages. The classes HTTPSocket and ServerDirectory are the main referencers in packages Protocols and RemoteDirectory; they are candidates to be central package classes (Figure 3.8 (p. 51)).

Main Internal Referencer Class. When vertical alignments are limited to the head, they reveal classes doing many internal and few external references. These classes often define the abstraction of the system. In Figure 3.8 (p. 51), the class Url only references classes within Url package.

Omnipresent Referenced Class. Classes of this kind are referenced by almost all the internal classes, and easily identifiable by filled rows in a surface. See the row of CrUML class in Figure 3.11 (p. 57). This makes sense for a facade class if it occurs a few times, but in ArgoUML we see this shape in most packages for Facade and Model (see Figure 3.11 (p. 57)); we may thus assess that the Facade pattern is misused.

3.8 User Case Study on Squeak Compiler

We performed a controlled user study to assess whether developers could easily use the blueprints. The case study we proposed is the Squeak Compiler system, which is composed of 4 packages (Kernel, ParseNodes, Support and Tests) containing a total of 33 classes (Figure 3.14 (p. 60)). We chose this case as we are familiar with the Squeak compiler. So we can better appreciate the information that the view provides. As in addition, compilers are systems that every one knows more or less, and, it is easier



(a) Outgoing Reference Blueprints for Compiler packages

(b) Inheritance Blueprints for Compiler packages

Figure 3.14: Global view in Compiler system. In this view, the class Parser is highlighted in green, the class ParseNode in red and the class DecompilerConstructor in blue.

for testers to make hypotheses. Another argument is that compilers generally contain interesting patterns and are not too small nor too big in terms of size and complexity.

3.8.1 Experimental Setup

We first explained the visualization to the testers and gave them the paper that describes the first version of the Package Blueprint [*Ducasse et al.*, 2007], as well as related slides. Then we showed them how to use our tool for detecting patterns in the Network system. The demo helps them learn how to use the views. Such demo is important since it shows step by step how to quickly get information from the views.

To define the questions, we have tested them internally to know if they are understandable and meaningful. Hereafter are the questions with comments on the rationale behind them:

1. *Can you identify the main abstractions/classes of each package?*

   With this question we want to know if the reader can quickly identify the main entities, and learn if they are meaningful.

   We know that the main classes in Compiler are: the class ParseNode defined in ParseNodes package, which is the top super class for all the parser node classes. It is also the super class of the classes Encoder and DecompilerConstructor (see Figure 3.14 (p. 60) (b)). The classes Parser and Encoder in Kernel package and DecompilerConstructor in Support package are main classes in Compiler. These classes uses heavily the ParseNodes classes and are the responsible of code parsing and compiling.

2. *Can you identify how these main classes interact within the package and within the system? Are there classes doing most of the internal/external references?*

   This question checks if testers get how a package blueprint helps them see the relationship between packages. The user learns how to select classes and finds if they are referenced or if they make references.

   It was clear for us that the ParseNode class and its subclasses are heavily referenced and used from the classes DecompilerConstructor, Parser and Encoder.

3. *How would you qualify the references from MessageNode class? Compare it to MethodNode?*

   This question checks if the testers are focusing on the understanding of a single package and comparing its classes.

   We know that MessageNode is the main referencing class in ParseNodes package: it refers and uses 5 classes of ParseNodes package and does few references to classes outside ParseNodes. While MethodNode class does not refer to any class in ParseNodes package and does many references to classes outside this package.

4. *How would you qualify the cohesion of Support package?*

   This question brings the user to focus on the head of the package blueprint as it gives information about the cohesion among enclosed classes. We check if the testers have understood that.

5. *Do you identify some misplaced dependencies with packages outside the compiler system?*

   By inspecting Compiler package view, we noticed several misplaced dependencies and we want to know if the user can find them. Those misplaced dependencies are: in Kernel package, the class ColoredCodeStream references classes Text, TextColor and TextEmphasis in Text package; the class Parser references TextMorphEditor class in TextSupport package and UIManager class in ToolBuilder package.

6. *Under the assumption that a package containing classes referenced by other packages should be loaded first, can you identify a loading order for the system?*

   We noticed that the three packages of the Compiler system depend on each other cyclicly, this just by clicking on each of them in the view. Does the user easily capture this?

7. *Using the inheritance view, what can you say about the shape of the ParseNodes package?*

   The parse node classes belong to the same hierarchy tree and are defined in one package. This is the common way of declaring a parser tree. It allows one to define visitors to walk the tree and it is easier to annotate it. We would know if the user arrives to this conclusion.

8. *Can you tell us something about the class ParseNode hierarchy?*

   We took the strangest things in the hierarchical view of the compiler to see whether the user can spot it out or not. All nodes inherit from ParserNode class except for PrimitiveNode class. This is not a good design since the node should be polymorphic to ParseNode.

9. *Do you think that you would have got the answers to our question in the time allocated without the help of visualization?*

   We want to know if the user finds our views handy to understand the compiler system. We also want to have suggestion to improve the usability of the view.

3.8.2 Results

The case study was conducted with 20 people, from master students to experienced researchers, with various programming skills and experience with software projects. We gave them a limited amount of time: one hour to perform the study.

For the first question, most of users identified the classes ParseNode and MessageNode in ParseNodes package, Parser in Kernel package, and DecompilerConstructor in Support package, as the main classes of the packages. This was expected: they did not identified Encoder class as a main class. They based their conclusions on the quantity of references and inheritance dependencies with these classes, and whether the referenced classes were within or outside of the system boundary. The remainder of users had exactly our estimation: they also identified Encoder as a main class in Kernel package – since it is referenced by the main class Parser.

For the second and third questions, all users have been able to correctly answer them.

For the 4th question, about 50% (11 users) of testers have identified the Support package cohesion as very low. Two users have identified Support package as very cohesive, without justifying their answers. The remainder of the users (9) skipped this question. We think that we had to define what we mean by package cohesion. On another hand, in the first version of the Package Blueprint [*Ducasse et al.*, 2007], packages that have not internal references do not have a head surface. Some users have found that such a visual mapping is not helpful, since they cannot analyze what they cannot see (*e.g.*, the package size, classes and internal references). In this version, we optimized the visualization in a way that users always see package size, classes and internal references, if any.

For the 5th question, only five users were not able to identify the misplaced dependencies that we identified. Some of them have declared that they did not understand the question. Similarly for the 6th question, most of users have found that the three packages (Kernel, ParseNodes and Support) depend on each other cyclicly and we cannot know the loading order of Compiler packages. Some of those users added that it is more probable to load the package ParseNodes at first – since other packages depend heavily on ParseNodes and extend its class ParseNode – this was a good answer. A couple of users said that we can easily and quickly answer this question by doing an automatic dependency analysis using Smalltalk cross referencer, rather than using the Package Blueprint.

For the 7th and 8th questions, more than 50% (12 users) captured that ParseNodes package contains a single domain defined by the class ParseNode. They found that the hierarchy of ParseNode class is coherent. They also spotted that it is not normal that the class PrimitiveNode does not belong to the hierarchy of ParseNode class. The remainder of users skipped these questions and mentioned that the questions are confusing.

Almost all the users concluded positively to the last question indicating that the visualization was useful. They underlined that the package blueprint was helpful to extract information about the Compiler system in a very short time and indicate that they would need really more time to do the same thing without the package blueprint.

Some of the testers proposed enhancements such as adding a fly-by-help to explain the nodes to ease the learning curve of the visualization. All those propositions were integrated in the package blueprint new version presented in this chapter.

3.9 Evaluation and Discussion

3.9.1 Evaluation

As illustrated in Section 3.6, Package Blueprint allows its user to extract information from the internal structure of a package, its clients as well as the provider packages it uses. Now we revisit some of the information that we listed in the beginning of the chapter.

Size. Package Blueprint highlights the complexity of the observed package in several

dimensions. For outgoing reference blueprints, the height of the body indicates the amount of external classes referenced, whereas the number of surfaces shows the number of referenced packages. Each individual surface height shows how many classes are referenced in the corresponding package. This gives us an estimate of the coupling between the package and this surface; to further evaluate the coupling strength, we should also look at the intensity of referencing classes in the surface because it represents the number of references. In addition, surface width indicates the number of referencing classes.

Combined together these visual properties offer a quick impression not just about the visualized package, but also about its classes: a thin package with a long body depends on a lot of classes because of few internal classes. If moreover the blueprint is heavily lined, *i.e.*, it references a lot of packages, so some of its referencing classes may be complex and fragile.

The same situation occurs with incoming reference and inheritance blueprints but from the view point of referencing packages/classes and inheritance dependencies.

Central or Peripheral. For outgoing reference blueprints, by looking at the border color of external classes (cyan or black), we can easily see if a package depends a lot on the framework or on the system. Also, through incoming reference blueprints, we can see if a package is used by different subsystems (central) or just by specific ones (peripheral).

Cohesion and Coupling. package blueprint also makes it possible to roughly compare how several packages are coupled with the observed one: larger surfaces indicate coupling to more classes and are positioned nearer to the head surface, while surfaces with more darker class squares represent packages which are more coupled in term of sheer number of references. We can also estimate cohesion by comparing internal coupling (size and overall intensity of the head surface) and external coupling.

Co-changes and Impact Analysis. Because package blueprint details how packages depend on each other, it hints at the fragility of the observed package to changes. Selecting a package or a class highlights surfaces or classes that reference the selected entity and are thus sensitive to its changes.

3.9.2 Discussion

Our approach has worked well on our case studies (it helped us to get important structural information efficiently). It should be noted that we were *not* familiar with the case studies such as the Network system before applying our approach. We have been able to locate many conceptual bugs. Our first evaluation with end-users is also promising, even if we are aware that the number of participants was not significant for drawing larger conclusions.

In conjunction with other tools. We do not consider that package blueprint should be used in isolation. In our recent work on remodularisation, we use DSM [*Laval et al.*, 2008, 2009] to spot cyclic dependencies, then we zoom on the packages and

use package blueprint to get a finer understanding of the package references. The synergy between DSM and package blueprint proved to be really useful. In addition, sometimes we complement the view using Distribution Map [*Ducasse et al.*, 2006a,b] to understand how a property (such as developers) spreads on a set of packages.

Let us now discuss some of the visualization choices we made.

Position Choices. We grouped the internal references at the top of the package blueprint, then ordered the surfaces from the ones having most external references at the top to the least at the bottom; inside a surface, we also ordered the rows from most referencing ones to the least. This way, we do not force the reader to scroll through big visualizations, and use the fact that the reader pays more attention to the top elements than to the bottom ones. We also tried to layout surfaces compactly so that we can easily move them. According to this principle, internal classes that do not do any reference are placed in the bottom of the left most column in the head.

Seriation. Rows within a surface are sorted according to the number of references they contain. In an earlier version we applied the dendrogram seriation algorithm [*Jain et al.*, 1999] to group lines having similar referencing classes. However the resulting views were not as meaningful as with a simple ordering. We thus plan to use seriation to group packages having similar surfaces *i.e.*, packages using similar packages.

In a package blueprint head, internal classes are ordered so that the head presents a symmetric matrix. This way, when the user focuses on the *i* column (*i.e.*, a column reserved for class *x*) s/he can easily see the information about the internal references within the package of this class by looking to the *i* row in the package blueprint head. Such an ordering reveals also the direct cyclic references within the package under consideration. In previous versions, the head only showed classes performing references [*Ducasse et al.*, 2007] and our users suggested such a change to be able to grasp package size.

Impact of Boundaries. We color classes that do not belong to the system in cyan. This way, users distinguish clearly the dependencies from/to classes packaged outside the analyzed system, from the dependencies among the analyzed system classes. This is a bit limited in inheritance blueprints because we do not distinguish well the true root classes —*e.g.*, Object or Model in Squeak — from other classes that are packaged outside the analyzed system.

We found it really effective to color surfaces so that the user can interactively mark entities on which s/he wants to focus on; this increases the usability of the tool and speeds up understanding packages.

Shapes. For the time being we represent the classes with squares only. We could convey more information by using several visually distinct shapes. But it is not clear which ones and how efficient the results will be since the shape size is intentionally quite small to provide a compact overview.

Package Nesting. Currently we do not support package nesting. A solution like the one proposed by Lungu *et al.* seems complementary to ours and interesting to deal with package nesting [*Lungu et al.*, 2006].

Outgoing vs. incoming. Having two views showing different flows of dependencies can be confusing and it took us several attempts and experiments to find a solution so that the reader can distinguish the incoming and outgoing flows.

3.10 Related Work

Several works provide or visualize information on packages. Many of these approaches treat software co-change, looking at coupling from a temporal perspective, whereas in this chapter we focus on the static structure of dependencies [*Beyer*, 2005; *Eick et al.*, 2002; *Froehlich* and *Dourish*, 2004; *Storey et al.*, 2005; *Voinea et al.*, 2005; *Xie et al.*, 2006].

Lungu *et al.* guide exploration of nested packages based on patterns in the package nesting and in the dependencies between packages [*Lungu et al.*, 2006]; their work is integrated in Softwarenaut and adapted to system discovery.

Sangal *et al.* adapt the dependency structure matrix from the domain of process management to analyze architectural dependencies in software [*Sangal et al.*, 2005]; while the dependency structure matrix looks like the package blueprint, it has no visual semantics.

Storey *et al.* offer multiple top-down views of a system, but these views do not scale very well with the number of dependencies [*Storey et al.*, 1997].

Ducasse *et al.* present Butterfly, a radar-based visualization that summarizes incoming and outcoming dependencies for a package [*Ducasse et al.*, 2005b], but only gives a high-level client/provider trend.

In a similar approach, Pzinger *et al.* use Kiviat diagrams to present the evolution of package metrics [*Pinzger et al.*, 2005].

Chuah and Eick use rich glyphs to characterize software artefacts and their evolution (number of bugs, number of deleted lines, kind of language...) [*Chuah* and *Eick*, 1998]. In particular, the time wheel exploits preattentive processing, and the infobug presents many different data sources in a compact way.

D'Ambros *et al.* propose an evolution radar to understand the package coupling based on their evolution [*D'Ambros* and *Lanza*, 2006b]. The radar view is effective at identifying outliers but does not detail the structure.

Those approaches, while valuable, fall short of providing a fine-grained view of packages that would help understanding the package shapes (the number of classes it defines, the inheritance dependencies of the internal classes, how the internal classes inherit from external ones...) and support the identification of their roles within an system.

3.11 Conclusion

In this chapter, we tackled the problem of understanding the details of a package with a focus on its dependencies in terms of relationships among classes. We described

the Package Blueprint, a visual approach for understanding package dependencies. Package Blueprint is a compact visualization supporting large overview without losing the essential details (references and inheritance among classes). Therefore it can be used to get a first impression of a system and also to understand fine-grained structures and relations.

While designing the Package Blueprint, we tried to exploit gestalt visualization principles and preattentive processing. We successfully applied the visualization to several large software systems and we have been able to point out core classes, misplaced ones, and badly designed packages. We also introduced interactivity to help the user focus and navigate within the system.

We validated the Package Blueprint usability by conducting tests with several software maintainers. The results were positive, even if the number of testers was low (20). Testers concluded that the Package Blueprint is useful for understanding and analyzing packages. They specially underlined that the Package Blueprint helps them to reduce the time and effort during maintenance tasks.

In this chapter, we showed that the Package Blueprint helps to understand package structure: *i.e.*, to answer the quantitive information questions that we listed in Section 2.3.1 (p. 23). We also showed that it helps to assess package cohesion based on class internal dependencies. This cohesion approach is most related to the Common-Closure Principle (CCP) of package cohesion (Section 2.3.2 (p. 23)). The limitation of the Package Blueprint is that it does not help well to understand: (1) package role and contextual information (Section 2.3.3 (p. 25)); (2) the usage of package interfaces and package cohesion based on the Common-Reuse Principle (CRP: Section 2.3.2 (p. 23)).

Chapter 4

Package Fingerprints: Visually Summarizing Package Interface Usage

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

4.1 Introduction

In the previous chapter we defined the Package Blueprint visualization which presents a condensed view of a package in terms of its relationships to other packages. It acts as a map and puts in situation the references between packages. Although that a package blueprint provides a compact view and shows dependencies on a per-class basis, it does not help users to group from a usage perspective the client/provider packages for the package under analysis, nor to identify the cardinalities of package interfaces and their use. But, as we underlined in Section 2.3 (p. 21), maintainers need understand package contextual information (Section 2.3.3 (p. 25)) and assess package design quality (*i.e.*, the principles of package cohesion: Section 2.3.2 (p. 23)).

On the other hand, in the existing literature of package design quality [*Abreu* and *Goulao*, 2001; *Melton* and *Tempero*, 2007; *Ponisio* and *Nierstrasz*, 2006; *Rising* and *Calliss*, 1992], we distinguish two main approaches for assessing package cohesion. The first approach defines the cohesion of a package in terms of the interdependencies between its internal classes. The second approach defines cohesion according to how the system uses the package classes. For instance, *if two classes of a package are used from the same client package, then they are considered as conceptually coupled, regardless of the explicit relationships that exist between them* [*Ponisio* and *Nierstrasz*, 2006]. This approach is meaningful to us, because we consider a package as functionality provider and not only a structural grouping of inter-dependent classes. *Contextual Cohesion* is the name we use for package cohesion regarding this approach.

As we explained in Section 2.5 (p. 29), many metrics have been defined to compute package cohesion and determine packages that are candidates for restructuring.

However, those metrics do not help maintainers when they face the problem of understanding how packages are used in general and how packages are in relation with each other in their provider/client roles. On another hand, the existing works on software visualization, together with the Package Blueprint presented in Chapter 3 (p. 39), fall short of providing a fine-grained view of packages that would help maintainers understand and assess the package interfaces, their usage and their contextual cohesion.

Contribution of the chapter

In this chapter, we present the *Package Fingerprint, a compact, rich and zoomable visualization to better support the understanding of package interfaces, relationships and the conceptual coupling of package classes (i.e., package contextual cohesion)*. The goal of this visualization is to help maintainers during their early contacts with unknown packages. We propose two complementary variants of the Package Fingerprint, structured around the distribution of references from or to the classes of the analyzed package: the *incoming fingerprint shows how the system uses the package classes, and highlights the cohesion of the analyzed package*, as defined by Ponisio [*Ponisio* and *Nierstrasz*, 2006]; the *outgoing fingerprint shows how the package classes use the system*.

The content of this chapter was the object of our paper submitted to the *Information and Software Technology (IST) journal* [*Abdeen et al.*, 2009a], as an extension of our paper published in *CSMR'08* [*Abdeen et al.*, 2008].

Structure of the chapter

In Section 4.2 (p. 70) we present the principles of the incoming and the outgoing fingerprints. Then, in Section 4.3 (p. 76), we show how to use the incoming fingerprint in practice, for analyzing and understanding package interfaces and their contextual cohesion. Section 4.4 (p. 79) presents the different zoom levels of a fingerprint and shows how to read a fingerprint from far away. Section 4.5 (p. 82) presents the outgoing fingerprint via a simple example, and Section 4.6 (p. 85) lists the relevant visual patterns in incoming and outgoing fingerprints. Finally, we discuss our approach and conclude in Section 4.7 (p. 98) and Section 4.8 (p. 102).

4.2 Package Fingerprint Principles

Our aim is to provide an approach that helps maintainers understand packages in their context, regardless of what happens inside packages – since this is considered as a hidden-information from the point of view of its system [*Ponisio* and *Nierstrasz*, 2006]. We will focus on a package as a provider and/or client offering and/or requiring functionalities to/from other packages within a system.

We propose two complementary views for *incoming* and *outgoing* references through the Fingerprints. The objective of Package Fingerprints is to provide an overview of package cohesion and coupling by stressing the client/provider relationships of the classes contained in the considered package. As such it is complementary

to traditional coupling/cohesion metrics [*Arisholm et al.*, 2004; *Briand et al.*, 1998]. Before going in detail, we setup the vocabulary and the intention of Fingerprints.

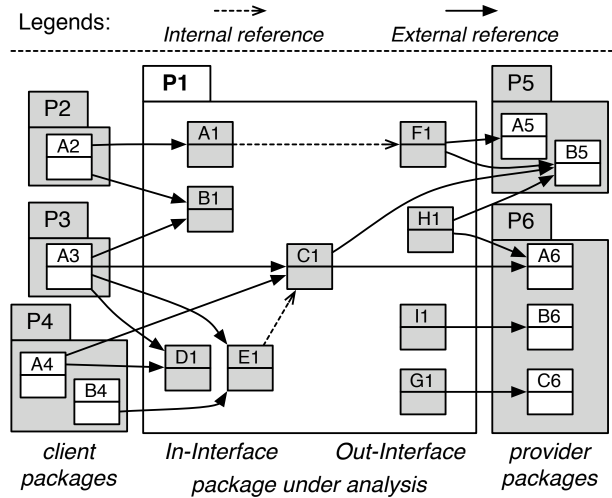
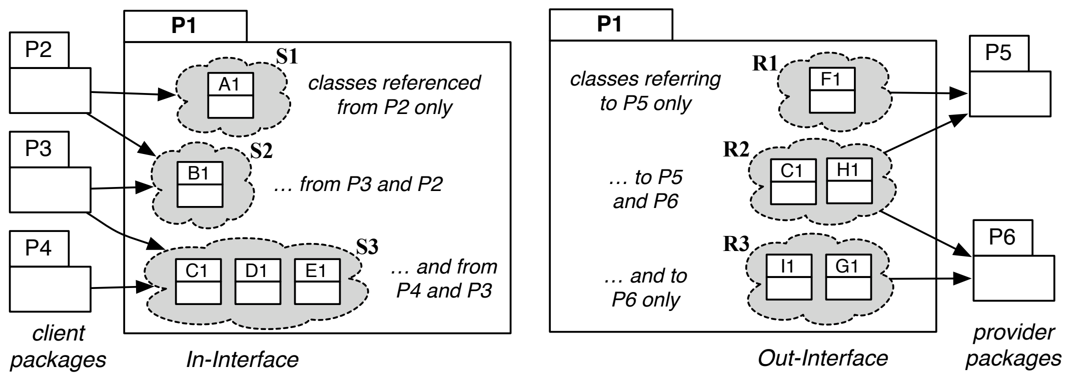


Figure 4.1: Terminology – An example of references between packages



(a) Grouping the classes of the In-Interface of P_1 by common *client* packages.

(b) Grouping the classes of the Out-Interface of P_1 by common *provider* packages.

Figure 4.2: Grouping incoming and outgoing references into In- and Out- interfaces.

4.2.1 Terminology

As shown in Figure 4.1 (p. 71) and 4.2, the size (*i.e.*, number of classes) of the In-Interface gives maintainers a quantified information about the dependency of the system on the package under-analysis P_1, while the number of referencing packages shows the importance of P_1 for the system. Similarly, the size of the Out-Interface of P_1 gives maintainers a quantified information about the dependency of P_1 on other

packages, while the number of referenced packages shows how much P_1 depends on the system.

Since referencing a class is an indicator of the usage of that class functionalities, referencing a group of classes in a consistent way is an indicator of the usage consistency of those classes. Such a referenced group, that we name a service, represents classes whose functionalities are consistently used together.

Definition **7 (Service)** *In the context of a package P, we mean by Service, the set of classes of P In-Interface which are referenced together by the same group of packages.*

On another hand, Martin [*Martin*, 2002b] defines a class responsibility as *a reason for change*. From the view point of inter-class references, if a class A refers to another one B, changes in B may be a reason for changes in A. At a high level of abstraction, if A refers to a package P, changes in P may be a reason for changes in A. In this context, we define a package reason for changing as follows:

Definition **8 (Reason for Changing)** *In the context of a package P, we mean by Reason for Changing, the set of classes of P Out-Interface which refer together to the same group of packages.*

4.2.2 Fingerprint Intention

To understand the multiple facets of a package, we group its classes according to their usage by other packages and their usage of other packages. Figure 4.2(a) (p. 71) shows the In-Interface classes of P_1 grouped into clusters as well as the references that point to those clusters, while Figure 4.2(b) (p. 71) shows the Out-Interface classes of P_1 grouped into clusters as well as the references that go out from those clusters. Figure 4.2(a) (p. 71) shows that P_1 provides three services (S_1, S_2 and S_3): the service S_3 is used by the client packages P_3 and P_4; additionally, P_3 with P_2 use the service S_2; the service S_1 is used by the client package P_2 only. Figure 4.2(b) (p. 71) shows that P_1 involves three reasons for changing (R_1, R_2 and R_3): R_1 represents the class F1 which refers to P_5, R_2 represents the classes C1 and H1 which refer to P_5 and P_6, while R_3 represents the classes I1 and G1 which refer to P_6.

Clustering the In-Interface and Out-Interface helps identifying the inter-dependencies between the package under analysis and the system, and thus which classes are conceptually coupled and which classes are not. At a higher level of abstraction, this helps answering the following questions:

- What services does the package provide?
- Which packages use those services?
- Does the package include classes that are always used together or not?
- Does the package include classes that use the same services/packages or not?
- Which are the reasons for changing the package?
- How are those reasons for changing distributed over the package classes?

The incoming fingerprint shows how the package under analysis is *used* by the system and how this use is distributed over its classes. The outgoing fingerprint shows how the package under analysis uses the remainder of the system. Since we use the same approach for both views, we only present the incoming fingerprint in details and briefly sketch the outgoing fingerprint further on.

Fingerprints have the four following properties: they are *compact* (only the references are shown), *zoomable* (different levels of information are proposed), *entity-based* in the sense that they focus on one package, and *semantically rich* since they present multiple types of information at a glance.

Figure 4.3 (p. 73) depicts the key visualization principles of an incoming Fingerprint with P_1 from Figure 4.1 (p. 71) as the package under analysis. We first present the basic layout before introducing additional features we give to convey more information on package relationships.

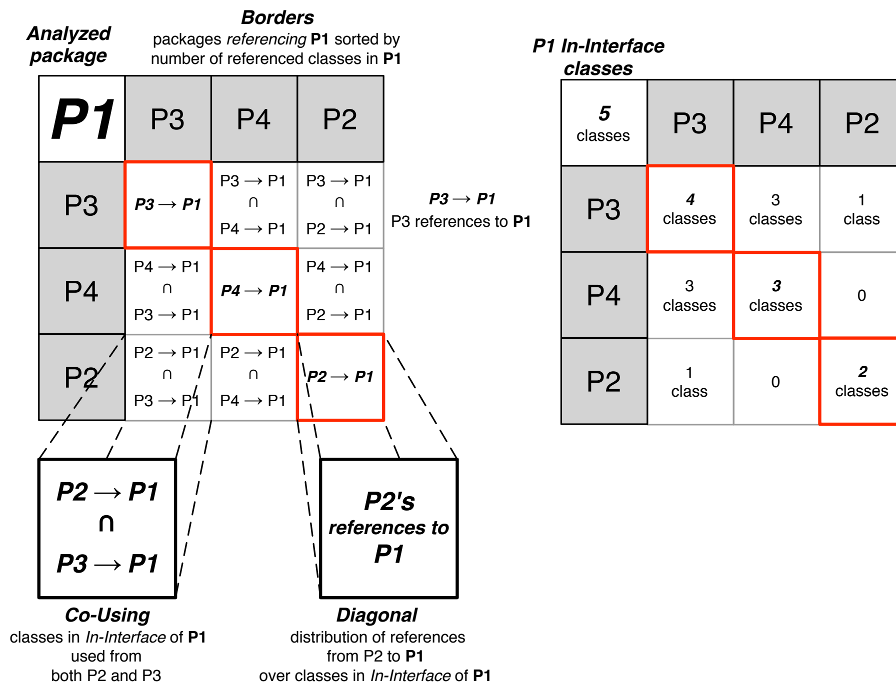


Figure 4.3: The Incoming Fingerprint skeleton with P_1 (Figure 4.2(a) (p. 71)).

4.2.3 Fingerprint Skeleton

The Fingerprint skeleton layout is the following:

Analyzed Package. The *top left corner* cell indicates global information about the package under analysis (here P_1): the size of its In-Interface and the internal references between its classes. Internal references are explained and illustrated in Section 4.2.4 (p. 74).

Referencing Packages. The cells at the *borders* of the fingerprint, *i.e.*, the leftmost column and the topmost row, both represent the referencing packages placed in the same order horizontally and vertically (*i.e.*, there is a symmetry). Packages are sorted according to the importance of their references: the more referenced classes a package refers to, the closer it is to the top left corner. Figure 4.3 (p. 73) shows the three packages that refer to P_1 in Figure 4.1 (p. 71): P_3, P_4, and P_2, referencing respectively four, three, and two classes inside P_1.

If two packages make the same number of references, we then group them using a similarity criterion. We define this latter in an incoming fingerprint, as the number of shared referenced classes among packages. For example, in Figure 4.2 (p. 71), we consider that P_4 is more similar to P_3 (3 referenced classes in common) than to P_2 (no referenced class in common). Conversely, we define the similarity of referenced classes by the number of referencing packages they share. Figure 4.2 (p. 71) shows that the similarity between C1 and D1 (2 common referencing packages P_3 et P_4) is higher than the similarity between C1 and B1 (1 common referencing package P_3). In any case, the ordering algorithm we have implemented always respects the number of references prior to similarity.

Cells. The *body* cells of an incoming fingerprint, *i.e.*, all cells except those on the leftmost column and the topmost row, each represents a subset of the In-Interface of the package under analysis. This subset contains the classes that are referenced by *both* packages placed at the heads of the cell's row and column. For a package P that is referenced by $P_1, \ldots, P_n$, a cell on row i and column j, $cell(i, j)$, represents the subset of classes of P that are referenced by both P_i and P_j (*i.e.*, $cell(i, 1)$ and $cell(1, j)$). Two situations occur: either a cell is on the *main diagonal* or not.

- The *main diagonal* presents the distribution of the In-Interface on the client packages. Figure 4.4 (p. 75) shows that $cell(3, 3)$ represents the classes (C1, D1, E1) referenced by P_4, *i.e.*, $cell(3, 1)$ and $cell(1, 3)$.
- The other cells present the classes referenced in *common* by both packages represented by the row and column heads. Figure 4.4 (p. 75) shows that $cell(2, 4)$ contains the class B1, referenced by both P_3 and P_2.

We define the size of a cell as the number of classes it represents. Hence in Figure 4.4 (p. 75), $cell(2, 2)$ has as size 4 and $cell(3, 3)$ has as size 3: both cells represent the classes C1, D1, and E1; in addition the $cell(2, 2)$ represents the class B1.

4.2.4 Enriching the Fingerprint Skeleton Layout

We enrich the skeleton of Figure 4.3 (p. 73) to convey extra information such as the amount of referenced classes in the analyzed package. For this purpose we use color intensity for cells, cell borders, and the position of classes within cells.

We selected those visual properties according to several research works that address the characteristics of efficient visualizations [*Tufte*, 2001; *Ware*, 2000]. Particularly, as our focus is on providing a first impression of a package and its context, and as we did in the previous chapter (Chapter 3 (p. 39)) for the definition of the

Package Blueprint, we want to exploit preattentive processing as much as possible to help spotting important information; and we stress that the Package Fingerprints visualization should respect the properties that we cited in Section 3.2 (p. 40).

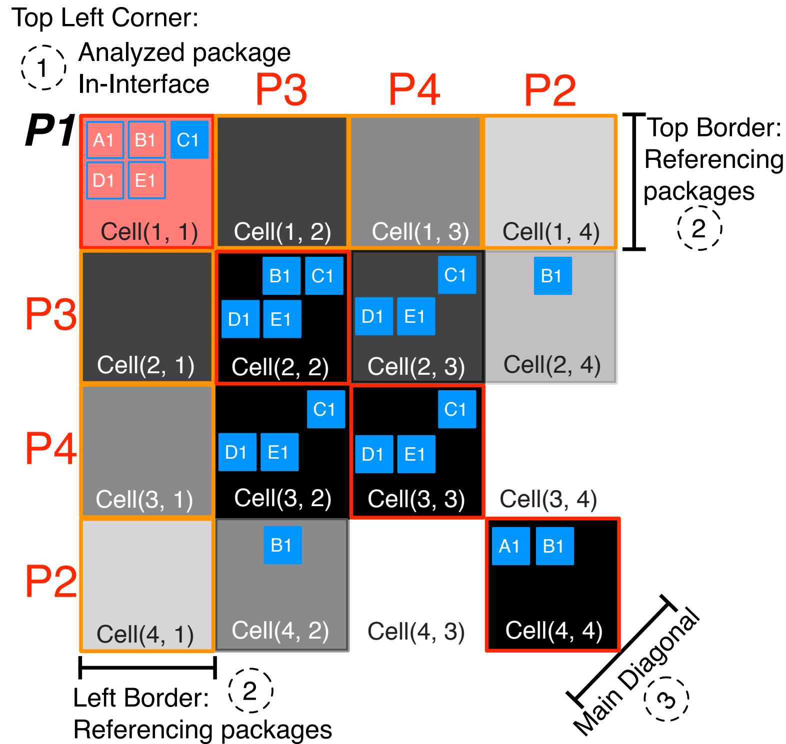


Figure 4.4: Showing the Incoming Fingerprint of P_1 (Figure 4.3 (p. 73)) with the classes involved in the relations inside each cell.

Cell Internals. Inside a cell, we visualize the package referenced classes as small filled squares.

To enable preattentive processing [*Healey et al.*, 1993], we give each class a fixed place which is the same in all the cells of a fingerprint. When a cell represents a package reference to a class of the analyzed package, the location of this class is colored: in Figure 4.4 (p. 75), since the class B1 is referenced by packages P_3 and P_2, the position corresponding to the class B1 is colored in the $cell(2, 4)$. This way all the cells will have the same geometrical size (*i.e.*, height and width), but the number of classes represented by the cell is given by the number of the colored squares inside that cell.

Information on Internal References. Information on internal references among classes of the analyzed package is visualized on the *top left corner* ($cell(1, 1)$). In Figure 4.4 (p. 75) we see that among the five referenced classes of P_1, only C1 is referenced internally (as it is colored). Additionally, since not all classes will appear in all cells, we use this corner cell to show all the placeholders for the classes that have incoming references, as bordered squares.

Colors. We use color hues to distinguish different entities in the fingerprint (*e.g.*, classes, packages), and to give more information about the references. The colors we use are: (1) shades of grey for all the cells in a fingerprint except the top left corner, (2) blue for the classes (3) red for the top left corner and for highlighting the borders of the main diagonal cells (4) orange to highlight the fingerprint borders, (5) gold to highlight borders of the referencing packages that are outside the scope of the system under analysis (called stubs thereafter).

Color Intensity. In addition to color hues, we use color *intensity* to give more information on the visualized entity: (1) for the top left corner, the darker the package, the bigger its In-Interface; (2) for the fingerprint borders, the darker a referencing package, the more classes it references in the analyzed package; (3) for the body, on *a given row*, the darker the cell, the more classes it represents. The darkness of a cell is calculated relatively to the size of the diagonal cell of that row. As consequence, the cells of the diagonal are black. On the fingerprint borders, we consider the color intensity for a referencing package as an additional visual information: as referencing packages are sorted according to the importance of referenced classes and similarity criteria (Section 4.2.3 (p. 73)), we use a same color intensity for referencing packages with a same number of referenced classes. Indeed, those packages are placed in different order but have the same color intensity. Figure 4.4 (p. 75) shows that P_3 is darker than P_4: the first package refers to 4 classes in P_1 while P_4 refers to 3 classes in P_1.

The color of the top left corner is based on an In-Interface size ratio: the size of the In-Interface of P_1 is 5 (Figure 4.2(a) (p. 71)) while the size of P_1 itself is 9 (Figure 4.1 (p. 71)). Thus the color intensity of this cell equals $5/9$.

In Figure 4.4 (p. 75), $cell(2,3)$ is darker than $cell(2,4)$, because the first contains 3 classes while the latter contains 1 class; $cell(4,3)$ is white (*i.e.*, the color intensity is zero) because no referenced class inside. $cell(3,2)$ is darker (it is black) than $cell(2,3)$ although they both contain the same set of classes: the reason is that the darkness of the former is relative to the size of $cell(3,3)$ while the darkness of the latter is relative to the size of $cell(2,2)$. This darkness relativity informs us that: for P_1, all the classes referenced by P_4 are also referenced by P_3 but some classes referenced by P_3 (*i.e.*, B1) are not referenced by P_4.

4.3 Decorticating a Fingerprint

In the following section we present an example that illustrates how a fingerprint is used to analyze package references. Figure 4.5 (p. 77) shows the incoming fingerprint of the Jboss render::renderer package (referred to as P here), visualized in the context of his subsystem, named theme. As a whole, Jboss is composed of 499 packages; theme is composed 15 packages totaling up 119 classes .

No Internal Reference. As depicted by Figure 4.5 (p. 77), none of the small squares on the top left corner cell (P) is filled: this means that there is no internal reference within the considered package. Actually, this package only contains Java interfaces.

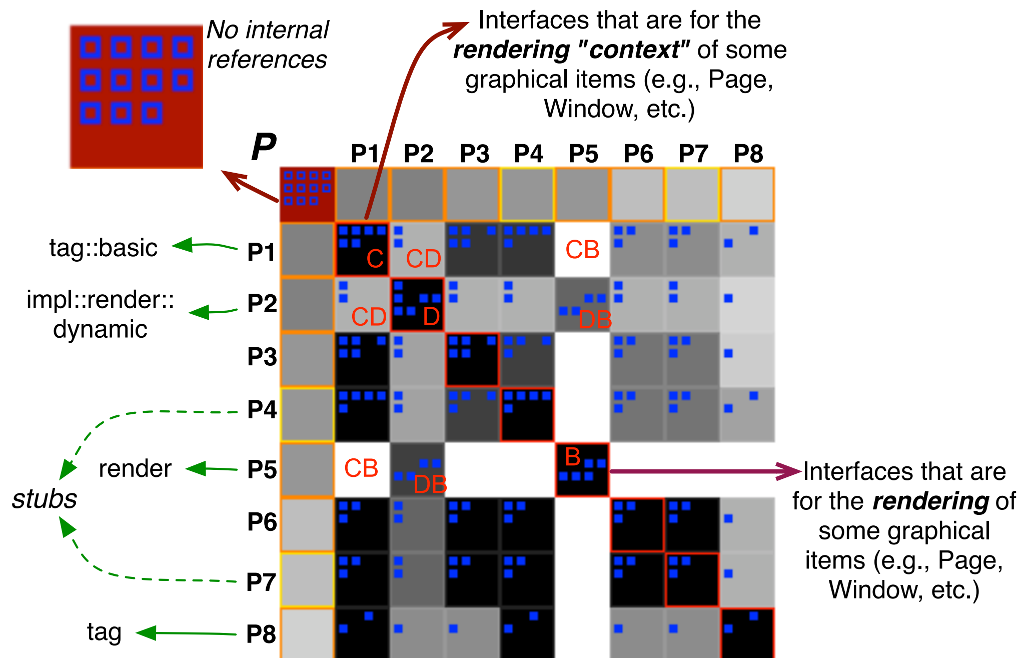


Figure 4.5: The Incoming Fingerprint of the package render::renderer, from the theme subsystem of Jboss.

Big Number of External Incoming References. The top left cell P is dark red, therefore most of the classes of render::renderer have incoming references from other packages. By looking at the number of squares in cell P we can estimate the size of its In-Interface (11 classes here).

Small Number of Referencing Packages. The fingerprint has a relatively small number of rows and columns: only 8 other packages reference classes of the package under analysis.

Two external packages, P_4 and P_7, have a gold border color, rather than orange. This means that they are stubs, *i.e.*, they are not part of the system under analysis theme. Indeed, when moving the cursor over these cells a fly-by-help reveals their names test::theme and test::theme::renderer. Thus those two packages are part of the test subsystem rather than theme, and probably mainly contain test classes. Moreover, since P (render::renderer) is only used by 6 of the 15 packages of his subsystem and 2 external test packages, it does not have a direct role outside the subsystem theme. Thus we can qualify render::renderer as a peripheral package.

Commonly Referenced Classes. Since the small squares representing classes keep their positions in every cell, they make it possible to spot patterns. For instance, most cells in the rows of P_6 and P_7 show the same 3-square shape, highlighting commonly referenced classes.

Dominant Package. As P_1 is the top/left-most package, we know that it makes most references to P. We can also see that all cells in the column of P_1 are black; this means that the corresponding packages (P_3, P_4, P_6, P_7 and P_8) refer to subsets of the classes that are referenced by P_1: P_1 is thus a dominant referencer of P.

Classes with different reasons for changing. At a first glance, the fingerprint body looks quite filled up: only one cell of the main diagonal (B) breaks the fill and causes a white cross hair shape. A white cell means that there is no shared reference to P between the two packages for this cell, *e.g.*, there is no shared reference between P_1 and P_5, nor between P_3 and P_5, *etc.*

The cell B contains 5 squares, for the 5 classes referenced by the package P_5. Cells denoted by DB represent the non empty intersection of cell D with cell B, *i.e.*, the four classes referenced from both P_5 (cell B) and P_2 (cell D).

Examining cell CD, which represents the common referenced classes from both P_1 (cell C represents 6 referenced classes) and P_2 (cell D represents 6 referenced classes), reveals that P_1 and P_2 have only 2/6 referenced classes in common. For this reason cell CD is lighter than cells C and D. Similarly, cell CB, which represents the common referenced classes from both P_1 and P_5 (cell B), reveals that P_1 and P_5 do not have common referenced classes. For this reason cell CB is clearly lighter (*i.e.*, white) than cells C and B.

Thus we learn that the analyzed package contains two disjointed subsets of classes: the first one with 6 classes (cell C) represents the subset which is referenced by all the client packages except P_5; the second one with 5 classes (cell B) represent the subset which is referenced only by P_5 and P_2. P_2 refers to classes of both subsets, but it refers to 4 classes from B (DB) and just 2 from C (CD). These subsets (C and B) hint at a possible way to split P into two more cohesive packages.

Based on that, we suspect that it is possible to re-modularize the package, for example by moving C classes to a new package. This will make the package under analysis (P) conceptually more cohesive while providing one group of classes (B) used together by P_5 and P_2. We check this hypothesis by reading the code of B and C classes. We learn that B classes represent the interfaces of item renderings (*e.g.*, PageRenderer, WindowRenderer, *etc.*), while C classes are the interfaces of item rendering contexts (*e.g.*, PageRendererContext, WindowRendererContext, *etc.*). The referencing package impl::render::dynamic (P_2) contains classes that implement some of the interfaces of B. The referencing package render (P_5) contains the class renderContext that refers to B interfaces. This class renderContext, which implements the facade pattern, is responsible of the communication with different objects whose types are declared via the interfaces (*e.g.*, PageRenderer, WindowRenderer, *etc.*). C interfaces are implemented by classes contained in different packages (*e.g.*, tag::basic, tag) which are responsible of different contexts of item rendering.

Reading the code reinforced the difference in the usage of both interface collections (B and C) the fingerprint revealed. It consequently reinforced our idea to move C classes to a new package, named for example render::rendererContext, for better modularization.

4.4 Reading the Fingerprint From Far Away

We introduce two levels of zoom-outs to: (a) keep the visualization compact and scalable over a number of referencing packages or the size of the interface; (b) support global visual patterns as presented in Section 4.6 (p. 85), while minimizing information loss compared to the details presented in Section 4.3 (p. 76).

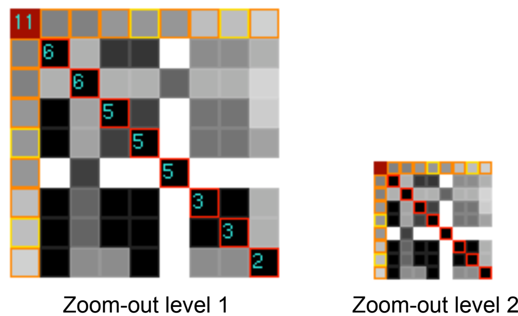


Figure 4.6: The Incoming Fingerprint of renderer package (Figure 4.5 (p. 77)) zoomed-out twice.

Zoom-out level 1. We do not visualize the cell internals. We only visualize in the main diagonal the size of each cell, *i.e.*, the number of referenced classes.

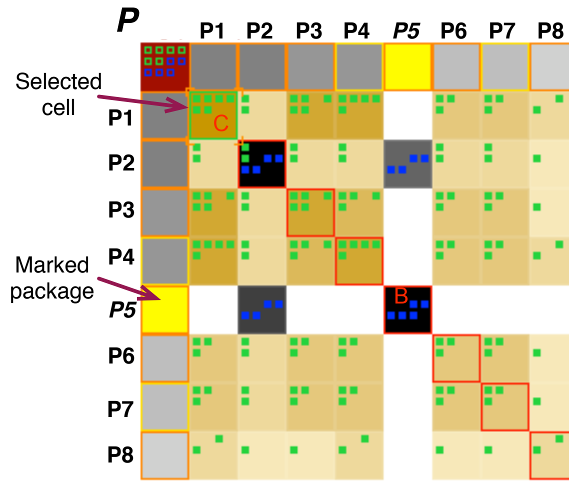
Zoom-out level 2. We visualize the fingerprint without the cell internal information and the size of main diagonal cells.

Figure 4.6 (p. 79) shows the fingerprint of the renderer package, illustrated in Figure 4.5 (p. 77), zoomed-out twice. In the first zoom-out we do not see the information about the classes represented by cells, but we can estimate the size of any cell using its darkness and the size of the main diagonal cell which is located on its row. This last information is hidden in the second zoom-out.

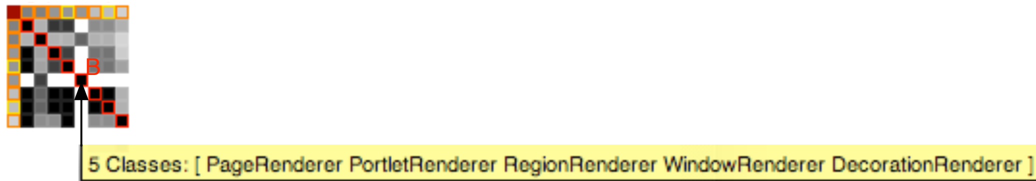
Interacting with the Fingerprint.

To help users detect quickly information within the Fingerprint, we have introduced an interaction mechanism to the visualization, as shown in Figure 4.7 (p. 80).

Figure 4.7(a) (p. 80) shows that the selection of a cell makes its fill color gold and its border color green. In addition it automatically selects all cells that display a subset of classes presented by the first selected cell. This highlights a family of packages based on their co-referencing of the analyzed package classes. The fill's color of the cells which are automatically selected is also gold but with different intensity. The cell which contains the biggest number of classes, is the cell with the darkest fill color. We do the same at the class level: The classes that are contained in the selected cell



(a) Interacting with the Fingerprint of renderer package (Figure 4.5 (p. 77)). P_5 is marked in *yellow* and the cell C is selected (*gold* fill and *green* border). Thus, all the classes of C are highligthed in *green*. In consequence, each cell that represents *only* a subset of those classes is also selected.



(b) Interacting with the zoomed-out Fingerprint of renderer package (Figure 4.5 (p. 77)). The cursor is over the cell B and a fly-by-help shows us B size and the set of the classes it represents.

Figure 4.7: Interacting with the Fingerprint.

get their fill color green. This highlights a family of the analyzed package classes based on their co-usage.

In addition to the selection and marking mechanisms, we have introduced a new interaction with the fingerprint: by moving the cursor over any cell a fly-by-help shows us the size of the cell and the set of the classes it represents (Figure 4.7(b) (p. 80)).

Reading the Fingerprint.

We believe that a package fingerprint, as described in Section 4.3 (p. 76), helps developers understand and analyze a given package, while the fingerprint zoom-outs help visualize large number of packages, easily navigate in the system and detect global information (*e.g.*, patterns, anomalies, etc.). To understand and analyze any package in detail, the developer can select it and zoom to its full fingerprint at any time.

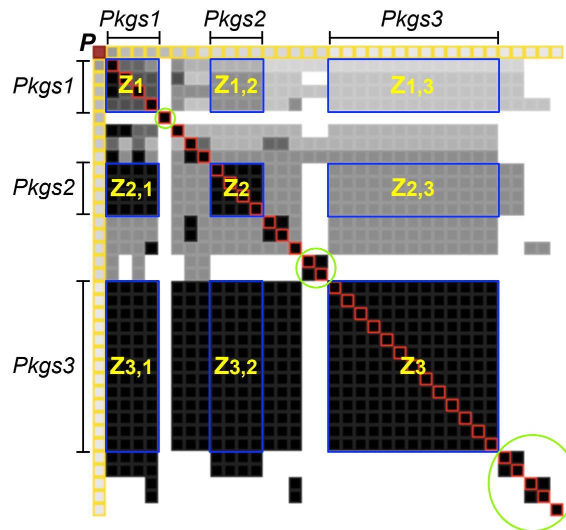


Figure 4.8: The Incoming Fingerprint of utils package, from plugins subsystem (Azureus Application).

Example. Figure 4.8 (p. 81) shows the incoming fingerprint of the package utils of the subsystem plugins, taken from Azureus system. In the following section we illustrate how to read this incoming fingerprint, and which relevant information we can get.

Size. At first glance, the size (*i.e.*, width or height) of the fingerprint is relatively large and all referencing packages are golden bordered. That means the utils package is referenced by a big number of packages that all are located outside the subsystem plugins.

Spread of external incoming references. The top left cell (P) is dark red, which means that most of the package classes are referenced from the outside, *i.e.*, the size of its In-Interface is relatively big.

Distinct part users. The fingerprint fill shows that some cells on the main diagonal (circled in green) are isolated within their row: *i.e.*, the rows are nearly completely white. These cells identify services provided by the analyzed package for only a couple of packages. Classes represented by those cells are considered as lightly coupled in the context of the package, and their presence degrades the package cohesion.

Systematic package external usage. The fingerprint fill shows a black filled rectangle Z_3 at the intersection of the rows and columns of the packages $Pkgs_3$. This indicates that the cells within Z_3 represent the same collection of classes that are referenced together by all packages $Pkgs_3$. In the same way, we can deduce that those classes are also referenced together by the packages $Pkgs_2$ and $Pkgs_1$: see the black filled rectangles $Z_{3,2}$ and $Z_{3,1}$. These set of classes are referenced together from most of the referencing packages: they are highly coupled within the package under analysis. Furthermore, the presence of dark/black rectangles

within the fingerprint body is an indicator of the package cohesion: *the more black space, the more cohesive the package is*.

Strength of use-based cohesive classes. Comparing black filled rectangles according to their size also provides another useful information related to the cohesion based on usage: *the larger a rectangle size is, the higher the coupling between the classes represented by it* –since more client packages used them together. For example, classes represented by the cells within the rectangle Z_2 are less coupled than the classes represented by the cells within the rectangle Z_3.

User heteregeneous references. The fingerprint body darkness is not symmetric. While the classes that are together referenced by both the packages $Pkgs_1$ and $Pkgs_3$ are represented by both the rectangles $Z_{1,3}$ and $Z_{3,1}$, the fills of those rectangles have different darkness: $Z_{1,3}$ is light grey and $Z_{3,1}$ is black. We deduce then that the classes referenced by $Pkgs_3$ form a small portion of the classes referenced by $Pkgs_1$. Thus, the dissymmetrical darkness of the fingerprint body indicates that the package In-Interface contains classes that are loosely coupled in the context of the package under analysis. As consequence, this is an indicator of a bad organization of classes.

4.5 Outgoing Fingerprint

Up to this point we limited our presentation to incoming references; we also propose the symmetrical view to help understanding how the package under analysis uses the rest of the system.

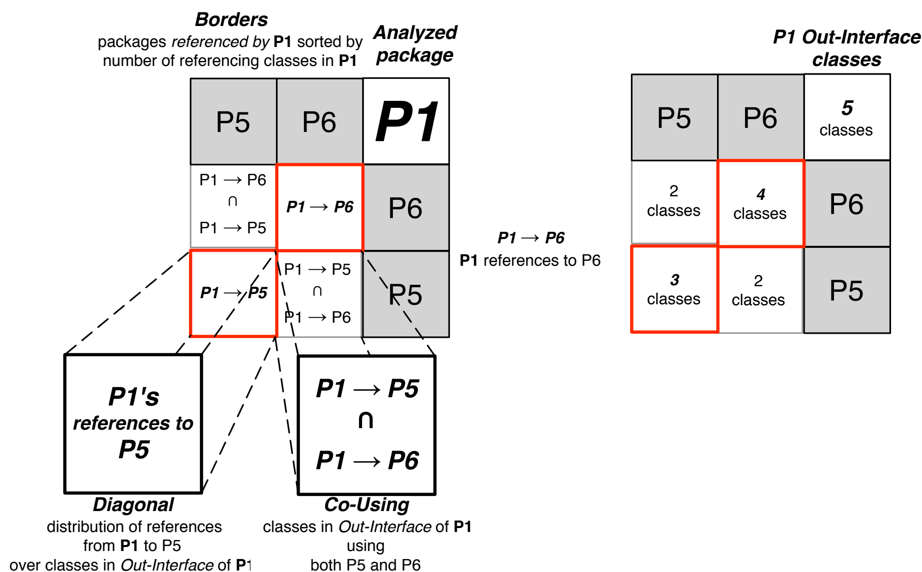


Figure 4.9: P_1 Outgoing Fingerprint skeleton (Figure 4.2(b) (p. 71)).

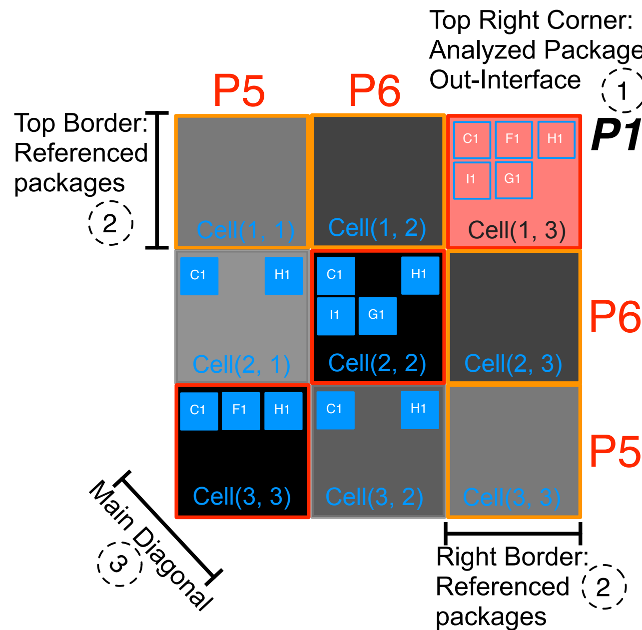


Figure 4.10: Showing the Outgoing Fingerprint of P_1 (Figure 4.2(b) (p. 71)) with the classes involved in the relations inside each cell.

Figure 4.9 (p. 82) and Figure 4.10 (p. 83) depict the key visualization principles of an outgoing Fingerprint with P_1 from Figure 4.1 (p. 71) as the package under analysis. The principles we described above for an incoming Fingerprint (Section 4.2.3 (p. 73) and Section 4.2.4 (p. 74)) are used exactly in the same way, except that we take into account outgoing references instead of incoming ones and referenced packages instead of referencing ones: the referenced packages and the Out-Interface of the package under analysis. In an outgoing fingerprint, the package under analysis is located on the top right most corner, *i.e.*, the *top right corner*, and the diagonal is crossing in the other direction. Also the referenced packages form the *right border* of the package outgoing fingerprint.

Reading Outgoing Fingerprint

Figure 4.11 (p. 84) shows the outgoing fingerprint of impl::api::user package. The fingerprint shows several important pieces of information:

A bad placed class. The package Out-Interface involves only two classes, UserEventBridge and UserEventIntercepter. In the top most corner, the square presenting the class UserEventIntercepter is not filled, which means that this class does not refer to classes inside the package under analysis impl::api::user. On the other hand, this class refers to classes packaged into three packages, the group $Pkgs_2$. We suppose then that it is better to move the class UserEventIntercepter to one of its provider packages. Inspecting UserEventIntercepter, we found that it has not incoming references nor inheritances inside its current package; it inherits from the class ServerInte, which is within the package portal::server. This last is one of

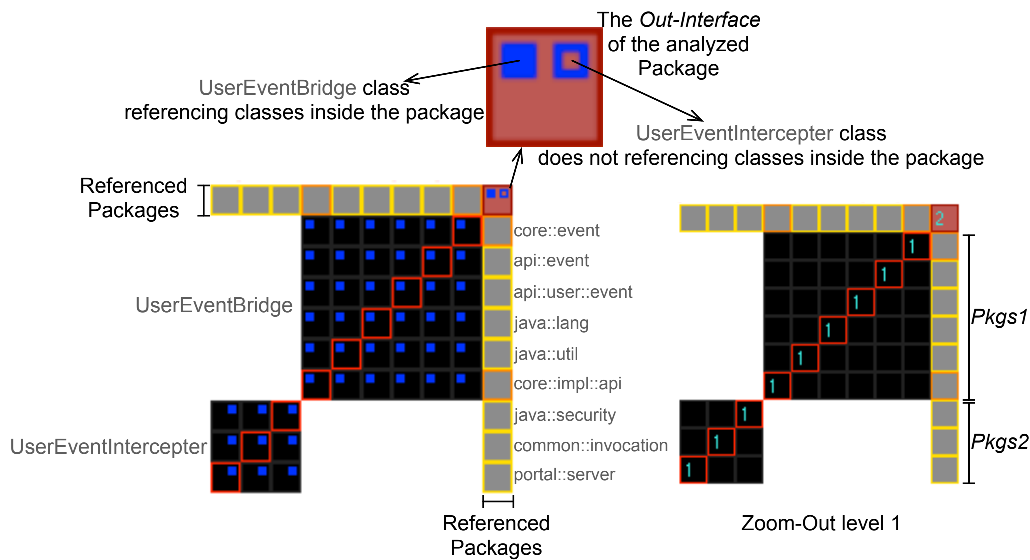


Figure 4.11: The Outgoing Fingerprint of impl::api::user package, from the subsystem Jboss.portal.core.

the provider packages.
That enforced our estimation and we think that moving UserEventIntercepter to the package portal::server will optimize the cohesion of both packages, the analyzed one and portal::server.

Distinct provider packages used by the package. There are two *distinct* groups of packages ($Pkgs_1$ and $Pkgs_2$ on the figure) being referenced by the classes of the analyzed package – since the body cells form around the main diagonal two distinct squares with uniform fill color, each group of the referenced packages is consistently accessed.

Distinct reasons for changing the package. The view also reveals the input source for each class of the package Out-Interface. The view shows that each class refers to *distinct* groups of packages/classes. Changes within the group $Pkgs_1$ directly impacts only the class UserEventBridge, while changes within the group $Pkgs_2$ directly impacts only the class UserEventIntercepter. Here we deduce that the package under analysis has two distinct reasons for changing (Definition 8 (p. 72)).

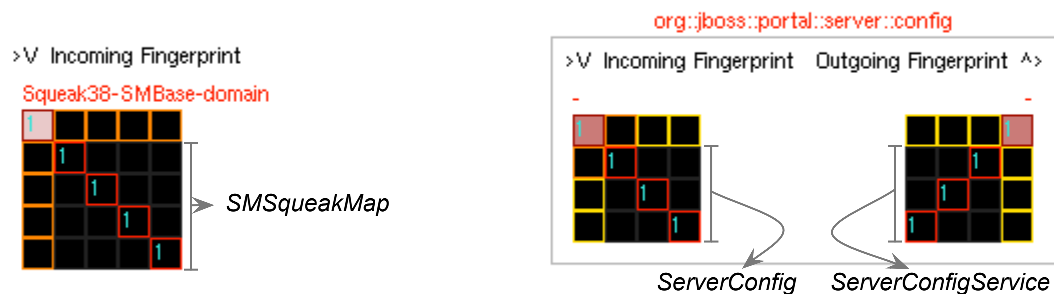
Thus the package outgoing fingerprint helps maintainers coarsely evaluate the package coupling with the rest of the system and the potential impact of changes on the package. Also it focuses on the similarity/coupling between the referencing classes and the cohesion of the considered package, from the provider point of view.

4.6 Relevant Visual Patterns

While applying Fingerprints to large systems (Squeak, Azureus, Jboss, ArgoUML) we identified some recurring visual patterns. We present here most frequent ones, knowing that several patterns could occur within a single fingerprint.

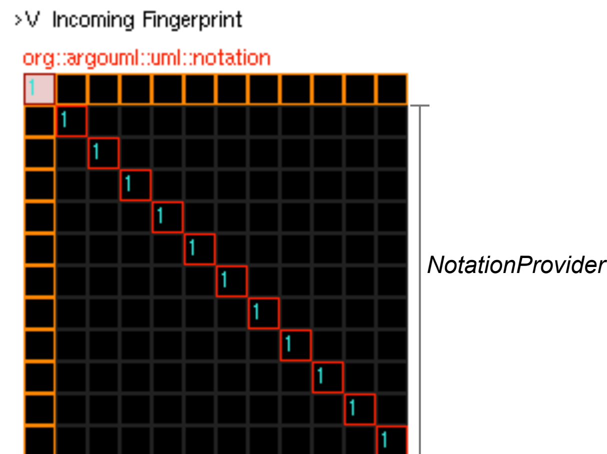
4.6.1 Black Fill Pattern

This pattern is characterized by a complete black fill of the fingerprint as shown in Figures 4.12 and 4.12(c). This pattern occurs when all the package interface classes are conceptually coupled: for an incoming fingerprint, all the In-Interface classes are referenced together by every referencing package, while for an outgoing fingerprint, all the Out-Interface classes refer together to every referenced package.



(a) The Incoming Fingerprint of *domain* package of the Squeak38::SMBase subsystem.

(b) The Incoming and the Outgoing Fingerprints of config package of the jboss::portal::sever subsystem.



(c) The Incoming Fingerprint of notation package of the argouml::uml subsystem.

Figure 4.12: Examples of *Black Fill* Fingerprints.

In our case studies, and in the context of the incoming fingerprint, this pattern occurs for small size In-Interface packages, particularly when they export only one class, or when the package is referenced by a small number of packages. Peripheral

packages often present this pattern.

Referenced as a single service. In this pattern, all the classes of the package In-Interface are referenced always together as a single service. Thus such a package is often characterized by a high degree of cohesion because all its classes tend to fulfill a single service, and the package design respects the package cohesion principles REP and CRP which are described in Chapter 2 (p. 15) (Section 2.3.2 (p. 23)).

Referencing all the same services. For outgoing fingerprints, this pattern occurs also for small size package Out-Interface, or when the package refers to a small number of packages. Exhibiting a black fill pattern reveals that all the classes of the package Out-Interface refer together to the same group of packages. Thus we can conclude that they have a high degree of similarity in terms of required services and responsibility. Also such packages respect the package cohesion principle CCP (Chapter 2 (p. 15) –Section 2.3.2 (p. 23)), –since all the package classes refer to the same group of packages, they have the same source of changes impact.

In consequence, packages that exhibit this pattern for incoming and outgoing fingerprints, may represent a good architecture design since: (1) they respect the three cohesion principles, (2) it is easy to know which services the package provides and to which packages it provides them, and (3) maintainer can see quickly which services/packages the package uses. Note that when several classes are doing consistently several and similar references to external classes, leading to an outgoing black fill, this pattern may reveal a lack of factorization within the package violating the DRY (Don't Repeat Yourself) principle [*Fowler et al.*, 1999].

Examples. Figure 4.12 (p. 85) shows some fingerprints that present this pattern.

A well encapsulated package. Figure 4.12(a) (p. 85) shows the incoming fingerprint of the package SMBase::domain of Squeak38, which defines the domain model of a source management system. It shows that SMBase::domain exports only one class (SMSqueakMap) to only four packages of Squeak38 system. Thus we know that the services provided by this package are exactly the role of SMSqueakMap class and we know that this class provides specific services – since it is referenced by only four packages within the system. Note that SMBase::domain contains 14 classes, but understanding its role requires understanding only one class of those classes. In such a context we say that the package design respects the hidden-information principle.

A provider of abstract service. Figure 4.12(c) (p. 85) shows the incoming fingerprint of notation package of argouml::uml subsystem. It shows that uml::notation exports only one class NotationProvider. By reading this class and its hierarchy we found that it is the interface which is implemented by every UML element notation (*e.g.*, AttributeNotation, MessageNotation, ObjectNotation, *etc.*). notation package includes all those classes (18 classes) but it provides them to the system via their top superclass NotationProvider.

A package with a single reason for changing. Figure 4.12(b) (p. 85) shows the incoming and outgoing fingerprints of config package of the jboss::portal::server subsystem. Both fingerprints present the *Black Fill* pattern. The outgoing fingerprint shows that the package Out-Interface contains only one class: ServerConfigService. By reading this class we found that it implements the interface ServerConfig which is the only class provided by the package: the incoming fingerprint shows that the package In-Interface contain only ServerConfig. Thus we deduce that the package has a single reason for changing, which is the class ServerConfigService. On the other hand, to understand the package role it is enough to understand the interface ServerConfig or its implementation provided by the class ServerConfigService.

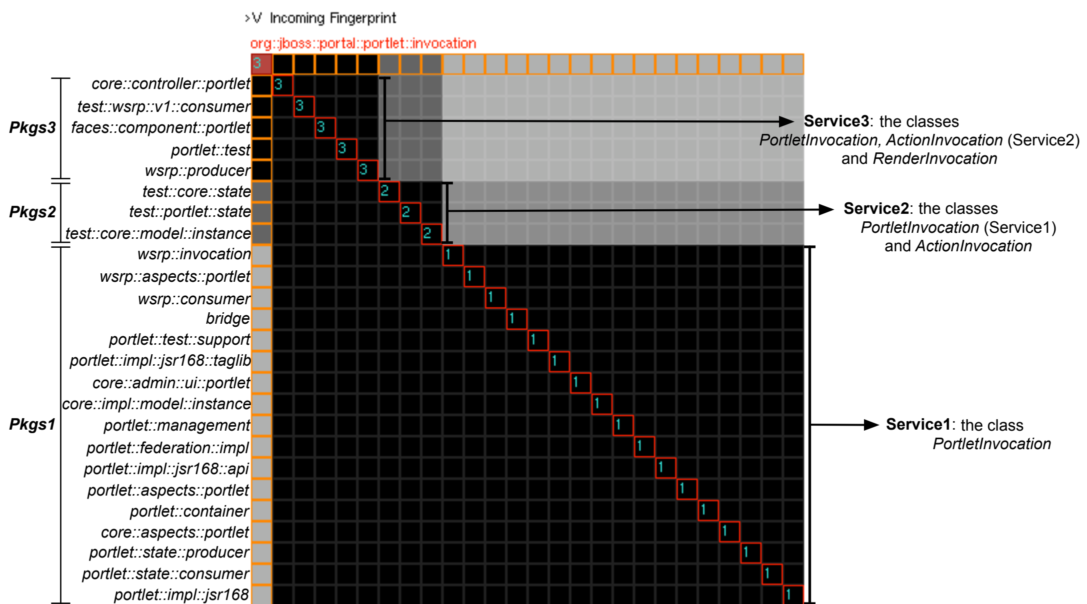


Figure 4.13: An example of the *Black-White* pattern: the Incoming Fingerprint of invocation package, from Jboss system.

Variation. The package invocation, shown in Figure 4.13 (p. 87), illustrates a variation of this pattern: the fingerprint fill appears as gray layers: under the main diagonal the cells are black and above it, they are in progressively lighter shades of gray. We call this variation *Black-White Fill*. The fingerprints that present this pattern are usually larger than those presenting *Black Fill*. Note that the presence of gray layers indicates a degradation of the package cohesion.

Providing a set of layered services. In incoming fingerprints, the *Black-White Fill* pattern indicates that the package In-Interface involves several groups of classes, where each group presents classes that are referenced together, as a single service, by a set of referencing packages.

In this pattern, those services are ordered (layered) from the bottom of the

fingerprint to the top: each service presents a sub-service of the services that are layered above it. Figure 4.13 (p. 87) shows that the most bottom service, $Service_1$ which presents the class PortletInvocation, is a sub service of the services $Service_2$ and $Service_3$: $Service_2$ presents, in addition to the class of $Service_1$, the class ActionInvocation; $Service_3$ presents, in addition to the classes of $Service_2$, the class RenderInvocation.
By relating the service importance to the number of packages that refer to it, in this pattern, the provided services are ordered by importance, from the bottom of the fingerprint to the top. Figure 4.13 (p. 87) shows that the bottom layer presents the class PortletInvocation ($Service_1$). $Service_1$ is referenced by all referencing packages: all cells into that layer are black. The classes PortletInvocation ($Service_1$) and ActionInvocation, which present $Service_2$, are referenced together by the groups of packages $Pkgs_2$ and $Pkgs_3$: within the layer denoted by $Service_2$, the cells which are placed in columns of $Pkgs_2$ and $Pkgs_3$ are black. The classes PortletInvocation, ActionInvocation ($Service_2$) and RenderInvocation, which present $Service_3$, are referenced together by only the group of packages $Pkgs_3$: within the layer denoted by $Service_3$, only cells which are placed in columns of $Pkgs_3$ are black.

Involving a set of layered reasons for changing. For outgoing fingerprints, the *Black-White Fill* pattern indicates that the package Out-Interface involves several group of classes, where each group present classes that refer together to a set of packages. Thus we deduce that each group involves a distinct reason for changing. Those groups are ordered (layered) from the bottom of the fingerprint to the top by reason-for-changing, where the group reason-for-changing is proportional to the number of packages that the group refers to.

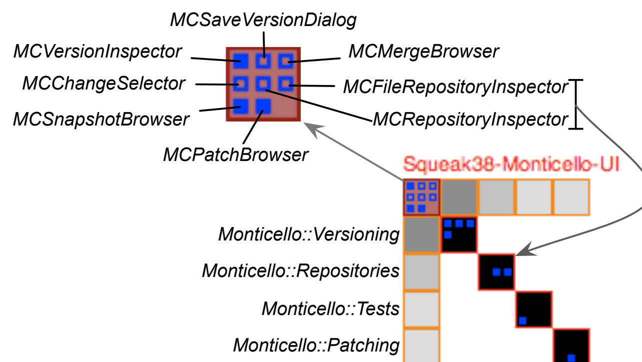
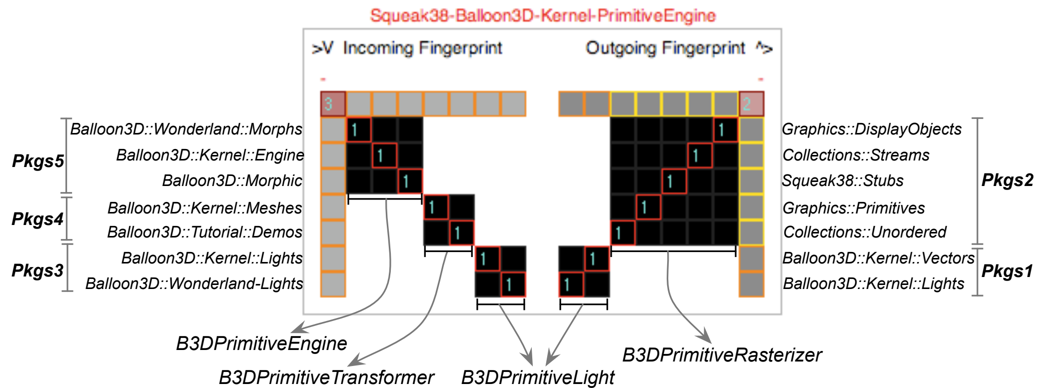
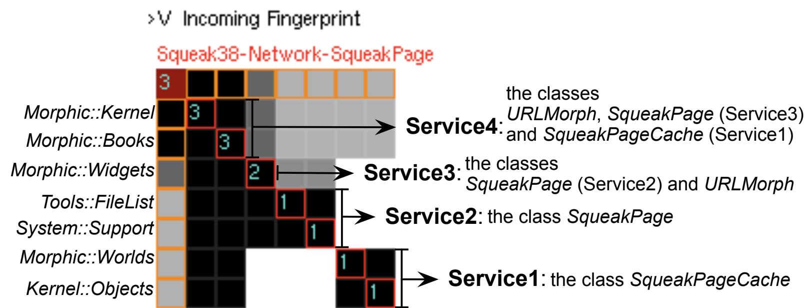


Figure 4.14: Arrow Pattern: the Incoming Fingerprint of UI package of the Squeak38::Monticello subsystem.



(a) The Incoming and Outgoing Fingerprints of Kernel::PrimitiveEngine package of the Squeak38::Balloon3D subsystem.



(b) The Incoming Fingerprint of SqueakPage package of the Squeak38::Network subsystem.

Figure 4.15: Variations of *Arrow* pattern.

4.6.2 Arrow Pattern

When the only non white cells are the diagonal cells, the fingerprint looks like an arrow.

Providing particular non-coupled services. For incoming fingerprints, the strict occurrence of this pattern appears when the package In-Interface involves several groups of classes, where each group presents classes that are referenced together, as a single service, by only one client package. On the other hand, in this pattern, each client package refers to only one service. In other words, the relationship between the provided services and the client packages is one-to-one. We deduce thus that the concerned package provides non-coupled services to the system. Since each service is used by only one client package, we also deduce that the provided services are particular (*i.e.*, are not general or core services from the point of view of the package system): relating the service's importance to the number of packages that use it, in this pattern, we can conclude that all package provided services have minimal importance.
Figure 4.14 (p. 88) shows the incoming fingerprint of UI package of the Squeak38::Monticello

subsystem. It shows that the package services are used separately, each service is used by only one package and all client packages belong to the same subsystem Squeak38::Monticello: the client package Monticello::Versioning uses the top service which presents the classes MCVersionInspector, MCSaveVersionDialog, MCMergeBrowser and MCChangeSelector; the client package Monticello::Repositories uses another service that presents the classes MCFileRepositoryInspector and MCRepositoryInspector; at the end, the client package Monticello::Patching uses the service which presents the class MCPatchBrowser.

Involving particular non-coupled reasons for changing. For outgoing fingerprints, the strict occurrence of this pattern appears when the package Out-Interface involves several groups of classes, where each group represents classes that refer together, as a single reason-for-changing, to only one provider package. Similarly to the case of the incoming fingerprint, each provider package is referenced by only one group of classes and the relationship between the provider packages and package reasons for changing is one-to-one. We deduce thus that the concerned package has several non coupled/mixed reasons for changing. On the other hand, since each reason-for-changing uses services of only one provider package, we deduce that package's reasons for changing are simple (*i.e.*, clear, or not complex).

Package may be a candidate for splitting. Since the occurrence of *Arrow* pattern indicates that the concerned package provides particular services that are used separately or/and it has several non coupled reasons-for-changing, the pattern indicates that such a package could be a candidate for splitting: moving some classes of the package In-Interface/Out-Interface to their referencer/referenced packages may optimize package internal cohesion and reasons for changing.
For example, Figure 4.14 (p. 88) shows that the incoming fingerprint of UI package has the *Arrow* pattern. One of the services that the package provides is the service that presents the classes MCFileRepositoryInspector and MCRepositoryInspector. The most top corner of the fingerprint shows that those classes have not incoming references within their package UI: they are presented by non-filled squares. By inspecting those classes, we found that they do not refer to classes within the UI package. We also found that both classes refer to classes within the package Monticello::Versioning and provide particular functionalities that are used only by Monticello::Repositories. But the package Monticello::Versioning is also a client package of UI package. Thus the classes MCFileRepositoryInspector and MCRepositoryInspector are related to cyclic-references between the analyzed package and Monticello::Versioning package. More, we found that the package Monticello::Repositories is also a provider to the package under analysis UI. Thus the classes MCFileRepositoryInspector and MCRepositoryInspector are related to cyclic-references between the analyzed package and Monticello::Repositories package. We thus deduce that moving the classes MCFileRepositoryInspector and MCRepositoryInspector to the package Monticello::Repositories is a good re-factoring –since that optimizes the internal cohesion of both packages UI and Monticello::Repositories and removes cyclic-references between UI package and the packages Monticello::Repositories and Monticello::Versioning.

Variation. A frequent variation of this pattern is when the fingerprint body appears as small squares, composed of multiple cells, around the fingerprint main diagonal, as in Figure 4.15(a) (p. 89). In this variation, the relationships between the package's services/reasons-for-changing and the package clients/providers is one service/reason-for-changing to *at least* one client/provider. Again, the presence of squares only around the fingerprint diagonal is a good indication that the functionality of the packages is not cohesive from the client/provider point of view. Note that the difference between this variation and the *Arrow* pattern that we described above, is: in this variation, the importance of a package service/reason-for-changing is proportional to the width of the square which represents that service/reason-for-changing.

Non coupled services (reasons-for-changing) with distinct importances. The incoming and the outgoing fingerprints of Kernel::PrimitiveEngine package, which are shown in Figure 4.15(a) (p. 89), both have the described pattern variation. The incoming fingerprint shows that Kernel::PrimitiveEngine package provides three classes. Those classes (B3DPrimitiveEngine, B3DPrimitiveTransformer and B3DPrimitiveLight) are used separately: the class B3DPrimitiveEngine is used by three client packages, denoted by $Pkgs_5$; the class B3DPrimitiveTransformer is used by two client packages, denoted by Pkg_4; the class B3DPrimitiveLight is used by two client packages, denoted by Pkg_3. We thus deduce that this package provides three non-coupled services and the class B3DPrimitiveEngine presents most important provided service –since it is used by three client packages rather than only two client package, as the classes B3DPrimitiveTransformer and B3DPrimitiveLight.

The package outgoing fingerprint shows that the package Out-Interface includes 2 non-coupled classes: the class B3DPrimitiveRasterizer refers to 5 packages (denoted by $Pkgs_2$), while the class B3DPrimitiveLight refers to 2 packages (denoted by $Pkgs_1$). We deduce thus that the reason-for-changing represented by the class B3DPrimitiveRasterizer is more important/complex than the reason-for-changing represented by the class B3DPrimitiveLight.

Variation. A variation of the *Arrow* pattern occurs for incoming fingerprints when some of the provided services, if not all, are used together by a few number of referencing packages. In this case, we say that the package services are loosely coupled and some of the referencing packages appear as dominant over the other referencing packages.

Providing loosely-coupled services. Figure 4.15(b) (p. 89) shows the incoming fingerprint of Network::SqueakPage package of the Squeak38 system. The incoming fingerprint has the described pattern variation. It shows that the packages Morphic::Kernel and Morphic::Books are dominant referencing packages. They refer to all the classes provided by Network::SqueakPage package (3 classes: SqueakPageCache, SqueakPage and URLMorph). The rest of referencing packages refer to distinct groups of those classes: The referencing packages Kernel::Objects and Morphic::Worlds refer to the class SqueakPageCache (*Service*$_1$); the referencing packages System::Support and Tools::FileList refer to the class SqueakPage (*Service*$_2$); the referencing package Morphic::Widgets refers, in addition to (*Service*$_2$), to the class

URLMorph. Thus the classes of the package In-Interface are used together by only two packages, while the package has seven referencing packages. We then deduce that the provided services are loosely coupled in the context of Network::SqueakPage package.

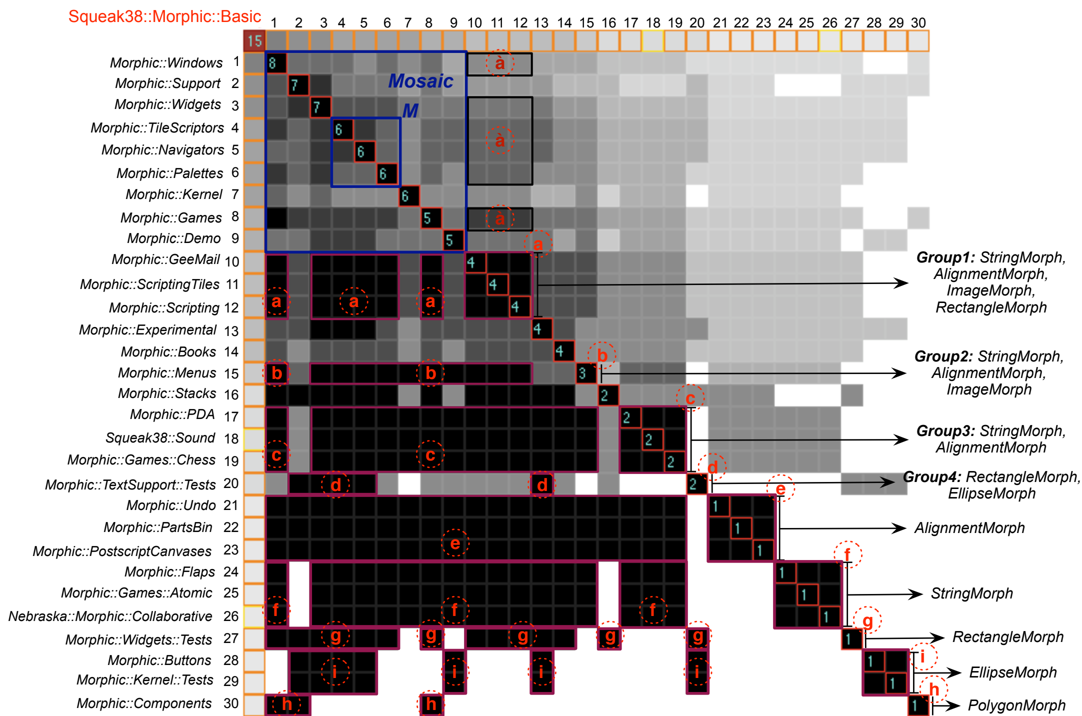


Figure 4.16: An example of the *Mosaic* pattern: the Incoming Fingerprint of Morphic::Basic package, from Squeak38 system.

4.6.3 Mosaic Pattern

In this pattern almost, if not all, the cells of the fingerprint fill are gray but they have not an homogenous darkness. Examples of this pattern are the incoming fingerprints of Basic package in Squeak38::Morphic subsystem (Figure 4.16 (p. 92)) and model package in Argouml system (Figure 4.17 (p. 93)).

Large package interface size. The *Mosaic* pattern occurs usually for packages whose interfaces (In-Interface and Out-Interface) contain a large number of classes. The incoming fingerprint of model package (Figure 4.17 (p. 93)) shows that the package In-Interface contains a large number of classes: 51 classes. This appears as a large In-Interface size whatever the size of the concerned package (*i.e.*, the number of classes that the package contains). In addition to that, only 14 classes among the In-Interface classes have incoming references inside the model

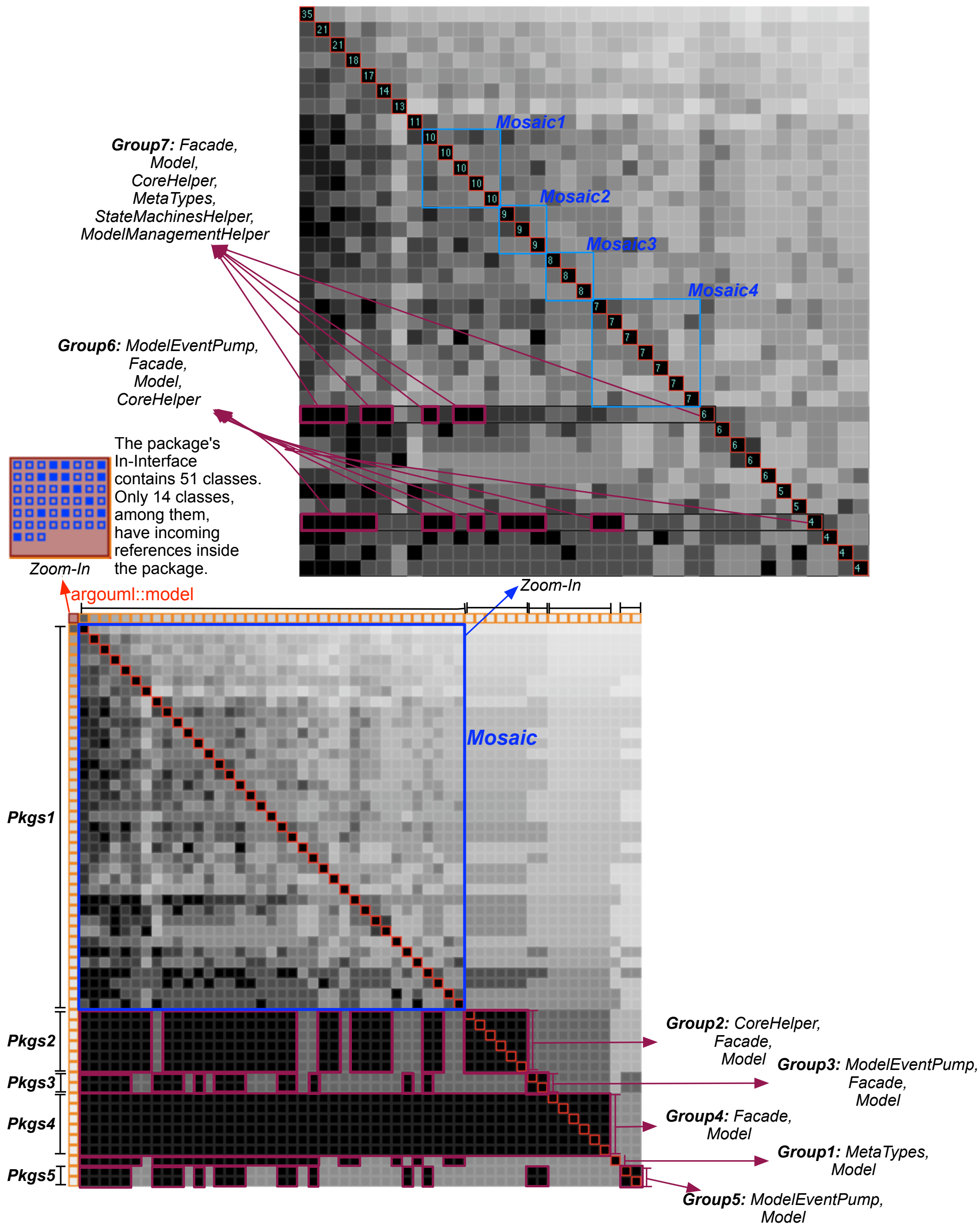


Figure 4.17: An example of the *Mosaic* pattern: the Incoming Fingerprint of model package, from Argouml system.

package. By inspecting model package, we found that it contains 112 classes. Thus its In-Interface contains about 45% of its classes.

Core and Central package. This patten occurs usually with giant fingerprints: the

package under analysis is coupled to a large number of other ones and its interface has a big size. In the case of incoming fingerprints, this means that the package provides a lot of services/functionalities that are used by an important number of packages within its system.

Examples. The Basic package whose incoming fingerprint (Figure 4.16 (p. 92)) has the *Mosaic* pattern is also a core package within its system Morphic. Basic package provides 15 classes to 30 packages. Only two packages of the referencing packages are stubs, *i.e.*, they do not belong to the Morphic system. Those stubs are Squeak38::Sound package (denoted by 18) and Nebraska::Morphic::Collaborative package (denoted by 26). Thus, Basic package provides 15 classes to 28 packages of the 45 Morphic packages. This means that more than 62% of the Morphic packages depend upon Basic package and this last is a core and a central package within *Morphic*.

Another example, the model package whose incoming fingerprint (Figure 4.17 (p. 93)) has the *Mosaic* pattern is a core package within its system Argouml. In addition to the fact that it contains 112 classes of the 1671 Argouml classes, it is also referenced by 54 packages of the 76 Argouml packages. Also the package In-Interface contains 51 classes, which means 71% of Argouml packages depend directly on 51 classes within the model package. This means that the whole Argouml system highly depends on the model package and this last plays the role of core and central package within Argouml.

Imprecision and difficulty in determining package usage and role. The occurrence of *Mosaic* pattern for incoming fingerprints indicates that the package under analysis provides a large number of functionalities that are accessed by a large number of packages in an arbitrary way, *i.e.*, non-consistent way. Thus in presence of this pattern, it is hard to know which functionalities are used together and which are not. As a result, it is hard to identify the role/functionality and to determine the contextual cohesion of the considered package.

Imprecise package contextual cohesion. Since it is hard to determine the package usage and role, it is also hard to have a precise contextual cohesion of such a package.

For example, Figure 4.17 (p. 93) shows that the model package suffers from the same problem that we discussed for the package Basic (Figure 4.16 (p. 92)): for the 51 In-Interface classes, the fingerprint clearly shows several groups of classes ($Group_1$..$Group_7$) — thanks to black zones within the fingerprint fill; those groups are composed of only 7 classes: Model, Facade, CoreHelper, ModelEventPump, MetaTypes, StateMachinesHelper and ModelManagementHelper. The classes Model and Facade, denoted by $Group_4$, are referenced together by the group of packages $Pkgs_4$ and also by all referencing packages except three (the three packages at the right of the fingerprint top border): $Pkgs_4$ row contains only black cells, except the last three cells in each row. Thus, the classes Model and Facade are highly coupled from a use point of view and are most important classes within model package.
Whatever, it still hard to evaluate the coupling of the remainder 44 classes –since that those classes are clearly co-used in a non-consistent way.

4.6.3.1 Analyzing Mosaic Patterns

Let us take a deep look at an example to see how a fingerprint reveals information even in presence of messy and large packages clearly identifiable as Mosaic Pattern.

Looking at the incoming fingerprint of the Basic package (Figure 4.16 (p. 92)) we see that there are several black zones within the fingerprint fill. Such zones, which are surrounded by red rectangles, indicate that some classes within the package In-Interface are referenced together by referencing packages.

Classes referenced together but with distinct groups of classes. Thanks to the black zone, denoted by a, around the main diagonal, we find that the referencing packages Morphic::GeeMail (10), Morphic::ScriptingTiles (11), Morphic::Scripting (12) refer to the same group of classes. This group represents 4 classes (as indicated by the value on the diagonal). Those 4 classes, denoted by $Group_1$, are StringMorph, AlignmentMorph, ImageMorph and RectangleMorph.
Other black zones in the rows of the referencing packages 10, 11 and 12 (rectangles denoted by a) indicate that those 4 classes ($Group_1$) are also referenced by the referencing packages Morphic::Games (8), Morphic::Palettes (6), Morphic::Navigators (5), Morphic::TileScriptors (4), Morphic::Widgets (3) and Morphic::Windows (1).

On another hand, rectangles denoted by $\grave{a}$, which are symmetrical to rectangles a relatively to the main diagonal, indicate that these last referencing packages (1, 3..6 and 8) refer to more classes than the 4 cited classes –since the fill color of cells involved within rectangles $\grave{a}$ is not as black as within rectangles a. If we look to the main diagonal cells: the referencing package Morphic::Windows (1) refers to 8 classes within the package under analysis, thus it refers to 4 classes in addition to $Group_1$ classes; the referencing packages 4..6 all refer to 6 classes within the package under-analysis, thus they refer to 2 classes in addition to $Group_1$ classes; etc.

More, thanks to black cells denoted by b, we see that the referencing package Morphic::Menus (15) refers to three classes among the $Group_1$ classes: StringMorph, AlignmentMorph and ImageMorph (denoted by $Group_2$). Also, thanks to black cells denoted by c, we see that the referencing packages Morphic::PDA (17), Squeak38::Sound (18) and Morphic::Games::Chess (19), all refer to two classes among the $Group_2$ classes: StringMorph and AlignmentMorph (denoted by $Group_3$).

Omni-referenced classses in Basic Package. Counting the number of columns involved into the rectangles c indicates that $Group_3$ classes are referenced together by 17 referencing packages: 1, 3..15 and 17..19, which means that it is good to keep $Group_3$ classes together in the same package.

Rectangles denoted by e indicates that the class AlignmentMorph is referenced by almost all referencing packages (packages 1..19 and 21..23) showing that this class is one of most important classes within the Basic package. Similarly we deduce the same thing for the class StringMorph, thanks to rectangles denoted by f.

We also found that the group $Group_4$, which represents the classes RectangleMorph and EllipseMorph, is referenced by 6 packages: 2..5, 13 and 20. On another hand,

the class RectangleMorph is referenced by other packages that do not refer to EllipseMorph: the referencing package 27 refers only to RectangleMorph (see the diagonal cell which is denoted by g). The same thing for EllipseMorph: the referencing packages 28 and 29 refer only to EllipseMorph (see the diagonal cells which are denoted by i). The class PolygonMorph is referenced by only 4 referencing packages ($1, 2, 8$ and 30).

Conclusion on Basics Package. From all that, we identify that the Basic package provides several classes to a large number of referencing packages and the usage coupling of those classes is not consistent. Thanks to some black zones, we spot that classes are referenced together by client packages. As conclusion, the usage coupling of the 44 In-Interface classes is done in an arbitrary way and there is no relevant information about class usage coupling.

About model package. While model package (Figure 4.17 (p. 93)) suffers from the same problem as the Basic package (Figure 4.16 (p. 92)), understanding and maintaining the model package is really harder than understanding and maintaining the Basic package: Basic In-Interface is smaller than model In-Interface, Basic is referenced by a smaller number of packages and a relatively large portion of its classes are clearly coupled — since they are used together in a consistent way.

Not surprisingly, the larger its interfaces are, the more difficult it is to understand a package. It is also clear that grouping all the basic functionalities of a system into one package (*e.g.*, model package: Figure 4.17 (p. 93)) is not a good design. It is always better to compose packages from a small number of related functionalities. This pattern indicates a bad organization or design of classes within the system: either the classes of the concerned package should be re-organized (*i.e.*, distributed over different packages); or new abstractions should be implemented, where classes that have similar behavior inherit from the same abstraction and the system classes refer to those abstractions instead of their implementations, as we have seen in Figure 4.12(c) (p. 85) (Section 4.6.1 (p. 85)).

Fingerprint vs. Metrics. Note that common reuse based cohesion metrics [*Ponisio* and *Nierstrasz*, 2006], indicate usually that such packages are cohesive. For example the value of CU [5] metric [*Ponisio* and *Nierstrasz*, 2006] is 0.63 for the Basic package and 0.7 for the model package: this means that the design of the latter is better than the design of the former and both packages are considered as enough cohesive, which is clearly not what the fingerprints revealed. We learned that the Fingerprint is much more than metrics: the Fingerprint shows which classes are coupled in a consistent way and which are not, what is the portion of those classes relatively to the package interface size, *etc.*

4.6.4 Diverse Patterns

We present some other less frequent but still interesting patterns with less details.

[5] CU (Common-Use) metric computes package cohesion from the reuse of the classes of the package in-interface. It takes its value between 0, the worst value of cohesion, and 1, the best value of cohesion. For more information see [*Ponisio* and *Nierstrasz*, 2006].

4.6.4.1 Unbalanced Pattern

This pattern occurs when an incoming or outgoing fingerprint appears clearly bigger than its counterpart (*i.e.*, its outgoing or incoming fingerprint). The Unbalanced-Incoming Fingerprint pattern indicates that the analyzed package plays a server role within the system, rather than a client role. The Unbalanced-Outgoing Fingerprint pattern indicates the reverse case. Two variants of this general patterns have special interest:

Giant Incoming Fingerprint. This variant reveals core/central and utility packages that provide basic services for the system. Figures 4.8, 4.17 and 4.16 show respectively that plugins::utils, argouml::model and Morpic::Basic exhibit this pattern.

Empty-Outgoing Fingerprint. The outgoing fingerprint is empty, *i.e.*, the package under analysis does not refer to any package in the system. This occurs for packages that include only abstract classes or/and interfaces. Such packages are not impacted by the system.

Empty-Incoming Fingerprint. The incoming fingerprint is empty and the package has no incoming references. It is the case of packages that include abstract classes that are implemented in other packages. This pattern appears for packages that are leaves in the package structure. This is often the case for UI application packages.

4.6.4.2 Golden Border Pattern

This patterns occurs when all the referencing packages are stubs (*i.e.*, are not part of the system under analysis). Thus, this pattern only occurs when the clients of the package under consideration do not belong to the analyzed subsystem (*e.g.*, Plugins package within Azureus Figure 4.8 (p. 81)). Such packages represent the border of the analyzed subsystem. This pattern is usually a good sign because it indicates that the system under analysis tends to be well layered.

Ideally, a subsystem should be composed of three distinct layers of packages: the first layer presents packages that refer only to packages outside the subsystem –thus they have *Golden Border* Outgoing Fingerprints– and are not referenced by packages outside the subsystem; the second layer presents packages that interact only with packages inside the subsystem; the third and last layer presents packages that refer only to packages inside the subsystem and are referenced by only packages outside the subsystem –thus they have *Golden Border* Incoming Fingerprints.
Whatever, analyzing and understanding subsystem architecture/layers need a global view of the analyzed subsystem. DSM views [*Sangal et al.*, 2005] are more suitable for such analysis.

On the other hand, if a package has *Golden Border* Outgoing and Incoming Fingerprints, this means that the concerned package is bad placed within the analyzed subsystem –since it has no incoming or outgoing references with the subsystem packages and it interacts only with packages outside the subsystem.

4.7 Discussion and Evaluation

4.7.1 Graphical concerns

Fingerprints are not magic; they show, albeit in a condensed form, the existing situation of the code. When packages are not well-designed the patterns are less apparent, still the visualization conveys the situation and the information about the use of the package by its clients or how the package uses the system. Our approach has worked well on our case studies and we have been able to locate many conceptual bugs and to spot several visual patterns. It should be noted that we were *not* familiar with the case studies before applying our approach.

Now we discuss some design points:

Position Choices. A reader often pays more attention to the top elements than to the bottom ones. Therefore, we grouped the internal references at the top corner of the package fingerprint, then ordered the related packages (*i.e.*, referencing packages in an incoming fingerprint; referenced packages in an outgoing fingerprint) from most related one (*i.e.*, most referencing package in an incoming fingerprint; most referenced package in an outgoing fingerprint) at the top to the least at the bottom.

Seriation. We ordered referencing (referenced) packages that make the same number of references by similarity based on common referenced (referencing) classes into the package under analysis: the largest number of common referenced (referencing) classes that two client (provider) packages have, the biggest similarity the two packages have; this way, the reader can see which packages access, or are accessed by, the same groups of classes. During the design of the fingerprint, we tried ordering packages differently, *e.g.*, by similarity regardless of how many references they make, but each time we lost important information *i.e.*, the position of most (least) referencing packages.

Impact of Boundaries. We colored the border of packages that do not belong to the system under analysis in gold. We found it really effective to use color to identify the currently selected entities so that the user can interactively mark entities on which s/he wants to focus; this increases the usability of the tool.

Zooming. We introduced two levels of zoom-outs with minimal information loss, so that the visualization remains compact and scalable over the number of the related packages or the size of the interfaces. This way, the user can visualize large systems, navigate in the system, spot global patterns and conceptual anomalies. Then s/he can focus on any package by zooming into the detailed fingerprint.

However, during our experiments, we found that detailed fingerprints do not scale as well as the zoomed-out views. Detailed fingerprints expose a lot of information, which makes it difficult to spot patterns or gather general information about the visualized package; this is especially true for giant packages whose interface and number of related packages are very large. In fact, in such cases, none of the detailed views we applied has scaled well. We think that

zooming mechanisms are very important in software visualization to solve this problem.

Placeholders. The placeholders in cell internals are essential to make preattentive processing work and thus to help users quickly spot which classes are coupled and where they are coupled. The negative impact of this principle is that all cells should be large enough to represent all possible classes in the package interface. This is one of the reasons why the detailed fingerprints do not scale so well.

4.7.2 About Coupling and Hints at Improvements

The presence of dark homogenous zones is a good indicator of the package cohesion. Note that common reuse based cohesion metrics [*Ponisio* and *Nierstrasz*, 2006], indicate usually that such packages are cohesive. Package Fingerprints are much more than metrics that give simple values. We illustrated in Section 4.6.3.1 (p. 95) that the Common-Use (CU) metric [*Ponisio* and *Nierstrasz*, 2006] indicates that the packages model (Figure 4.17 (p. 93)) and Basic (Figure 4.16 (p. 92)) are both considered as enough cohesive and the design of the former is better than the design of the latter, which is clearly not what the fingerprints revealed.

Fingerprints Ability. Incoming Fingerprint helps maintainer answer the following questions about a given package:

- Which In-Interface classes are used together, in a consistent way, as a single service? And which are not used together, also in a consistent way, as distinct services?
- Where is a group of In-Interface classes used as a single service? and where is it mixed with distinct classes of the concerned In-Interface?
- How many classes of In-Interface classes are consistently used together (*i.e.*, coupled)? And how many are not?
- Which referencing packages refer to a given group of In-Interface classes? And which ones do not refer to that group?
- Which In-Interface classes are highly coupled (*i.e.*, used together by a large number of referencing packages)? And which ones are loosely coupled (*i.e.*, used together by a small number of referencing packages)?
- Which In-Interface classes are considered as most important (*i.e.*, classes that are referenced by most referencing packages)? And which In-Interface classes are less important?

Outgoing Fingerprint works similarly to Incoming Fingerprint but from the point of view of package outgoing references (Out-Interface), referenced packages and package reasons-for-changing, instead of the the point of view of package incoming references (In-Interface), referencing packages and package provided services.

Fingerprints Limitations. Package fingerprints focus on the package contextual cohesion, afferent and efferent coupling, and co-use of internal classes. However, they do not provide a good map for internal references; our aim is to support understanding packages through their interfaces, regardless what happens inside them. With package fingerprint, we consider related packages (*e.g.*, referencing packages in an incoming fingerprint) as black boxes; we only pay attention to package classes while we look at its fingerprint.

Since package fingerprints do not show complete information about package internal references, hints at improvements, which are revealed from fingerprints, should be assessed together with other information/views of package internal references and of the inheritance relationship between classes.

For example, we illustrated in Section 4.6.2 (p. 89) that an *Arrow* pattern indicates that the concerned package may be a candidate for splitting — since it provides distinct non-coupled services. For such a case, before deciding to split the concerned package, maintainer needs to know if classes that are not contextually coupled interact with each other. In other words, s/he needs to verify if there are references or inheritance relationships among classes that are used by distinct packages, before deciding to split the package or to move some classes of that package to other ones.

We consider package fingerprints as complementary views to the Package Blueprint resulting from Chapter 3 (p. 39) work, where the Package Blueprint provide a good map for internal dependencies and shows dependencies between packages on a per-class basis, but they give limited information about the coupling and co-use of classes.

Package Fingerprints are a dense and compact visualization, they were designed to have such property. Still users may have difficulty extracting all the information from them. Our current work lacks a serious user study. We performed some limited studies with members and students of our team not working on Fingerprint. Our preliminary results show that a first level of understanding is easy to get: identifying groups of co-referencing/co-referenced classes; identifying distinct provided services and distinct reasons-for-changing; identifying referencing and referenced packages; etc. Those users found that, the direction of the diagonal as well as the small annotations we put on top of the fingerprint to distinguish incomoing/outgoing fingerprint are very helpful. Fingerprint supports also fly-by-help. The fly-by-help use suggests that showing the names of the packages on the side may really help creating a deeper context. In addition, the fly-by-help showing the referenced/referencing classes make visualizations less abstract.
We learned that a deeper level of information extraction, with packages that have very large interfaces and their classes are coupled in a non-consistent way (*e.g.*, the Mosaic pattern, Section 4.6.3.1), is much more difficult to grasp. This suggests that Fingerprint is good for fast overview, but further usability enhancements and studies are required.

4.7.3 Related Work

Several works focus on understanding packages. We are interested here on those based on visualizations. Sangal *et al.* adapted the dependency structure matrix (DSM) from the domain of process management to analyze architectural dependencies in software [*Sangal et al.*, 2005]. DSM presents a consistent visualization that offers a system overview. While the visualization scales for large systems, it is poor in terms of precise information about the package. DSM cells contain a number indicating the number of references made between packages. However DSM did not focus on packages cohesion and co-use or co-usage of classes.

Package fingerprints are based on similar principles, but provide more visual information and help identify groups of packages with similar dependencies. A fingerprint exploits pre-attentive processing using color, contrast, and the principle of placeholders. In addition, a fingerprint by focusing on a package at a time qualifies in a finer-grained way the dependencies.

X. Dong *et al.* [*Dong* and *Godfrey*, 2007] present the High-level Object Dependency Graph (HODG) that helps capturing, from a high-level point of view, possible usage dependencies among coarse-grained software entities, namely packages. In their approach, they interpret the usage dependencies between classes in the context of their hierarchy and present a new graph of the system under analysis. While the given graph is helpful for understanding the considered system from a high-level point of view, it does not give any information about package cohesion nor about the co-use or the similarity between classes. Also, their graph visualization still difficult to be interpreted by human eyes because within it, the nodes have different sizes but without any meaningful dimension. The HODG has not visual semantics and it uses numbers to visualize almost all information.

Several works explore packages and their structure but few of them reveal information on their relationships and dependencies. In Softwarenaut, Lungu *et al.* help system discovery by guiding exploration of nested packages [*Lungu et al.*, 2006]. Storey *et al.* also worked on system exploration, supporting zoom-out facilities and forces-based graph layouts [*Storey et al.*, 1997]. However the work did not focus on co-use or co-usage of classes.

There is a plethora of software metrics on cohesion: from the bogus LCOM ([*Chidamber* and *Kemerer*, 1994]) to more advanced LCOM* metrics [*Briand et al.*, 1998]. Ponisio *et al.* introduced the notion of use cohesion [*Ponisio* and *Nierstrasz*, 2006], which is at the foundation of the fingerprint. Hautus defines a new metric that indicates the percentage of changes to be made in order to make a package structure acyclic [*Hautus*, 2002]. While he focuses only on the cyclic dependencies, he does not provide any utility that helps understanding packages or indicating their cohesion or similarity.

Kuhn *et al.* used information retrieval to exploit linguistic information. He introduced semantic clustering to group source artifacts that use similar vocabulary [*Kuhn et al.*, 2007]. He uses vocabulary topics to reveal the intention of the code and the similarity between its artifacts, then he provides a consistent visualization.

A number of approaches give summarized information on package relationships and their evolution: the Butterfly by Ducasse *et al.* gives a high-level client/provider trend of package dependencies [*Ducasse et al.*, 2005b]; Pzinger *et al.* show the evolution

of package metrics using Kiviat diagrams [*Pinzger et al.*, 2005]; Chuah and Eick use rich glyphs to characterize software artifacts and their evolution (number of bugs, number of deleted lines, kind of language...) [*Chuah* and *Eick*, 1998]. In particular, the timewheel exploits preattentive processing, and the infobug presents many different data sources in a compact way; finally, D'Ambros *et al.* reveal package coupling by showing evolutions that are correlated in time [*D'Ambros* and *Lanza*, 2006b].

Other works treat and visualize information about software co-change evolution, looking at coupling from a temporal perspective, and software development teams and activities [*Beyer*, 2005; *Eick et al.*, 2002; *Froehlich* and *Dourish*, 2004; *Storey et al.*, 2005; *Voinea et al.*, 2005; *Xie et al.*, 2006]. Such approaches are completementary to ours in the sense that we only focus on the static nature of the packages and their relationships. While those approaches are valuable and provide fine-grained views of packages that may help understanding the contextual coupling and cohesion inside packages, they fall short on the analysis of a single version of a system.

4.8 Conclusion

In this chapter, we tackled the problem of understanding the details of package relationships from a usage perspective. We described the package fingerprints, and their use as a visual approach for understanding package relationships, contextual cohesion, and the conceptual coupling of their classes. While designing the Package Fingerprint, we exploited pre-attentive processing using color properties and placeholders saving principle. We also introduced interactivity and multi-selection mechanism to help the user during the analysis task.

We successfully applied the visualization to several large systems and we have been able to quickly point out badly designed packages, and to extract relevant patterns.

While applying Fingerprints to large systems that contain radically different packages in terms of internal size and package references, the visualization generally scaled well and the detection of the different patterns presented in this chapter was always possible.

Chapter 5

Automatically Measuring and Optimizing Modularization Quality

5.1 Introduction

A well modularized system enables its evolution by supporting the replacement of its parts without impacting the complete system. A good organization of classes into identifiable and collaborating subsystems eases the understanding, maintenance, test and evolution of software systems [*DeRemer* and *Kron*, 1976].

However code decays: as software evolves over time with the modification, addition and removal of new classes and dependencies, the modularization gradually drifts and looses quality [*Eick et al.*, 2001]. A consequence is that some classes may not be placed in suitable packages [*Griswold* and *Notkin*, 1993]. To improve the quality of software modularization, optimizing the package structure and connectivity is required.

Although that Package Blueprint and Package Fingerprints help understanding packages, they do not propose alternative modularizations. In addition, similarly to every visualization-based approach, it is always the responsibility of maintainers to take decisions about changes and to assess changes impacts on the modularization quality.

On the other hand, as we explained in Section 2.7 (p. 34), few previous works address the problem of automatically optimizing existing software modularization, and those works often change (to various degrees) the existing package structure of a software system. In such a case, it can be difficult for a software engineer to understand the resulting structure and to map it back to the situation that maintainer knows.

The aim of this chapter is, to support automatic optimization of existing package structure by explicitly taking into account the whole modularization quality and the original class organization; and avoiding creating new packages or related abstractions.

Contribution of the chapter

In this chapter, we present an approach for automatically optimizing existing software modularizations by reducing connectivity among packages, in particular cyclic-connectivity. The objective of the optimization process is inspired by well known package *cohesion* and *coupling* principles already discussed in Section 2.3.2 (p. 23). We limit ourselves to *direct* cyclic connectivity and restrict our optimization actions to moving classes over existing packages.

Our approach is based on *Simulated Annealing* [*Ferland* and *Costa*, 2001; *Kirkpatrick et al.*, 1983], which is a *neighborhood (local) search-based* technique. Simulated Annealing is inspired by the annealing process in metallurgy [*Kirkpatrick et al.*, 1983]. We chose this technique because, it suits well our problem, *i.e.*, *local optimization of an existing solution*. Moreover, it has been shown to perform well in the context of automated OO class design improvement [*O'Keeffe* and *Cinnéide*, 2006, 2008] and more generally, in the context of software clustering problems [*Mitchell* and *Mancoridis*, 2002, 2008].

In that respect, this chapter proposes two main contributions, that are the object of our paper published in *WCRE'09* [*Abdeen et al.*, 2009b]:

- Firstly, we define a suite of metrics, based on the principles of package cohesion and coupling [*Martin*, 2002a] (Section 2.3.2 (p. 23)): Common Closure Principle (CCP), Common Reuse Principle (CRP) and Acyclic Dependencies Principle (ADP). The aim of those metrics is to help in automatically assessing the quality of a modularization, as well as, the quality of a package within a given modularization.

- On the other hand, we present an approach, using simulated annealing technique, for the automatic reduction of package coupling and cycles by only moving classes over packages while taking into account the existing class organization and package structure. In our approach, maintainers can define (1) the maximal number of classes that can change their package, (2) the maximal number of classes that a package can contain, and (3) the classes that should not change their packages or/and the packages that should not be changed.

Structure of the chapter

In the next section (5.2) we define a metric suite to assess the quality of a modularization or a package within a given modularization. This approach follows the principles of package coupling and cohesion which we underlined in Section 2.3.2 (p. 23). After that, we present and detail our optimization algorithm in Section 5.3 (p. 107), and we define evaluation functions that our algorithm uses to automatically evaluate both modularization and package quality. We validate our approach using real large software systems and discuss the results in Section 5.4 (p. 112). In Section 5.5 (p. 119) we position our approach with related works, before concluding in Section 5.6 (p. 121).

5.2 Modularization Quality

Our goal is to automatically optimize the decomposition of a software system into packages so that the resulting organization of classes/packages, *mainly*, reduces connectivity and cyclic-connectivity between packages. This goal is inspired from well known quality principles already underlined and discussed by Brian *et al.* [*Briand et al.*, 1998], Fowler [*Fowler*, 2001] and Martin [*Martin*, 2002a] (Section 2.3.2 (p. 23)) and in particular from the following principle: *packages are desired to be loosely coupled and cohesive to a certain extent* [*Fowler*, 2001]. In such a context, we need to define metrics that evaluate package *cohesion* and *coupling*.

In addition, Martin underlined that *cyclic dependencies* between packages are considered as an anti-pattern for package design [*Martin*, 2002a].

In this section we define two suites of metrics: the first is used when evaluating modularization quality; the second is used when evaluating modularity quality of single package within a given modularization.

Note that all metrics we define in this section take their value in the interval $[0..1]$ where 1 is the optimal value and 0 is the worst value.

5.2.1 Measuring Modularization Quality

Inter-Package Dependencies. According to Common Closure Principle (CCP) [*Martin*, 2002a], *classes that change together should be grouped together*. In such a context, optimizing modularization requires reducing the sum of inter-package dependencies ($IPD = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Ext.Out.D}}|$) [*Briand et al.*, 1998; *Fowler*, 2001]. Since we do not change the dependencies between classes during our optimization process, we use the sum of inter-class dependencies ($ICD = \sum_{j=1}^{|\mathcal{M}_C|} |c_{j_{Out.D}}|$) as normalizer. We define the metric $\mathcal{CCQ}$ to evaluate the Common Closure Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{CCQ}(\mathcal{M}) = 1 - \frac{IPD}{ICD} \tag{5.1}$$

Inter-Package Connections. According to Common Reuse Principle (CRP) [*Martin*, 2002a], *classes that are reused together should be grouped together*. In such a context, optimizing modularization requires reducing the sum of inter-package connections ($IPC = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Out.Con}}|$) [*Briand et al.*, 1998; *Fowler*, 2001]. We define the metric $\mathcal{CRQ}$ to evaluate the Common Reuse Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{CRQ}(\mathcal{M}) = 1 - \frac{IPC}{ICD} \tag{5.2}$$

Inter-Package Cyclic-Dependencies. According to Acyclic Dependencies Principle (ADP) [*Martin*, 2002a], *dependencies between packages must not form cycles*. In such a context, optimizing modularization requires reducing the sum of inter-package cyclic-dependencies ($IPCD = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Out.Cyc.D}}|$). We define the

metric $\mathcal{ADQ}$ to measure the Acyclic Dependencies Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{ADQ}(\mathcal{M}) = 1 - \frac{IPCD}{ICD} \quad (5.3)$$

Inter-Packages Cyclic-Connections. As for cyclic dependencies between packages, reducing cyclic connections between packages is required, where reducing inter-package cyclic dependencies does not necessarily reduce inter-package direct cyclic-connections ($IPCC = \sum_{i=1}^{|\mathcal{M}_P|} |p_{i_{Out.Cyc.Con}}|$).
We define the metric $\mathcal{ACQ}$ to evaluate the Acyclic Connections Quality of a modularization $\mathcal{M}$ as follows:

$$\mathcal{ACQ}(\mathcal{M}) = 1 - \frac{IPCC}{ICD} \quad (5.4)$$

5.2.2 Measuring Package Quality

In addition to metrics represented in Section 5.2.1 (p. 105), we define a set of metrics that help us determine and quantify the quality of a single package within a given modularization.

To normalize the value of those metrics we use the number of dependencies related to the considered package ($|p_D|$) with $|p_D| > 0$.

Package Cohesion. We relate package cohesion to the direct dependencies between its classes. In such a context, we consider that the cohesion of a package p is proportional to the number of internal dependencies within p ($|p_{Int.D}|$). This is done according to the Common Closure Principle (CCP) [*Martin*, 2002a]. We define the metric of package cohesion quality similarly to that in [*Abreu* and *Goulao*, 2001] as follows:

$$CohesionQ(p) = \frac{|p_{Int.D}|}{|p_D|} \quad (5.5)$$

Package Coupling. We relate package coupling to its efferent and afferent coupling (Ce, Ca) as defined by Martin in [*Martin*, 2005]. Package Ce *is the number of packages that this package depends upon* ($|p_{Pro.P}|$). Package Ca *is the number of packages that depend upon this package* ($|p_{Cli.P}|$). According to the common reuse principle, we define the metric of package coupling quality using the number of package providers and clients as follows:

$$CouplingQ(p) = 1 - \frac{|p_{Pro.P} \cup p_{Cli.P}|}{|p_D|} \quad (5.6)$$

Package Cyclic-Dependencies. For automatically detecting packages that suffer from direct-cyclic dependencies we define a simple metric that evaluates the

quality of package cyclic dependencies (*CyclicDQ*) using the number of package cyclic dependencies:

$$CyclicDQ(p) = 1 - \frac{|p_{Cyc.D}|}{|p_D|} \quad (5.7)$$

Similarly we define another metric that evaluates package cyclic connections quality (*CyclicCQ*) using the number of package cyclic connections:

$$CyclicCQ(p) = 1 - \frac{|p_{Cyc.Con}|}{|p_D|} \quad (5.8)$$

5.3 Optimization Technique (Methodology)

To optimize package connectivity, we use an optimization procedure that starts with a given modularization and gradually modifies it, using small perturbations. At each step, the resulting modularization is evaluated to be possibly selected as an alternative modularization. The evaluation of modularization quality is based on metrics defined in Section 5.2.1 (p. 105). This section describes our optimization approach and algorithm.

5.3.1 Technique Overview

To address the problem of optimizing modularization, we use a heuristic optimization technique based on simulated annealing algorithm [*Ferland* and *Costa*, 2001; *Kirkpatrick et al.*, 1983]. Simulated annealing is an iterative procedure that belongs to the category of Neighborhood Search Techniques (NST).

Algorithm 1 (p. 108) shows an overview of the optimization algorithm. The optimization process performs series of local searches with the global search parameter $T_{current}$. $T_{current}$ represents in simulated annealing technique the current temperature of the annealing procedure which started with the value T_{start}. A local search consists of num ($num \geq 1$) searches of suboptimal solution. At each of them, a new modularization $\mathcal{M}_{trial}$ is derived from a current one $\mathcal{M}_{current}$ by applying to this latter a modification. The derivation of $\mathcal{M}_{trial}$ from $\mathcal{M}_{current}$ is performed by the *Neighborhood* function. Then, the algorithm evaluates the $\mathcal{M}_{trial}$ and $\mathcal{M}_{current}$ *fitness* using the *Fitness* function $\mathcal{F}$, where the bigger is the value of $\mathcal{F}(\mathcal{M})$, the better is modularization $\mathcal{M}$: if $\mathcal{M}_{trial}$ is better than $\mathcal{M}_{current}$ then $\mathcal{M}_{trial}$ becomes the $\mathcal{M}_{current}$; then, if $\mathcal{M}_{current}$ is better than the current best modularization $\mathcal{M}_{best}$, $\mathcal{M}_{current}$ becomes the $\mathcal{M}_{best}$. At the end of each local search, the parameter $T_{current}$ decreases and another local search starts with the new value of $T_{current}$. Decreasing $T_{current}$ is the responsibility of *CoolingSchedule* function. This latter is defined according to Keeffe *et al.* discussion [*O'Keeffe* and *Cinnéide*, 2008] using a geometric cooling scheme: $CoolingSchedule(T) = 0.9975 * T$. Local searches are repeated until reaching T_{stop} ($T_{current} \leq T_{stop}$).

To circumvent the problem of *local optima* [*Ferland* and *Costa*, 2001], a less-good modularization can be accepted with some probability: a less-good modularization

Algorithm 1 Optimization Algorithm

Require: $T_{stop} \geq 1, T_{start} > T_{stop}$, $num > 1$ and $\mathcal{M}_{original}$
Ensure: $\mathcal{M}_{best}$
  $\mathcal{M}_{best} \leftarrow \mathcal{M}_{original}$
  $\mathcal{M}_{current} \leftarrow \mathcal{M}_{best}$
  $T_{current} \leftarrow T_{start}$
  –starting global search–
  while $T_{current} > T_{stop}$ **do**
    –starting local search–
    for $i = 1$ to num **do**
      –generating a new modularization and evaluating it–
      $\mathcal{M}_{trial} \leftarrow Neighborhood(\mathcal{M}_{current})$
      if $\mathcal{F}(\mathcal{M}_{trial}) > \mathcal{F}(\mathcal{M}_{current})$ **then**
        $\mathcal{M}_{current} \leftarrow \mathcal{M}_{trial}$
        if $\mathcal{F}(\mathcal{M}_{current}) > \mathcal{F}(\mathcal{M}_{best})$ **then**
          $\mathcal{M}_{best} \leftarrow \mathcal{M}_{current}$
        end if
      else if $AcceptanceCondition$ **then**
        –accepting a worse modularization–
        $\mathcal{M}_{current} \leftarrow \mathcal{M}_{trial}$
      end if
    end for
    –end of local search–
    $T_{current} \leftarrow CoolingSchedule(T_{current})$
  end while
  –end of global search–
  Return $\mathcal{M}_{best}$.

$\mathcal{M}_{trial}$ can replace $\mathcal{M}_{current}$ under some conditions *AcceptanceCondition*. Simulated annealing technique defines acceptance conditions in a way that the probability of accepting a less-good modularization decreases over time. We define *Acceptance-Condition* as follows: $r > e^{-\frac{T_{current}}{T_{start}}}, r \in [0..1]$. The value of r is generated randomly in the interval $[0..1]$. The function $e^{-\frac{T_{current}}{T_{start}}}$ takes its value in the interval $[0..1]$ $\forall T_{current} \geq 0$, and $T_{current} \leq T_{start}$. It increases along the optimization process –since $T_{current}$ decreases. By doing so, the probability of accepting a less-good modularization decreases over time.

In the next section we define the *Fitness* function that evaluates the quality of a given modularization.

5.3.2 Evaluating Modularization Quality (Fitness)

As for any search-based optimization problem, the definition of the fitness function represents a central concern as it guides the search. We define our fitness function as a combination of the metrics defined in Section 5.2.1 (p. 105). We define dependency quality ($\mathcal{DQ}$) for a modularization $\mathcal{M}$ as the weighted average of *Common Closure Quality*

($\mathcal{CCQ}$) and *Acyclic Dependencies Quality* ($\mathcal{ADQ}$); and we define connection quality ($\mathcal{CQ}$) for $\mathcal{M}$ as the weighted average of *Common Reuse Quality* ($\mathcal{CRQ}$) and *Acyclic Connections Quality* ($\mathcal{ACQ}$). To give higher intention to cyclic dependencies/connections between packages we define a factor of importance γ ($\gamma = \frac{\beta}{\alpha}, \beta > \alpha \geq 1$):

$$\mathcal{DQ}(\mathcal{M}) = \frac{\alpha * \mathcal{CCQ}(\mathcal{M}) + \beta * \mathcal{ADQ}(\mathcal{M})}{\alpha + \beta} \quad (5.9)$$

$$\mathcal{CQ}(\mathcal{M}) = \frac{\alpha * \mathcal{CRQ}(\mathcal{M}) + \beta * \mathcal{ACQ}(\mathcal{M})}{\alpha + \beta} \quad (5.10)$$

Both functions $\mathcal{DQ}$ and $\mathcal{CQ}$ take their values in the interval $[0..1]$ where 1 is the optimal value. The final fitness function is defined by the average of $\mathcal{DQ}$ and $\mathcal{CQ}$:

$$\mathcal{F}(\mathcal{M}) = \frac{\mathcal{DQ}(\mathcal{M}) + \mathcal{CQ}(\mathcal{M})}{2} \quad (5.11)$$

Our hypothesis is: optimizing $\mathcal{F}$ will reduce inter-package dependencies and connections, particularly cyclic ones.

Furthermore, in addition to *AcceptanceCondition* for less-good modularizations we defined in Section 5.3.1 (p. 107), the optimization process may accept a less-good resulting modularization only if the number of inter-package dependencies decreases (*i.e.*, $\mathcal{DQ}$ increases). We expect such a decision facilitates the reduction of inter-package cyclic dependencies.

In addition to the *Fitness* function, the optimization approach should allow maintainers to specify constraints on possible alternative modularizations ($\mathcal{M}_{best}$). In the next section we present the constraints that our optimization approach supports.

5.3.3 Modularization Constraints

In addition to the fitness function, our approach allows maintainers to define distinct constraints that should complete the evaluation process and guarantee maintainers' requirements. The rationale behind those constraints is to control the optimization process when optimizing a given modularization. *e.g.*, putting a major partition of classes into one package can effectively reduce inter-package (cyclic-) dependencies/connections; such an approach is clearly not the best one to optimize software modularization. This section presents three constraints to control the optimization process. Section 5.3.4 (p. 110) explains how the optimization process favors these constraints when deriving new modularizations.

5.3.3.1 Controlling package size

To avoid having very large and dominant packages, we introduce the following constraint: the size of every package (p_{size}) should always be smaller than a predefined number (*size*$_{max}$). We define *size*$_{max}$ for every package p relatively to its size in the original modularization $p_{size_{V0}}$: $size_{max} = \delta + p_{size_{V0}}, \delta \geq 0$. Maintainers can define δ according to the context of the concerned software system. We cannot determine upfront the good interval in which δ should be taken. In the scope of this chapter, we define δ as the *theoretical package size* in the original modularization $\mathcal{M}_0$, which equals

to the ratio: $\frac{|\mathcal{M}_{0_C}|}{|\mathcal{M}_{0_P}|}$. It is worth to note that maintainers can define a different δ for each package: *e.g.*, for a large package p, δ may be defined to 0; this way, p will never be larger than before.

5.3.3.2 Controlling modularization modification

Maintainers should be able to define the *limit* of modifications that the optimization process can apply on the original modularization $\mathcal{M}_0$ when it proceeds. In other words, the optimization process must take into account the *maximal authorized distance* (*distance*$_{max}$) between resulting modularizations and $\mathcal{M}_0$. In our context, for two modularizations that entail the same set of classes, we define the *distance* between them by the number of classes that changed packages. This way, *distance*$_{max}$ can be defined simply as the maximal number of classes that can change their packages.

5.3.3.3 Controlling modularization structure

Moreover, we found that it is very helpful to allow maintainers decide whether some classes should not change their package and/or whether given packages should not be changed. We say that such classes/packages are *frozen*. This constraint is particularly helpful when maintainers know that a given package is well designed and should not be changed: *e.g.*, if a small package p contains a couple of classes that extend classes from other packages, p may be considered a well designed package, even if it is not cohesive. Similarly, it is also helpful when maintainers know that some classes are well packaged together and should not change their package.

This constraint may also be used when maintainers need refactoring propositions for a specified set of classes: in such a case, maintainers may specify the rest of classes as *frozen*. Maintainers may also limit the scope of the possible refactoring propositions to a specified set of packages: in such a case, maintainers may specify the rest of packages as *frozen*.

5.3.4 Deriving New Modularization (Neighbor)

The neighborhood function ($\mathcal{N}$) is the second main concern of the optimization process. Defining $\mathcal{N}$ requires: (1) the definition of the set of modifications that $\mathcal{N}$ can use to derive new modularizations, (2) and the definition of a process that derives a new modularization from another one. This section presents our definition of $\mathcal{N}$.

Since we search near optimal modularization by applying near minimal modification to the original modularization, we limit the set of modifications that $\mathcal{N}$ can use to only: moving a class c from its current package p_{source} to another one p_{target}. In this context, we say that c is the modification $actor$ (c_{actor}). To minimize search-space we reduce the selection-space of p_{target} to the set of client and provider packages of c_{actor}: $p_{target} \in ({c_{actor}}_{Pro.P} \cup {c_{actor}}_{Cli.P})$.

We specify the derivation of a neighbor modularization of a modularization $\mathcal{M}$ by 4 sequential steps: (1) selecting p_{source}, (2) selecting c_{actor}, (3) selecting p_{target} and then (4) moving c_{actor} to p_{target}. Selections in the first three steps are done arbitrary using a probability function. The probability function gives higher probability to the *worst* package into $\mathcal{M}_P$ to be selected as p_{source}, to the *worst* class into p_{source} to

be selected as c_{actor} and to the *nearest* package to c_{actor} to be selected as p_{target}. The selection mechanism performs similarly to a roulette wheel selection, where each class/package is given a slice proportional to its probability to be selected and then we randomly take a position in the roulette and pick the corresponding class/package.

It is worth to note that packages and classes that are defined as *frozen* (Section 5.3.3.3 (p. 110)), do not belong to the selection spaces: a *frozen* package will never be a p_{source} or p_{target}, and a *frozen* class will never be a c_{actor}.

The following subsections explain our definition of the probability of being selected as p_{source}, c_{actor} or p_{target}. Note that in our definition of this probability we use the factor γ, as defined in the fitness function (Section 5.3.2 (p. 108)), to pay more attention to cyclic dependencies/connections.

5.3.4.1 Selecting p_{source}

The worst package in $\mathcal{M}_P$ is, the highest probability to be selected it has. We relate package badness to the quality of its cohesion, coupling and of its external dependencies (*i.e.*, the density of cyclic dependencies/connections related to the concerned package). We define the badness of package by using the metrics: *CohesionQ*, *CouplingQ*, *CyclicDQ* and *CyclicCQ* (Section 5.2.2 (p. 106)): where we relate *CohesionQ* and *CyclicDQ* to package dependency quality ($\mathcal{DQ}$). Also we relate *CouplingQ* and *CyclicCQ* to package connection quality ($\mathcal{CQ}$). We define package quality functions, *similarly to modularization quality functions defined in* Section 5.3.2 (p. 108) :

$$\mathcal{DQ}(p) = \frac{\alpha * CohesionQ(p) + \beta * CyclicDQ(p)}{\alpha + \beta} \tag{5.12}$$

$$\mathcal{CQ}(p) = \frac{\alpha * CouplingQ(p) + \beta * CyclicCQ(p)}{\alpha + \beta} \tag{5.13}$$

Both functions $\mathcal{DQ}$ and $\mathcal{CQ}$ take their values in the interval $[0..1]$ where 1 is the optimal value.
Finally, we define package badness based on the average of $\mathcal{DQ}$ and $\mathcal{CQ}$:

$$\mathcal{B}adness(p) = 1 - \frac{\mathcal{DQ}(p) + \mathcal{CQ}(p)}{2} \tag{5.14}$$

In addition to satisfy constraints discussed in Section 5.3.3 (p. 109), we define the probability of selecting a package p as p_{source} by: $\rho * \mathcal{B}adness(p)$, where ρ is a factor that takes its value in the interval $[0..1]$. It is the average of two sub-factors (ρ_1, ρ_2):

- ρ_1 is based on p_{size}: relatively to p size in the original modularization $\mathcal{M}_0$ ($p_{0_{size}}$), a package whose size increased has a higher probability to be selected than a package whose size decreased. By doing so, we expect that the package size in resulting modularizations will be similar to that in the original modularization;

- ρ_2 is based on the number of new classes into p: relatively to p_0, a package that acquired the largest number of new classes (*i.e.*, classes are not packaged in p_0) has the highest probability to be selected. By doing so, we favor moving classes that already changed their packages until they find their optimal package.

5.3.4.2 Selecting c_{actor}

The worse a class in p_{source} is, the highest probability to be selected it has. We relate class badness to the number of external dependencies related to the class ($|c_{Ext.D}|$) and to the number of its external cyclic-dependencies (related to its package p):

$$\mathcal{B}adness(c) = \frac{\alpha * |c_{Ext.D}| + \beta * |c_{Ext.D} \cap p_{Cyc.D}|}{\alpha + \beta} \tag{5.15}$$

In addition, to satisfy the constraint of *distance*$_{max}$ (Section 5.3.3.2 (p. 110)), when the distance (d) between resulting modularizations and the original one increases, classes that have already changed their packages have higher probability to be c_{actor}. In this context, we use the factor $\rho = 1 - \frac{d}{distance_{max}}$. If $\rho \leq 0$ then only classes which already changed their original packages can move. Only if $0 < \rho \leq 1$ then the optimization process can move more classes over packages but with a probability ρ. Thus we define the probability of selecting a class c as c_{actor} as following:

$$\begin{cases} 0 & \rho \leq 0, not(isMoved(c)) \\ \rho * \mathcal{B}adness(c) & \rho > 0, not(isMoved(c)) \\ \mathcal{B}adness(c) & isMoved(c) \end{cases} \tag{5.16}$$

Where the predicate $isMoved(c)$ is true if c has already moved from its original package.

5.3.4.3 Selecting p_{target}

The nearest package to c_{actor} is, the highest probability to be selected it has. We simply relate the nearness of a package p to a class c to the number of dependencies that c has with p classes ($|c_D \cap p_D|$) and to the number of cyclic dependencies between c and p ($|c_D \cap p_{Cyc.D}|$):

$$\mathcal{N}earness(p, c) = \frac{\alpha * |c_D \cap p_D| + \beta * |c_D \cap p_{Cyc.D}|}{\alpha + \beta} \tag{5.17}$$

To satisfy the constraint on p_{size} ($size_{max}$ defined in Section 5.3.3.1 (p. 109)), when package size increases its probability to be selected as $p_{targert}$ decreases. In this context, we use the factor $\rho = 1 - \frac{p_{size}}{size_{max}}$. If $\rho \leq 0$ then the package size should not increase anymore. Only if $0 < \rho \leq 1$ then the package size can increase but with a probability ρ which decreases when p_{size} increases. Thus we define the probability of selecting a package p as p_{target} for a class c, as following:

$$\begin{cases} 0 & \rho \leq 0 \\ \rho * \mathcal{N}earness(p, c) & 0 < \rho \leq 1 \end{cases} \tag{5.18}$$

5.4 Experiments and Validation

To validate our optimization approach, we applied it to several software systems that differ in terms of: number of classes ($|\mathcal{M}_C|$), number of packages ($|\mathcal{M}_P|$), number of inter-class dependencies (ICD); number of inter-package dependencies (IPD),

Table 5.1: Information about used software applications.

$Original$	$\|\mathcal{M}_C\|$	ICD	$\|\mathcal{M}_P\|$	IPD	$IPCD$	IPC	$IPCC$	$maxP_{size}$	$\frac{\|\mathcal{M}_C\|}{\|\mathcal{M}_P\|}$
JEdit	802	2683	19	1430	1032	110	26	173	42.2
ArgoUML	1671	7432	76	5661	1406	517	63	156	22
Jboss	3094	8859	455	7219	296	1898	41	80	6.8
Azureus	4212	13945	380	10929	1319	2037	136	213	11

connections (IPC), cyclic dependencies ($IPCD$) and cyclic connections ($IPCC$). Table 5.1 (p. 113) shows information about the original modularization of those software systems.

Since the search process is not deterministic, we applied our algorithm 10 times for each software system and we calculated the average of modularization parameters cited in Table 5.1 (p. 113). We used the parameters T_{start}, T_{stop} and num (Algorithm 1 (p. 108)) with value 50, 1 and 30 respectively. On another hand, we weighted cyclic dependencies/connections to be three times more important than noncyclic dependencies/connections. We performed our experience twice: the first time, we did not use the constraint *distance*$_{max}$. In the second time, we limited *distance*$_{max}$ to 5%, which means that only 5% of classes can change their original packages.

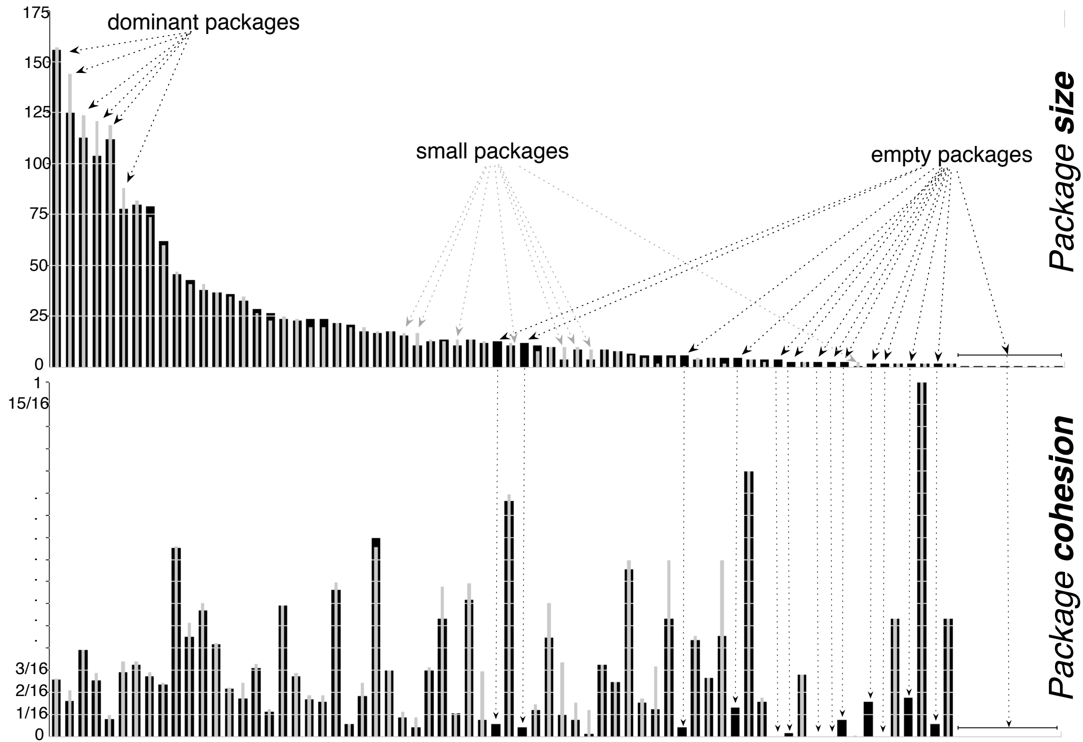


Figure 5.1: Package size and cohesion into *ArgoUML original* (dark gray) and *resulting* (light gray) modularizations. Packages have the same order in diagrams.

General optimization. Table 5.3 (p. 115) shows optimization results. In the resulting

modularization for JEdit (*JEdit*$_1$), 10.2% of inter-package dependencies IPD, 23.3% of inter-package cyclic-dependencies $IPCD$, 24% of inter-package connections IPC and 37.2% of inter-package cyclic-connections $IPCC$ have been removed. This significant improvement of inter-package connectivity was obtained by moving only 8.9% of the classes ($d = 8.9\%$). Similarly for other case studies, the optimization process has improved original modularizations by moving a relatively small number of their classes. When limiting $distance_{max}$ to 5%, the algorithm obtained similar results.

Class distribution and package size. Table 5.4 (p. 116) shows that some packages were *empty* in resulting modularizations -since their classes moved to other packages. For example, in *ArgoUML*$_1$, 25.4% of packages were empty. By inspecting packages in the original modularization we found that those empty packages are packages which have originally very small sizes (*i.e.*, in average two or three classes) and have low quality for cohesion, coupling and/or cyclic dependencies. This conclusion were also true for the other case studies.

Figure 5.1 (p. 113) shows an overview about package size and cohesion for the original modularization of *ArgoUML* and for the resulting modularizations (Table 5.3 (p. 115)). We can see that empty packages in the resulting modularizations are packages whose sizes are small and whose cohesion is relatively worse. On another hand, Figure 5.1 (p. 113) shows also that the size of some small packages, annotated by *small packages*, is increased in the resulting modularization. Dominant packages, annotated by *dominant packages*, is a main cause of bad distribution of classes: moving classes from dominant packages to small ones generally produces more dependencies and connections among packages.

Maintainers can avoid moving classes from small packages to dominant ones by limiting the $size_{max}$ (Section 5.3.3.1 (p. 109)) of dominant packages to their original size: $p_{size_{max}} = p_{size_{V0}}$. This way, the dominant package size will never increase and the optimization process will search better modularizations by moving classes among/to smaller packages.

Fortunately, in the case of *JEdit*$_1$, only 10.5% (2/19) of packages are empty (Table 5.4 (p. 116)), where Table 5.3 (p. 115) shows that our optimization process has effectively optimize package connectivity in *JEdit*$_1$.

Now as a future work, we have to perform a deep manual validation since in presence of late-binding and frameworks, some small packages may extend larger ones and as such may have a real reason to exist. Note that defining such packages as *frozen* (Section 5.3.3.3 (p. 110)) will keep those packages existing.

While some packages became empty, Table 5.4 (p. 116) shows that the average package size ($\frac{|\mathcal{M}_C|}{|\mathcal{M}_P|}$) for the resulting modularizations is really close to the average package size for the original ones Table 5.1 (p. 113). Similarly, we can see that for the maximum package size ($maxP_{size}$). This shows that the optimization algorithm conserves the original system shape.

Package quality optimization. Table 5.5 (p. 116) shows that package quality average is also optimized: cohesion quality average ($CohesionQ_{Avg}$), coupling quality

average (*CouplingQ*$_{Avg}$) and cyclic-dependency quality average (*CyclicDQ*$_{Avg}$) for resulting packages are also almost all optimized, even if *distance*$_{max}$ is limited to only 5%. This can be seen also in Figure 5.1 (p. 113).

In only one case (*JEdit*$_1$ and *JEdit*$_2$), the package coupling quality (*CouplingQ*) decreased with a very good improvement of *CyclicDQ* and *CohesionQ*. We explain this by the fact that the optimization process gives more importance to inter-packages cyclic-dependencies. Indeed, *CyclicDQ* had a very bad value in the original modularization *JEdit* (Table 5.2 (p. 115)): the ratio of inter-package cyclic-dependencies ($\frac{IPCD}{ICD}$) shows that 38.4% of inter-class dependencies form cyclic-dependencies between packages. Moreover there are 802 classes distributed over only 19 packages, so that the search space for generating new modularization is limited.

Table 5.2: Package Quality in original modularizations

$Original$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicsDQ_{Avg}$
JEdit	28.8%	91.4%	40.6%
ArgoUML	17.2%	76.6%	81.6%
Jboss	12.5%	61.8%	96.4%
Azureus	11.7%	72.3%	84.7%

Table 5.3: Optimizations on Inter-Package Connectivity. The top table shows the percent of reduction of IPD, .. , IPCC (Table 5.1 (p. 113)) into resulting modularizations. The biggest negative value is, the best optimization is. The bottom table shows these information when $distance_{max}$ is specified and limited to 5%.

$Optimization_1$	IPD	$IPCD$	IPC	$IPCC$
$JEdit_1(d = 8.9\%)$	-10.2%	-23.3%	-24.0%	-37.2%
$ArgoUML_1(d = 8.3\%)$	-04.4%	-09.0%	-32.7%	-31.8%
$Jboss_1(d = 11.9\%)$	-08.3%	-37.7%	-18.5%	-51.2%
$Azureus_1(d = 9.5\%)$	-06.0%	-23.2%	-6.2%	-28.4%

$Optimization_2$	IPD	$IPCD$	IPC	$IPCC$
$JEdit_2(d = 05.0\%)$	-06.5%	-09.4%	-20.6%	-24.3%
$ArgoUML_2(d = 05.0\%)$	-02.5%	-04.2%	-25.9%	-24.9%
$Jboss_2(d = 05.0\%)$	-03.2%	-12.6%	-11.3%	-21.6%
$Azureus_2(d = 05.0\%)$	-03.2%	-09.3%	-05.7%	-16.7%

Consistency of resulting modularizations. since our optimization approach uses random selection, different executions produce different modularizations. To evaluate the consistency of our optimization approach, we have applied it 10 times on each case study (Table 5.1 (p. 113)). As a result, each system has 10 modularizations $[\mathcal{M}_1..\mathcal{M}_{10}]$. Table 5.6 (p. 116) shows the average distance between every pair $(\mathcal{M}_i, \mathcal{M}_j)$. For example, between resulting modularizations for *JEdit*, there are, *in average*, only 3% of classes that have not the same packages. For *Jboss*, only 5.6% of the classes have different packages in distinct resulting modularizations.

Table 5.4: Modifications on Package Size. The top table shows the percent of empty packages (Table 5.1 (p. 113)), the biggest and the average package size into resulting modularizations. The bottom table shows these information when $distance_{max}$ is specified and limited to 5%.

$Optimization_1$	$EmptyP$	$maxP_{size}$	$\frac{\lvert\mathcal{M}_C\rvert}{\lvert\mathcal{M}_P\rvert}$
$JEdit_1(d = 8.9\%)$	10.5%	176	47.2
$ArgoUML_1(d = 8.35\%)$	25.4%	157	29.3
$Jboss_1(d = 11.9\%)$	22.4%	79	8.8
$Azureus_1(d = 9.48\%)$	15.8%	219	13.2

$Optimization_2$	$EmptyP$	$maxP_{size}$	$\frac{\lvert\mathcal{M}_C\rvert}{\lvert\mathcal{M}_P\rvert}$
$JEdit_2(d = 05.0\%)$	5.3%	176	44.6
$ArgoUML_2(d = 05.0\%)$	21%	155	27.9
$Jboss_2(d = 05.0\%)$	14.7%	81	7.9
$Azureus_2(d = 05.0\%)$	12.9%	215	12.7

Table 5.5: Optimizations on Package quality. The top table shows the average optimizations on package quality into resulting modularizations. Values are based on Table 5.2 (p. 115). The biggest positive value is, the best optimization is. The bottom table shows these information when $distance_{max}$ is specified and limited to 5%.

$Optimization_1$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicDQ_{Avg}$
$JEdit_1$	+06.1%	−00.8%	+10.4%
$ArgoUML_1$	+08.1%	+07.4%	+00.4%
$Jboss_1$	+08.5%	+11.6%	+01.8%
$Azureus_1$	+05.8%	+03.6%	+05.4%

$Optimization_2$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicDQ_{Avg}$
$JEdit_2$	+05.4%	−02.2%	+05.2%
$ArgoUML_2$	+06.0%	+07.4%	+00.1%
$Jboss_2$	+04.0%	+09.4%	+00.5%
$Azureus_2$	+03.9%	+04.6%	+02.2%

We mainly relate this very good consistency of resulting modularizations to the improvements we introduced to the neighbor function $\mathcal{N}$ Section 5.3.4 (p. 110) (*i.e.*, the probability function to being selected).

In conclusion, the obtained results are very convincing. For all the case studies, the new modularizations are clearly better than the original ones. Moreover, our optimization process produces very similar results.

Table 5.6: Resulting Modularization Consistency. Table shows the average distance between ten resulting modularizations for each application.

$Optimization_1$	$Distance_{Avg}$
$JEdit_1$	3%
$ArgoUML_1$	4.1%
$Jboss_1$	5.6%
$Azureus_1$	4.9%

Optimization of package quality with constraints on package size. To illustrate how to control our optimization process, we took, as example, the case study of *ArgoUML* and specified the following constraints:

1. *Do not increase the size of dominant packages:* to avoid increasing the size of dominant packages, we considered that the maximal package size is 35 (*i.e.*, $size_{max} = 35$). This way, our optimization process will not move classes to such packages.
   Note that, in the original modularization of *ArgoUML* there are 14 packages

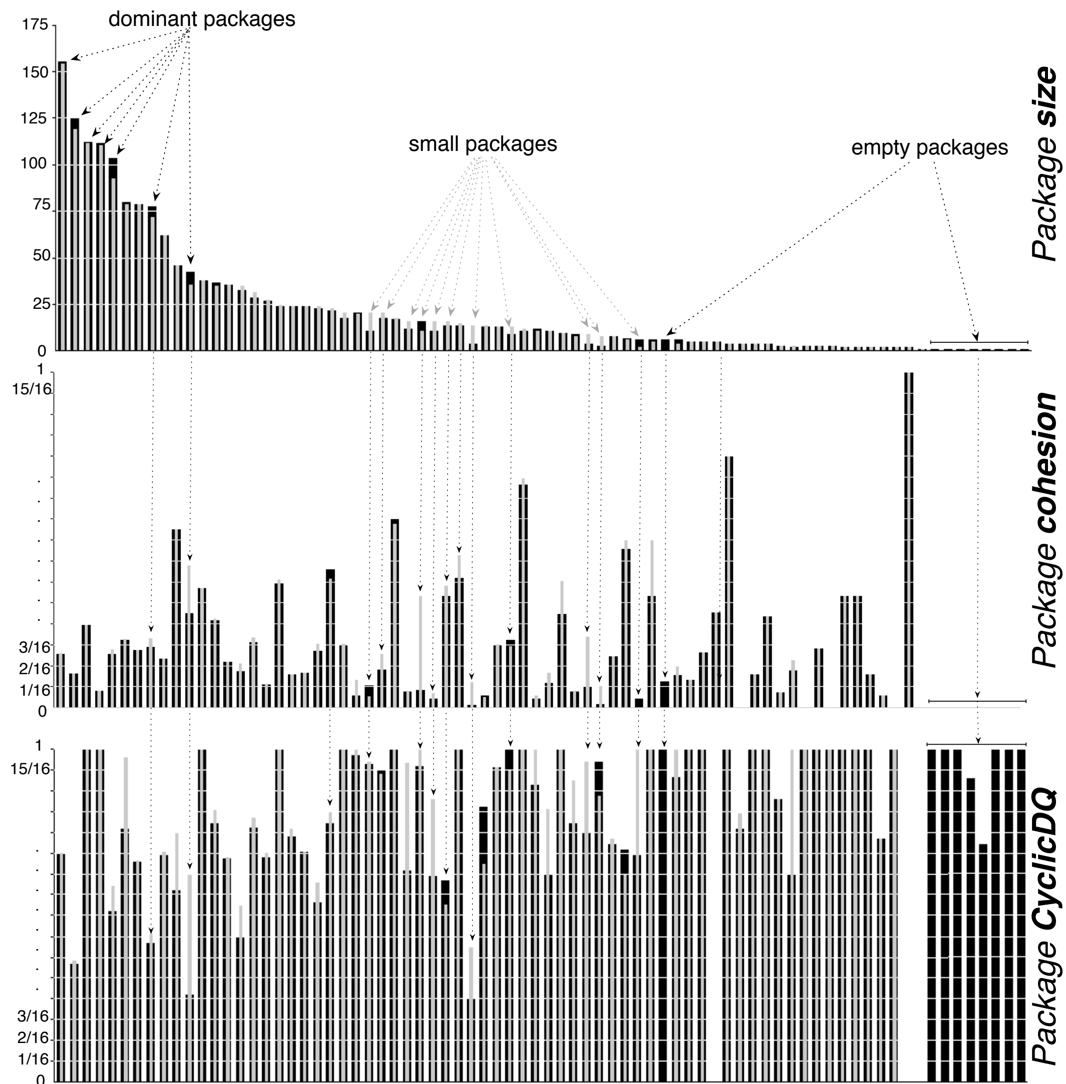


Figure 5.2: Package size, cohesion quality (*CohesionQ*) and cyclic dependency quality (*CyclicDQ*) into *ArgoUML original* (dark gray) and *resulting* (light gray) modularizations. Packages have the same order in diagrams. The constraints are: (1) the size of packages that entail more than 35 should not increase ($size_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*).

that their size is greater than 34. Which means that, this way, about 18% of *ArgoUML* packages can not be target packages for class moving.

2. *Do not move classes from small packages:* to avoid removing small packages, we considered that the classes of small packages should not be moved (*frozen* classes). We defined a small package as a package that entails less than 6 classes, but more than 1 class. We think that a package that entails only one class has no sense.
   Note that, in the original modularization of *ArgoUML* there are 21 packages that their size is smaller than 6 and greater than 1. Those packages entail 68 classes. Which means that, this way, about 4% of *ArgoUML* classes can not change their packages.

Table 5.7: Modifications on Package Size for *ArgoUML*. It shows the percentage of empty packages (Table 5.1 (p. 113)), the biggest and the average package size into resulting modularizations. The constraints are: (1) the size of packages that entail more than 35 should not increase ($size_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*).

$Optimization_1$	$EmptyP$	$maxP_{size}$	$\frac{\|\mathcal{M}_C\|}{\|\mathcal{M}_P\|}$
$ArgoUML_1(d = 5.5\%)$	11.8%	155	24.9

Table 5.8: Optimizations on Package Quality for *ArgoUML*. Values are based on Table 5.2 (p. 115). The biggest positive value is, the best optimization is. The constraints are: (1) the size of packages that entail more than 35 should not increase ($size_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*).

$Optimization_1$	$CohesionQ_{Avg}$	$CouplingQ_{Avg}$	$CyclicDQ_{Avg}$
$ArgoUML_1(d = 5.5\%)$	+04.2%	+04.4%	+02.5%

Table 5.9: Optimizations on Inter-Package Connectivity for *ArgoUML*. It shows the percentage of reduction of IPD, .. , IPCC (Table 5.1 (p. 113)) into resulting modularizations. The biggest negative value is, the best optimization is. The constraints are: (1) the size of packages that entail more than 35 should not increase ($size_{max} = 35$); (2) the classes that are packaged in small packages ($1 < p_{size} < 6$) should not be moved (*i.e.*, they are *frozen*).

$Optimization_1$	IPD	$IPCD$	IPC	$IPCC$
$ArgoUML_1(d = 5.5\%)$	−01.4%	−15.6%	−8.5%	−12.7%

Figure 5.2 (p. 117) shows package size, cohesion quality (*CohesionQ*) and cyclic dependency quality (*CyclicDQ*) into *ArgoUML* original and resulting modularizations. The diagram of package size shows that the size of dominant packages has not increased. It also shows that the size of some dominant packages, denoted by *dominant packages* has decreased. On the other hand, only 9 packages are empty in resulting modularizations: 8 among them are packages that entail only one class in the original modularization.

A summary of Figure 5.2 (p. 117) is that: our optimization process optimized the quality of most packages (*CohesionQ* and *CyclicDQ*) in *ArgoUML*, without moving classes from small packages to dominant ones. This is also shown in Table 5.7 (p. 118), Table 5.9 (p. 118) and Table 5.8 (p. 118).
Table 5.7 (p. 118) shows that the resulting modularization shape ($maxP_{size}$ and $\frac{\mathcal{M}_C}{\mathcal{M}_P}$) is very similar to the original modularization shape (Table 5.1 (p. 113)). While Table 5.9 (p. 118) and Table 5.8 (p. 118) show that package quality ($CohesionQ_{Avg}$, $CouplingQ_{Avg}$ and $CyclicQ_{Avg}$), as well as, package connectivity are optimized.

5.5 Related Works

Our work is mostly related to work on software modularization and decomposition [*Abreu* and *Goulao*, 2001; *Harman* and *Hierons*, 2002; *Harman* and *Tratt*, 2007; *Mancoridis* and *Mitchell*, 1998; *Mancoridis et al.*, 1999; *Mitchell* and *Mancoridis*, 2002, 2006, 2008; *Seng et al.*, 2005].

Mancoridis and Mitchell, [*Mancoridis* and *Mitchell*, 1998; *Mancoridis et al.*, 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 population of random modularizations. The clustering algorithm clusters each of the random modularization and selects the result with the largest quality as the suboptimal solution. Recently, they used Simulated Annealing technique to optimize resulting clusters [*Mitchell* and *Mancoridis*, 2002, 2006] [*Mitchell* and *Mancoridis*, 2008]. Their optimization approach 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.

Harman *et al.* [*Harman* and *Tratt*, 2007] introduces a non-exhaustive hill climbing approach to optimize and determine a sequence of class refactorings. Similarly to our approach, they also restricted their approach to only move methods (classes in our context) over existing classes (packages in our context). The goal of their approach is reducing the class coupling, based on the Coupling Between Objects (CBO) metric [*Briand et al.*, 1998]. To avoid having very large classes, they also used the dispersion of methods over classes (the standard deviation of methods per class metric) as a factor to measure the quality of resulting class refactoring sequences.

Abreu *et al.* [*Abreu* and *Goulao*, 2001] used hierarchical agglomerative clustering methods to decompose software classes into packages. Their clustering methods starts with a set of classes considering that each class is placed within a singleton cluster.
The goal of their approach is also increasing package internal dependencies (*i.e.*, package cohesion). In addition to the package cohesion, they used the dispersion of classes over packages (*i.e.*, package size dispersion) as a factor to measure the modularization quality.

Seng *et al.* [*Seng et al.*, 2005] and Harman *et al.* [*Harman* and *Hierons*, 2002] proposed genetic algorithms to partition software classes into subsystems (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 works is increasing package internal dependencies. Seng *et al.* consider also cyclic-dependencies between packages as anti-pattern for package design quality.

Our approach has several advantages compared to those works.

Considering original modularizations: our approach tackles the problem of optimizing existing software modularizations rather than the problem of software re-modularization. Indeed, our optimization approach starts from one original modularization instead of an initial population of modularizations or a flat set of classes. Although we use an optimization technique similar to that in Mitchell *et al.* work, we restrict ours to moving classes over existing packages rather than creating new packages since we want to minimize the distance for a maintainer between the initial situation and the resulting one. Even if some prior works support the notion of importing a defined clustering [*Tzerpo* and *Holt*, 1997] and restrict modifications to only moving classes over existing packages [*Harman* and *Tratt*, 2007], we did not found, in the software re-modularization literature, approaches that explicitly take into account the original modularization structure as we do. Our approach allows maintainers to specify the maximal number of classes that may change their packages ($distance_{max}$).

Controlling the optimization process: Our approach allows maintainers to specify, in addition to the constraint $distance_{max}$, a set of constraints: (1) the package maximal size ($size_{max}$); (2) the packages/classes which should not be changed/moved (*frozen* entities). As a consequence, it allows them to control the optimization process. Although those constraints are simple, they are very important and helpful for the automatic optimization of software modularization. For example, the constraint $size_{max}$ could be used to avoid increasing the size of dominant packages. The constraint *frozen* could be used to specify that a group of classes should always be together in their original package –even if that package is not cohesive from the point of view of the *CohesionQ* metric (Equation 5.5 (p. 106))

Doing near minimal modifications: differently from those cited works, we introduce a probability function that improves the derivation of neighbor modularizations by taking into account the distance between resulting modularizations and the original one, in addition to other constraints (Section 5.3.3 (p. 109)) and package quality parameters (Section 5.2.2 (p. 106)). The great advantage of the probability function is finding better modularization by doing near-minimal modifications: Section 5.4 (p. 112) shows that the distance between resulting modularizations and the original one is very small.

Reducing inter-package dependencies and *connections*: another advantage of our approach is that we use an evaluation function consisting of a combination of multiple metrics. This allows us to have a much richer quality model than the approaches cited above which are mostly based on the unique goal of maximizing package internal dependencies. Although those approaches aim at reducing the global number of inter-package dependencies (*e.g.*, the fitness metric MQ used by Mitchell *et al.* [*Mitchell* and *Mancoridis*, 2002, 2006, 2008]),

they do not take into account the number of coupled packages (*i.e.*, the number of inter-package connections). As consequence, they do not check whether package coupling is reduced or not along the optimization process. In addition, those cited prior works do not consider package cycles. Excepted Seng *et al.* [*Seng et al.*, 2005], they consider inter-package cyclic-dependencies, without taking in account inter-package cyclic-connections.

5.6 Conclusion and Future Work

In this chapter, we addressed the problem of optimizing existing modularizations by reducing the connectivity, particularly the cyclic-connectivity, among packages. We proposed an optimization algorithm and a set of metrics that our optimization process uses to automatically evaluate the quality of a modularization. When designing our optimization approach, we exploited several principles of package design quality to guide and to optimize the automatic derivation of new modularizations from an existing one. We limited the optimization process to only moving classes over existing packages. We also introduced constraints related to package size, to the number of classes that are allowed to change their packages and to the classes/packages that should not be moved/changed. The results obtained from 4 case studies on real large software systems showed that our optimization algorithm has been able to reduce, significantly, package coupling and cycles, by moving a relatively small number of classes from their original packages. These results are important because the chosen software systems have radically different original modularizations (in terms of number of classes/packages, inter-class/inter-package dependencies, *etc.*).

As future work, we intend to enhance our approach by: (1) supporting indirect cyclic-dependencies among packages, (2) taking into account visibility of classes and particular cases of classes (*e.g.*, inner classes in Java). Indeed in this chapter, we considered that classes are always public and then can change their packages, (3) setting up a real validation supported by proper statistics and qualitative analyses of resulting source code structures.

Chapter 6

Conclusions and Future Work

In the life, there is no problem,
there is a sequence of solutions

Although packages are important to cope with the complexity of software systems, packages, themselves, are complex and play distinct roles. However, as software systems evolve, the software modularization drifts, and as a consequence, it should be maintained. In that respect, maintainers need to understand the structure of packages, their intra and inter dependencies, their roles, *etc.* They also need to assess the modularization quality, as well as the quality of a single package within the concerned modularization; and looking for good alternative modularizations without breaking the shape of the original modularization.

We have reviewed various existing approaches that tried to solve a substantial body of the problem of the maintenance of software system modularizations. Our review reveals that these approaches usually focus on few features about packages. The existing approaches for understanding packages fall short of providing a fine-grained view of packages that would help understanding the package shapes. The existing metrics for assessing the quality of packages focus only on some aspects of the package shape without an explicit relation to the package design principles, and they fall short of assessing the quality of a given modularization. The existing approaches for the automatic remodularization of software systems suffer from several limitations: (1) they have as unique goal to maximize package intra-dependencies, without considering other parameters of the modularization quality; (2) they do not allow maintainers to specify constraints on alternative modularizations; (3) they often produces new modularizations that are completely different from the original ones.

In this dissertation we argue for the need for: (1) approaches that help to understand package shapes at a fine and coarse grained level; (2) metrics that help to automatically assess, from different perspectives, the quality of a modularization, as well as the quality of a single package within a given modularization; (3) an approach

that help to automatic searching for good alternative modularizations.

Our proposal consists of:

The visual map Package Blueprint: a compact visualization that helps in understanding fine-grained structures and dependencies (internal and external) of packages. We defined three variants of the Package Blueprint: (a) the incoming reference package blueprint maps the incoming references that point to the package classes; (b) the outgoing reference package blueprint maps the outgoing references that exit the package classes; (c) the inheritance package blueprint maps the outgoing inheritances that exit the package classes.

In Chapter 3 (p. 39) we illustrated that the Package Blueprint for a given package reveals a lot of information about the package shape and dependencies. For example, a package blueprint reveals: (1) the package size; (2) the package internal and external complexity; (3) the number of its client and provider classes/packages; (4) the distribution of its client/provier classes over the client/provider packages; (5) the importance of each client/provider package; (6) the importance of each referencing/referenced class; (7) the scope of the incoming/outgoing dependencies: *e.g.*, whether if the package refers to classes belonging to other subsystems than the package subsystem, or to classes belonging to the package subsystem; (8) the package role: whether if the package is central to the analyzed system or peripheral; (9) the package cohesion based on the direct dependencies among the package classes –a package blueprint shows, in detail, if the concerned package respects, or not, the Common Closure Principle (CCP), as described in Section 2.3.2 (p. 23); *etc.*

A result of our validation of the Package Blueprint is the identified set of visual patterns that help to quickly identify package and class shapes.

The visual map Package Fingerprint: a compact, rich and zoomable visualization to better support the understanding of package interfaces, relationships and the conceptual coupling of package classes (*i.e.*, package contextual cohesion). The goal of this visualization is to help maintainers during their early contacts with unknown packages. We defined two complementary variants of the Package Fingerprint, structured around the distribution of references from or to the classes of the analyzed package: (1) the incoming fingerprint shows how the system uses the package classes, and highlights the conceptual cohesion of the analyzed package –an incoming fingerprint shows, in detail, if the concerned package respects, or not, the Common Reuse Principle (CRP), as described in Section 2.3.2 (p. 23); (2) the outgoing fingerprint shows how the package classes use the system and highlights the coupling of the package classes from the point of view of provider packages.

Package Fingerprints are orthogonal to the Package Blueprint. Maintainers may use the Package Fingerprints to understand and characterize packages at a high level of abstraction. After that, they may use the Package Blueprint to reveal package's detail and see the distribution of dependencies over classes.

Outgoing Fingerprint works similarly to Incoming Fingerprint but from the point of view of package outgoing references (Out-Interface), referenced packages and package reasons-for-changing, instead of the the point of view of

package incoming references (In-Interface), referencing packages and package provided services.

A package metric suite: we defined a suite of metrics that help to compute the quality of a single package within a given modularization. Our metrics follow the principles of package cohesion and coupling: Common Closure Principle (CCP), Common Reuse Principle (CRP) and Acyclic Dependencies Principle (ADP), that we described in Section 2.3.2 (p. 23). The aim of these metrics is to automatically identify candidate packages for restructuring (Chapter 5 (p. 103)). These metrics may also be used to have a first impression about a given package. In Chapter 5 (p. 103) we illustrated how to use these metrics to identify candidate packages for restructuring. We also defined a strategy (measure) to determine the place quality of a class within its package, based on the class external dependencies: class $\mathcal{B}adness$ (Equation 5.15 (p. 112)). In addition, we defined a strategy to propose suitable refactoring of a given class, based on the external dependencies that the class has with its client/provider packages: the $\mathcal{N}earness$ between a class c and a package p (Equation 5.17 (p. 112)).

A modularization metric suite: we defined a suite of metrics that help to compute the quality of a modularization. The aim of these metrics is to automatically assess the quality of a modularization and the impact of changes within the concerned modularization. These metrics also follow the principles of package cohesion and coupling cited above.

In Chapter 5 (p. 103) we illustrated how to use these metrics in our approach of automatic optimization of a software modularization. These metrics can be used in any similar approach and also to have a first impression about the quality of a given modularization.

A search-based optimization approach: we defined an approach based on the local search-based technique Simulated Annealing to automatically reduce package coupling and cycles. Our approach searches for good alternative modularizations by doing near minimal modifications on the original modularization. It also allows maintainer to control the optimization process by defining divers constraints on the possible alternative modularizations: (1) the maximal number of classes that may change their package; (2) the maximal number of classes that a package may contain; (3) the classes that should not change their package; (4) the packages that should not be changed.

In Chapter 5 (p. 103) we validated our optimization approach on real large software systems and showed that the resulting modularization effectively optimizes the connectivity among packages. The obtained results illustrate that: (1) our optimization approach is characterized by a very good consistence, which is a good sign; (2) it optimizes existing software modularization by moving a relatively small number of classes; (3) it does a relatively small modifications on the package size.

Maintainers may use our optimization approach to have several alternative modularizations. They then can use our metrics with our visualizations to understand these alternative modularizations and choose one of them as the most

suitable solution. Since the distance between resulting modularizations and original ones is relatively small, it should be easy to map back the alternative modularizations. In addition, since our visualizations support interactive mechanisms, maintainers can mark classes that changed their packages by a given color. This way, they can easily detect changes and understand the alternative modularizations.

According to this overview, our proposal covers the set of requirements, that we identified in Section 2.8 (p. 37), for the maintenance of large and complex software modularizations.

6.1 Open Issues

In this dissertation we exploit only explicit dependencies among classes, which can be class inheritance, class access and method call. However, beyond the kinds of dependencies explored in our proposal, there are further kinds of dependencies among packages. For example, the class extension dependency that we described in Section 2.2 (p. 16). We think that the techniques represented in this dissertation can be refined with new dependency kinds.

We have shown that packages play different roles and offer different views. However, our visualization approaches do not exhaustively analyze every possible perspective of packages. In this dissertation, we focus on the static structure of packages. We think that our visualization approaches are complementary to approaches that analyze: package evolution, the spread of properties over packages and the software architecture design.

We defined a suite of metrics that compute the quality of software modularizations and packages from different perspectives. However, we think that our metrics need a deeper validation that characterizes these metrics and their correlations. In addition, our metrics do not cover all the principles of package cohesion and coupling that we described in Section 2.3.2 (p. 23): *e.g.*, our metrics do not cover the Reuse-Release Equivalence Principle. Therefore, other new metrics are needed.

In this dissertation, we also defined an approach for the automatic reduction of package cycles and coupling. Our approach takes into account the quality of software modularization, in addition to several constraints that maintainers may define. However, we think that our approach can be refined to: (1) support indirect cyclic-dependencies among packages; (2) take into account visibility of classes and particular cases of classes (*e.g.*, inner classes in Java). Indeed in our approach, we considered that classes are always public and then can change their packages; (3) support more types of constraints on classes and packages: *e.g.*, maintainers should be able to specify if a given class can be moved to only an identified set of packages.

Bibliography

Abdeen, H., **Alloui**, I., **Ducasse**, S., **Pollet**, D., and **Suen**, M.: (2008). Package reference fingerprint: a rich and compact visualization to understand package relationships. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 213–222. IEEE Computer Society Press.

Abdeen, H., **Ducasse**, S., **Pollet**, D., and **Alloui**, I.: (2009a). Package fingerprint: a visual summary of package interfaces and relationships. *Under submission at the Information and Software Technology (IST) Journal*.

Abdeen, H., **Ducasse**, S., **Sahraoui**, H., and **Alloui**, I.: (2009b). Automatic package coupling and cycle minimization. In *International Working Conference on Reverse Engineering (WCRE)*, pp. ?–? IEEE Computer Society Press.

Abreu, F. B. and **Goulao**, M.: (2001). Coupling and cohesion as modularization drivers: are we being over-persuaded? In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 47–57.

Allen, E. and **Khoshgoftaar**, T.: (2001). Measuring coupling and cohesion of software modules: An information theory approach. In *International Software Metrics Symposium*.

Anquetil, N. and **Lethbridge**, T.: (1999). Experiments with Clustering as a Software Remodularization Method. In *Working Conference on Reverse Engineering (WCRE)*, pp. 235–255.

Arisholm, E., **Briand**, L. C., and **Foyen**, A.: (2004). Dynamic coupling measurement for object-oriented software. *Transactions on Software Engineering (TSE)*, 30(8):pp. 491–506.

Bauer, M. and **Trifu**, M.: (2004). Architecture-aware adaptive clustering of oo systems. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 3–14. IEEE Computer Society Press, Washington, DC, USA.

Berard, E. V.: (1993). *Essays On Object-Oriented Software Engineering*, volume 1. Prentice-Hall.

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

Bertin, J.: (1983). *Semiology of Graphics*. University of Wisconsin Press.

Beyer, D.: (2005). Co-change visualization. In *International Conference on Software Maintenance (ICSM), Industrial and Tool volume*, pp. 89–92.

Bieman, J. and **Kang**, B.: (1995). Cohesion and reuse in an object-oriented system. In *ACM Symposium on Software Reusability*.

Bieman, J. and **Kang**, B.: (1998). Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):pp. 111–124.

Bieman, J. and **L.M.Ott**: (1994). Measuring functional cohesion. *IEEE Transactions on Software Engineering (TSE)*, 20(8):pp. 644–658.

Briand, L. C., **Daly**, J. W., and **Wüst**, J.: (1998). A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):pp. 65–117.

Briand, L. C., **Daly**, J. W., and **Wüst**, J. K.: (1999a). A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):pp. 91–121.

Briand, L. C., **Daly**, J. W., and **Wüst**, J. K.: (1999b). Using coupling measurement for impact analysis in object-oriented systems. In *International Conference on Software Engineering (ICSE)*, pp. 475–482.

Callebaut, W. and **Rasskin-Gutman**, D.: (2005). *Modularity: Understanding the Development and Evolution of Natural Complex Systems*. MIT press.

Chapman, W. L., **Rozenblit**, J., and **Bahill**, A. T.: (2001). System design is an np-complete problem: Correspondence. *Systems Engineering*, 4(3):pp. 222–229.

Chidamber, S. R. and **Kemerer**, C. F.: (1994). A metrics suite for object oriented design. *Transactions on Software Engineering (TSE)*, 20(6):pp. 476–493.

Chuah, M. C. and **Eick**, S. G.: (1998). Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):pp. 24–29.

Clarke, J., **Dolado**, J. J., **Harman**, M., **Jones**, B., **Lumkin**, M., **Mitchell**, B., **Rees**, K., and **Roper**, M.: (2003). Reformulating software engineering as a search problem. In *IEEE Proceedings on Software*, volume 3, pp. 161–175.

D'Ambros, M. and **Lanza**, M.: (2006a). Applying the evolution radar to postgresql. In *Workshop on Mining Software Repositories (MSR)*, pp. 177–178.

D'Ambros, M. and **Lanza**, M.: (2006b). Reverse engineering with logical coupling. In *Working Conference on Reverse Engineering (WCRE)*, pp. 189 – 198.

D'Ambros, M. and **Lanza**, M.: (2006c). Software bugs and evolution: A visual approach to uncover their relationship. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 227–236. IEEE Computer Society Press.

D'Ambros, M. and **Lanza**, M.: (2007). Bugcrawler: Visualizing evolving software systems. In *European Conference on Software Maintenance and Reengineering (CSMR)*, p. to be published. IEEE Computer Society Press.

D'Ambros, M., **Lanza**, M., and **Lungu**, M.: (2006). The evolution radar: Integrating fine-grained and coarse-grained logical coupling information. In *International Workshop on Mining Software Repositories (MSR)*, pp. 26–32. ACM.

Demeyer, S., **Ducasse**, S., and **Nierstrasz**, O.: (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

Demeyer, S., **Tichelaar**, S., and **Ducasse**, S.: (2001). *FAMIX 2.1 — The FAMOOS Information Exchange Model*. Technical report, University of Bern.

Denker, M. and **Ducasse**, S.: (2007). Software evolution from the field: an experience report from the Squeak maintainers. In *Proceedings of the ERCIM Working Group on Software Evolution*, volume 166 of *Electronic Notes in Theoretical Computer Science*, pp. 81–91. Elsevier.

DeRemer, F. and **Kron**, H. H.: (1976). Programming in the large versus programming in the small. *IEEE Transactions on Software Engineering (TSE)*, 2(2):pp. 80–86.

Dong, X. and **Godfrey**, M.: (2007). System-level usage dependency analysis of object-oriented systems. In *International Conference on Software Maintenance (ICSM)*, pp. 375–384. IEEE Computer Society Press.

Doval, D., **Mancoridis**, S., and **Mitchell**, B. S.: (1999). Automatic clustering of software systems using a genetic algorithm. In *the Software Technology and Engineering Practice (STEP)*, p. 73. IEEE Computer Society Press, Washington, DC, USA.

Ducasse, S., **Abdeen**, H., **Pollet**, D., **Suen**, M., and **Alloui**, I.: (2009). Understanding packages: The package blueprint. *Under submission at the IEEE Transactions on Software Engineering (TSE) Journal*.

Ducasse, S., **Gîrba**, T., and **Kuhn**, A.: (2006a). Distribution map. In *International Conference on Software Maintenance (ICSM)*, pp. 203–212. IEEE Computer Society Press, Los Alamitos CA.

Ducasse, S., **Gîrba**, T., **Lanza**, M., and **Demeyer**, S.: (2005a). Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pp. 55–71. Franco Angeli, Milano.

Ducasse, S., **Gîrba**, T., and **Wuyts**, R.: (2006b). Object-oriented legacy system trace-based logic testing. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 35–44. IEEE Computer Society Press.

Ducasse, S., **Lanza**, M., and **Ponisio**, L.: (2005b). Butterflies: A visual approach to characterize packages. In *Software Metrics Symposium (METRICS)*, pp. 70–77. IEEE Computer Society Press.

Ducasse, S., **Pollet**, D., **Suen**, M., **Abdeen**, H., and **Alloui**, I.: (2007). Package surface blueprints: Visually supporting the understanding of package relationships. In *International Conference on Software Maintenance (ICSM)*, pp. 94–103. IEEE Computer Society Press.

Eick, S., **Graves**, T., **Karr**, A., **Marron**, J., and **Mockus**, A.: (2001). Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering (TSE)*, 27(1):pp. 1–12.

Eick, S., **Graves**, T., **Karr**, A., **Mockus**, A., and **Schuster**, P.: (2002). Visualizing software changes. *IEEE Transactions on Software Engineering*, 28(4):pp. 396–412.

Emerson, T.: (1984). A discriminant metric for module cohesion. In *International Conference on Software Engineering (ICSE)*.

Farrugia, A.: (2004). Vertex-partitioning into fixed additive induced-hereditary properties is np-hard. *the electronic journal of combinatorics*, 11.

Feathers, M. C.: (2005). *Working Effectively with Legacy Code*. Prentice Hall.

Fenton, N. and **Pfleeger**, S. L.: (1996). *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, 2nd edition.

Ferland, J. A. and **Costa**, D.: (2001). Heuristic search methods for combinatorial programming problems.

Flanagan, D.: (1999). *Java In a Nutshell: 3rd Edition*. O'Reilly, 3rd edition. Http://hell.org.ua/Docs/oreilly/javaenterprise/jnut/index.htm.

Fowler, M.: (2001). Reducing coupling. *IEEE Software*, 18:pp. 102–104.

Fowler, M., **Beck**, K., **Brant**, J., **Opdyke**, W., and **Roberts**, D.: (1999). *Refactoring: Improving the Design of Existing Code*. Addison Wesley.

Froehlich, J. and **Dourish**, P.: (2004). Unifying artifacts and activities in a visual tool for distributed software development teams. In *International Conference on Software Engineering*, pp. 387–396. IEEE Computer Society Press, Washington, DC, USA.

Gîrba, T., **Kuhn**, A., **Seeberger**, M., and **Ducasse**, S.: (2005). How developers drive software evolution. In *International Workshop on Principles of Software Evolution (IWPSE)*, pp. 113–122. IEEE Computer Society Press.

Griswold, W. G. and **Notkin**, D.: (1993). Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):pp. 228–269.

Harman, M.: (2007). The current state and future of search based software engineering. In *Future of Software Engineering (FOSE)*, pp. 342–357. IEEE Computer Society Press, Washington, DC, USA.

Harman, M. and **Hierons**, R.: (2002). A new representation and crossover operator for search-based optimization of software modularization. In *the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1351–1358. Morgan Kaufmann Publishers.

Harman, M. and **Tratt**, L.: (2007). Pareto optimal search based refactoring at the design level. In *the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1106–1113. ACM.

Hautus, E.: (2002). Improving Java software through package structure analysis. In *International Conference Software Engineering and Applications*, pp. ?–?

Healey, C. G.: (1992). *Visualization of Multivariate Data Using Preattentive Processing*. Master's thesis, Department of Computer Science, University of Bristish Columbia.

Healey, C. G., **Booth**, K. S., and **Enns**, J. T.: (1993). Harnessing preattentive processes for multivariate data visualization. In *Proceedings of Graphics Interface (GI)*, pp. 107–117.

Healey, C. G., **Booth**, K. S., and **Enns**, J. T.: (1995). Visualizing real-time multivariate data using preattentive processing. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 5(3):pp. 190–221.

Henderson-Sellers, B.: (1996). *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall.

Jain, A. K., **Murty**, M. N., and **Flynn**, P. J.: (1999). Data clustering: a review. *ACM Computing Surveys*, 31(3):pp. 264–323.

Kirkpatrick, S., **Jr.**, C. D. G., and **Vecchi**, M. P.: (1983). Optimization by simulated annealing. *Science*, 220(4598):pp. 671–680.

Kuhn, A., **Ducasse**, S., and **Gîrba**, T.: (2007). Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):pp. 230–243.

Kuhn, A., **Loretan**, P., and **Nierstrasz**, O.: (2008). Consistent layout for thematic software maps. In *Working Conference on Reverse Engineering (WCRE)*, pp. 209–218. IEEE Computer Society Press, Los Alamitos CA.

Langelier, G., **Sahraoui**, H., and **Poulin**, P.: (2005). Visualization-based analysis of quality for large-scale software systems. In *IEEE/ACM international Conference on Automated software engineering (ASE)*, pp. 214–223. ACM, New York, NY, USA.

Lanza, M.: (2001). The evolution matrix: Recovering software evolution using software visualization techniques. In *International Workshop on Principles of Software Evolution (IWPSE)*, pp. 37–42.

Lanza, M.: (2003). *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. Ph.D. thesis, University of Bern.

Lanza, M. and **Ducasse**, S.: (2002). Beyond language independent object-oriented metrics: Model independent metrics. In F. B. **e Abreu**, M. **Piattini**, G. **Poels**, and H. A. **Sahraoui** (Editors), *Proceedings of the 6th International Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pp. 77–84.

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

Lanza, M. and **Marinescu**, R.: (2006). *Object-Oriented Metrics in Practice*. Springer-Verlag.

Laval, J., **Bergel**, A., and **Ducasse**, S.: (2008). Assessing the quality of your software with moqam. In *FAMOOSr, 2nd Workshop on FAMIX and Moose in Reengineering*, pp. 28–31.

Laval, J., **Denier**, S., **Ducasse**, S., and **Bergel**, A.: (2009). Identifying cycle causes with enriched dependency structural matrix. In *Working Conference on Reverse Engineering (WCRE)*, pp. ?–?

Lehman, M. and **Belady**, L.: (1985). *Program Evolution: Processes of Software Change*. London Academic Press, London.

Liu, X., **Swift**, S., and **Tucker**, A.: (2001). Using evolutionary algorithms to tackle large scale grouping problems. In *the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 454–460.

Lung, C.-H., **Xu**, X., **Zaman**, M., and **Srinivasan**, A.: (2006). Program restructuring using clustering techniques. *Journal of Systems and Software*, 79(9):pp. 1261–1279.

Lungu, M., **Lanza**, M., and **Gîrba**, T.: (2006). Package patterns for visual architecture recovery. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 185–196. IEEE Computer Society Press, Los Alamitos CA.

Lutz, R.: (2001). Evolving good hierarchical decompositions of complex systems. *Journal of Systems Architecture*, 47(7):pp. 613–634.

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

Maini, H., **Mehrotra**, K., **Mohan**, C., and **Ranka**, S.: (1994). Genetic algorithms for graph partitioning and incremental graph partitioning. In *Supercomputing*, pp. 449–457. IEEE Computer Society Press, Los Alamitos, CA, USA.

Mancoridis, S. and **Mitchell**, B. S.: (1998). Using automatic clustering to produce high-level system organizations of source code. In *International Workshop on Program Comprehension (IWPC)*, pp. 45–52. IEEE Computer Society Press.

Mancoridis, S., **Mitchell**, B. S., **Chen**, Y., and **Gansner**, E. R.: (1999). Bunch: A clustering tool for the recovery and maintenance of software system structures. In *International Conference on Software Maintenance (ICSM)*, pp. 50–59. IEEE Computer Society Press, Oxford, England.

Martin, R. C.: (1996). Granularity. Www.objectmentor.com.

Martin, R. C.: (2000). Design principles and design patterns. Www.objectmentor.com.

Martin, R. C.: (2002a). *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall.

Martin, R. C.: (2002b). Srp: The single responsibility principle. Www.objectmentor.com.

Martin, R. C.: (2005). The tipping point: Stability and instability in oo design. Software Development.

Melton, H. and **Tempero**, E.: (2007). The crss metric for package design quality. In *the Australian Computer Science Conference (ACSC)*, pp. 201–210.

Meyer, B.: (1989). The new culture of software development: Reflections on the practice of object-oriented design. In *Proceedings TOOLS*, pp. 13–23.

Mitchell, B. S.: (2002). *A heuristic search approach to solving the software clustering problem*. Ph.D. thesis, Drexel University, Philadelphia, PA, USA. Adviser-Mancoridis, Spiros.

Mitchell, B. S. and **Mancoridis**, S.: (2002). Using heuristic search techniques to extract design abstractions from source code. In *the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1375–1382. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Mitchell, B. S. and **Mancoridis**, S.: (2006). On the automatic modularization of software systems using the bunch tool. *Transactions on Software Engineering (TSE)*, 32(3):pp. 193–208.

Mitchell, B. S. and **Mancoridis**, S.: (2008). On the evaluation of the bunch search-based software modularization algorithm. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 12(1):pp. 77–93.

Mitchell, B. S., **Mancoridis**, S., and **Traverso**, M.: (2004). Using interconnection style rules to infer software architecture relations. In *the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1375–1387. Seattle, Washington.

Mišić, V. B.: (2001). Cohesion is structural, coherence is functional: Different views, different measures. In *International Software Metrics Symposium (METRICS)*. IEEE.

Morris, K.: (1989). *Metrics for Object-Oriented Software Development Environments*. Master's thesis, Sloan School of Management. MIT.

Myers, G. J.: (1978). *Composite/Structured Design*. Van Nostrand Reinhold.

O'Keeffe, M. and **Cinnéide**, M. O.: (2006). Search-based software maintenance. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 249–260. IEEE Computer Society Press, Los Alamitos, CA, USA.

O'Keeffe, M. and **Cinnéide**, M. O.: (2008). Search-based refactoring for software maintenance. *Journal of Systems and Software*, 81(4):pp. 502–516.

Parnas, D. L.: (1972). On the criteria to be used in decomposing systems into modules. *CACM*, 15(12):pp. 1053–1058.

Patel, S., **Chu**, W., and **Baxter**, R.: (1992). A measure for composite module cohesion. In *International Conference on Software Engineering (ICSE)*, pp. 38–48.

Petre, M.: (1995). Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):pp. 33–44.

Pinzger, M., **Gall**, H., **Fischer**, M., and **Lanza**, M.: (2005). Visualizing multiple evolution metrics. In *ACM Symposium on Software Visualization (SoftVis)*, pp. 67–75. St. Louis, Missouri, USA.

Pollet, D., **Ducasse**, S., **Poyet**, L., **Alloui**, I., **Cîmpan**, S., and **Verjus**, H.: (2007). Towards a process-oriented software architecture reconstruction taxonomy. In R. **Krikhaar**, C. **Verhoef**, and G. **Di Lucca** (Editors), *European Conference on Software Maintenance and Reengineering (CSMR)*, pp. 137–148. IEEE Computer Society Press. Best Paper Award.

Ponisio, L. and **Nierstrasz**, O.: (2006). Using context information to re-architect a system. In *Software Measurement European Forum (SMEF)*, pp. 91–103.

Ponisio, M. L.: (2006). *Exploiting Client Usage to Manage Program Modularity*. Ph.D. thesis, University of Bern, Bern.

Pressman, R. S.: (1994). *Software Engineering: A Practitioner's Approach*. McGraw-Hill.

Rising, L. and **Calliss**, F. W.: (1992). Problems with determining package cohesion and coupling. *Software - Practice and Experience*, 22(7):pp. 553–571.

Sangal, N., **Jordan**, E., **Sinha**, V., and **Jackson**, D.: (2005). Using dependency models to manage complex software architecture. In *ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 167–176. ACM, New York, NY, USA.

Seng, O., **Bauer**, M., **Biehl**, M., and **Pache**, G.: (2005). Search-based improvement of subsystem decompositions. In *the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1045–1051. ACM, New York, NY, USA.

Serban, G. and **l. G. Czibula**: (2007). Restructuring software systems using clustering. In *International Symposium on Computer and Information Sciences (ISCIS).*, pp. 1–6.

Stevens, W. P., **Myers**, G. J., and **Constantine**, L. L.: (1974). Structured design. *IBM Systems Journal*, 13(2):pp. 115–139.

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

Storey, M.-A. D., **Čubranić**, D., and **German**, D. M.: (2005). On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *ACM symposium on software visualization (SoftVis)*, pp. 193–202. ACM Press.

Storey, M.-A. D., **Wong**, K., **Fracchia**, F. D., and **Müller**, H. A.: (1997). On integrating visualization techniques for effective software exploration. In *Symposium on Information Visualization (InfoVis)*, pp. 38–48. IEEE Computer Society Press.

Sullivan, K. J., **Griswold**, W. G., **Cai**, Y., and **Hallen**, B.: (2001). The structure and value of modularity in software design. In *ESEC/FSE-9*, pp. 99–108. ACM.

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

Tufte, E. R.: (1997). *Visual Explanations*. Graphics Press.

Tufte, E. R.: (2001). *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition.

Tzerpo, V. and **Holt**, R. C.: (1997). The orphan adoption problem in architecture maintenance. *Reverse Engineering, Working Conference on*, 0:p. 76.

Voinea, L., **Telea**, A., and **van Wijk**, J. J.: (2005). CVSscan: visualization of code evolution. In *ACM Symposium on Software Visualization (Softviz)*, pp. 47–56. St. Louis, Missouri, USA.

Ware, C.: (2000). *Information visualization: perception for design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Wettel, R. and **Lanza**, M.: (2007a). Program comprehension through software habitability. In *International Conference on Program Comprehension (ICPC)*, pp. 231–240. IEEE Computer Society Press.

Wettel, R. and **Lanza**, M.: (2007b). Visualizing software systems as cities. In *IEEE International Workshop on Visualizing Software For Understanding and Analysis (VISSOFT)*, pp. 92–99.

Wiggerts, T.: (1997). Using clustering algorithms in legacy systems remodularization. In I. **Baxter**, A. **Quilici**, and C. **Verhoef** (Editors), *Working Conference on Reverse Engineering (WCRE)*, pp. 33–43. IEEE Computer Society Press.

Wysseier, C.: (2005). *Interactive 3-D Visualization of Feature-Traces*. Master's thesis, University of Bern, Switzerland.

Xie, X., **Poshyvanyk**, D., and **Marcus**, A.: (2006). Visualization of CVS repository information. In *Working Conference on Reverse Engineering (WCRE)*, pp. 231–242. IEEE Computer Society Press, Washington, DC, USA.

Yourdon, E.: (1979). *Classics in Software Engineering*. Yourdon Press.