Exploiting Client Usage to Manage Program Modularity

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

María Laura Ponisio

von Argentinien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Prof. Dr. Stéphane Ducasse
Institut für Informatik und angewandte Mathematik

Exploiting Client Usage to Manage Program Modularity

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

María Laura Ponisio

von Argentinien

Leiter der Arbeit:
Prof. Dr. Oscar Nierstrasz
Prof. Dr. Stéphane Ducasse
Institut für Informatik und angewandte Mathematik

Von der Philosophisch-naturwissenschaftlichen Fakultät angenommen.

Bern, 30.06.2006

Der Dekan:

Prof. P. Messerli

Abstract

Over the last thirty years designers have tried to cope with software complexity by organizing system entities into modules, *i.e.,* groups of entities. However, the creation and organization of modules is not straightforward. The criterion with which these modules are built impacts in the maintainability and development of the system. Designers have different interests and personal views of the same system, views that are difficult to communicate and to extract from the code. Poor understanding of this organization increases the complexity of the system *e.g.,* by favoring the addition of duplication and of unexpected rippling effects. This, in turn, lowers the flexibility of the system to changing requirements and leads to a sharp increase in their maintenance cost.

To overcome these problems, we present a methodology to manage the locality in object-oriented systems. We develop a model that exploits the contextual information, *i.e.,* the way objects are used by their clients, to understand and improve the organization of classes in the system. With our model we take full advantage of the contextual information of modules to evaluate their cohesion, find misplaced classes, detect hot spots and find the different views that its clients have.

In our experimental validation we apply the contextual information to understand, maintain and describe systems. Our methodology is applied successively together with metrics, visualization techniques, and an optimization method named simulated annealing to reverse-engineer object-oriented systems. All in all, we provide a methodology to understand and improve the modularization of object-oriented systems, in an effort towards simplicity.

To my parents María Cristina Noval
and Jorge Roberto Ponisio

To my grandparents Cristina Torrente,
Emilio Higinio Noval,
Angélica Emma Epíscopo,
and Domingo Angel Ponisio

Acknowledgments

First and foremost, I would like to express my gratitude to Prof. Oscar Nierstrasz for his guidance, encouragement and for being always there when I needed him. Oscar, it was a pleasure working with you.

I would like to thank to Prof. Stéphane Ducasse for the long discussions, energy, interest in my work and for letting me share unforgettable moments with his family.

I am grateful also to Prof. Houari Sahraoui for generously giving me ideas, for his patience, for accepting to review my thesis, and for coming over to my Ph. D. defense.

Many thanks to Prof. Hanspeter Bieri for his warm welcome in the university of Bern and for accepting to chair the defense.

Very special thanks go to my family. I thank my parents, María Cristina Noval and Jorge Roberto Ponisio, my grandparents Cristina Torrente, Emilio Higinio Noval, Angélica Emma Episcopo, and Domingo Angel Ponisio, and my aunt, uncle and cousin Ana María Noval, Alejandro Azadte and Paula Prieto for their terrific support, encouragement and love.

I would also like to thank my friends for sharing their life with me. Thanks to Luis Cleve and Stella Córdoba for their unconditional advice and support. Special thanks to Lelli Samanta, Marcel Zumbühl, Gian Luca Zumbühl for making me feel part of the family. Many thanks to Andreas Schlapbach for helping me, for his company and for the many interesting lunches together. Many thanks also to Therese Schmid, for her unconditional help, generosity, understanding, patience, open mind, wise advice, smile, and for teaching me swiss german.

Thanks to Martín Ricardo Miguel Auriemma, Martín Mantovani, and Christine Violeau for keeping the friendship despite the geographic distance.

Very special thanks to my neighbor, Gertrude Kummer, who during these three years took care of me as a grandmother would. She called in with warm dishes in winter and with flowers in spring. Finally, she patiently taught me German through my senses and through my heart.

Many thanks to all the professors who have helped me during my thesis. I especially thank Prof. Gustavo Rossi, my mentor and advisor in the licentiate thesis in Argentina, for believing in me and for introducing me to the world of research. Many thanks to my advisor in the master thesis in France, Prof. Jacques Noyé for the valuable discussions. I thank Prof. Linda Ott for discussing some ideas with me and for giving me advice and feedback about my work. Thanks also to Prof. Rolf Hänni for the interesting talks. Thanks to Paul Asman, who have helped me and collaborated with me during my work.

Many thanks also to Orla Greevy, Tudor Girba and Oana Girba for reviewing my thesis and for their comments.

Next I would like to thank all the former and current members of the Software Composition Group. It was a pleasure to work with you: Juan Carlos Cruz, Marcus Denker, Markus Gälli, Lukas Renggli, Marc-Philippe Horvath, Adrian Kuhn, Adrian Lienhard, Alexandre Bergel, Frank Buchli, Roel Wuyts, Juan Carlos Cruz, Mauricio Seeberger, Daniele Talerico, David Vogel, Niklaus Haldimann, Christoph Hofer, Philippe Marschall, Gabriela Arévalo, Florian Thalmann, and Pascal Zumkehr, Matthias Rieger, Michele Lanza, and Nathanael Schärli.

Finally, I would like to send my best to all the great people I have met and talked to at conferences, meetings, and workshops.

Usually, people ask me if it was worth it to enlarge the scenario and find my way in Switzerland. My answer is yes; I am very happy of every single step, because it led me to meet these people.

June 30, 2006
María Laura Ponisio

Table of Contents

Chapter 1

Introduction

Modularity is the single attribute of software that allows a program to be intellectually manageable [Myers, 1978]. It leads to simple, changeable programs, which make coding and maintenance easier, faster and less expensive [Pressman, 1994]. It limits the impact of change and reduces complexity [Yourdon, 1979]. To take advantage of these benefits, engineers strive for maximizing the locality of the objects in the system by dividing it into pieces containing every and only related objects of a kind. For example, we define in the same package all the classes related to the implementation of a feature.

However, as the system evolves, this modularity is easily broken. The old recipes for modularizing, namely to define a clean interface, exploit the advantages of abstraction, minimize coupling and maximize cohesion, fail to keep the system modularized through its evolution. The interfaces of modules become polluted during maintenance tasks with obsolete elements, or excessively loaded with responsibilities specific to unconnected clients of the module. This breakage in the modularisation lowers the flexibility to changing requirements of the system and increases its maintenance cost.

Despite the thirty years of research effort, the methodology to deal with locality is still unsatisfying. We claim that one of the reasons behind this failure is the methodology for its evaluation, which relies on the assumption that it is *only* influenced by interactions between elements of the module. This assumption is completely unfounded. If we want to understand and manage locality, we need to examine (as well as the internals) attributes external to the module, namely the context of the module.

To exploit the way modules interact, we concentrate on packages representing modules and containing classes. *Contextual information* is the name we use to express the information concerning the interaction between a package and the rest of the system.

In the context of class-based object-oriented systems, this work exploits the context information of packages to manage locality of classes.

1.1 The Problem of Managing Program Modularity

Legacy systems are systems that often play a key role in an organization and cannot simply be thrown away and replaced [Demeyer *et al.*, 2002]. They must therefore be maintained and upgraded. We focus on object-oriented legacy systems because early adopters of object-oriented technology need at this moment to reverse engineer legacy object-oriented systems, and because due to the popularity of this paradigm, current systems are being implemented in object-oriented languages, which in turn results in future object-oriented legacy systems to maintain.

Reverse engineering is a prerequisite to system maintenance and evolution. Understanding the code in object-oriented systems is more difficult than in procedural ones, as reading object-oriented code is more difficult than reading procedural code [Dekel, 2002]. Absence of reading order, incremental class definition, polymorphism and late binding are challenges that object-oriented approach adds when analyzing software systems [Wilde and Huitt, 1992].

Moreover, software is maintained by people. It is people who create software units, people who reuse them, and people who have to change the code preserving existing behaviour. Preserving behaviour is a challenge [Feathers, 2005] because users depend on it and it can involve the risk of breaking apparently unconnected areas of the system. If developers change the system without understanding it, they may break existing behavior, thus introducing bugs. If they introduce bugs, change, or remove the behaviour on which users depend, the system fails and developers lose the user's trust [Feathers, 2005]. What, then, facilitates software maintenance for the maintainers? There are several techniques. One approach is to construct a mental picture of the system, such as the one obtained with an appropriate visualization technique. Another, as in other aspects of life, is to group together in one place all the elements that are related. In computer science, the movement claiming this wisdom started more than thirty years ago and was called Modular Design [Yourdon, 1979].

To support software maintenance, engineers strive for maximizing system's modularization at several levels of granularity: method, class, package, and subsystem. At the level of packages, engineers strive to define in the same package, classes that are used together [Martin, 2000]. In other words, they try to improve the *locality* of the classes defined in a package. Even though enforced at development time, system modularity becomes obsolete at a later time, when more development and new maintenance tasks

obscure the original role of the package through the introduction of new semantics. The various agents acting on the system, *e.g.,* developers of different teams, maintainers, and client programs, each have its particular view of a certain package, a view that, applied in each maintenance task, clouds and deviates from the originally intended role of the package. Sparse related classes break the modularity of the system, complicating the reuse of modules, and increasing the cost of further maintenance tasks.

Research effort has been applied to automatically classify similar entities such as methods in classes and classes in packages. Most of these techniques acknowledge only the attributes *internal* to the entity. That approach sometimes appears to contradict the view of the developer, who sees entities as related even when they have no specific dependence between them. Furthermore, there is no such a thing as a unique view of the system where its elements can be partitioned in such a way that every agent consider the elements of a module as *local* to this module.

As locality of elements of a system depends on the guiding principle of the observer, the system evolving according to the sometimes conflicting interest of the maintainers, and the impossibility of a single way to partition the system making every entity local for each of its users, we believe that it is evident the necessity of managing locality.

Research Question:

How can we understand and improve the organization of classes into modules?

We strive to find a technique that facilitates the organisation and reorganisation of an evolving object-oriented system. Starting from the *leap of faith* that by maximizing the locality of software elements, the system becomes more manageable by developers, our hypothesis is that through understanding context information of packages, the system structure is revealed. We validate our hypothesis by creating and implementing such locality-manageability techniques and by showing the characteristics of the system that they reveal.

1.2 Our Proposal: Contextual Information

Our ultimate goal is to improve the modularity of systems. We believe that to define classes that are used together in the same place (for instance, in the same package) facilitates maintenance tasks. Under these circumstances, we claim that to characterise packages we need not only to acknowledge the elements *in* the package, but also to acknowledge the perspective of *clients*, in other words by including context information.

Everyday life is full of tasks that involve grouping similar items. Let us consider the example of a library. Namely, how do readers find software engineering books. The search can be performed using internal attributes, such as by subject matter and author, for example. But it can also exploit information that is *outside* the item, such as all the books that colleagues read. If every member of our research team reads a certain book, chances are that this book is probably related to our research interests, and therefore useful for us. In other words, the characterization mechanism is refined by the addition of information regarding how a book is *used*.

Item's *usage* is concerned with external attributes of software entities. Traditionally, software entities are characterised by internal attributes. Considering also external attributes enriches software entities' description by describing them from the perspective of their interactions.

Thesis:

To express system structural properties we need to recognize contextual information as an explicit phenomenon.

1.3 Contributions

The main contributions of this thesis can be summarized as follows:

1. A detailed analysis of the existing different methods to compute and visualize locality of entities as the system evolves.
2. A formal model to manage locality by capturing contextual information.
3. A measurement to capture the locality of classes [Ponisio and Nierstrasz, 2006b].
4. A coarse-grained view to visualize interaction of packages [Ducasse *et al.*, 2004].
5. A novel technique to visualize the interaction of modules [Ducasse *et al.*, 2005b].
6. A semi automatic technique that provides hints as how to re-locate classes to improve locality of every package in the system [Ponisio and Nierstrasz, 2006a].

1.4 Structure of the Dissertation

This dissertation is structured as follows:

Chapter 2 (p.7) elaborates on the problem of managing the locality in evolving systems and previous research effort on this field.

Chapter 3 (p.25) presents the model for contextual information, the basis of the further research that infer information from the system from the way entities are used.

Chapter 4 (p.39) presents examples of the use of contextual information for describing packages, namely the introduction and analysis of a measurement that describes a package according to the use that their clients make of its interface classes, and detection strategies based on Contextual Information that detect packages with specific design issues.

Chapter 5 (p.61) presents butterfly blueprints, a visualization technique to characterize packages based on the contextual information of packages. It elaborates on the possible uses of butterflies to reason about the evolving system.

Chapter 6 (p.83) presents an idea that associates the contextual information and the problem of solving combinatorial optimizations to produce a technique that automatically finds new combinations of classes that increase the locality of the system.

Chapter 7 (p.95) contains the conclusion of this work. It includes a discussion on the current applications of the contextual information, with its advantages and drawbacks, lessons learnt during the validation of the ideas presented here, and a view of future applications suggested by our approach.

Appendix A (p.101) The Implementation of the Model. The model proposed in this dissertation was implemented in a tool named ALCHEMIST that served as support for the implementation and analysis of the novel measurements, the visualization techniques, and the optimization of locality. This chapter elaborates on the design issues of this implementation, the architectural overview, and the integration of ALCHEMIST in the reengineering environment.

Appendix B (p.105) We validated the ideas presented in this dissertation by applying them to several case studies. This chapter describes the case studies.

Chapter 2

Organizing Classes in Object-Oriented Systems

In the object-oriented paradigm the unit of a class is too small to gain an understanding of a system as a whole [Zenger, 2002]. The hope is then in forming groups of related classes. But how can we organize classes in such a way that developers can later find them efficiently?

Any chosen organization may prove to be neither straightforward nor obvious for a given developer. Between two opposing forces, namely the need to increase modularization and unavoidable interactions between groups, only the designer criteria, interests and experience act as guidelines for the engineer when deciding where to define a class [Demeyer *et al.*, 2002]. As a consequence of these ad hoc criteria, classes can be misplaced. If conceptual classes are defined in different packages, developers working in a particular package may miss class definitions and create a new class clone of the class he does not see because it is defined in an unexpected place. This practice leads to duplicated code and ripple effects with minor changes effecting multiple packages and aggravates the task of maintenance.

Moreover, as the system evolves, the organisation of classes into packages degrades. Different clients of the package need different interfaces [Fowler, 1997], which imposes different views regarding which classes are related, and therefore, in which package a class should be defined. In other words, the *"locality"* of a group of related classes is altered or destroyed by dispersing the classes into different packages. These different perspectives transform the original design into an unclear one, which in turn hinders the maintenance of the system. The question now is how to understand and regain high

locality of the classes.

2.1 Introduction

What makes some systems easier to maintain for people than others? 50% to 75% of the overall cost of a software system is devoted to its maintenance [Lientz and Swanson, 1980]. During maintenance, software professionals spend at least half their time reading and analysing software in order to understand it [Corbi, 1989] [Basili, 1997]. Classes contain abstractions of code elements that are essential for understanding and maintaining the system. However, they are too small gain understanding of a system. Maintenance can be facilitated by organising classes into highly modular groups, *i.e.,* where the group contains only related classes and coupling between groups are minimised. In such groups it is easier to retrieve related items, and they facilitate understanding of the system. However, this organization brings out the problem of where to define each class.

This chapter analyzes existing approaches in different fields used to manage locality. It establishes current limits of the state of art, and a set of open problems.

2.1.1 Locality of Classes

The term *locality* has different meanings in different disciplines. In computer science it is a condition in which items accessed temporally closely are also physically close. This concept deals with the process of accessing a single resource multiple times. The reason behind this is the manner in which computer programs access data. Thus, in the realm of compilers, optimization techniques like *caching* (*i.e.,* techniques that modify a system to make it more efficient) exploit locality, for it dictates to put physically close objects that are accessed temporally closely. In the realm of software reengineering, and in the particular context of object-oriented programming of this dissertation, a *temporal* access has no meaning, since object-oriented languages hide memory allocation issues. It is superseded by a *functionality* access.

Locality is a property of a group of classes where the ones that provide the same functionality are defined together. Previous approaches captured the degree to which classes belong together, and named cohesion this property of a set of classes [Fenton *et al.*, 1994]. The difference between cohesion and locality is that cohesion traditionally refers to internal attributes of the group of classes. Locality, on the other hand, considers also external attributes, *i.e.,* it considers elements outside the group of classes. Locality takes the point of view of the user, that is, not from the point of view of entities being

related, but from the point of view of the interface exposed by the group and how the clients view and access that interface.

2.1.2 Locality and Maintenance

For developers to find classes efficiently, classes implementing functionality of related concepts must be defined in the same group, or *package*. If this fails, developers looking for a class during the maintenance of the system, spend time searching and may overlook related classes. Failing locality diminishes developer's understanding, increases class-search times, menace reuse, and increases the risk of duplication. When locality is broken, the effects of a change propagate to apparently independent pieces of software. Developers cannot predict the impact of a change, become reluctant to refactor the code (*i.e.,* to transforme the source-code of a program without changing its external behavior [Fowler *et al.*, 1999]) and the system loses flexibility with respect to changing requirements. All in all, neglecting locality management leads to systems that are rigid to change.

The management of locality is not straightforward. Here are some difficulties for instance that the developer finds when he is looking where to define a class:

There is no perfect place to define a class. A reasonable location to define a class depends of the point of view of the developer.

It is not easy to identify packages containing related classes. Classes are too small modular units to understand the system as a whole. Therefore, developers need facilities to characterize packages as means to understand the system and to know where to add the new class.

Traditional cohesion conflicts with developer's concept of related classes. The developers may be afraid of creating packages that will be evaluated as non-cohesive for traditional cohesion measures.

The big picture of locality of all the packages in the system is missing. Patterns revealed by visualization techniques show developers how to structure a system, but it is difficult to foresee the best probable criterion to aggregate classes.

Classes need to be re-located as systems evolve. After changes in the system, it may be no longer reasonable for class A to be defined in package P.

Approaches solving the problem of locality should prevail over some questions, namely: what is the best way to describe packages to understand their interaction? Which is the right information to visualize? How can it be achieved in an scalable way? There are different types of dependencies that connect classes (for instance inheritance definition,

method invocation, and references as in a class instantiation). Which type is relevant to characterize packages? How can different dependency types be combined?

2.2 Existing Approaches to Manage Locality

In the last thirty years, research effort has been spent on finding methods to evaluate and increase the locality of the elements in structured programming as well as in object oriented systems. Already in 1974, Stevens, Constantine and Meyer [Stevens *et al.*, 1974] observed that *"programs that were easy to implement and maintain were those composed of simple, independent modules"*, and encouraged to increase cohesion as a way to lower coupling. Unfortunately the proposed approach for evaluating cohesion was manual.

Later work focused on finding mechanisms to automatically measure the structure of a program. Examples of this work are automatic measurements for cohesion and coupling [Bieman and L.M.Ott, 1994][Fenton *et al.*, 1994], remodularizing a system by applying clustering techniques [Mancoridis and Mitchell, 1998][Anquetil and Lethbridge, 1999], and producing visualization of the locality [Pintado and Tsichritzis, 1988].

In last three decades there have been a number of research efforts that help developers understand object-oriented systems. But those approaches neglect the perspective of the client, by focusing on the *internal* attributes of software entities, and obviating analysis of the package context.

Efforts made in the last three decades to help developers understand object-oriented systems can be organized in the following groups: measurements (there is a substantial amount of work to derive module's coupling, cohesion and complexity), visualization (visualization techniques reveal structure of the system, providing knowledge that guide developers through different analysis tasks), clustering and automatic function optimizations methods (semi-automatic techniques exploit both developer's knowledge, and automatic techniques to group entities), offering new language constructs and design principles (both used as facilities and guidelines for development respectively).

In the following sections we give a short account of these topics.

2.2.1 Measurements

There is a substantial body of work on deriving metrics to improve the structured design of the system, in particular to measure cohesion, coupling and complexity [Fenton *et*

al., 1994] [Zuse, 1990] [Briand *et al.*, 1996] [Henderson-Sellers, 1996], [Briand *et al.*, 1998a].

Cohesion

In 1974, Stevens, Myers and Constantine introduced the concept of cohesion in the context of structured design [Stevens *et al.*, 1974], together with the notion that increasing cohesion and minimizing coupling helps developers to cope with complexity. Since then, many metrics have been defined to compute the cohesion of a module [Allen and Khoshgoftaar, 2001][Bieman and Kang, 1995][Bieman and Kang, 1998][Bieman and L.M.Ott, 1994][Briand *et al.*, 1998b][Chidamber and Kemerer, 1991][Chidamber and Kemerer, 1994][Emerson, 1984][Henderson-Sellers, 1996][Lakhotia, 1993][S. Patel, 1992].

In large systems, developers need to analyze packages [Fowler, 1997]. There exist many cohesion measures for modules in structured programming and classes in object-oriented systems which can eventually be extended to packages. However, there are few cohesion measures devoted to packages as sets of classes [Morris, 1989] [Fenton and Pfleeger, 1996] [Allen and Khoshgoftaar, 2001]. Emerson presents a measure to compute cohesion applicable to modules in the sense of Pascal procedures [Emerson, 1984]. His measure is based on a graph theoretic property that quantifies the relationship between control flow paths and references to variables. Bieman and Ott compute cohesion using a slice abstraction of a program based on data slices [Bieman and L.M.Ott, 1994]. Patel *et al.* [S. Patel, 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. It uses 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.

Bunge defined the notion of similarity between two objects as the intersection of the sets of their properties [Bunge, 1974]. This definition is the basis of the first measurement intended to capture the cohesion of a class know as *lack of cohesion*.

In object-oriented programming, Chidamber and Kemerer propose a measure for class cohesion named LCOM [Chidamber and Kemerer, 1991] [Chidamber and Kemerer, 1994], criticized and improved by Henderson-Sellers's LCOM* [Henderson-Sellers, 1996]. Hautus [Hautus, 2002] proposes a tool to analyze the structure of Java programs and a metric to determine the quality of the package architecture. 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 module and intramodule in terms of the subgraph's information and cohesion in terms of intramodule coupling. However this

approach does not take into account classes, inheritance and their relationships.

Bieman and Ott [Bieman and L.M.Ott, 1994] measure functional cohesion; whereas Bieman and Kang's TCC [Bieman and Kang, 1995] measures cohesion for classes. Their measure assesses cohesion using the number of pairs of methods in a class that access common instance variables. They provide an intramodule cohesion measure for cohesion based on the design level information [Bieman and Kang, 1998].

Misic adopts a different perspective and measures the cohesion of a package as an external property of a module [Mišić, 2001]. Following an approach closer to ours, he claims that the internal organization of a module isn't enough to determine its cohesion. Morris follows this line by computing module cohesion considering the fan-in of the contained objects [Morris, 1989].

Morris defined DCO (Degree of Cohesion of Objects), a measure that considers the context of the subject object [Morris, 1989].

Definition 1 Degree of Cohesion of Objects *(DCO), according to the author is:*

$$\mathrm{DCO}(P) = \frac{\textit{total fan-in for objects}}{\textit{total number of objects}}$$

This definition leaves unclear what objects are. Our interpretation of the measure is based on packages and classes, and it is the following: the total number of client packages considering dependency types inherits and references for every class in the package divided by the total number of classes in the package. Under this setup, DCO can take values from 0 to greater than 1. Furthermore, this definition leaves unclear what fan-in is. We interpret fan-in as the number of packages that are clients of the subject package. This is the fan-in interpretation used by JDepend [JDepend, 2005]. We will discuss the measure fan_in in the following section.

Fenton proposed an inter-modular measure of cohesion, Cohesion Ratio (CR) [Fenton and Pfleeger, 1996]. Taking modules to be classes,

Definition 2 CR *is the ratio between functionally cohesive classes and the total number of classes.*

$$\mathrm{CR}(P) = \frac{\textit{number of classes having functional cohesion}}{\textit{total number of classes}}$$

To compute Cohesion Ratio we need to know first if a class is functionally cohesive. There are several methods to know if a class is functionally cohesive, *e.g.,* determining it after inspection of the code. We chose to determine it using the measure Tight Class Cohesion (TCC), defined as the relative number of directly connected methods, where

two methods are directly connected if they access a common instance variable of the class. The threshold to consider a class cohesive was 0.7. The values obtained with CR range from 0 to 1.

As CR's definition shows, CR (like other traditional measures) disregards the context where the subject package operates. In other words, it neglects perspective of the clients of the package. As a result from that it misses to detect packages whose role is given by the context rather by connections between classes defined in them.

Regarding the definition of cohesion, we take our cue from Fenton and Pfleeger's definition of cohesion [Fenton and Pfleeger, 1996] since individual elements can also contribute to the "same task" even if they are only coincidentally associated. Fenton and Pfleeger propose a definition of cohesion of a module that captures the essence being general enough to be applied regardless of the concrete definition of the module. He defines cohesion as the degree to which elements in a module belong together [Fenton *et al.*, 1994].

The different approaches may consider a module as a set of processing elements, a class, or a cluster, depending on programming paradigm. They focus on the *explicit* dependencies and interactions between the classes *within* the package under study. A package, however, may be conceptually cohesive even though its classes exhibit no explicit dependencies.

Summary:

We need techniques to capture that classes belong together from the developers point of view. Information based on the content of the package is not enough to capture cohesion.

Inter-Module Measures

Inter-module measures are traditional coupling measures for modules. Coupling between packages is necessary to delegate responsibilities [Berard, 1993], but it has the disadvantage that changes in a package P with clients may ripple to those clients. Analysis of coupling reveals change propagation. For instance, when any package in the system depends on P, P is a central to the application.

fan_in refers to a measure counting the number of packages that are clients of a package P, *i.e.,* those that depend upon classes within P (see Section 3.3 (p.29)) [Henderson-Sellers, 1996]. This measure is also known as *afferent couplings* [JDepend, 2005].

Definition 3 *Fan_in (*fan_in*) as the number of packages client of the analysed package.*

$$\textit{fan\_in(P)}=\mid \mathsf{clients}(P) \mid$$

Where a package Q is client of P if Q depends on P, which happens when there exists at least a dependency from Q to P.

The definition of fan_in leaves unspecified dependencies of which type (see Section 3.2 (p.26)) to take into account in order to consider a package a client of another. Our interpretation of fan_in acknowledges dependency types inherits (fan_in inherits), or references (fan_in references). Once again, Section 3.2 (p.26) discuss these dependency types; whereas Section 3.3 (p.29) gives a formal definition of clients (P).

If P does not depend on any package, changes produced on other packages do not affect it: P is independent. Conversely, when P depends on many packages, it receives the ripple effects from those packages. The measure fan_out indicates P independence [JDepend, 2005]. Henderson-Sellers applies fan_out to classes and associates high values of it with bad object-oriented design [Henderson-Sellers, 1996].

Definition 4 *We define Fan_out as the number of provider packages.*

$$\textit{fan\_out(P)}=\mid providers(P) \mid$$

Definition 5 *Instability is an indicator of the package resilience to change [JDepend, 2005]. It refers to the ratio of efferent coupling* fan_out *to total coupling (*fan_out + fan_in*) .*

$$I(P) = \frac{\mathsf{fan\_out}(P)}{\mathsf{fan\_out}(P) + \mathsf{fan\_in}(P)}$$

The range for this metric is 0 to 1, with 0 indicating a completely stable package and 1 indicating a completely instable package.

Conceptual Frameworks

As the number of measures for coupling and cohesion increased, it became clear the need for conceptual frameworks to compare and categorize metrics related to cohesion and coupling [Hitz and Montazeri, 1995; Briand *et al.*, 1998b; Briand *et al.*, 1998b; Briand *et al.*, 1999]. We have chosen two complementary conceptual frameworks to analyze the behavior of cohesion measures. The first one is the well-known and criticized

set of properties proposed by Weyuker [Weyuker, 1988], which defines nine properties that good complexity measures should have [Fenton *et al.*, 1994].

Is cohesion a complexity measure? According to Berard, semantic cohesion is an " *externally discernible concept that assesses whether the abstraction represented by the module (here a class) can be considered as a whole semantically"* [Berard, 1993].

A complexity measure describes how complex it is to understand a program [Henderson-Sellers, 1996]. It could be argued that packages containing classes highly interrelated are cohesive and therefore complex. Moreover, taking into account Fenton's definition of cohesion (which says that cohesion of a module is the degree to which its elements belong together), packages highly cohesive reveal to the developer the concept behind the group of classes forming that package. There is in any case a relationship between cohesion and complexity, and we agree with Henderson-Sellers in that class cohesion is a module measure for procedural complexity.

Weyuker [Weyuker, 1988] defined a set of nine properties as basis for evaluation of complexity measures. The proposed set of properties applies originally to programs, but it can be easily extended classes and packages in object-oriented programming. Several authors refer to it to explain properties of their proposed measures [Chidamber and Kemerer, 1991]. Weyuker's axiomatic approach does not prove that a measure is good or bad. Zuse, for instance, has proven that some of these properties are mutually incompatible. Even measures that practice has proven to be useful under certain contexts as, for instance, Mc Cabe's cyclomatic complexity [McCabe, 1976], do not satisfy all the properties. It represents, however, a set of guidelines to understand the behavior of a complexity measure. A case in point is the first property: *"A measure should not rank every program equally complex."* Naturally, it is not useful at all to have a measure so coarse as to not to reflect a different in the attribute measured. The fifth property, for instance, *"Adding code to a program cannot decrease its complexity."* helps the reader to analyze the measure under similar scenarios. In that regard, it agrees with Zuse, who propose to analyze complexity measures by observing their values before and after performing atomic actions on the module where the measure is applied. For instance, it the measure is applied to a graph with nodes and edges, an atomic action is to add an edge. Under these circumstances, we agree with Henderson-Sellers in that Weyuker's axioms, in spite of being severely criticized, *do* provide a conceptual framework which can be adapted and refined in an object-oriented framework [Henderson-Sellers, 1996].

The second conceptual framework, is the one proposed by Briand *et al.* [Briand *et al.*, 1996]. This conceptual framework is a generic framework based on mathematical concepts, and in particular it offers a discussion of a cohesion measure.

The may cohesion conceptual frameworks shows the degree of difficultly in finding a good universal criteria for evaluating cohesion measures. Part of such criteria depends

on the technique to compute the measure. For instance, Chidamber and Kemerer's LCOM cohesion measure for class cohesion (based on methods accessing attributes) requires a different technique than Mc Cabe's complexity (based on graphs deduced from the code) to be evaluated. Once again, Zuse proposes to evaluate complexity measures by applying atomic actions to the object of study and prove that the measure values correspond to an intuitive behavior [Zuse, 1990]. This is the approach, for instance, used by Henderson-Sellers to prove that LCOM is not a good cohesion measure. Briand claims the importance of a clear definition free of ambiguities. In other words, a reader must be able to recompute it. For the sake of comparison, it is a desirable property for a cohesion measure to be within a range, so that packages with maximum and minimum cohesion can be clearly indentified. Cohesion measures must, as any other measure, satisfy the representational theory of measurement [Fenton *et al.*, 1994], which asserts that *"a measurement mapping M must map entities into numbers and empirical relations into numerical relations in such a way that the empirical relations preserve and are preserved by the numerical relations"*. In other words, if the attribute to measure is how-hot-is-it, and if today is hotter than yesterday, a measure for how-hot-is-it must assign a higher number to the evaluation of today than to the evaluation of yesterday. Furthermore, this must be done regularly and with any degree of refinement (if today it was a bit hotter than yesterday, the numbers must also reflect it). Finally, Fenton claims that *"A measure must be viewed in the context in which it will be used. Validation must take into account the measurement's purpose; a measure may be valid for some uses, but not for others"*.

In summary, a good cohesion measure must be defined without ambiguities and must satisfy the representation condition of measurement. This means that once submitted to atomic actions, it must behave accordingly to what is intuitively expected, (for that Weyuker's axioms are good guidelines, though it must be noted that, as Zuse proved, not all of Weyuker's properties can be satisfied by the same complexity measure).

2.2.2 Visualization

Graphical representations have been used extensively as comprehension aids [Stasko *et al.*, 1998]. Software visualization reveals patterns in the entities visualized. Chuah and Eick visualise with their infobug technique files in a glyph-oriented way [Chuah and Eick, 1998]. Infobug is a visualization layout that contain many details. It tries to support the understanding of files in a glyph oriented way. Glyphs are graphical objects representing data through visual parameters. The authors use glyphs for viewing project management data (such as evolution aspects, programming languages used, and errors found in a software component). A piece of software is visualized as if it were an insect, conveying an incredible huge amount of details representing attributes of the piece in a

small space. This technique, so rich in details, is useful for describing a limited number of pieces of software. Unfortunately, when the size of the groups of entities to analyze is large (such as hundreds), this visualization is less useful for the image is cluttered with many details.

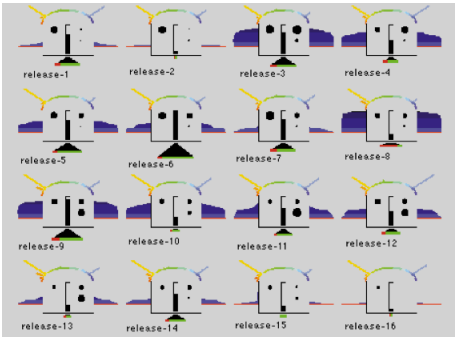

Figure 2.1: Example of graphic view using infobug. Each insect-like glyphs shows one software release.

Polymetric Views [Lanza and Ducasse, 2003] is a technique to visualize software entities such as classes, methods, attributes and packages. It presents views of those software entities in terms of graphs formed by rectangles and links connecting them. This approach presents a serialized way to produce views. The developer associates programatically a software entity to a form, and a connection between entities to links. With this approach, developers have the possibility to map attributes of the entity represented by the rectangle (or other shape) to the size of the rectangle (or shape). For example, if we let a rectangle represent a class, the height of the rectangle represents the amount of methods defined in the class. The more methods defined in the class, the taller the rectangle.

Pinzger *et al.* use a similar technique: a graph with its nodes having the shape of compass-type plots. RelVis, presents compass-type plots (first used by [Sharble and Cohen, 1993]) and draws them into bigger graphs linked by lines representing the strength of the connection [Pinzger *et al.*, 2005]. However, no attempt is made to represent hierarchical structure of the packages.

Recently, Langelier *et al.* proposed a technique [Langelier *et al.*, 2005] for the three-dimensional visualization of large systems. The visualization, based on the tree-map and sunburst representations [Stasko and Zhang, 2000], provides a graphic view of the relevance of packages. With this technique, packages are represented in a flat surface with its size and location organized as dictated by a tree-map view. Classes defined in packages are presented as three-dimensional boxes. This visualization of the system

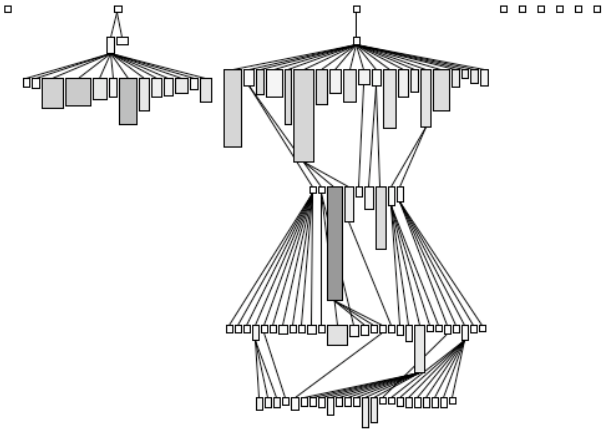

Figure 2.2: Polymetric View showing the inheritance tree of a set of classes. Rectangles represent classes. The height of the boxes represents the number of methods in each class, the width represents the number of attributes, and the color represents the lines of code.

resembles the aspect of a city observed from the sky, where the space between packages represents the streets and the classes represent the buildings. This visualization characterizes packages by the classes defined in them (for instance, the number and height of the buildings), but it does not describe package interaction. For instance, the user can at a first glance discover packages containing classes with a large number of methods, but not which packages contain the classes that are more inherited.

Regarding the exploitation of contextual information, Pintado provided a visualization of the relationships between objects in its Affinity Browser [Pintado and Tsichritzis, 1988]. Figure 2.3 (p.18) shows the Affinity Browser revealing affinity between classes [Pintado, 1995]. It does not characterise packages.

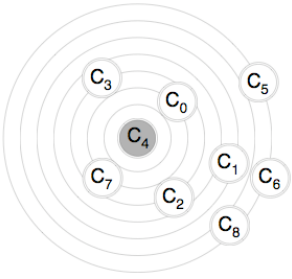

Figure 2.3: Affinity Browser display showing a set of classes.

Bischofberger, proposes Sotograph [Sotograph, 2003], a technique to understand the structure of large systems. Its Sotograph tool navigates software systems at the level of packages containing classes, and subsystems containing packages. Figure 2.4 (p.19) shows two subsystems, each containing four packages, and each package containing four classes. The figure shows also some relationships existing between classes, packages and subsystems.

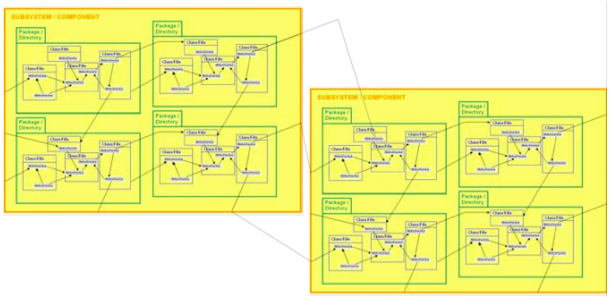

Figure 2.4: Sotograph displaying two subsystems, eight packages, sixteen classes, and some relationships between classes.

Summary:

To manage class locality, developers need visual aids; for example, graphic views characterizing packages and systems. Graphic views must reveal the basic organization of the system, and describe package interaction.

2.2.3 Semi-automatic Analysis Techniques

With large-scale software systems, the complexity increases, for the classes are of large number, *i.e.*, thousands. As such a big amount of classes would be unmanageable without grouping them into packages. These structures form hierarchical organizations that convey also information about the system. Visualizations of large-scale systems must find a way to represent this hierarchical organization.

Langelier *et al.* claim that regarding the understanding and evaluation of software quality, automatic analysis techniques give only limited results applied to problems that are complex and where the factors are poorly understood as is software evolution [Langelier *et al.*, 2005]. They propose as an alternative to combine automatic analysis techniques with human expertise. For that they propose a framework to visualize large-scale software systems. In this framework they represent classes as three dimensional boxes that are included in two dimensional boxes representing the packages containing the

classes. The distribution of classes in packages does not have special semantics. On the contrary, the two dimensional boxes representing packages are organized to represent the hierarchical organization of the packages. To this regard they use two layout techniques: *Treemap* [Johnson and Russo, 1991] and *Sunburst* [Stasko and Zhang, 2000].

Tzerpos and Holt present an hybrid process for recovering the architecture of the software [Tzerpos and Holt, 1996]. In their approach, the authors cluster files into subsystems and group also these subsystems to create a diagram of the architectural structure. The grouping of files needs a property to guide the clustering algorithm. The property in this case was that the files belonged to the same group if they had the same *covering* file. A file f1 *covers* another f2 if they are in the same subsystem and all the calls from files outside the subsystem are calls to f1, which can be considered the entry point of the subsystem. This property is based on the assumption that subsystems in the particular case study of the presented work usually have a single point of entry. The approach had a drawback, however, namely files that could not automatically placed in a subsystem using the covering property. To solve this problem, the authors consulted the developers and used their knowledge of the system (they call it live information) to manipulate the diagram representing the system visualization.

Architecture Recovery and program Comprehension Architecture Recovery is a subject that can exploit any of the techniques in the aforementioned topics to understand the big elements of the system and their interaction. This subject usually combines several techniques to obtain a description of the elements conforming the architecture of the system, and improve its understanding.

In the software clustering area, Anquetil proposes Modularization Quality (MQ). Together with the tool Bunch [Mancoridis *et al.*, 1999], MQ uses the dependencies between modules of two distinct subsystems and the ones between the modules of the same subsystem (in that differs from ours) to determine the cohesion of clusters and to reward the creation of highly cohesive ones.

Schwanke's information sharing heuristic [Schwanke, 1991] *"If two procedures use several of the same unit-names, they are likely to be sharing significant design information"*, used to compute a similarity metric between two procedures, is analogous to the package use heuristic that we use to compute IOC. The innovation of our approach regarding Schwanke's similarity metric is that IOC captures the extent to which the functionality *exposed* by a package is related, while Schwanke uses the interconnections where a package is client as well as the ones where it is provider.

Finally, Mancoridis uses clustering analysis to produce high-level system organizations of code [Mancoridis and Mitchell, 1998].

Visualization techniques complement clustering [Jain *et al.*, 1999] to recover the architecture of the system. A substantial amount of work has exploited this combination to understand the system [Schwanke, 1991] [Schwanke *et al.*, 1989; Schwanke, 1991], [Koschke, 2000], Clustering has proven to be useful to understand the code and to organize it at high-level [Mancoridis and Mitchell, 1998], [Mancoridis *et al.*, 1999], [Tzerpos and Holt, 2000], [Tzerpos and Holt, 1999]. However, to our best understanding, there is no aid to comprehend forces pulling a class towards a package and forces pulling it towards a different package resulting of the use of the class. This is challenging problem because packages offer different perspectives, and because maximizing locality in a package must not result in destroying locality in the whole system.

Summary:

An approach to manage locality must automatically detect organizational units that follow design flaws. But the concept of locality can be applied to entities of many levels, classes in packages being only one level of granularity. The approach must, therefore, apply to different levels of granularity (method, class, package, subsystem, and system.

2.2.4 Offering New Language Constructs

The problem of managing locality is dependant of characteristics of the programming language such as the ability of the programming language to enforce private and public types and to provide support for contracts. For example Ada allows the programmer to define packages with interfaces where everything that is declared inside the package and not in the interface is hidden from the outside. One way of facilitate management of class locality is to modify the programming language by creating new constructs that make package interfaces more flexible. Work has been done extending programming languages by adding constructs that control the visibility of class extensions [Mezini and Ostermann, 2002; Zenger, 2002].

Bergel introduced a simple calculus for modules and a set of operators expressing encapsulation policies, composition rules, and extensibility mechanisms [Bergel, 2005]. In that work, the author presents a formalism to express semantics of different module systems which allows the author to make a taxonomy of different module systems. His work provides the basis for expressing various packaging mechanisms using a common foundation. In the same work, the author analyses the problem of unanticipated changes through class extensions. Class extensions are a way to incrementally modify existing classes alternative to subclassing. As a solution to the class extensibility problem, the author provides a module system for object-oriented languages with method

addition and replacement. He introduces "classboxes": changes made by a classbox are only visible inside the classbox or in classboxes that import it.

Adding constructs to the language increases complexity and make systems rigid.

2.3 Summary

Organizing classes in large object-oriented systems in such a way that developers can at a later stage easily find classes, or class locality, is an old problem. A substantial body of work has tried to solve it from different perspectives: measures, clustering, visualization, semi-automatic architectural recovery and program comprehension techniques, forward engineering (extending the set of constructs of the language), and design principles. In this chapter we have established current limits of the state-of-art regarding the problem of class locality. Our conclusion after analysis of state of art is that certain issues are still open. We have identified a set of interesting open problems that we identify and address in this thesis. Below, we present these issues together with different open requirements of class locality analysis:

Package characterization. A class is too small as a unit of understanding of large systems. Thus, there is a need to form groupings of classes, for example as packages, to support understanding of a system at a coarse level, and to characterize packages and the role that they play in the system.

Recognition of package perspectives. When considering a package in isolation it may be marked as non cohesive. However, if we consider the same package in the context of how it is used by other parts of the system, it then may prove to be cohesive. This circumstance indicates the need to observe and characterize packages from different perspectives.

Improved detection of cohesion. When classes conceptually related are defined in the same organizational unit, developers find them efficiently. Approaches capturing locality must detect classes that belong together, even when there is no coupling between them.

Combination of package properties. The approach should allow us to reason about multiple properties of packages, and to combine them. For example, it should support techniques to detect automatically key packages having a combination of structural and non-structural properties. When reverse engineering large class-based object-oriented systems, the amount of packages to analyze make it difficult to spot packages that are key in the design of the system, or that are flawed.

Global graphic view of the system. Before a reverse engineering effort, developers need to know where to start. A graphic view, representing the big picture of the system, reveals the organization of the system. Moreover, graphic views complement automatic analysis techniques to understand and evaluate software quality. In particular, where there are poorly understood factors in the system analysis, graphic views reveal patterns in the interaction between packages, and patterns in the organization of classes into packages.

Application to different levels of granularity. Developers need time to apply and master a new technique before gaining understanding. Therefore, it is desirable to use the same technique when analyzing locality at different granularity levels. An approach to manage locality should provide the basis to capture it at method level (methods organized in classes), class level (classes organized in packages), and package level (packages organized in subsystems).

In Chapter 3 (p.25) we present Contextual Information, our meta-model to manage locality and to characterize packages. We validate out model by using it in various analysis, which we present in Chapter 4 (p.39), Chapter 5 (p.61) and Chapter 6 (p.83).

Chapter 3

Contextual Information: Modeling What a Package Is

The previous chapter presented problem on managing the organization of classes into packages. In this dissertation, we solve this problem by observing the way packages are used. In contrast to traditional approaches based on structural properties, we observe the interaction between packages, and use this information to manage the organization of classes. This way of thinking will allow us to:

— Define a measure for locality.

— Implement a technique to find misplaced classes.

— Build visualizations depicting locality in every package of the system.

— Characterize package from a wider perspective.

— Characterize subsystems based on packages interactions.

This chapter presents the underlying model which is the basis for achieving the aforementioned tasks.

3.1 Introduction

With existing class-based languages, the notions of class and package present some controversy. On the one side, classes are the centre of class-based languages. They

are descriptions of objects, for a class describes the structure of all the objects generated from it [Abadi and Cardelli, 1996]. On the other side, as the number of classes increases, flat models present the disadvantages of non-structured systems. Packages appear then as middle-level structures of organization [Fowler, 1997]. In the UML, a package is *"a general-purpose mechanism for organising elements into groups"* [Booch *et al.*, 1998]. The concept of a package is useful in systems composed of many classes, for packages group classes together for ease of use, maintenance and reusability [Quatrani, 1998]. By grouping classes into packages, developers can look at the higher level of the model. Modularization at package level facilitates understanding by dividing the application into chunks containing classes conceptually related that are easier to manipulate. Moreover, organization into packages facilitates distributing developing to several teams, and reuse of software pieces through imports. As a result, packages are an important mechanism for dealing with scale.

Packages offer different perspectives to developers coming from different context. When the system evolves, maintainers modify it according to particular perspectives (each reflecting maintainer's convenience from his or her perspective). As a consequence, the organization of classes into packages degrades as the system evolves. Original packages lose their goal of containing only conceptually related classes. The question is then how to capture the essence and role of the package during maintenance of the system, when additions and removals of classes deviated the package from its original design. We manage program modularization by analyzing packages in their interaction with their context. Our approach relies therefore on packages, classes and dependencies. In the following sections, we define our solution based on these elements and the interpretation of contextual information.

3.2 Grounds of the Model

Our source model consists of classes, packages, and dependencies. We characterize packages from their structure and their interaction with the context. A package's interaction is based on the dependencies between classes in the package and classes defined in other packages, the Contextual Information of the package. Our model is simple. We perform static analysis to reify interaction between objects to detect explicit dependencies between classes. There exists a wide spectrum of dependencies between classes; for our analysis, we focus on an essential subset:

1. *Inheritance*: a class is a subclass of another. A subclass inherits behaviour and state froms its parent. (inherits dependencies).

2. *State*: a class may directly access state field inherited from its ancestors. (accesses

dependencies).

3. *Class Reference*: a class makes an explicit (*i.e.,* static) reference to another for example by instantiating the class (references dependencies). We limit our scope to consider static relationships and do not include run-time interactions.

4. *Message sends*: An object requests another object to carry out an operation by sending a message to it (sends dependencies).

Dependencies are directed. Direction is crucial because it makes explicit which class is a client and which one is a provider of services. This information constitutes the basis to capture stability and impact of change of the connected classes.

Dependencies are non-transitive representing that one package insulates its clients from the effect of changes performed in its providers.

Each dependency represent an interaction between two classes; whereas classes are grouped into packages. But what is a package? D'Souza [D'Souza and Wills, 1999] defines a package as *"A named container for a unit of development work. All development artifacts (including types, classes, compiled code, refinements, diagrams, documentation, change requests, code patches, architectural rules and patterns, tests, and other packages) are in some package. A package is treated as a unit for versioning, configuration management, reuse, dependency tracking, and other purposes. It also provides a scope for unique names of its elements."*. The variety of elements included and the policies applied to them according to the programming language and environment originate many notions of a package. The definitions vary according to characteristics given to packages such as the mechanism of visibility that they contain, or allowance for class extensions. We abstract from those language-specific characteristics and focus on kernel characteristics that are relevant for the logical view of the model.

Bergel [Bergel, 2005] introduced a calculus for modules. In the formalism presented by the author, a module is an abstraction over environments; environments being a set of mappings. A module is represented as a function that takes an environment and returns an environment. In that work, modules have different properties such as extensibility and merging. Modules also contain class definitions. The author's simple calculus can be utilized to express semantics of different module systems. In particular it can express semantics for module systems for class-based object-oriented programming. In the model that we propose packages are modules as defined by Bergel. Taking therefore packages as container of class definitions, we define a package as follows:

Definition 1 *A package is a collection of class definitions.*

An important characteristic of packages is that a package can *import* another package and extend it. Classes defined in the importing package know, and may therefore send messages to, or *depend* on classes defined in the imported package. This interaction

between classes defined in different packages is represented in our model by dependencies that go beyond package boundaries, determining dependencies between packages. A package therefore exposes any class defined in it. In terms of Bergel's module system, in our model there is no package encapsulation. Figure 3.1 (p.28) depicts this situation.

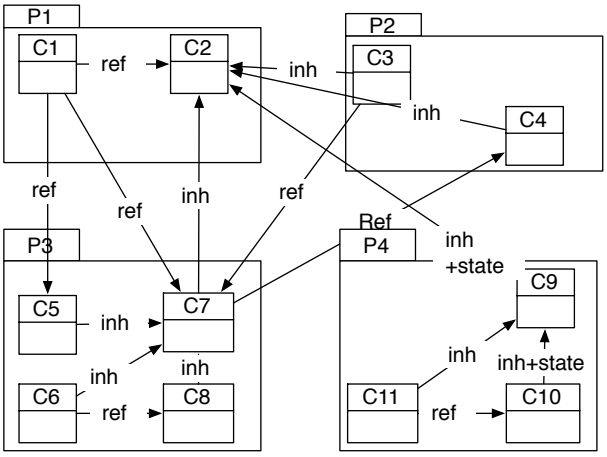

Figure 3.1: Example of 4 packages, 10 classes, and different types of dependencies.

As with classes, the dependency direction between packages determines client-provider relationships. A class defined in package P with at least one class being client of any class defined in package Q, makes P a client of Q. It is the same in the opposite direction: a class defined in package P with at least one class being provider of any class defined in package Q, makes P a provider of Q. We say that P *consumes* or *provides* services respectively.

Packages usually contain other packages making it possible to decompose models hierarchically. Hierarchical organisation reveals important insights about the system such as which parts of the system impact the most if they change, or the system's overall structure. As a consequence, our model supports packages containing packages.

Definition 2 *A subsystem is a collection of packages.*

Once we have established definitions of packages and subsystems, we introduce our discussion of package context. The term context has a wide range of tacit understanding according to the field where it is applied. For instance, a well known definition of context was proposed by Dey and Abowd [Dey and Abowd,]. They define context as *"any information that can be used to characterise the situation of an entity"*. In other words, if something *can* be used to characterise a given entity, then that something *is* context.

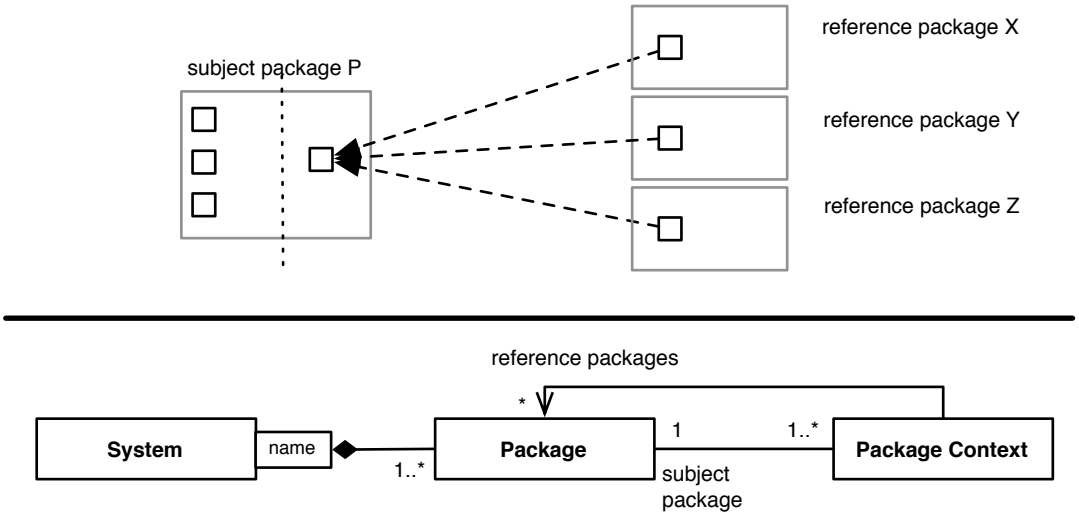

Figure 3.2: Relationship between the System, Package and Package Context.

This definition is useful in the field of context-aware applications, where it comes from, but not for our approach. The problem is that it relies on the ability of something to characterise an entity, but in our approach we do not know a priori if the context will characterise them. However, Schilit *et al.* claim that the important aspects of context are where you are, who you are with, and what resources are nearby [Schilit *et al.*, 1994] . This is closer to our definition:

Definition 3 *Package context of a package P is the collection of its clients and providers, together with the collection of dependencies between classes defined in P and clients or providers of P.*

Figure 3.2 (p.29) shows the core of our meta-model displaying entities System, Package, and Package Context. These entities represent the subject package's Contextual Information displayed above.

3.3 The Model

Our source model consists of classes, packages, and dependencies. To express the cohesion measures unambiguously we provide the following set-theoretic formalism.

An object-oriented system consists of a set of classes, $\mathcal{C}$, where A, B, C range over

classes.

$$A, B, C \in \mathcal{C}$$

Let $\mathcal{P}$ be some partitioning of $\mathcal{C}$, where P, Q, R range over partitions, or packages.

$$P, Q, R \in \mathcal{P}$$

There are dependencies between classes. Each dependency is of kind inherits, accesses, references, or sends.

$$\mathsf{inherits}, \mathsf{accesses}, \mathsf{references}, \mathsf{sends} \subseteq \mathcal{C} \times \mathcal{C}$$

Each dependency determines a client and provider

$$\mathsf{depends} \subseteq \mathcal{C} \times \mathcal{C}$$

$$\mathsf{depends} = \mathsf{inherits} \cup \mathsf{accesses} \cup \mathsf{references} \cup \mathsf{sends}$$

The *clients* of a class are the classes that depend on it:

$$\mathsf{clients}(C) = \{A \in \mathcal{C} \mid A \text{ depends } C\}$$

The *providers* of a class are the classes on which it depends:

$$\mathsf{providers}(C) = \{A \in \mathcal{C} \mid C \text{ depends } A\}$$

There are package dependencies. A package P may contain classes that have clients in other packages. These classes constitute the interface of P.

$$\mathsf{interface}(P) = \{C \in P \mid \mathsf{clients}(C) - P \neq \emptyset\}$$

The classes of P that do not belong to the interface of P are internals.

$$\mathsf{internals}(P) = P - \mathsf{interface}(P)$$

A package P may have clients, packages whose classes depend on classes of P.

$$\mathsf{clients}(P) = \{Q \in \mathcal{P} \mid \exists B \in Q, \exists C \in \mathsf{interface}(P), B \in \mathsf{clients}(C)\}$$

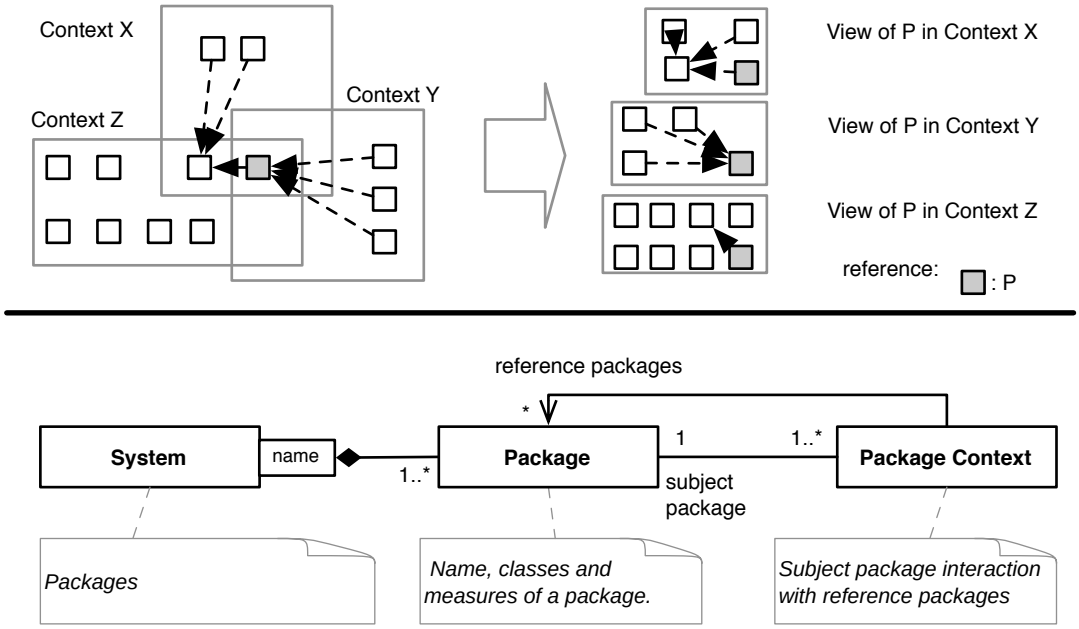

Figure 3.3: Relationship between the System, Package and Contextual Information.

3.4 Package Context as First-Class Entity

We model Package Context as a first-class entity of our model. Package Context reveals properties of the package and of the system that would remain obscure if only the internal elements of the package were considered. A package P can be isolated (*i.e.*, not referenced at all, P depending on no package itself) if observed in the context of system X and be a highly referenced piece of system Y when observed under another context. Besides exposing complexity of interaction between P and other packages, analysing Package Context as a separated phenomenon reveals P's behaviour and role in the system. In particular, it reveals *views* of P depending on the perspective in which P is observed. A package considered non-cohesive in isolation may prove to be cohesive when used by clients. A package can be observed from different perspectives. A package together with a perspective define a role that the package play. All the roles that a package plays together with its internal structure characterize the package. This information reveals package's properties, such as conceptual bounds between classes defined in it. All in all, it characterizes part or the whole system.

Figure 3.3 (p.31) shows the relationships between Contextual Information, package and

different views of a package P. We observe several packages interacting. Those packages are grouped in contexts representing views. Consider three teams working in the same big system. Context X represents the are of interest of the user-interface team, Context Y that of the finance team, and Context Z that of the database team. Every team is concerned with package P, but under different perspectives. P internal attributes, such as the number of classes defined in P, are a small part of P's characterization. However, the context in which P operates reveals its importance. For instance, P being heavily accessed in context Y describes it as a critical package. However, observed in context Z and having no clients in Z, we would be inclined to mark it as non-important. We conclude that P and its context have properties (namely, the interaction package-context) and therefore can be treated as a first class entity. Enriching package characterization with Contextual Information , allow us to determine a package's cohesion, to find misplaced classes, to capture package's characteristics such as stability, and to determine characteristics of parts of systems such as impact of changes.

3.4.1 Determining Contextual Cohesion

Complex systems are decomposed into cohesive packages with the goal of limiting the scope of changes: if our packages are cohesive, we hope that changes will be limited to the packages responsible for the features we are changing, or at worst the packages that are immediate clients of those features. But how should we measure cohesion? Traditional cohesion metrics focus on the *explicit* dependencies and interactions between the classes *within* the package under study. A package, however, may be conceptually cohesive even though its classes exhibit no explicit dependencies.

Figure 3.4 (p.33) shows the two packages whose classes are apparently unrelated. Observers of $P1$ and $P2$ would mark both packages as not cohesive. However, considering Contextual Information we see that classes of $P1$ have different clients, whereas classes in $P2$ have the same clients, which could be an indicator of some cohesion in $P2$. In contrast to $P1$ and $P2$, classes in $P3$ are highly bound.

We propose a *contextual measure* that assesses the cohesion of a package based on the degree to which its classes are *used together by common clients*. The measure is computed considering any of the following dependency types: inherits, accesses, references, and sends.

We apply these metrics to various case studies, and contrast the degree of cohesion detected with that of traditional cohesion metrics. In particular, we note that object-oriented frameworks may appear not to be cohesive with traditional metrics, whereas our contextual metrics expose the implicit cohesion that results from the framework's clients.

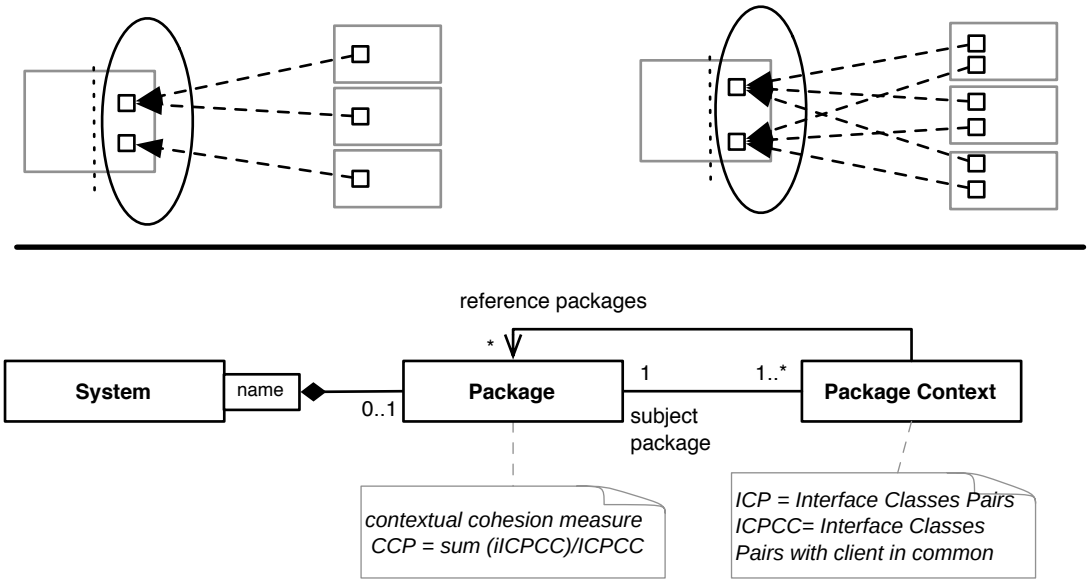

Figure 3.4: Computing cohesion from Contextual Information.

3.4.2 Finding Misplaced Classes

Part of the problem of organizing class definitions in packages through the system's evolution is to capture the degree to which a class belongs to a package. As a consequence, classes can be misplaced, leading to duplicated code and ripple effects with minor changes affecting multiple packages.

We claim that contextual information is the key to re-architecture a system. Exploiting contextual information, we propose a technique to detect misplaced classes by analyzing how client packages access the classes of a given provider package. We use the contextual measure mentioned in the previous section to guide a simulated annealing algorithm to obtain optimal placements of classes in packages. The criterion is that classes reused by common clients must be defined in the same package. The result is the identification of classes that are candidates for relocation.

Figure 3.5 (p.34) shows an example of two resulting patterns detected with this technique. Representing packages as a rectangle, Figure 3.5 (p.34) depicts two packages, S and T with 18 and 7 classes defined in them (the smaller rectangles). Classes are colored as the package to which they are pulled through coupling dependencies. In other words, the small clear rectangle in S tells us that by moving a class from S to T we improve locality in both packages. However, not every class with clients in other

packages must be re-located, and to reflect that issue Figure 3.5 (p.34) show the degree of happiness of each class. In the figure, rectangles representing classes have different height. Class-representing rectangle height describes the degree of coupling between the subject class and other classes in the subject package. In particular, the height of it indicates the amount of clients that the class has in the package where it is defined. For example, in Figure 3.5 (p.34) S shows a class candidate to be moved to T with no clients in S; whereas T presents two classes candidate to be moved to S, but that have clients in T. The larger the rectangle is, the more clients a class has in the package where it is defined, which we refer to as the *"happiness"* of a class.

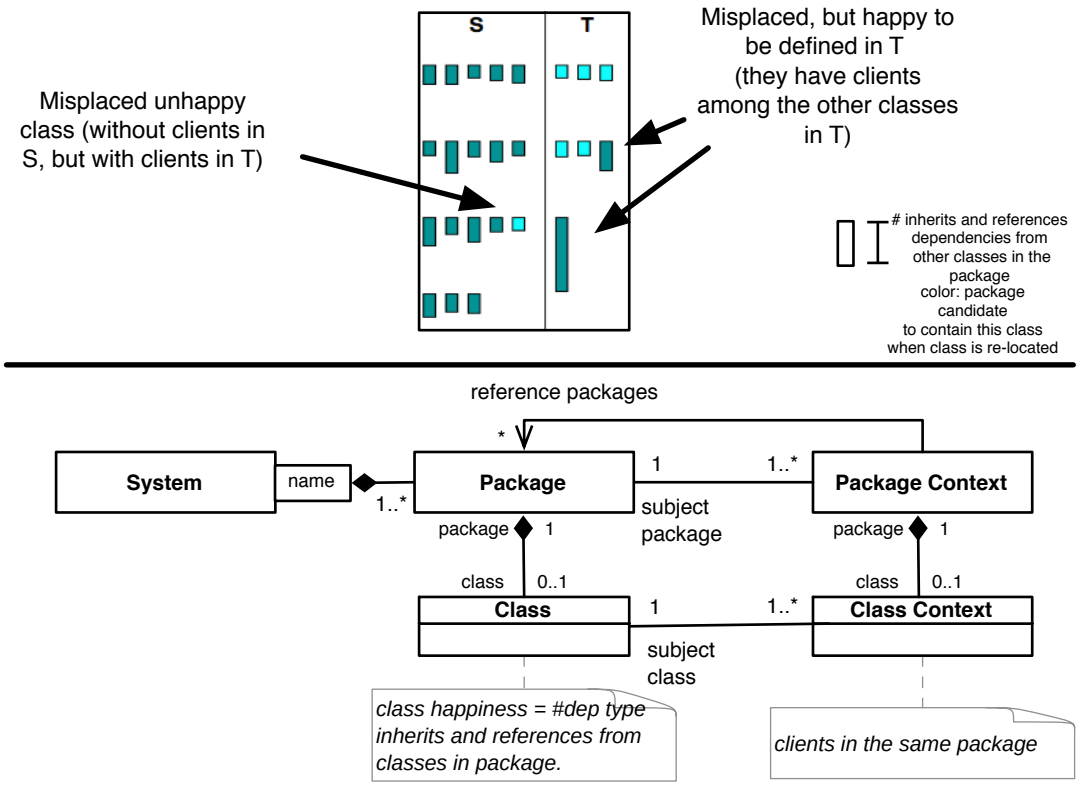

Figure 3.5: Finding misplaced classes with Contextual Information.

We have mentioned the notion of moving a class from a package to another. Our meta-model allows us to measure an application, move a class definition to a different package, and recalculate locality after the relocation without performing any modification to

the actual code. With this technique we can analyse the modularization that the system would have *if* specific class relocations were applied to it. We need an entity to represent the system representation with its moves. We map this entity to our system entity. After each re-location Package Context and Class Context are updated accordingly.

3.4.3 Capturing the Role of Packages in Systems

As discussed in Section 2.2.1 (p.13) inter-module measures quantify the interaction between modules. The benefit of those measures is that they give an idea of how complex that interaction is. The shortcoming of inter-module measures is that they reveal if a package is highly coupled, but they overlook the essence of a package. Intra-module measures, on the other hand, may indicate a package's complexity, but once again they disregard the different perspectives of observing a package. It is still difficult to quickly grasp the structure of a package, and to understand how a package interacts with the rest of the system. In particular, we seek the answer to the following questions:

— What is the importance of a package in terms of its intrinsic properties such as the number of classes it contains and its efferent and afferent relationships? How many clients rely on it?

— Does the package use several other packages or is it more self-contained?

— What is the impact of changes in the relationships between packages?

— Can we identify patterns or repeating package characteristics?

— How is a package structured: does it only extend other packages via inheritance, or does it define itself some complex hierarchies? When classes are subclassing other classes what are exactly the relationships that link them (state, behavior)?

We suggest a visualization technique based on direct measures derived from Contextual Information that help us to answer the questions above. Figure 3.6 (p.36) shows the relationship between direct contextual measures and package characterization through butterfly visualization. The measures example values refer to the situation depicted in Figure 3.1 (p.28). Section 5 (p.61) explains our approach and how its application reveals insights about packages and systems.

Butterfly blueprints take into account the context in which the package is inserted, and characterize a package on the basis of its context's features.

Contextual Information reveal the application's composition by identifying packages capable of producing much impact in the system, packages that are stable and unstable, and patterns in package interaction.

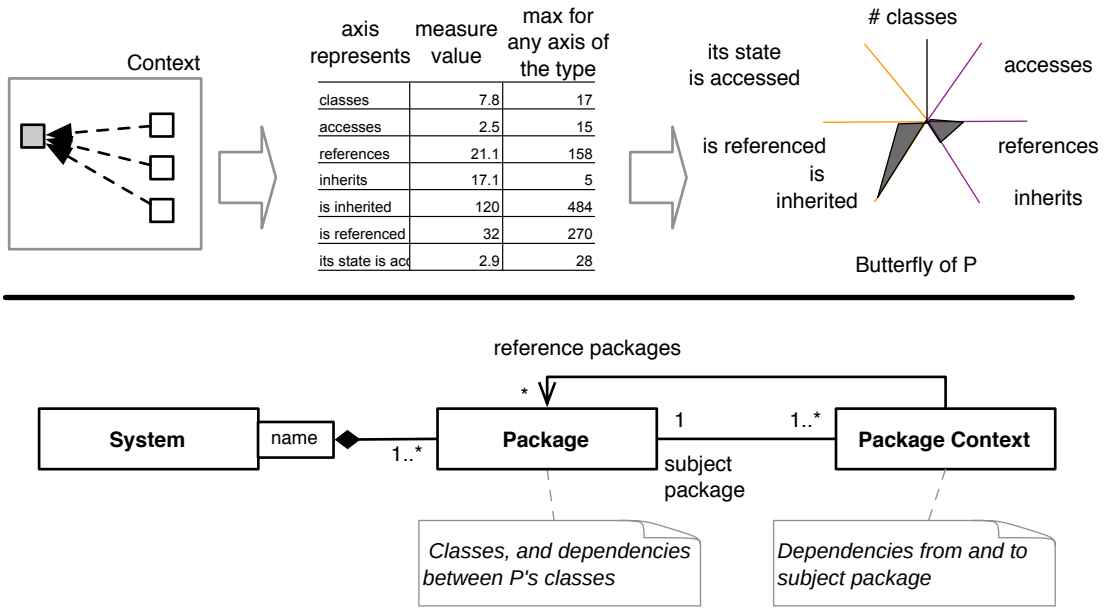

Figure 3.6: Characterizing Packages through direct measures derived from Contextual Information.

3.5 Summary

Observing the context in which a package P operates exposes P's characteristics that remain hidden taking P's internal attributes into account. For instance, a package with low cohesion considered in isolation might prove to be cohesive when used by clients. If the context change, so does the interaction of P with the context and P's characterization.

Thinking about packages together with their context puts them under a new light. Answers to questions such as where do I put this class? (locality) depend not only on the internals of P, but also on the P's clients. In particular, thinking in terms of Contextual Information will allow us to create measures that can be used to characterise cohesion in packages (Section 4 (p.39)), detect package that are core to the application (Section 4 (p.39)), visualize systems by capturing package interaction (Section 5 (p.61)), and detect misplaced classes (Section 6 (p.83)).

In this chapter we have presented the underlying model which is the basis for achieving the aforementioned tasks. We have already presented applications of the exploitation of Contextual Information in reverse engineering; whereas the next chapters discuss those

applications in detail.

Chapter 4

Understanding Packages from the Outside

In the previous chapter, we claimed that by adding Contextual Information, cohesion and coupling measurements reveal situations that traditional measurements cannot detect. In this chapter we define a new measure that overcomes these shortcomings. By acknowledging the way a package is used, the new measure detects cohesion even when there exist few or no explicit dependencies among classes inside the package and therefore, it reveals cohesion where traditional measurements fail to detect it. We show a case in point where framework packages designed to be hooked by client applications are marked as not-cohesive by traditional measurements, whereas measurements exploiting Contextual Information capture the cohesiveness of their classes.

As systems grow, maintainers need automatic identification of packages such as those being central to the application and complex. Current approaches for automatic detection take into account either coupling or only internal attributes of the package. Based on our approach of exploiting Contextual Information, this chapter presents novel detection strategies for packages. The novel detection strategies detect packages with the aforementioned characteristics. We apply our approach to a large case study and show detected packages exposing design issues.

The following sections present system properties revealed through exploitation of Contextual Information. We show Contextual Information revealing package's cohesion, detecting automatically packages with specific design characteristics and revealing a module's different perspectives according to the use that different clients make of it.

4.1 Contextual Package Cohesion (CPC)

Complex systems are decomposed into cohesive packages with the goal of limiting the scope of changes: if our packages are cohesive, we hope that changes will be limited to few packages, and conceptually related ones. Ideally, packages perform only one kind of task, and all their classes are related to the accomplishment of this task.

To achieve this, developers need guidelines for evaluating the relationship among the classes in the package. One of the most useful guidelines for evaluating this relationship in development and maintenance is cohesion, described by Fenton and Pfleeger [Fenton and Pfleeger, 1996] as *"the extent to which its individual components are needed to perform the same task."*

But how to measure cohesion? Traditional cohesion metrics focus on the interactions between the classes *within* the package under study. A package, however, may be conceptually cohesive even though its classes exhibit no explicit dependencies. Cohesion does not have to be represented in explicit relations between the classes of the package, an often-used basis for cohesion measurements [Berard, 1993]. Cohesion may be represented in the "purpose" of the package [Mišić, 2001]. Our approach represents the way a package is used based on dependencies between classes.

Given a set $\mathcal{D}$ of explicit dependencies between classes defined in packages, CPC counts how many classes in the package are used by the same client. In Figure 4.1 (p.41) (a) all the classes are accessed by classes in the two client packages. But in Figure 4.1 (p.41) (b) class H seems not to belong to the package $P1$.

We define cohesion as a property of a package measuring the degree to which its interface classes are used together. Exploiting information regarding the interaction between a package and the rest of the system, we work under the heuristic that two classes provide related functionality if they have clients in common. If classes defined in the same package have the same clients, then they are used together, and make the package cohesive. This is related to Bunge's notion of similarity between two objects [Bunge, 1974]. Bunge's made reference to real objects in the concrete world. The notion is not circumscribed to programming languages. As we mentioned in Section 2 (p.7), this idea formed the basis of Chidamber and Kemerer's LCOM measure for class cohesion [Chidamber and Kemerer, 1991; Henderson-Sellers, 1996]. We define Contextual Package Cohesion of a package as follows:

Definition 6 *Contextual Package Cohesion (CPC) of a package is the sum of pairs of classes from the interface of a package having a common client package (f), divided by the number of pairs that can be formed with all classes in the interface.*

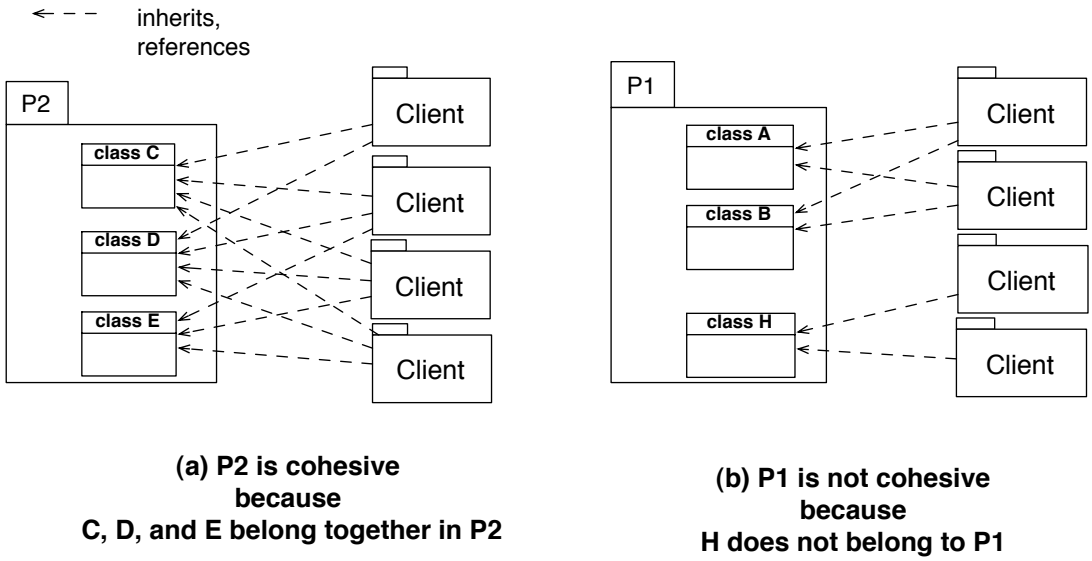

Figure 4.1: Deriving package's cohesion from the context.

$$\mathsf{cpc} = \sum_{a,b \in I} \frac{\mathsf{f}(a,b)}{\#Pairs}$$

Where

$$\begin{array}{lcl} I & = & \mathsf{interface}(P) \\ \#Pairs & = & \frac{|I| \times (|I|-1)}{2} \\ C & = & \mathsf{clients}(a) \cap \mathsf{clients}(b) \\ \mathsf{f}(a,b) & = & \begin{cases} 1, & \textit{if } C \neq \emptyset \\ 0, & \textit{otherwise} \end{cases} \end{array}$$

4.1.1 Discussion of cpc

To investigate cpc, we discuss its behavior regarding properties of two well know conceptual frameworks, Weyuker's and Briand's, which were already discussed in Section 2.2.1 (p.14). It is interesting to analyze cpc under the axioms presented by those frameworks because that analysis explains the behavior of cpc and reflects its limitations under determined conditions. In particular, we observe the lack of the expres-

siveness of cpc when the subject package has no client, or when systems are poorly structured.

The proposed set of Weyuker's axioms applies originally to programs and complexity measures. We have discussed in Section 2.2.1 (p.14) the application of Weyuker's axioms to object-oriented programming and concluded that analysis of a measure under those axioms was useful also for measures designed for object-oriented programming. We also explained that cohesion was a complexity measure. It remains to be shown that cpc is a complexity measure. In the definition of cpc above, we observe that cpc indicates complexity of interaction between a package P and its context. High values of cpc reveal that classes in P have the same clients. We believe that such uniformity in the use of classes facilitates understanding of the program. As observed in Section 2.2.1 (p.14), the property required for complexity measures was to describe how complex it is to understand a program. cpc satisfy that property and therefore cpc is a complexity measure, which is why it makes sense to analyze cpc using Weyuker's axioms.

The following sections discuss cpc in terms of the aforementioned conceptual frameworks.

Behavior of cpc under Weyuker's Properties for Complexity Measures

As we noted in Section 2.2.1 (p.14), it is interesting to observe how a measure satisfies (or not) Weyuker's set of properties because it reveals the measure's behavior. To analyze cpc we apply Weyuker's properties to packages and their context. Let us observe how cpc behaves with respect to these properties.

Property 1 *"A measure should not rank every program equally complex."* cpc satisfies this property. Figure 4.1 (p.41) shows two packages with different values of cpc. In this case cpc(P1)=0.66, and cpc(P2)=1.

Property 2 *"Let c be a non-negative number, then there are only finitely many programs of complexity c."* In the case of cpc, two packages with different load of accesses can have the same cpc value. We learn that cpc does not does not distinguish between very little used and heavily used packages, provided that the accesses to classes from client packages is proportional, and therefore it does not satisfy this property.

Property 3 *"There are distinct programs P and Q having the same complexity measure value."* Figure 4.2 (p.44) shows an example of distinct packages P and Q, where cpc(P)=1 and cpc(Q)=1/3. Therefore cpc satisfies this property.

Property 4 *"Even though two programs compute the same function, it is the details of the implementation that determine the program's complexity."* Our approach to capture cohesion abstracts implementation details that do not belong to a package's interface. From this point of view, implementation details do not determine complexity, and therefore cpc does not satisfy this property.

Regarded complexity as we do, determined by the classes in the interface of a package P and P's context, two packages computing the same function may have different values of cpc. And therefore cpc does not satisfy this property.

Indeed, two versions of the *same* library accessed by similar clients may offer opposite extremes in complexity. Consider the case of a framework with two versions F and G. Framework version F has a dedicated package as interface to the client. F computes the same function as G, but exposes its responsibilities in a single package (F). Version G does not have such interface package. Its clients must extend and interact with many packages of G, lacking of a single point concentrating interaction with the framework. Packages in this client are forced to extract little pieces of functionality here and there. How is then the complexity of F+F's context compared to that of G+G's context? Maybe F requires more coupling between its classes, but maintenance effort and understandability of F+ F's context is likely to be lower than those of G+G's context *for developers of the clients*. Regarding developers of F, each time they perform a change they have one point to check for no rippling effects; whereas those of G must remember several points that could be affected (those that clients consume). All in all, when we consider a package *in its context*, the way it is accessed determines its complexity.

Property 5 *"Adding code to a program cannot decrease its complexity."* This property and the next one are mutually incompatible. In object-oriented programming developers usually complain that they do not understand the code because they see too little of it. A complexity measure based on size would satisfy this property, but a complexity measure based on comprehensibility would not [Fenton and Pfleeger, 1996]. The final goal of cpc is to capture cohesion as developers see it in their programs. Comprehensibility is a pillar of cpc, and therefore cpc (as well as Mc Cabe's cyclomatic complexity) does not satisfy this property. Figure 4.2 (p.44) a) depicts an example where cpc does not satisfy this property.

Property 6 *"There can be two program bodies of equal complexity which, when separately concatenated to a same third program yield (two larger) programs of different complexity."* Figure 4.2 (p.44) b) shows by example that cpc satisfies this property. Contrasting with property five, which has to do with size, this property is based primarily on comprehensibility, and therefore it is a desirable property for cpc to satisfy.

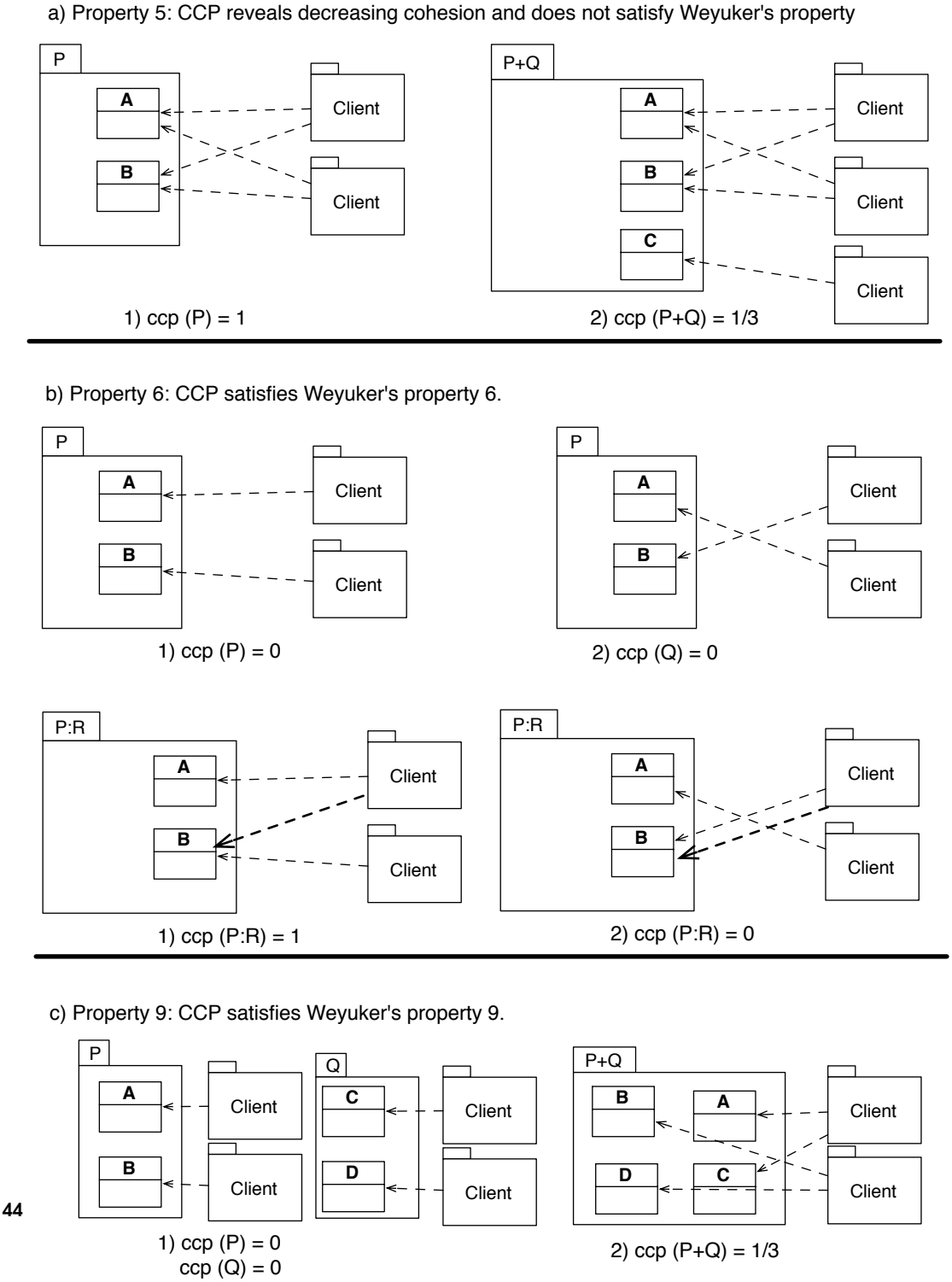

Figure 4.2: Examples of behavior of Contextual Package Cohesion (CPC) regarding Weyuker's properties.

Property 7 *"There are program bodies P and Q such that Q is formed by permuting the order of the statements of P and measured values of P and Q are different."* This property is an example of non-applicability to measures for class-based object-oriented programs. Property seven does not apply to the context of classes and packages in which cpc is defined because in object-oriented programming there is no notion of class ordering, and therefore the condition of *"permuting the order of the statements"* cannot be satisfied.

Property 8 *" If P is a renaming of Q, then P and Q have the same complexity."* Clearly, this property is desirable. cpc, being symmetric (every package has the same complexity that itself, provided it is evaluated in the same context), satisfies this property.

Property 9 *"There exists programs P and Q such that the sum of their complexities is lower than evaluating the complexity of the result of merging P and Q."* Figure 4.2 (p.44) b) shows an example of such programs P and Q. Therefore, cpc satisfies this property.

Behavior of cpc under Briand's *et al.* framework

cpc does not satisfy some cohesion properties specified by Briand *et al.* [Briand *et al.*, 1996]. One of those properties is monotonicity, which states that adding intramodule relationships does not decrease cohesion. The other is the cohesive modules property, which states that cohesion of a module obtained by putting together two unrelated modules is not greater than the maximum cohesion of the two original modules.

Therefore, according to this framework, cpc is not a cohesion measure. However, the reason behind cpc not satisfying the properties mentioned above is that those properties are for cohesion measures based on explicit interconnections between entities in the module. We believe that the case is the same as with Weyuker's axioms, where cohesion is observed only from explicit relationships between elements of the model. The assumption that a cohesion measure has only to do with explicit dependencies is unfounded and, derived from traditional approaches for cohesion, it neglects the potential connection existing between two elements observed from outside the module.

cpc takes a novel, different view, namely that of detecting *reasons* for two classes being together further than the existence of explicit interaction between them. Once again, in essence the problem of cpc not satisfying the properties of monotonicity and cohesive modules is analogous as that of cpc not satisfying Weyuker's properties 4 and 5, namely, that cpc measures classes that are used together. Under this approach, a

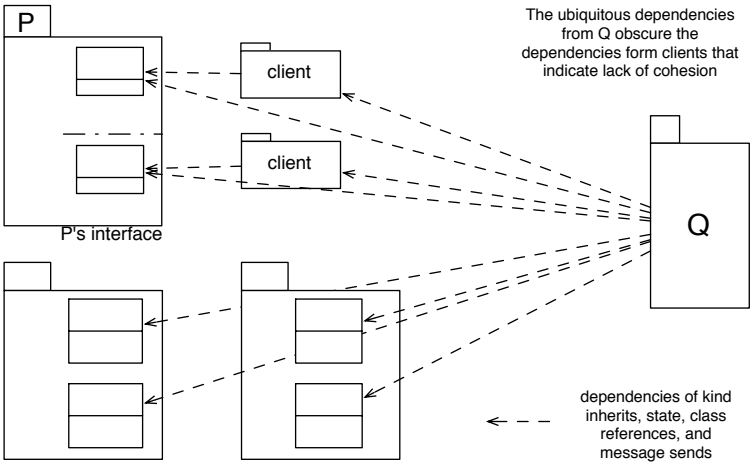

Figure 4.3: Example of limitation of Contextual Package Cohesion (CPC).

client using two classes establishes an implicit link between them: it makes them a pair, regardless of explicit connections between the classes. If CPC is high, there are many pairs of classes having the same clients in the interface of the subject package. We conclude that the aforementioned properties are only useful for discussing CPC.

To give useful values for cohesion, CPC has the limitation of requiring a certain system modularization. Relying on clients of a package P to compute P's cohesion, computing CPC in systems lacking modularization produce useless results. Indeed, as Figure 4.3 (p.46) shows, the presence of a package Q that is client of every class in the system makes cohesive all the packages (with the exception of the Q). One solution to this issue is to filter out such packages as Q before computing CPC.

In Section 2.2.1 (p.14) we discussed the criteria for evaluating cohesion measures. Coming back to **Chapter 2** (p.7) we observe that CPC has a clear definition, and, after having analyzed CPC under Weyuker's and Briand's framework, we observe that once submitted to atomic actions, CPC behaves accordingly to what we would expect. We conclude that under certain restrictions, CPC reveals cohesion where traditional measures do not. Moreover, together with other measures, CPC exposes information that can be useful to perform analysis tasks such as maintenance. For instance, CPC combined to traditional coupling measures reveals packages reflecting design issues (such as packages complex packages that are key to the system and difficult to reuse. In the next step, we will compare the behavior of CPC with that of traditional cohesion measures, *i.e.,* those centred

on internal relationships in a module.

4.1.2 cpc Compared to Traditional Cohesion Measures

As mentioned in the previous section, one advantage of including context to measure cohesion is that it reveals implicit dependencies, namely those derived of being used by the same client. Traditional measures acknowledge only explicit dependencies. By adding information about how the package is used, we make its cohesion relative to the other packages. This reveals implicit dependencies between classes in a package (in particular, in the package's *interface*) which cannot be detected by observing only classes inside the package and their dependencies.

Taking the point of view of how classes in a package are used, package-context interaction determines the degree of difficulty in analysing, maintaining, testing, and modifying software. With values obtained after measuring packages and their context with cpc, we make statements like " classes in package P are used together by clients". The more classes in P have clients in common, the more cohesive P is. Packages should not have some classes accessed by some clients and a disjoint set of classes used by other clients. When this happens, P loses part of the concept that bound its classes together to the eyes of developers. Moreover, it results in an inadvertent coupling between client packages.

We compare cpc with traditional cohesion measures CR and DCO (discussed in **Chapter 2** (p.7)) showing through examples found in actual applications where cpc improves (or not) over traditional measures to analyze maintenance, design, extensibility, and modifiability.

Cohesion Ratio (CR) A package marked as non-cohesive by the measure Cohesion Ratio (CR) may prove to be cohesive when used by its clients. We have found examples showing this in real applications. In particular, we refer to CODECRAWLER, a small application based on two frameworks. **Appendix B** (p.105) provides further description of CODECRAWLER. In Figure 4.4 (p.48) we see two packages of CODECRAWLER, `CCHotDraw` and `CCMoose`, dedicated to connect CODECRAWLER with frameworks HotDraw and Moose. `CCHotDraw` and `CCMoose` might have classes with few dependencies between them. They are, however, cohesive. The binding between its classes being their connection with the framework. As we concentrated our attention on the cohesion of the packages, we observed that `CCMoose` seemed to be *not* cohesive when its cohesion was evaluated using CR (due to the relative lack of dependencies among classes in `CCMoose`). However, `CCMoose` revealed itself to be cohesive when evaluated using cpc. The other package, `CCHotDraw`, had high values of cohesion when evaluated using all

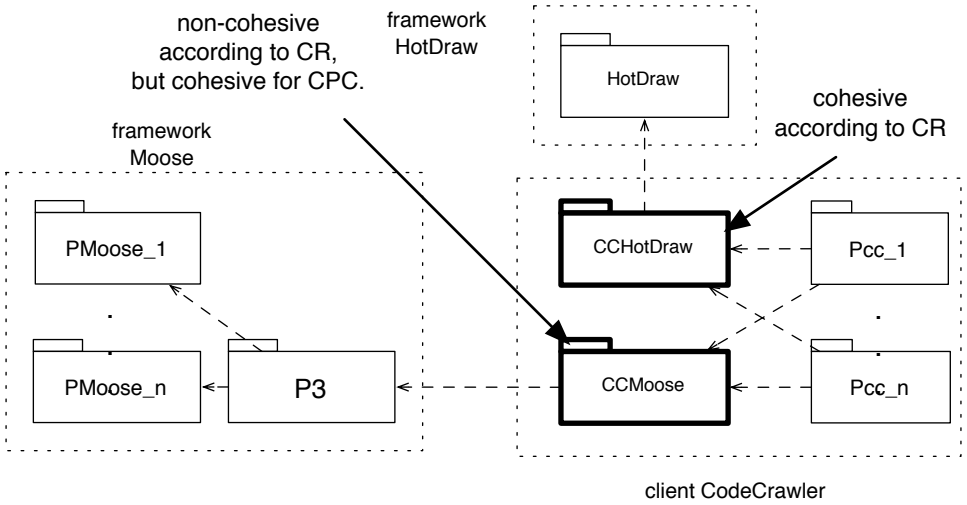

Figure 4.4: Example of client application with packages dedicated to interact with its frameworks.

the measures except CPC. This means that `CCHotDraw` is cohesive when evaluated in isolation, but its classes are accessed by different clients and there are few classes accessed by the same client. This is because the reason behind the existence of `CCHotDraw` is to "abstract the framework" to the rest of CODECRAWLER, as analysis of the code revealed.

To understand CR compared to CPC, let as consider the ratio of internal dependencies in a package. We define it as the number of dependencies between classes defined in a package P divided by the maximum possible number of dependencies between classes defined in P.

$$\mathsf{IDR}(P) = \frac{\mid \{\mathsf{depends} \mid A \in P, B \in P, A \text{ depends } B\} \mid}{\frac{|classes|(|classes|-1)}{2}}$$

Where $\frac{|classes|(|classes|-1)}{2}$ is maximum of number of dependencies among classes defined in P (each class depending on all the rest). The resulting values are positive and may be greater that one if the dependency type analyzed allows more than one dependency between classes. This is the case of references, where, for instance, methods in a client class can reference a provider class many times. The values range from 0 to 1 if inherits dependencies are considered, when there is a maximum of one possible dependency between two classes.

This set up is analogous to that of CR in the sense that it uses explicit dependencies between classes in a package to compute its cohesion, disregarding contextual information. We have found in CODECRAWLER three packages with low values of IDR based on references. Two of these packages obtained high values of cohesion when we computed it with cpc based on references dependencies.

Degree of Cohesion of Objects (DCO) Studies performed on CODECRAWLER showed two examples where DCO gave low values and cpc based on inherits and cpc based on references signaled cohesion (*e.g.,* considering inheritance cpc(`CCBase`)=0.83, and considering explicit referencing cpc`CCBase`0.66) but DCO considered them the third, and the fourth packages less cohesive of CODECRAWLER respectively.

We conclude that under certain conditions (such as packages with clients, and absence of packages accessing systematically most of the classes defined in the system), cpc indicates cohesion when traditional metrics indicate lack of cohesion.

4.2 Detection Strategies Enriched with Contextual Information

Measures are useful to detect design problems. Detection strategies, based on concrete data (*e.g.,* measures), allow us to reason at a more abstract level. A detection strategy is defined by its author as *the quantifiable expression of a rule, by which design fragments that are conformant to that rule can be detected in the source code* [Marinescu, 2002; Marinescu, 2004].

Detection strategies are expressions that combine several *traditional* measures to detect design problems [Raţiu *et al.*, 2004]. The result of applying a detection strategy to a system is a set of packages conforming to the package characteristics specified in the detection strategy. In this way, detection strategies make possible to *quantify* design imprints in the system.

Previous approaches derived detection strategies to detect, among others, god classes [Raţiu, 2003] and classes following a particular evolution pattern such as stable god classes [Gîrba, 2005]. Our contribution is to define detection strategies that include contextual information. In the following, we show novel detection strategies detecting packages having key design imprints such as packages that are complex, core of the system and not cohesive.

Consider the following inter-module measures (presented in **Chapter 2** (p.7)) describing package's coupling:

— fan_in using inherits

— fan_in using references

— fan_out using inherits

— fan_out using references

Close observation reveals that these measures depend on the context where a package operates. More precisely, on the view that a package offers based on the context where it operates. In the following sections we use them together with CPC to build detection strategies aiming to detect complex, reused, and non-cohesive packages in the system.

Complex Package Detection Strategy

Complex Package refers to a package whose classes are highly interrelated or complex themselves. A *Complex Package* carries on a difficult-to-decompose chunk of system intelligence. They are the packages containing classes of high complexity, and highly related by structural or non-structural dependencies.

To detect a *Complex Package* we look for packages whose classes have a certain complexity, or are highly coupled to other classes in the package. The *Complex Package* detection strategy is a rule (see Equation 4.1 (p.50)) capturing this description and based on the following measures:

— Internal Dependencies Ratio (IDR) using only inherits dependency type (**Chapter 3** (p.25), Section 4.1.2 (p.48))

— Internal Dependencies Ratio (IDR) using only references dependency type (**Chapter 3** (p.25), Section 4.1.2 (p.48))

— CR (Cohesion Ratio) (**Chapter 2** (p.7), Section 2 (p.12))

$$ComplexPackage(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall P \in S' \\ (\mathsf{IDRinherits} > value1) \wedge (\mathsf{IDRreferences} > value2) \vee \\ (\mathsf{CohesionRatio} > value3)) \end{array} \right. \quad (4.1)$$

Where $value1$, $value2$ and $value3$ depend on the measure values in each specific system.

Independently of the number of classes defined in a package, the interrelation between

its classes as well as their complexity compromises modularization of this specific piece of software. For this reason a *Complex Package* is difficult to maintain.

Complexity and Size In the definition of the rule to detect *Complex Packages* of Equation 4.1 (p.50) we have decided to not include package size as an element to filter packages. Authors have already questioned the measure of size to describe complexity [Zuse, 1990; Henderson-Sellers, 1996]. The idea behind our decision is that a package containing many classes is not necessarily more difficult to maintain, nor did we find evidence that it makes system maintenance more complex. On the contrary, we have found systems containing large packages that do not compromise the system's maintainability. A case in point are large libraries built by third parties: as long as they provide clear, stable interfaces, their size does not add to the complexity of their clients. Furthermore, including size as an element to filter complex packages has the disadvantage that packages with few, but complex classes are filtered out. We performed experiments adding the constraint of size (*i.e.,* number of classes) as a filter to obtain every *Complex Package*. As a result, packages defining classes known to be key to the system (due to the fact that many depended on them) were filtered out. We believe that a package being central to the application and containing a God Class (God Classes are big and complex classes, and which are known to be a source of maintainability problems [Riel, 1996]) can be complex, even though containing a solitary class. We conclude that size is not an indicator of the complexity of a package in the case of our analysis, and therefore we do not include it in our detection rule, but analyze size separately.

Core Package Detection Strategy

Core Package refers to a package that has many clients. A change (*e.g.,* modification or removal of a class) in a *Core Package* may impact in many parts of the system, meaning that it may produce ripple effects in a large amount of packages. The rule to detect *Core Packages* (see Equation 4.2 (p.51)) captures this description based on the following measures (discussed in **Chapter 2** (p.7), Section 3 (p.14)):

— fan_in using references

— fan_in using inherits

$$CorePackage(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall P \in S' \\ (\text{fan\_ininherits} > value1) \vee (\text{fan\_inreferences} > value2)) \end{array} \right. \quad (4.2)$$

Given a change (*e.g.,* addition, modification or removal of a class) in a package, developers must check a large part of the system for rippling effects. We say that *Core Packages* have big *suspected* impact. In Equation 4.2 (p.51) we observe that incoming dependencies references and incoming dependencies inherits do not participate in the rule. The reason behind is that a package P may have one or few clients that use it heavily. As a consequence, if this detection strategy were included the condition ($\vee$) of large incoming dependencies, P would be marked as provider even with one client. This situation goes against the intuition of this detection strategy, which is to detect packages potentially dangerous because a change in those will affect *many* other parts of the system. Moreover, if this rule would require large incoming dependencies, packages having many clients that access them poorly would be filtered out, once again against the idea of this rule.

Client Package Detection Strategy

In the same sense that there are heavily-used packages (*Core Packages*), there are packages that use many packages, but that are themselves seldom used. This rule (see Equation 4.1 (p.50)) captures this description based on the following measures:

— fan_in using inherits and references

— fan_out using inherits and references (**Chapter 2** (p.7), Section 4 (p.14))

$$ClientPackage(S) = S' \left| \begin{array}{l} S' \subseteq S, \\ \forall P \in S' \\ (\mathsf{fan\_in} < value1) \wedge (\mathsf{fan\_out} > value2) \end{array} \right. \tag{4.3}$$

Because *Client Packages* do not have (or have few) clients, ripple effects of changes produced in *Client Packages* do not propagate. As a consequence, it is desirable to have this kind of package as containers of implementation details, sometimes depending on the platform where the application operates. This is why *Client Packages* usually belong to upper layers of the system's architecture (for instance containing code covering other packages, such as tests).

Unstable Package Detection Strategy

Some packages require services from many others; whereas others do not. A package is instable if its number of provider packages is high. This rule is defined in analogous way

to the previous ones capturing this description based on the following measures:

— fan_out using inherits

— fan_out using references

This differs from Martin's instability [Martin, 1997], in that their instability is based on counting the number of dependencies towards clients (*e.g.,* Outgoing dependencies); where this rule is based on provider packages (*e.g.,* fan_out). These packages receive rippling effects from many others (their providers). The presence of many *Unstable Packages* that are also *Core Packages* render the system fragile, for changes propagate to and from them.

4.2.1 Package Characterization Applied

In order to study the need for cpc, we evaluated its usefulness to perform analysis tasks on software systems, and we start by detecting packages key in the system. *Core Packages* are, by definition, packages that have many clients. They contain important classes, because change effects in those classes propagate to many client packages, making *Core Packages* key packages in the system.

If key packages are used in a non-cohesive way, maintenance tasks performed on such package will affect packages that are not clearly conceptually related, and developers will therefore find it difficult to predict the extent of ripple effects (since there is no clear concept exposed by the package). As a partial solution to this problem we propose to detect complex, provider (heavily accessed), and contextual non-cohesive packages. This contextual approach contrasts with previous approaches that exploited internal attributes of packages (*e.g.,* size).

Detecting complex, core, client and unstable packages allow developers to predict the scope of changes during maintenance, and to avoid violating design decisions during development (such as to avoid having dependency cycles between packages of different architectural layers). Moreover, understanding the system in terms of provider and client packages on one side, and contextual cohesion on the other, provides guidelines to add classes in package containing classes conceptually related to the new one.

In the following sections, Contextual Information supports detection strategies to obtain packages with special characteristics (as the ones mentioned above) on real applications. We use the detection strategies presented above to detect packages (and their context) with design flaws, such as packages that are complex, reused and not contextually cohesive (*i.e.,* cpc-cohesive). Henceforth, we will use the term cohesion instead

of contextual cohesion (unless explicitly indicated) to refer to cohesion of the package derived from the way it is used, as defined by CPC.

Complex, Core, and Cohesive

Of all the packages detected as complex, after a change not every one has the same impact on the rest of the system. A *Complex Package* with many client packages is more critical than one having no client packages. Even under the same complexity, maintaining the first requires to check its interface with many client packages, whereas maintaining the second does not. Moreover, if a *Core Package* is complex (*i.e.,* its classes depend on one another such as in a high IDR) the question now is if the interface interaction with its context is complex or not. We believe that in *Complex Packages* that are highly referenced, cohesion is important, for it tells about complexity of package interaction. Our goal is then to detect such packages.

Detection strategies detect packages that are complex and provider, whereas Contextual Package Cohesion reveals if the package is cohesive in its context or not. In this section we apply detection strategies to an in-house developed application (CODECRAWLER) and a commercial tool (BASEVISUALWORKS). For details of the systems used as examples of package characterization used in this section we refer to Section B (p.105).

BASEVISUALWORKS We start by analyzing large packages, in the case of BASEVISUALWORKS those with more than 20 classes (smaller packages were analyzed separately in an analogous way). After filtering out packages with less that twenty classes, we obtained 26 large packages that were classified as complex and less complex using the rule for complex packages defined above. Detection strategies need parameters (*e.g.,* $value1$) to filter packages that follow the rule. These parameters depend on the application. In the case of BASEVISUALWORKS, the application from which we selected complex packages that appear in Figure 4.5 (p.56), these parameters for complexity rule were: IDRinherits > 0.15, IDRreferences > 0.15, CohesionRatio > 0.5. For the reuse rule the parameters were fan_ininherits > 9, fan_inreferences > 15. Finally, packages where considered cohesive regarding inheritance if cpcinherits > 0.7 and they where considered cohesive regarding references if cpcreferences > 0.7.

We observe that the parameters to filter packages determine the result of the search. Only by comparing the values for the rest of the packages and after trial and error could we find these parameters.

Figure 4.5 (p.56) shows the twenty-six packages detected as large by the rule of Equation 4.1 (p.50), and categorized into complex (Equation 4.1 (p.50)), provider (Equation 4.2 (p.51)) according to detection strategies defined above, and cohesive according to CPC.

A software product may be more or less reusable depending on the complexity of the context that it requires to operate [Mili *et al.*, 2002]. We believe that a *Complex Package* containing every class related to a concept (intuitively clear to the developer) is harmless if it abstracts details behind a simple *i.e.,* cohesive interface. Figure 4.5 (p.56) shows six such packages (those with crosses under provider and one of the cohesion rows). Their relevance is given by their many clients and their facilities to reuse by their interface-classes forming subsets accessed by the same client or clients.

After filtering large packages and applying detection rules we obtained nine complex packages, seven of which are providers, two of which show neither inherits nor references cohesion. Among the packages classified as non-complex, only four were detected to be providers, three of which were classified as cohesive regarding inheritance. One package was marked as provider and not cohesive.

CODECRAWLER We applied the rule to detect complex, provider, and cohesive packages to CODECRAWLER, a smaller application. Because the size of the application was manageable (only 8 packages) we avoided filtering large packages. Figure 4.6 (p.57) shows these five complex packages of CODECRAWLER (first and second column), the result of applying *Core Package* detection strategy (third column), the result of applying *Client Package* detection strategy (fifth) column, and finally, if each package was found cohesive or not using either inherits or references with a certain threshold (fourth column).

We applied the *Complex Package* detection strategy with the following parameters:

— IDRinherits > 0.05,

— IDRreferences > 0.1, and

— CohesionRatio > 0.1

As a result, it detected five packages (`CCCore`, `CCHotDraw`, `CCUI`, `CCGlyphs` and `CodeEvolver`). Code examination revealed that these packages are very different. The following paragraphs explain in which ways the detected packages were different.

Core complex but easy to reuse Two of the five *Complex Packages* detected are complex core (`CCGlyphs` and `CCCore`). Moreover, though complex and heavily accessed, they are stable, *i.e.,* depending on few packages (not seen in the figure). Only `CCCore` is cohesive, *i.e.,* interface classes having the same clients. We believe that cohesive complex packages have characteristics of being easy to reuse. It does not mean, however, that used in different contexts (such as used by packages of an application client of CODECRAWLER) they would reveal the same characteristics. To start with, a package-context

	Large Package	Complex	Core	Cohesive Inh	Cohesive Ref
	Magnitude-Numbers	x	x	x	
	UIBasics-Collections	x	x	x	
	Kernel-Exception Handling	x	x	x	
	Tools-Programming	x	x	x	
	UIBasics-Support	x	x	x	
	Graphics-Images	x	x		x
	Collections-Streams	x	x		
	External-Types	x			
	Tools-Modules	x			
	OS-Support		x	x	
	Interface-Support		x	x	
	Kernel-Methods		x	x	
	IUBasics-Componenrs			x	x
	UIBasics-Controllers			x	x
	IULooks-Mac			x	
	UILooks-MSWin			x	
	UIBuilder-Specifications			x	
	PackageCategories			x	
	System-Compiler-Support			x	
	Interface-Events				x
	Interface-Events-Trackers				
	UILooks-WinXP				
	UILooks-Motif				
	UILooks-MacOSX				
	XML				
	Graphics-Fonts		x		
Total	**26**	**9**	**11**	**15**	**4**

Figure 4.5: Examples of detection strategies applied to the 26 largest packages of BASEVISUALWORKS.

	Package	Complex	Core	Cohesive Inh of Ref	Client
	CCGlyphs	x	x		
	CodEvolver	x			x
	CCCore	x	x	x	
	CCHotDraw	x		x	
	CCUI	x			x
Total	**5**	**5**	**2**	**2**	**2**

Figure 4.6: Detection strategies applied to CODECRAWLER's packages

change will make `CCGlyphs` and `CCCore` will no longer be core packages, provided that in the new context they do not have enough afferent dependencies (*i.e.,* in the case of low fan_in using inherits and low fan_in using references). Moreover, a package accessed cohesively (*i.e.,* having high cpc) in one package context the package might be accessed in a non-cohesive way in another package context.

Detecting Bad Smells The Dependency Inversion Principle [Martin, 1997], tells us to structure object-oriented applications in such a way that packages implementing high-level policies do not depend on packages implementing low level details. According to this principle, a provider package depending on client packages is a hint of a design flaw. Packages containing more conceptual, abstract or basic functionality should not depend on *Client Packages*. Martin's Dependency Inversion principle tells that modules that implement high level policy should not depend upon the modules that implement low level details. The author claims that adhering to this principle produces reusable and maintainable modules. Detecting *Client Packages* and *Core Packages* is a first step to detect places in the system where the Dependency Inversion principle is not followed. Figure 4.6 (p.57) shows that there are packages that are complex and provider, but not cohesive (`CCGlyphs`, `CCUI` and `CodEvolver`). Observing the code of `CCGlyphs` we noticed that developers had built this package for one single purpose: to contain glyphs. By looking at the names of the classes we can observe that every class-name has the suffix Glyphs. `CCGlyphs` seemed a highly cohesive package that had been wrongly marked as non-cohesive by our contextual approach. However, cpc being lower than the threshold we specified, indicated the presence of at least a class accessed in a separate way. More detailed inspection of the code revealed an implementation particularity breaking the cohesion: a foreign reference due not to a conceptual reference, but to an implementation detail. In particular, a method in a client package iterates through all the subclasses of a certain class in `CCGlyphs` not referenced for other matters.

Detecting Pieces of Design Figure 4.6 (p.57) shows that `CodEvolver` and `CCUI` are complex, though not cohesive client packages. The combination not-provider-and-client tells us that they belong to a higher architectural layer. They are places to implement low level policies (*i.e.*, containing user-interface implementation details) that depend on the context where CODECRAWLER operates, or extensions to the application. In particular, `CCUI` contains details regarding the implementation of CODECRAWLER's user interface and `CodEvolver` constitutes a later addition to CODECRAWLER.

4.3 Package Role Depends on the Client Usage

Packages play different roles according to the context where they are evaluated. A package with low cohesion considered in isolation might prove to be cohesive when used by clients. As an example, we propose in this section to observe the interaction between packages of a framework and packages of two of the framework clients. We believe that cohesion observed by developers in such packages depends on the client application of reference, rather than depending only on internal connections between classes defined in the framework's package.

In order to verify that cohesion of a package P could be different if P were analyzed in different contexts, we counted the interaction between framework and client packages, and generated bar charts by stacking the total number of connections to the framework and from the client.

Figure 4.7 (p.59) a) and b) show the same framework, MOOSE (see **Appendix B** (p.105)) used by two different clients. The top part of the figure, a), shows the usage of framework packages by packages belonging to the client named Chronia. The framework is on the left, and the client is on the right. Bars show, stacked, the total number of dependencies of four dependency types. In the framework (left) each bar counts the number of dependencies *to* each package of the framework. On the client side, each bar shows the number of dependencies *from* the respective package of the client *to* packages of the framework.

The dependency types are the ones defined in Chapter 3 (p.25) together with types *method overridden - overrides method* and *ancestor called - ancestor candidate*. Method overridden happens when a method from a class defined in the framework is overridden in the client. Overrides method happens when a class in the client overrides a method defined in the framework. In a similar way, ancestor called happens when a method in the client invokes a method in the framework. More precisely, when it invokes a method that has the selector of a method defined in the client. This way of counting invocations results in counting connections that do not exist. As a consequence, the number

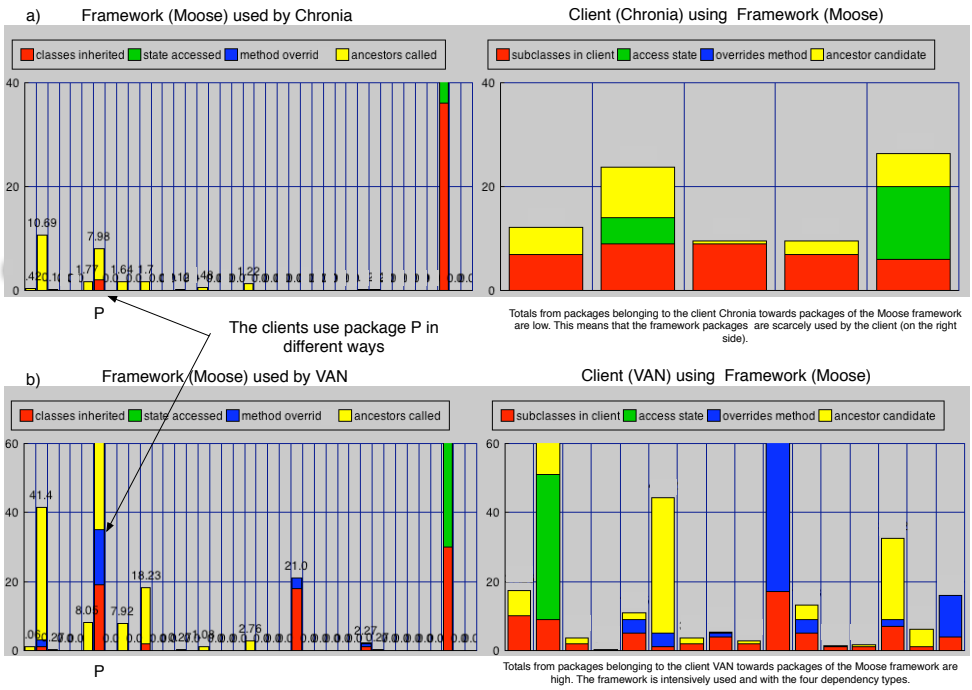

Figure 4.7: More intensive use of the framework Moose, this time used by another client. Packages in the client (on the right side) heavily access packages in the framework.

of these dependencies is significantly higher than the number of other dependencies, and therefore we decided to make it smaller by dividing it for a value. To sum up, the sum of dependencies ancestors called (for framework packages) and ancestor candidate (for client packages) are divided by a value (in this case, 100) to prevent this type of dependency from hiding the other values in the chart.

In Figure 4.7 (p.59) the client VAN uses extensively the framework, including package P; whereas the client Chronia uses less the packages in the framework, and in particular, package P. We learn from comparing a) and b) that P is used in very different ways, and that its role depends, at least in part, o its clients.

4.4 Summary

This chapter shows that Contextual Information provides extra insights: it reveals a different way to compute package cohesion, exposing deficiencies of cohesion as tra-

ditionally regarded, it reveals complex difficult-to-reuse packages, and can be used to reason about design issues.

In this chapter, we have defined a novel measure for package cohesion. We have also presented novel detection strategies for packages, and se have shown evidence that a module with low cohesion considered in isolation might prove to be cohesive when used by clients.

We conclude that Contextual Information enriches traditional approaches to characterize packages based exclusively on internal attributes. The different dependency types participating in package interaction add complexity to the process of characterizing packages. To overcome this shortcoming, we will recur to visualization techniques. The following chapters continue the exploitation of Contextual Information to reveal locality, and package and system characteristics with the aid of visualizations.

Chapter 5

Context Visualization: Characterizing Package Interaction

In the previous chapters, we defined contextual information and built measures to understand the locality of classes in packages. In this chapter we present the application of contextual information to get a mental picture of the structure of a system in our way towards its maintenance. We combine direct and complex measures to get a high level image of the packages and subsystems in which systems are organised. Our views represent the system by visually characterizing packages through their interactions, and building charts that have the shape of a compass-type plot.

5.1 Introduction

Understanding packages as grouping of classes, is a crucial step to manage the overall structure of large object-oriented systems. Packages convey semantics and design intentions of programmers. They are artefacts to deploy and structure applications. Package interaction reveals the stability of the system by exposing areas whose changes ripple through the system. Unfortunately, package interaction is difficult to grasp due to the different nature of the involved elements such as inheritance-based and non-inheritance based coupling, different kinds of connections between packages, and locus of impact, *i.e.*, import and export connections.

In this chapter we tackle this problem by defining a novel visualization named *butterfly blueprints*. Based on direct measures, each butterfly blueprint is a diagram that characterises one package by its interaction with the context. This visualization technique condenses the interaction information enough to fit the characterization of all the application packages in one screen.

Our goal in this chapter is to provide appropriate visualizations to analyse the contextual information of packages in order to help experienced and novice programmers to understand and manipulate systems. The following sections present views that reveal essential properties of large object-oriented systems.

5.2 Butterfly Views

Butterfly views are dedicated radar charts built from simple package metrics based on a language-independent meta-model. They help us to find the answers to the following questions:

— What is the importance of a package in terms of its intrinsic properties such as the number of classes it contains and its efferent and afferent relationships? How many clients rely on it?

— Does the package use several other packages or is it more self-contained?

— What is the impact of changes in the relationships between packages?

— Can we identify patterns or repeating package characteristics?

— How is a package structured: does it only extend other packages via inheritance, or does it define itself some complex hierarchies? When classes are subclassing other classes what are exactly the relationships that link them (state, behavior)?

5.2.1 Package Polymetric Views

Package polymetric views [Ducasse *et al.*, 2004] are polymetric views [Lanza and Ducasse, 2003] designed to provide a coarse-grained visualization of packages. Figure 5.1 (p.63) describes the meaning of the elements present in a package polymetric view; whereas Figure 5.2 (p.64) shows, as an example, a polymetric view of ALCHEMIST (Section B (p.106)). In this figure, we observe that packages A (in the bottom of the figure) and C are heavily accessed, apparently being the core of the application; whereas package F is a client package, having dependencies to many providers. The figure also reveals a dependency

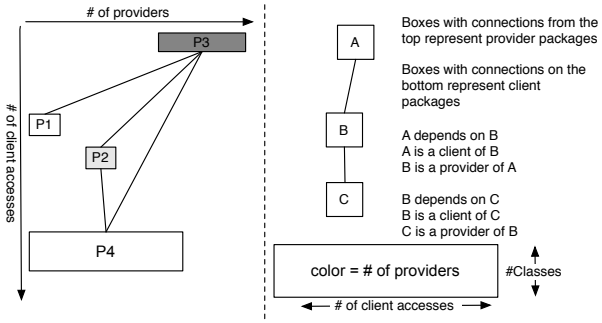

Figure 5.1: Description of package polymetric view. Rectangles represent packages, and edges represent dependencies between classes defined in different packages. Packages that contain classes heavily accessed appear on the bottom of the view; whereas the amount of providers that a package has is represented by placing the package on the right.

cycle involving packages A, C, and D. Cycles between packages are source of problems, for instance, concerning the order of loading.

Figure 5.1 (p.63) reveals that package polymetric views are coarse. We observe that they do not express the different dependency types connecting packages, being, for instance, difficult to spot cycles even in a small system as the one in example.

Figure **??** (p.**??**) shows an example of this polymetric view applied to a large system. Once again, this approach provides a coarse view of the system, but it does not scale.

We conclude that this approach is useful only as a first approach to understand the system. Developers could gain some limited understanding of the system by observing it first with a package polymetric view, use this view to select packages for further analysis, and obtaining more information with other visualization techniques applied to the selected package. All in all this polymetric view reveals the need for a more refined approach able to show, for instance, package connections in the different dependency types.

5.2.2 Principles of Butterfly Visualization

Butterfly visualization are based dividing a circle area with a number of axes, each one representing the value of a measure. Butterfly views represent a package's interaction with its context. To represent this complex interaction is not straightforward, since the order of axes determines the shape of butterfly views. To keep the shapes expressive and simple to read is important to compare packages through their butterfly views. To that

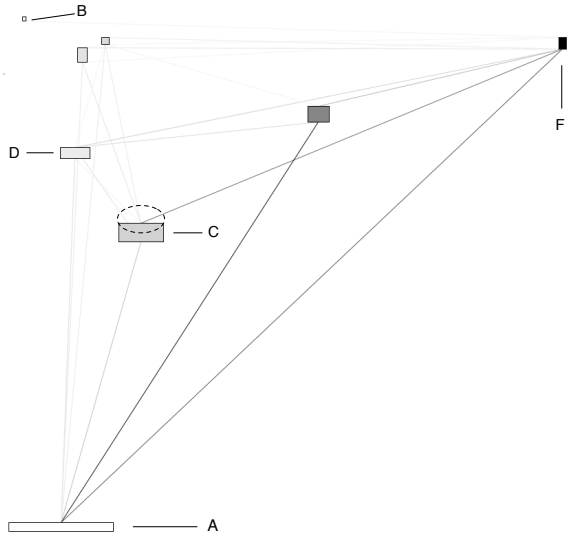

Figure 5.2: Example of polymetric view showing all the packages of CODECRAWLER.

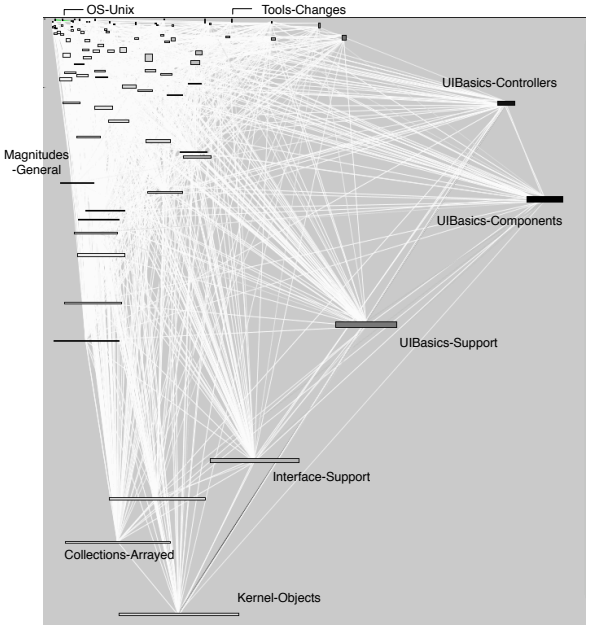

Figure 5.3: Package Polymetric view showing the system BASEVISUALWORKS.

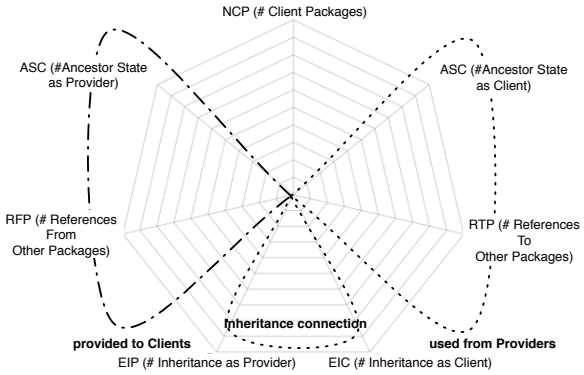

Figure 5.4: Principles of the butterfly view based on direct measurements.

regard we exploit the characteristic of symmetry: butterfly views are symmetric regarding their vertical axis, keeping inheritance connection description in the bottom.

Butterfly views are based on minimal information revealing the essence of a package. Table 5.1 (p.81) lists the measures we actually compute. In this table the term *external dependencies* denotes dependencies that originate from other packages and target classes of the analyzed package or vice versa. The metric example values refer to the situation depicted in Figure 3.1 (p.28). Figure 5.4 (p.65) explains the semantics of butterfly views based on direct measurements.

We propose *absolute* and relative measures for a package. Absolute measures are direct measures obtained from counting dependencies. However, the use of facade classes could obscure the visualization of the structure of the package. Butterflies based on relative measures capture the effect of increased internal dependencies.

— *Absolute measures* count the dependencies of a given kind and direction. An example of an absolute metric of a package is references (Number of Class References To Other Packages) which is the number of class references to classes belonging to other packages (providers) from classes belonging to the analyzed package (a client). This metric is useful to assess whether a package (and its classes) is heavily using other packages.

— *Relative measure* show the relationship between the amount of internal and external dependencies of a given type and direction in the package. They follow the pattern:

$$property/(property + internalproperty)$$

For instance, the relative measure RRTP (RelativeNumber of Class References To Other Packages) divides references by the total number of class references in a package, thus creating a normalized measure (*i.e.,* between 0 and 1) that denotes to what extent a package is self-contained (low RRTP) or not (high RRTP).

5.2.3 Global Butterfly Views

Absolute measures originate butterfly views named *global views*. Figure 5.5 (p.67) displays the GLOBAL BUTTERFLY of the packages CCCore, CCBase and CCUI, which belong to the CODECRAWLER case study (see Section B (p.105) for a description of CODECRAWLER).

CCBase. Its shape leaning towards the left shows that this package is essentially a provider package. In addition it shows that the state of the classes in the package is directly accessed by clients subclasses (for example CCCore) and that the package also accesses state of other packages. A close inspection of the code reveals that the references to other packages are the ones to default types such as String and Collection.

CCCore. It is a central package of CODECRAWLER. This is reflected by the fact that the butterfly has two even, long and horizontally symmetric wings. It uses the package CCBase. The view indicates that this package uses 86 external classes while it defines 22 classes. The classes it defines are referenced from other packages too (is_referenced (Number of Class References From Other Packages) = 58). inherits (Number of External Inheritance as Client) shows that this package inherits from 10 classes in the other packages, but this package is also extended (is_inherited (Number of External Inheritance as Provider) = 3). This package does not directly use state from the superclasses which is an indication of good design. We also learn that its state is directly accessed by subclasses defined in other packages (is_accessed (Number of Ancestor State as Provider) = 2).

As the package contains 22 classes and inherits (Number of External Inheritance as Client) is 10, we learn that the package inherits solely from a couple of root classes but that it is composed of inheritance hierarchies.

CCUI. The GLOBAL BUTTERFLY of CCUI shows that it is mainly a client: its classes directly access attributes of provider superclasses (accesses (Number of Ancestor State as Client) = 152). This package will be impacted if the superclasses located in other

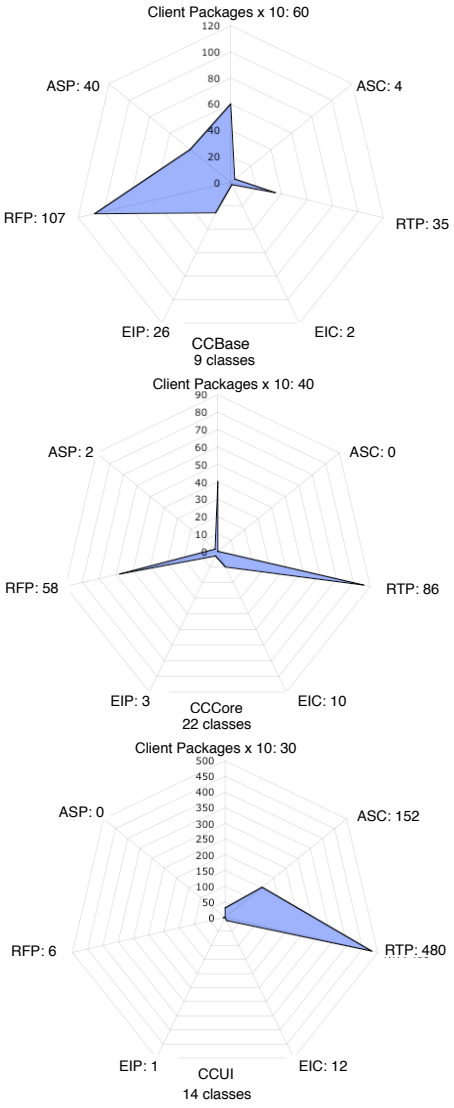

Figure 5.5: Butterfly Direct view on three packages.

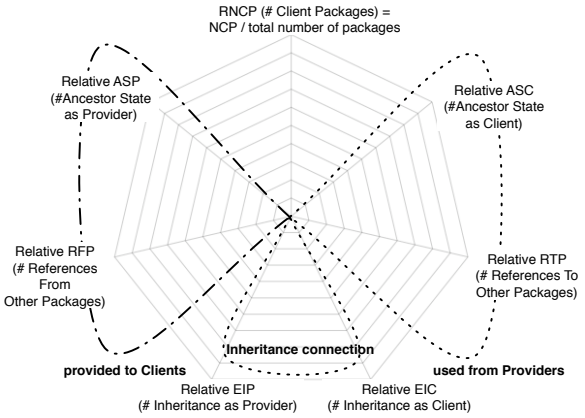

Figure 5.6: Principles of the Butterfly Relative view.

packages change. The high-value (480) of references (Number of Class References To Other Packages) is due to the manual building of menus *e.g.,* direct instantiations of MenuItem. This shape was expected, because CCUI contains all the CODECRAWLER UI elements.

5.2.4 Relative Butterfly Views

Relative measures originate *relative views*. They complement globalviews. While the GLOBAL BUTTERFLY provides information about a package, it does it by measuring the package in the context of the complete system. However, it is difficult to assess how a property exists *in the context* of the package itself. For example, the information that a package defines a lot of classes is refined when we know that most of the classes are inheriting from a class defined inside the package itself or when most of the classes are subclasses of an external class. Figure 5.6 (p.68) describes the principles of relative butterfly views. To minimize context-switching a RELATIVE BUTTERFLY has the same axes as the GLOBAL BUTTERFLY, but uses relative metrics described in Table 5.1 (p.81).

As the following example illustrates, there is an interplay between the two views. In particular the information displayed by the GLOBAL BUTTERFLY allows one to qualify the finer level of description given by the RELATIVE BUTTERFLY. Figure 5.7 (p.70) shows an example of butterfly relative views applied to the same three packages as before.

Example. Figure 5.7 (p.70) shows the RELATIVE BUTTERFLY views of three packages of CODECRAWLER: CCBase, CCCore and CCUI.

CCBase. We see that its classes do not directly access state, since RASP (Relative Number of Ancestor State as Provider) and RASC (Relative Number of Ancestor State as Provider) are 1. This happens even when such classes are accessing the state of external superclasses (accesses (Number of Ancestor State as Client)= 4) and their state is accessed by clients classes (is_accessed (Number of Ancestor State as Provider)= 40. As the value of REIC (Relative Number of External Inheritance as Client)is 0.25, we learn that this package has 3 times more internal inheritance than it is inheriting from others. REIP (Relative Number of External Inheritance as Provider) = 0.81 indicates that it subclassed more from the outside than from the inside. In fact it indicates that it is subclassed 26 times from other packages whereas there are only 6 inheritance dependencies in the package. However, it could still be the case that its classes are much more subclassed: A class can be subclassed by a class in another package that then acts as another hierarchy root to numerous classes.

CCCore. Considering CCCore, we see that it does not access the state of other packages (RASC (Relative Number of Ancestor State as Client) = 0). It has more references to the outside than references between the classes inside the package (RRTP (RelativeNumber of Class References To Other Packages) = 0.8) and it has a bit more references from other packages (RRFP (Relative Number of Class References From Other Packages) = 0.73) than internal class references. REIP has a value of 0.2 which means that the package has a lot more internal inheritance relationships than it has direct subclasses.

CCUI. Regarding CCUI we see that the REIC (Relative Number of External Inheritance as Client) value of CCUI($REIC = EIC/(EIC + PII)$) is 1. This confirms that it does not define an inheritance hierarchy. Interpreting RRTP (RelativeNumber of Class References To Other Packages) whose value is 97%, we learn that the package classes have few class references among them, because there are 480 references to external classes and only 3% of internal references (*i.e.,* 14 internal references). RRFP (Relative Number of Class References From Other Packages) is 32%, since there are 6 external and 14 internal references (is_referenced (Number of Class References From Other Packages)).

Example. Figure 5.8 (p.71) shows the RELATIVE BUTTERFLY views of three packages of CODECRAWLER: CCBase, CCCore and CCUI.

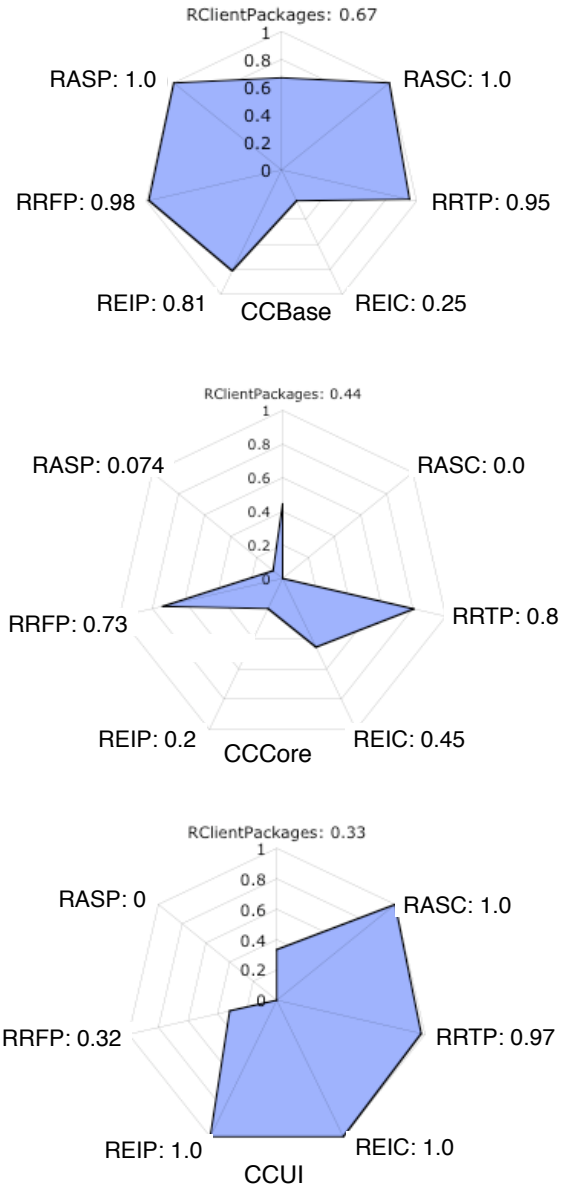

Figure 5.7: Examples of relative butterfly views of three packages of CODECRAWLER.

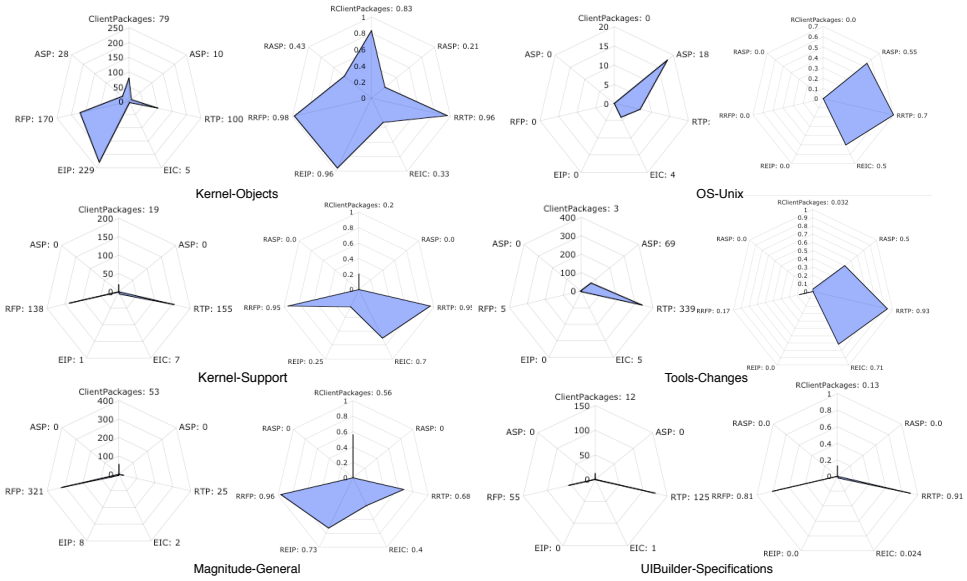

Figure 5.8: Example of butterfly direct view and relative view of six packages from BASEVISUALWORKS.

Figure 5.8 (p.71) shows an example of butterfly direct view and butterfly relative view applied to six packages of BASEVISUALWORKS (see **Appendix B** (p.105)).

5.3 Butterfly System Blueprint

Based on butterfly views, butterfly blueprints condense package-context interaction visualization enough to fit the characterization of many application packages in one screen. This picture of the system as a whole conveys information about package organization in subsystems, and facilitates detection of visual patterns.

Figure 5.9 (p.72) depicts the process of mapping measures to the icon simplifying a butterfly GLOBAL BUTTERFLY.

To compare butterfly blueprints, metrics values are normalised. Two butterfly blueprints can only be compared if their corresponding axes can be compared. We normalise therefore each of the seven axes to the maximal drawing length of *that* axis, as opposed to the maximal drawing length of any axis, being this a difference with butterfly views (GLOBAL BUTTERFLY and RELATIVE BUTTERFLY).

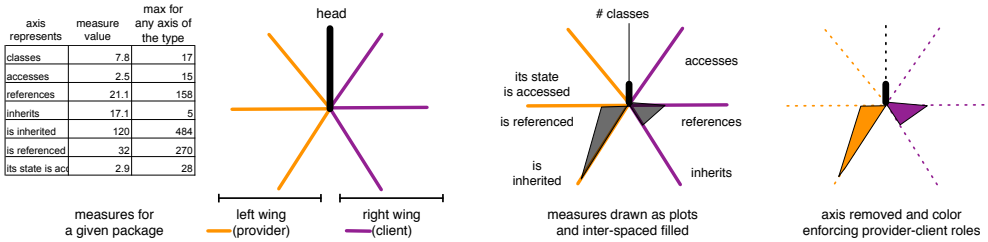

Figure 5.9: Mapping from measurement values to butterfly blueprint icons

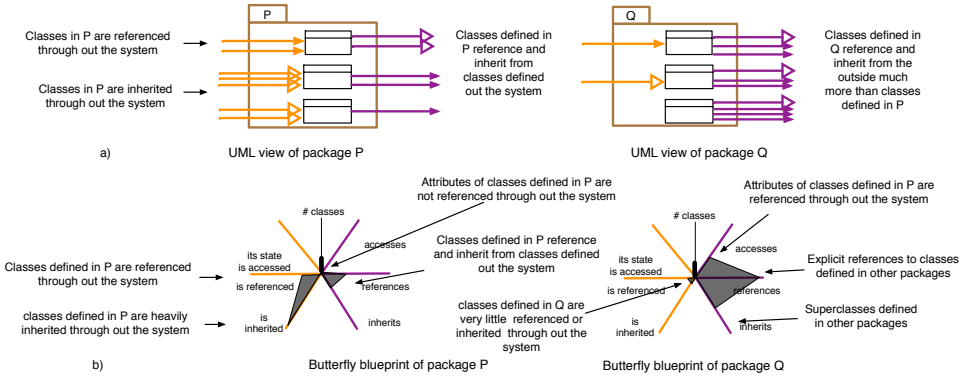

Figure 5.10: Principles of the butterfly view based on direct measurements.

Figure 5.10 (p.72) shows principles of butterfly blueprints.

5.3.1 Understanding Systems with Butterfly Blueprints

Understanding packages and their context is key to develop systems that are able to respond to change.We seek answers to the following questions:

Which are the packages containing classes used throughout the system? The maintenance of these packages might produce a cascade of changes in dependent packages. When the extent of the ripple effects originated by a change cannot be predicted, the impact of making that change cannot be estimated. As a consequence, neither can the cost of the change be estimated, nor can the affected areas be predicted.

Can we detect violation of design principles? For instance, is a package implementing common functionality depending on another implementing details? When a

package with classes containing implementation details defines heavily inherited classes, the package becomes difficult to reuse.

Can we package interaction in blueprints? Packages may represent code ownership, or the deployment process. Consequently, package interaction describes the complexity of the application design and deployment. This interaction is usually too complex to be depicted in one screen with a useful level of detail using nodes and edges. We propose a visualization of large systems that on the one side shows detailed context-information for all the packages, and on the other fits the characterization of the packages within the eye span. By having the characterizations in one screen, developers can compare them and get insights about the system such as the distribution of complexity.

What kind of patterns can we find? Packages devoted to singular tasks have special shapes. By comparing the shapes of diagrams, we can derive knowledge about role and function of the packages such as packages concentrating most of the implementation, or the ones with poor interaction with the context.

We chose to visualize direct connections that reveal the essence of the package interaction with the context. The information that we use is based on three types of connections: inherits (inheritance-based coupling), class references (non-inheritance-based coupling) and access to inherited attributes (inheritance-based). P being a package, our approach builds a butterfly blueprint of P on the dependencies described in table 5.2.

Figure 5.11 (p.74) shows part of the butterfly blueprints obtained on BASEVISUALWORKS, a large application (see **Appendix B** (p.105)). Groupings of blueprints represent subsystems forming part of the organization of packages in BASEVISUALWORKS.

The following sections explain the patterns that we can identify in this view.

Big Independent Provider

Big Independent Provider denotes packages containing major inheritance hierarchy root classes, or that are exceptionally referenced, being relatively independent (a sign of light stability).

In the butterfly views, this phenomenon is represented by a butterfly with a larger left wing than the others and a yellow-needle like shape on one or more of the left axis.

Figure 5.11 (p.74) shows package `Kernel-Objects`, with its inheritance definition axis (down, left) significantly bigger than its additional axes and the other packages. This

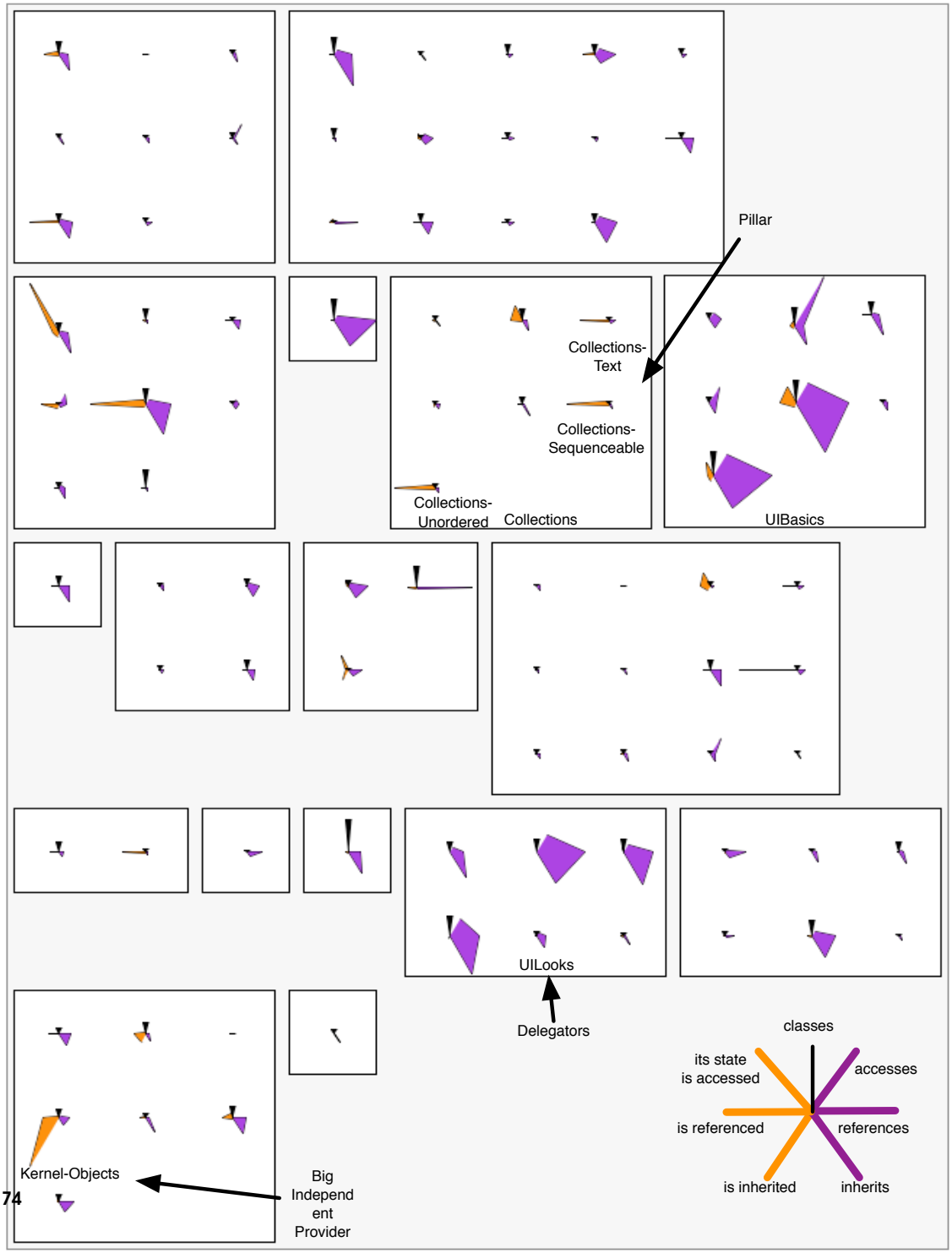

Figure 5.11: Butterfly blueprint of BASEVISUALWORKS

indicates that this package contains classes root of the hierarchy. Indeed, `Kernel--Objects` contains some important classes such as `Boolean`, `True`, and `False`. Many of its classes are heavily subclassed and considerably referenced.

In the same line of examples, subsystem `Collections` shows 3 packages `Collections--Text`, `CollectionsSequenceable`, and `Collections-Unordered` having a yellow-needle like shape and a very small right wing. The indicates that these packages are considerably referenced,. These packages contain the implementation of the programming structures related to the class `Collection`. `Collections` is highly referenced, which shows it as a subsystem capable of producing a high impact in the case of a change.

Delegator

A *Delegator*is a package that delegates the details. The corresponding butterfly shape has wide right wings and small or no left wing. It characterises packages that consume more services than what they provide.

In Figure 5.11 (p.74) we can detect packages of the subsystem `UILooks` implementing the user interface. The butterflies of the packages have only right (*i.e.,* provider representative axis) wings and a big head. This tells that these packages subclass and reference classes from other packages, and have some classes defined in them, but they have no clients.

Close examination of the code showed that these packages depend on the platform where BASEVISUALWORKS is running. They contain therefore implementation details such as preferred bounds of views and icons to be displayed. Because they contain platform-depending details, and very few clients, we deduce that packages implementing high level policy do not depend upon the modules that implement low level details (Martin's Dependency Inversion Principle [Martin, 1997]), which is a good practice of designing for reuse.

Pillar

Pillar is a package containing classes that are extremely used. As an example, let us observe the package that contains the longest axis in the system, `Collections-Arrayed`. Figure 5.12 (p.76) shows a comparison of the packages of the subsystem `Collections` and the package `Collections-Arrayed`, one of the five exceptionally coupled packages.

The packages depicted in Figure 5.12 (p.76) in the subsystem `Collections` are the same packages of the system `Collections` depicted in Figure 5.11 (p.74) plus the package

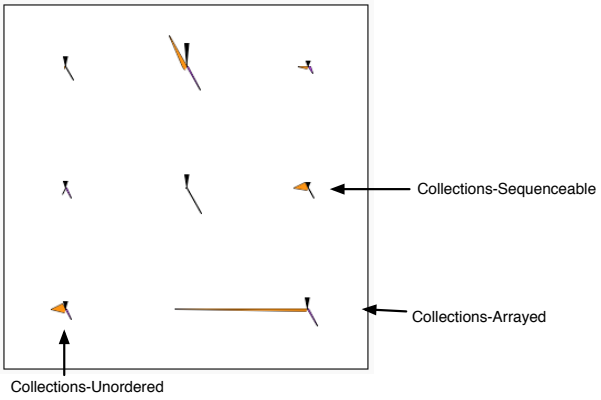

Figure 5.12: This example illustrates the of the difference between `Collections-Arrayed`, containing the classes most referenced, and the rest of the packages in `Collections`. Butterfly blueprints of other packages in `Collections` are minimised and changed due to the inclusion of an exceptionally referenced package.

`Collections-Arrayed`. However, their shapes seem to be different. `Collections-Arrayed` is an exceptional package containing the classes most referenced in BASEVISUALWORKS. There are five packages as exceptional as `Collections-Arrayed`, depended on or depending on the six different dependency types depicted in the butterfly. All the axis sizes are normalized. And there are only five exceptionally coupled packages. Therefore, including those exceptions in the design, obscures the information presented from the other packages, by diminishing the size of the 87 remaining packages. Those five packages were therefore excluded from the representation of the system offered in Figure 5.11 (p.74).

5.3.2 Application-Specific Patterns

Similar shapes are clear signs of similarity between the packages. In some cases all the packages in the same subsystem have the same butterfly shape. The subsystem `UILooks` in Figure 5.11 (p.74) shows an example of that. Butterfly blueprints of its packages make evident that those packages share the same shape. Figure 5.13 (p.77) zooms in the `UILooks` subsystem. All the packages take a heavy consumer role (big right wing), being insignificantly inherited or referenced from other packages (small left wings).

In this example, the reason why all the packages in the subsystem `UILooks` have the same shape is that each package concentrates a similar set of responsibilities. More concretely, each implements the instantiation of a set of widgets required to build a

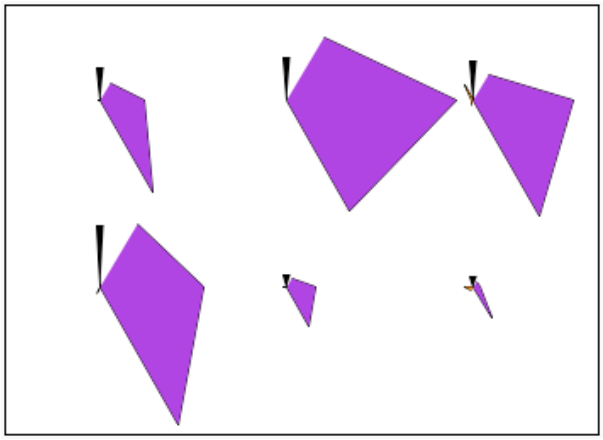

Figure 5.13: This view illustrates the similarities in shapes between the packages of subsystem `UILooks`

window, depending on the selected look and feel (Windows, OS/2, Macintosch, etc). A known use of the abstract factory pattern, `UIBuilder` (defined in another subsystem) has the responsibility of constructing a window from a set of widget specifications, but without any knowledge as to what type of look-policy object it is dealing with [Alpert *et al.*, 1998].

The shape of these packages is what one would expect from a subsystem devoted to the user interface. The reason behind big right wings is that their classes heavily request the services of classes in UIBasics, responsible for the implementation details. The reason behind the yellow left wings being small is that `UIBuilder` (the window builder) is unaware of the type of what type of look policy object it is dealing with. Finally, the reason why there is a left wing at all is that the policies inherit UILookPolicy, defined together with `UIBuilder` in the package `UIBuilder`.

From a more general perspective, with butterfly views we learn that many of the subsystems of BASEVISUALWORKS show this style of organization, with most of the packages in the same subsystem having similar butterfly blueprints. A few examples are System-Name Spaces, Interface, `Collections`, UIBasics, External, UIBuilder, Graphics, UILooks, Tools and Kernel.

If the visualization of a group of packages in a subsystem follows a pattern, then we can expect to detect a package that breaks the pattern of the group, or a subsystem devoted to a particular task. Comparing diagrams is a way to detect a package that violates design architectural decisions. From a more general perspective, we have observed that a case in point is packages implementing graphical user interface functionality. These

packages usually depend for implementation details on other packages specific for the platform and look-and-feel. This dependency translates into classes heavily referencing and inheriting from the context, which in turn imposes a particular shape in the diagram resulting from the application of the visualization technique.

5.3.3 ArgoUML

Figure 5.14 (p.79) shows the butterfly blueprint overview of argoUML. Java namespaces (named also packages in the context of this work) are distributed into higer-level namespaces (subsystems).

A first approach reveals that the most heavily used package is `uml`. In fact, most of the the packages are just clients, some are mainly provider, and only one package, `uml` is client and provider. The butterfly blueprint of `uml` compared with the rest of the butterfly blueprints, reveals a significant amount of inheritance and references to and from `uml`, signalling it as a key package in the system. It shows that most of the functionality is defined subsystems `uml`.

`kernel` package, which is heavily referenced, but also inherited, is free from the impact of change in other packages. Having 0 access to attributes and few inheritance dependencies and references to other packages.

We detect that 9 of 13 subsystems are devoted to only one package, meaning that most of the subsystems are one-purpose. A few examples of those are `ocl`, `persistence`, `moduleloader` and `notation`.

5.4 Summary

This chapter presented a novel approach to obtain graphical views of an object-oriented system, understanding its packages and coping with its complexity. We seek to capture the organization of packages in an object-oriented system and to characterize packages and their interaction.

Out contribution is two radar visualizations named butterfly views that help to understand and categorize packages. The butterfly views not only show how a package relates to the rest of the system, but also how it is internally structured.

Our contribution includes also a novel visualization technique named butterfly blueprints, which produces graphical views showing the system in one screen.

Figure 5.14: Butterfly view applied to argo UML

Butterfly blueprints answer the aforementioned requirements by revealing patterns in the package interaction in the system. We applied our approach to three different applications to find interaction patterns, namely *Big Independent Provider*, *Delegator* and *Pillar*. The patterns allowed us to detect core packages, and revealed design issues such as co-locating in the same subsystem classes with related knowledge.

Name	Description
PP	*(Number of Provider Packages).* Number of *package* providers of a package. PP(P1)=1, PP(P2)=2, PP(P4)=1.
fan_in	*(Fan_in).* Number of *packages* that depend on a package. fan_in(P1)=3, fan_in(P3)=2, fan_in(P4)=0.
references	*(Number of Class References To Other Packages).* Number of class references from classes in the measured package to classes in other packages. RTP(P1)=2, RTP(P2)=1 ,RTP(P3)=1, RTP(P4)=0.
RRTP	*(RelativeNumber of Class References To Other Packages).* references divided by the sum of references and the number of internal class references.
is_referenced	*(Number of Class References From Other Packages).* Number of class references from classes belonging to other packages to classes belonging to the analyzed package. RFP(P1)=0, RFP(P2)=1, RFP(P3)=3, RFP(P4)=0
RRFP	*(Relative Number of Class References From Other Packages).* is_referenced divided by the sum of is_referenced and the number of internal class references.
PIIR	*(Number of Internal Inheritance Relationships).* Number of inheritance relationships existing between classes in the same package. PIIR(P1)=0, PIIR(P2)=0, PIIR(P3)=3, PIIR(P4)=2
RPII	*(Relative Number of Internal Inheritance Relationships).* PIIR divided by the sum of PIIR and is_inherited. RPII(P1)=0, RPII(P2)=0, RPII(P3)=1, RPII(P4)=1.
inherits	*(Number of External Inheritance as Client).* Number of inheritance relationships in which superclasses are in external packages. EIC(P1)=0, EIC(P2)=2, EIC(P3)=1, EIC(P4)=1
is_inherited	*(Number of External Inheritance as Provider).* Number of inheritance relationships where the superclass is in the package being analyzed and the subclass is in another package. EIP(P1)=4, EIP(P2)=0, EIP(P3)=0, EIP(P4)=0
REIP	*(Relative Number of External Inheritance as Provider).* is_inherited divided by the sum of PIIR and is_inherited. REIP(P1)=1, REIP(P2)=0, REIP(P3)=0, REIP(P4)=0.
accesses	*(Number of Ancestor State as Client).* Number of accesses to instance variables defined in a superclass that belongs to another package. ASC(P3)=0, ASC(P4)=1
RASC	*(Relative Number of Ancestor State as Client).* accesses divided by the sum of accesses and ASCI. Where ASCI, Number of Ancestor State Client Internal to the Package is the ancestor state class dependencies internal to the package. We consider only dependencies from a class that is inside the package to other classes of the same package.
is_accessed	*(Number of Ancestor State as Provider).* Number of times that instance variables of classes belonging to the analyzed package are accessed by classes belonging to other packages. ASP(P1)=1, ASC(P4)=0
RASP	*(Relative Number of Ancestor State as Provider).* is_accessed divided by the sum of is_accessed and the number of gives ancestor state dependencies between classes when both classes belong to the package.
CC	*(Number of Class Clients).* Number of external *class* dependencies that are clients of a package. Sum over the number of the class dependencies (ancestor state, class reference and inheritance) that refer to a package. CC(P1)=4, CC(P2)=1, CC(P3)=3, CC(P4)=0.
NCP	*(Number of Classes in a Package).* Number of classes in the package. NCP(P1)=2.

Table 5.1: Package Measures used in the Global and Relative butterfly views.

Type of Connection	**Description**	**Measure Name**	**Name in global view**	**How to count the connections**	**Client-Provider Role**
Inheritance	A class is subclass of another.	inherits	Number of External Inheritance as Client (inherits)	Add up the number of classes defined in P having superclasses defined in the rest of the packages.	Client
		is inherited	Number of External Inheritance as Provider (is_inherited)	The total number of inheritance relationships in which the superclass is defined in P, and the subclasses are defined in packages other than P.	Provider
Class Reference	Statement that explicitly reference a class, *e.g.,* during class instantiation	references	Number of Class References To Other Packages (references)	The total number of class references from classes in P to classes defined in other packages.	Client
		is referenced	Number of Class References From Other Packages (is_referenced)	The total number of explicit references to classes in P made *by* methods of classes not defined in P.	Provider
Inherited Attribute Access	Method references attribute defined in superclass.	accesses inherited attribute	Number of Ancestor State as Client (accesses)	The total number of attribute references *by* methods belonging classes not defined P.	Client
		attribute referenced in subclass	Number of Ancestor State as Provider (is_accessed)	The total number of references (from packages other than P) *to* the attributes of the classes defined in P.	Provider
-	-	NOC	-	Number of classes defined in the package.	-

Table 5.2: Measures used in butterfly blueprints.

Chapter 6

Optimized Re-architecturing: Understanding the Future

In the previous chapters we developed techniques to facilitate maintenance of big legacy systems by describing how they organize its classes into packages. However, a particular organization may be neither straightforward nor obvious for a given developer. As a consequence, classes can be misplaced, leading to duplicated code and ripple effects with minor changes effecting multiple packages. Exploiting contextual information, we propose a technique to detect misplaced classes by analysing how client packages access the classes of a given provider package. We define locality as a measure of the degree to which classes reused by common clients appear in the same package. We then use locality to guide a simulated annealing algorithm to obtain optimal placements of classes in packages. The result is the identification of classes that are candidates for relocation.

6.1 Improving Locality

Where was that class that I always forget to update each time I make a change on this one? Where was the class that was doing something similar to what this one does? Our ultimate goal is to improve the system in the sense that classes that are used together are in the same package. As a result, we propose an operational technique (a procedure) to represent and manage the location of classes in packages in large object-oriented systems. But first we have to understand and represent how classes in a

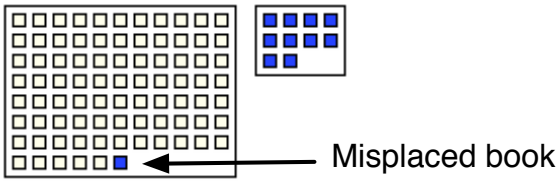

Figure 6.1: In this library, among the white books there is one read by the people that read only blue books.

package belong together. We exploit the information present in the context, i.e., how the classes are used. In particular, we define Locality (loc) as Contextual Package Cohesion (cpc) evaluated using references dependencies. Consequently, its computation is that of cpc as defined in 4.1. Let us present a metaphor from everyday life that describes the idea behind our approach to capture the locality of the classes.

Motivating Example

Books related to the subject white may belong to different editorial offices, have different authors and have, in short, no explicit attribute that connects them, but they also belong to a group: the group of books consulted by readers of subject white. If we were organizing a library, it would make sense to put on the same shelf the books that belong to the same field of study. That is straightforward if we know the field of each book, but not if we are unaware of it. However, even without knowledge about the contents of the books, we could observe that every group of readers is interested mainly in one field. If that is the case, a book read only by people interested in subject blue that is on the same shelf as books read by students interested in subject white would call our attention and most people would agree that it is out of place. Figure 6.1 (p.84) depicts this situation.

We expand this idea to understand locality in large object oriented systems. More concretely, to describe the locality of the classes of a package. Now Figure 6.1 (p.84) does not represent books anymore, but code. It depicts four packages of a system containing classes. We believe that most of the developers would agree that the locality of these packages would increase if the blue class where defined in the blue package, instead of being defined in the white package.

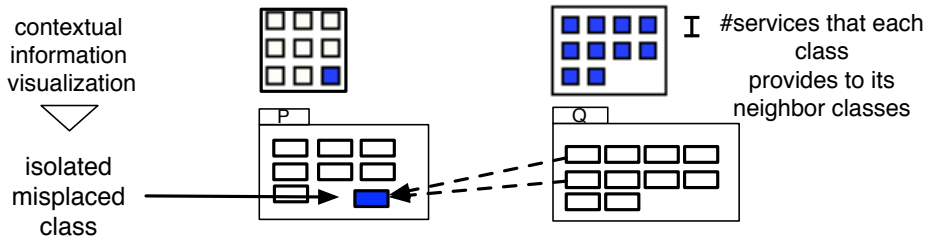

Figure 6.2: Two packages and their representation in contextual information visualization.

6.1.1 Visualization of Locality

This example presented the idea behind our technique to find classes that are misplaced: two classes are related if they are used by the same package. Software doesnt have any form. Opposed to mechanical simulation, where real objects are represented in two or three dimensions, software is intangible. We choose a rectangle as the shape to represent packages and classes. The goal is to capture characteristics of the packages or classes that we consider relevant for understanding the code, and to associate them through metrics with the rectangle size and color. Figure 6.2 (p.85) shows the mapping from code to our visualization.

The big rectangle represents a package, and the small ones inside represent the classes defined in the package. The color shows the locality of the classes. If the class is misplaced, it is drawn as a rectangle inside the package where it is defined and painted with the color of another package (a potential destination).

Understanding the distribution of classes into packages is one way of analyzing the pros and cons of restructuring a large-scale software system. With visualization described in the previous section, the developer receives hints about which classes to move and where, but nothing about what he should tackle first. Contextual information provides a solution to this problem. It captures reasons why the class should not be re-located, which are the clients a class has in the subject package. In other words, it describes the “happiness” of a class.

6.1.2 The Happiness of a Class

Classes interact to perform tasks. They are naturally sociable. Therefore they are happy when they have clients in package where they are defined. The more services a class provides to its neighbors, the happier it is. With the locality, we detect classes whose

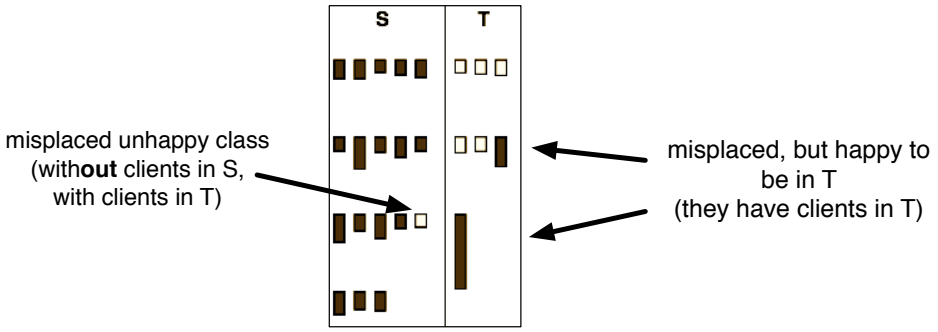

Figure 6.3: Two packages and their representation in contextual information visualization.

clients belong to foreign packages. When a class is used by some foreign clients and the neighbors are used by a complete different group, this circumstance pulls the class out. But this information omits the reasons why the class is in the package. For instance, there can be structural reasons supporting the placement of a class in a determined package.

By adding size to the rectangles representing classes, the proposed visualization shows misplaced classes and provides hints to the developer about which class re-locations are more important than others. Figure 6.3 (p.86) depicts this situation.

6.1.3 Average Locality

We define the average locality (avloc) of a system (S) as the average locality in its packages.

Definition 7 *Being S a system and P a package in S,*

$$avloc(S) = \frac{\sum \mathsf{loc}(P)}{\mid numberofpackages \mid}$$

6.2 A Combinatorial Optimization Algorithm at the Service of Software Maintenance

In 1983 Kirkpatrick et al. [S. *et al.*, 1987] proved that the search of an optimal solution in combinatorial optimization algorithms is analogous to a method that models the search

of a state of balance of a solid. In the field of thermodynamics, that method is well known as simulated annealing (SA). When using simulated annealing to make glass, for example, the system starts at a high temperature with molecules moving randomly. The higher the temperature, the more the molecules move. After some time the system cools down. Eventually, it cools down enough for the molecules to form a glass, which is the resulting solid. If the glass contains defects, the system is heated again followed by the cooling time. This process is repeated until finding a satisfactory result, in our example, a glass with none or with acceptable defects, or having reached a condition to stop. Because of the analogy existing between the method that models the search of a state of balance in a solid, and the combinatorial optimization method, the later was named simulated annealing as well. SA is an efficient technique to find the maximum value of a function of many independent variables. This function is usually called an objective function. It represents a quantitative measure of the goodness of a complex system [13], being dependent on the detailed constitution of the system. The travelling salesman problem is a classical example of a combinatorial optimization problem. Given a graph representing N cities and the cost of travelling between two cities, the problem is to plan the route of the salesman visiting each city once, and minimizing the total cost. This problem is similar to our problem of re-architecturing a system by optimizing its design. Therefore, we apply SA to optimize the average locality of packages. However, our interest is not in finding the optimal solution. Even if this solution could be found, nothing guarantees that all the developers will agree that this is the ideal partitioning of the system. Our interest is in obtaining the movement of classes around packages that led the optimization function to be closer to the optimal value. It is a subset of those movements that we visualize. The optimization function is then (1 - avloc), which is based on avloc as it was defined in the previous section, and ranges between 0 and 1. In the context of thermodynamics, molecules move randomly. In the context of using context information to re-architecture a system in software engineering, the movement of molecules represents the movement of classes from one package to another. This suggests the presence of constraints specific to the software system that determine which classes can moved. Due to these constraints and for reasons of performance, the movements in our approach are guided rather than random. Experiments giving total freedom in choosing randomly the next package to be acted upon were inefficient. We therefore optimized the algorithm. The modification consisted in (a) ordering the packages according to a criterion, for instance the cohesion of the package, and (b) restricting the permitted movements. The random movements where applied first on those packages with lower cohesion. And a class would move to a different package only if this package is client of the package where the class is defined.

6.3 Validation

In order to study the effectiveness of our approach, we apply it to three real systems and show its potential to perform common analysis tasks. Our goal is to understand the organization of classes into packages, in an effort to detect violation of the architecture of the system. For instance, a generally accepted principle states that code should always exhibit low coupling and high cohesion. We aim to find visual patterns revealing conformance or breakage of this principle. Moreover, the architecture of the system (e.g., layered, blackboard, etc.) should be respected. In a similar way, we aim to find visual patterns indicating a design flaw in this regard. All in all, we use our contextual information approach, in particular, client usage, to grasp the class organization, and to analyze package coupling and cohesion.

6.3.1 Applications

We apply the proposed contextual information approach to the following systems:

ALCHEMIST is package analysis and measurement too. It consists of 138 classes distributed into six packages.

CODECRAWLER is a small software visualization tool. It contains 244 classes distributed in eight packages and uses three more packages. CODECRAWLER was built under the same design principles as ALCHEMIST. But while ALCHEMIST is still heavily evolving, CODECRAWLER is already a mature tool. Figure 6.4 (p.89) shows a locality visualization of CODECRAWLER. The uniformity of color shows a high modularization of the packages. Only three out of the eleven packages show classes of a second color besides their own. None of the classes hinted to be relocated has a particularly strong coupling with its neighbours (i.e. the others in the package), indicative of good design.

BASEVISUALWORKS is an industrial system. Including library classes, it contains 2022 classes distributed into fourteen subsystems. Of those subsystems, we chose to analyze in this example the subsystem `Collections`, which contains 228 classes distributed among eight packages. `Collections` is strategic because it contains the classes related to or supporting the *"collection"* programming structure.

See **Appendix B** (p.105) for more information about the case studies mentioned above.

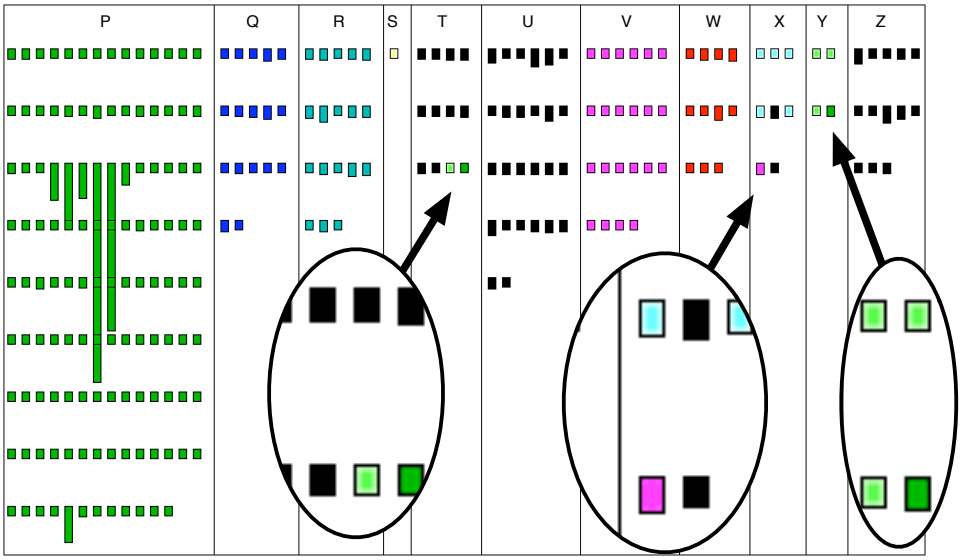

Figure 6.4: CODECRAWLER example. Uniformity of one color in most of the packages indicates high modularization. Only in three specific places there are potentially misplaced classes.

6.3.2 *Library*

Figure 6.5 (p.90) shows the locality of ALCHEMIST obtained using the contextual information. We observe a package bigger than the rest. Q contains 86 classes, which is more than half of the classes of the system. Most of the classes in this package are almost flat squares, indicating low coupling among classes in this package. As an exception, there is a tall red rectangle indicating a class with high coupling with its neighbours. The connections are either subclassing (this class being the root), or referencing (this class being referenced statically e.g., instantiated by the other classes in the package). The visualization shows that there is no evidence that moving its classes to another package it would increase the average locality of the system: all the classes painted in red are in Q. It could be, for instance, a big library.

6.3.3 *Newcomer*

The visualization of ALCHEMIST shows a close relation between packages `UI`, `Base`, and `ClientFrmrkInt`. In Figure 6.5 (p.90) the packages are highlighted in the bubble. ALCHEMIST has a layered architecture. Packages P and `Base` belong to the lowest layer,

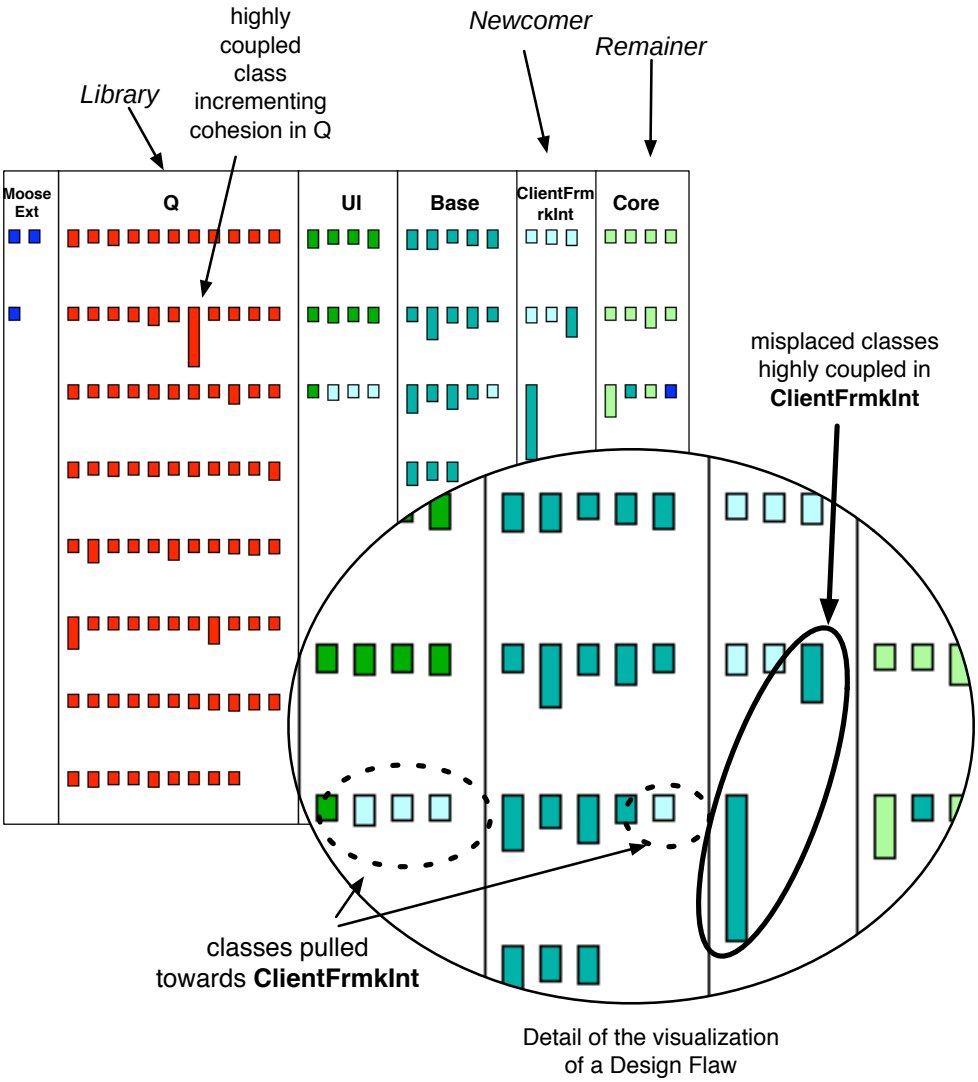

Figure 6.5: ALCHEMIST example. More than one color in a package indicates potential lack of locality. Size of smaller rectangles represent forces pulling the class towards the package.

and packages ClientFrmrkInt and UI in the layer immediately above. It is important to enforce the design decision of having Base in the lowest layer because Base contains classes used by most of the packages.

Close inspection to Base and ClientFrmrkInt in the visualization reveals a cyclic dependency: ClientFrmrkInt depends on Base, and Base depends on ClientFrmrkInt. Moreover, the small cyan (clear) rectangle in Base, indicate that ClientFrmrkInt consumes services from Base; whereas the large blue (dark) rectangles in ClientFrmrkInt are used by Base. Because we know that Base belongs to a lower layer than ClientFrmrkInt, observing Figure 6.5 (p.90) we learn that there is a cyclic dependency between two packages belonging to different layers.

It is true that this situation could be noticed also with simpler layouts, for example by marking explicitly the connections between packages with lines. But what our visualization shows that others do not show, is that there are forces pulling those cyan (clear) classes towards ClientFrmrkInt and pulling two classes in ClientFrmrkInt towards Base. Moreover, these classes in ClientFrmrkInt that are pulled away have higher coupling with the other classes defined in ClientFrmrkInt than the average of classes in ClientFrmrkInt (they are taller). Meaning that they are, therefore, happy to be defined in ClientFrmrkInt. We believe that the combination of these two factors indicates a design flaw.

Closer inspection of the code confirmed the violation of the layered architecture. As a consequence of the later addition ClientFrmrkInt, classes in Base became clients of those on ClientFrmrkInt, adding dependencies from a package in a lower layer (Base) towards a package in a higher layer (ClientFrmrkInt). This situation was a consequence of ClientFrmrkInt being a later add-on, and those foreign cyan (clear) classes in UI and Base being added only to satisfy needs of the new client, ClientFrmrkInt.

6.3.4 *Remainer*

Continuing with Figure 6.5 (p.90), we observe that in package Core, most of the classes have square shapes rather than rectangles, an indication of classes weakly coupled, revealing a potentially low-cohesive package from a traditional point of view. Besides, two of the lowest-coupled classes are pulled towards two different packages.

Inspection of the code indicated that they were obsolete classes, one implementing facilities to debug the code of a class defined in MooseExtensions (therefore its blue color), and the other an obsolete class statically referenced by the user interface defined in UI. Once the code was analyzed, it became clear that these classes were misplaced.

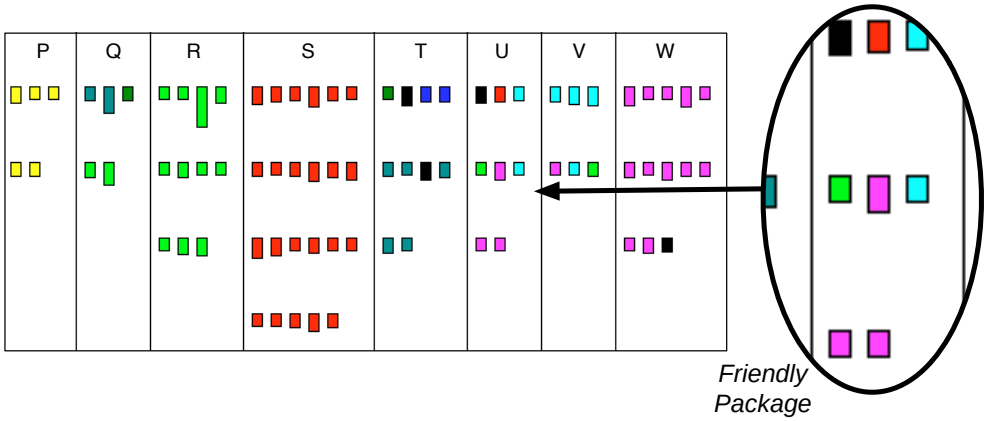

Figure 6.6: Collections Example. Package U shows extremely low locality. The distribution of its classes indicates it is completely anomalous, or a package with a special purpose.

6.3.5 *Friendly*

Figure 6.6 (p.92) shows the contextual information visualization of the `Collections` subsystem from BASEVISUALWORKS. We observe that in packages P, R and S all the classes have the same color, an indication of high locality in those packages. However, in packages Q, T, U, and W the rectangles representing classes have different colors, an indication of low locality. In summary, half of the packages in Collections have high locality and the other have poor locality.

The structure of package U is particular: the many colors of the rectangles in of U indicate that there are strong forces pulling its classes apart. This means that its classes are separately accessed by a number of packages of the subsystem, concretely by four out of the eight packages.

Packages T and W define three classes that seem to be pulled towards U (the little black rectangles). However, two of those classes are happy to be in T. Their size, being larger than the rest of the classes in T, indicates that they provide services to classes defined in T.

Observed with traditional approaches based only on internal attributes, the classes in U would expose low coupling among them, and U would indicate common low cohesion. Context information shows how exceptional U is. Its classes are so much distributed, that the visualization indicates an anomalous package or a package having a special purpose. It could indicate the existence of a design decision supporting the presence of U . With contextual information visualization showing this pattern, the developer

is encouraged to do more investigation before doing a possible dangerous refactoring. Analysis of code confirmed the uniqueness of U: it contains abstract classes which are root of inheritance hierarchies contained in the neighbor packages, and must therefore be loaded first.

6.4 Summary

The contribution of this chapter is a novel technique to analyze the locality of classes defined in packages in large object-oriented systems. To support the analysis of locality of classes into packages, we define a process and a visualization layout.

Traditional approaches ignore Contextual Information to convey insights about the organization of the classes into packages. This chapter shows examples where our technique detects misplaced classes from the way the classes are used. We have applied our technique to three real systems and showed how it facilitates their understanding.

We have found four visual patterns of locality, namely *Library*, *Newcomer*, *Remainer* and *Friendly*.

Chapter 7

Conclusions

Managing the organization of classes into packages is crucial, as the criterion by which these packages are built impacts in the maintainability and evolution of the system. In particular, developers cope with system complexity by defining related classes in the same package, since organizing classes into packages with high locality helps them to manage the propagation of changes.

We have reviewed various approaches used to manage the organization of classes into packages. Our research reveals that previous approaches typically focus on analysis of the content of the package, thus missing vital information about the context that surrounds the package. However, individual developers have different views of packages, depending on the developer's particular interest or perspective of the system to perform his specific task. These different perspectives of packages aggravate the process of organizing classes into packages. Based on our research in related fields, we have identified and specified several requirements to manage system modularization into packages: (a) characterization of the interactions between packages, (b) recognition of package perspectives, (c) combination of package properties, (d) different levels of abstraction and granularity.

Our solution is to manage the organization of packages by exploiting the way in which clients use packages. We model package-context as a first class entity and argue for the need to include contextual information in the analysis of classes, packages and systems. By this we do not mean to neglect the value of characterizing packages by their content, rather we complement such approaches. We enrich the perspective of a package by considering its contextual information. We found contextual information, in particular, client usage, to be useful because it provides us alternative views of classes and packages.

As a validation of our approach we have presented several analyses built on contextual information and client usage of packages:

Package Cohesion enriched with information of client usage. It shows cohesion in packages whose classes are *used* together, even when the classes contained in a package present few relationships binding them.

Detection strategies obtained from the combination of package contextual properties. They are rules to detect packages conforming to this rule. We have defined rules to detect complex, core and client packages, and together with our contextual cohesion, we identify flawed packages, and packages conforming to design issues.

Relativity of cohesion as a package may be marked as non cohesive in isolation, but in fact is cohesive when used by clients. A case in point are framework packages designed to be containers of hooks for clients.

We built *coarse and fine grained visualizations for package characterization* to provide both first-contact insight and detailed understanding of package interaction. Examples of these are our package polymetric views and butterfly blueprints respectively. Polymetric views show amount of coupling between packages, whereas butterfly blueprints show in detail, separated by type, the interaction between a package and its context. The visual patterns reveal the roles played by the packages for example whether it plays the role of a client, a provider or a combination of both.

Our *Visualization of system and subsystems* show butterfly blueprints grouped by the subsystem where they belong. We detect subsystems where its packages follow the same pattern. We observe the complete system in the same graphical view to facilitate comparisons.

Visualization of class locality. This novel visualization exploits the magnet metaphor, depicting forces attracting and repelling a class to its package. In the same graphical view, we observe the degree of services that a class provides to other classes defined in its package, what we refer to as the *happiness* of each class. This visualization reveals patterns indicating the distribution of locality, *i.e.,* classes to re-locate and their dependencies that are reasons for the class remain in the package where it is defined.

Semi-automatic management of class locality. We combine locality visualization and contextual cohesion to derive class re-locations. The recommended re-locations are obtained from the application of an optimization algorithm that is guided by a function based on contextual cohesion.

7.1 Discussion

Our approach supports management of the organization of classes into packages. The analysis tasks listed in the previous section show how our model satisfies the properties required to manage the organization of classes:

Characterization of package interaction. The entity package context holds dependencies that form the interaction between classes in a package and classes in the rest of the system. The techniques used in the analysis above use Contextual Information to provide detailed information that characterizes the package.

Recognition of package perspectives Every analysis is based on the assumption of a package being contained in a context, and described from a fixed perspective. Beyond the one to one relationship between a package and a package context explored in this dissertation, Contextual Information has the potential to identify different perspectives by allowing a package to have several package contexts (Chapter 3 (p.25)).

Combination of package properties Detailed characterization of package interaction need to combine properties of the packages. For example, butterfly views (Chapter 5 (p.61)) reveal the package role based on a combination of several dependency types.

Different levels of abstraction and granularity The idea of characterizing a software entity by how it is used by its clients should be applicable at different granularity levels. In particular, it should be applicable both at a class and at a package level. In the case of classes, we could consider the class as a container of attributes and methods. In the case of systems, we could consider the system as a container of packages. In this dissertation we have exploited Contextual Information to characterize entities of different levels of granularity, namely packages and classes. For example, butterfly blueprints (Chapter 5 (p.61)) describe subsystems by the packages that they contain, and the visualization of class happiness (Chapter 6 (p.83)) describes the client usage of a class by other classes in the package. The application of all the techniques presented in this dissertation to different levels of granularity is, however, an open issue.

7.2 Open Issues

Most of the novel techniques presented in this dissertation exploit dependencies between classes, which can be of type inherits, accesses, references, and sends. However,

beyond the types of dependencies explored in our research, there are further types of dependencies that connect packages, for example class extensions (a way to incrementally modify existing classes alternative to subclassing, mentioned in Chapter 2 (p.7)). We believe that by extending and refining the techniques presented in this work with new dependency types, there is potential to achieve a even more expressive means to characterize a system.

Furthermore, we have observed that computing cohesion derived from client usage requires a certain degree of modularity in the system. We would like to derive improved measures for contextual cohesion that overcome, at least in part, this shortcoming.

We have shown that packages offer different views according to the perspective under which they are observed, and we have mentioned the need to understand packages in their context, rather than characterizing it in an absolute way. However, our analysis does not exhaustively analyze the effect of every possible perspective in the package characterization. We believe that the introduction of package-context opens a new perspective on how to obtain realistic results characterizing systems. Old cohesion measures are too limited to detect cohesion in packages when classes belong together, but have few connections among them. We have observed that properties derived from the package together with the package's context are equally important to internal attributes of the package. As example, we put analysis of package cohesion, where a package considered non cohesive when observed in isolation proves to be cohesive when analyzed in its context, for instance used by its clients.

We have exploited client usage for packages and classes. However, we believe that our focus on client usage of packages and our explicit representation of this as an attribute to describe the package is applicable to different levels of granularity of a software system. At a coarse level of granularity, the system could be considered as the encompassing entity. A more fine-grained perspective could be obtained by applying our approach at the class level. As a result, there is potential to apply cohesion obtained from usage exploitation, detection strategies combining usage properties, and the visualizations explored in this research in different levels of granularity.

7.3 Lessons Learned

Our approach appears to violate the generally accepted image that cohesion has to be computed considering exclusively the contents of the package. We have learned that to capture the degree to which classes belong to a package, sometimes we need to step back and look at the package from the outside, rather than focusing on its contents.

During this work we have observed how relative it is to catalogue packages as cohesive or not cohesive, and that to attempt to classify software entities as “good” or “bad” is useless and naive. We acknowledge the different views of packages. Our approach helps a developer to understand object-oriented systems by allowing him to interactively investigate locality from his own perspective.

We have learned the importance of adopting a flexible approach to package analysis. Our complementary techniques help developers to get an overview of the system initially, and then to focus on crucial areas.

Searching to understand the complex net of interaction between software entities is a characterization in itself. Graphics views of those interactions reveal the role of entities. In fact, searching to understand those interactions revealed the model basis of our analysis, and solutions to the problem of managing program modularity.

We recognized the value of making the context of a package explicit. By having a package-context entity we were able to remove an excessive load of responsibilities on the entity representing a package, and to explicitly characterize a package by its interaction with its context. From an implementation point of view, we obtained a clearer distribution of responsibilities and simpler algorithms to answer our research questions about coupling.

Appendix A

Alchemist: Organizing Classes into Packages

We implemented our model to support analysis of contextual information in our tool called ALCHEMIST. Based on our meta model, ALCHEMIST is a package analysis and metrics tool that bridges the gap between the specification of our model, and the analyses presented in this dissertation. It offers a space to recognize packages perspectives, and to represent in a uniform way the structural and context-based properties of a package.

ALCHEMIST is based on the MOOSE reengineering environment. In this appendix, we show the position of ALCHEMIST in the reengineering environment.

A.1 Architectural Overview

ALCHEMIST is a package analysis tool built on a dynamic reengineering environment named MOOSE [Ducasse *et al.*, 2005a; Nierstrasz *et al.*, 2005]. MOOSE can store multiple models, providing the underlying foundation to hold multiple package contexts.

A meta-model describes the way the system can be represented. MOOSE is implements the FAMIX language independent meta-model [Demeyer *et al.*, 2001]. At the core of ALCHEMIST is the implementation of Contextual Information, which is an extension of FAMIX. In particular, this foundation of our model gave us the possibility, to re-locate classes without actually effecting the code. We can, therefore, analyze the locality of

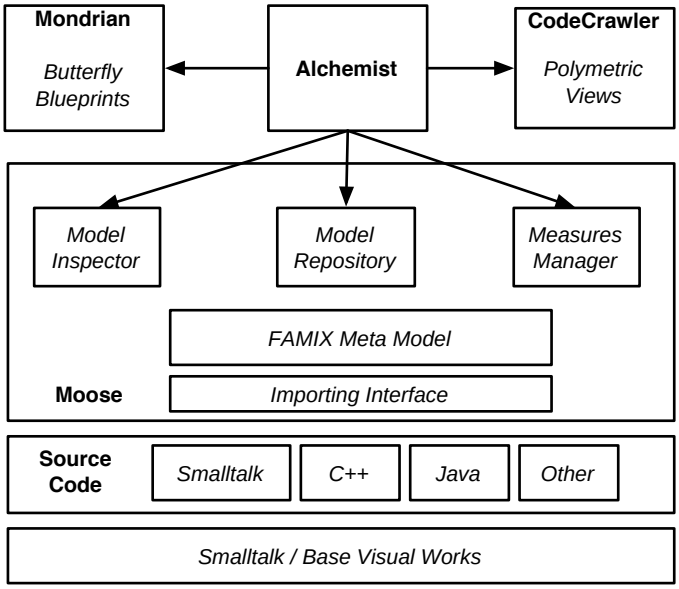

Figure A.1: Position of ALCHEMIST in the overall architecture of MOOSE. Different tools have been developed on top of MOOSE. ALCHEMIST, our analysis tool, uses some of those tools. It also accesses the model repository of MOOSE that stores multiple models of the analyzed systems. MOOSE imports source code written in different languages. ALCHEMIST extends the FAMIX meta-model with Contextual Information.

packages as *if* classes were re-located without the need to move a class from one package to another.

Figure A.1 (p.102) shows the position of ALCHEMIST in the overall architecture of the MOOSE reengineering environment. The systems to be analyzed can be implemented in different languages, for example Java, Smalltalk, and C++. An import/export interface imports the source code of the analyzed systems into the model repository of MOOSE. ALCHEMIST exploits basic analysis tools available in MOOSE and manipulates FAMIX objects representing objects of the real source code.

MOOSE and most of the tools built on top of it have been developed using the BASE-VISUALWORKS (**Appendix B** (p.105)), an environment to program in the Smalltalk programming language. In Smalltalk all objects are always available. Often during analysis we need to query specific objects. Smalltalk contributes to the success of the analyses by giving us the possibility to query objects in a flexible way.

A.2 Integrating ALCHEMIST

MOOSE makes the meta-model explicit, which facilitates the development of tools that cooperate. In particular, the meta-modeling framework offered by MOOSE gives us the possibility to combine software measures and visualizations offered by different tools. As mentioned before, ALCHEMIST is built on MOOSE. It is also integrated with other tools named CODECRAWLER and MONDRIAN.

ALCHEMIST uses CODECRAWLER to generate the Polymetric View (Chapter 5 (p.61)). Typically, CODECRAWLER maps entities to nodes and relationships to edges. In the case of ALCHEMIST, packages map to nodes and dependencies between packages map to edges. The nodes have different sizes, representing the values of different measures applied to packages.

We also use a tool named MONDRIAN [Meyer *et al.*, 2006] to build butterfly blueprints (see Chapter 5 (p.61)). In this case, the butterfly views are built as objects in ALCHEMIST, and handled to MONDRIAN together with the desired layout.

A.3 Summary

ALCHEMIST is a package analysis and metrics tool built on MOOSE, a reengineering environment. MOOSE facilitates the integration of tools by making the meta-model explicit. By default, MOOSE implements the FAMIX meta-model. ALCHEMIST extends the FAMIX meta-model with contextual information. Besides, several tools are built on top of MOOSE. ALCHEMIST integrates with two of those tools to achieve the analysis techniques explained in this dissertation.

Appendix B

Applications used in Case Studies

CODECRAWLER

description CODECRAWLER is a small software visualization tool [Lanza and Ducasse, 2003][1] CODECRAWLER relies on a graph model.

why is it interesting: It serves to illustrate examples in detail.

number of packages: 8

number of classes: 1402

number of LOC: 9088

BASEVISUALWORKS

description BASEVISUALWORKS [VisualWorks, 2003] is a large portion of the Cincom VisualWorks Smalltalk environment[2]. It is an industrial system, developed over the last 15 years. It defines all the runtime entities of a smalltalk environment (classes, methods, strings, characters, collections, graphical display, memory objects) but also the compiler framework, the coding tools (debugger, code browsers), the OS support and all the widgets offered by the graphical framework.

why is it interesting: It is a large industrial system

[1] See http://www.iam.unibe.ch/ scg/Research/CodeCrawler/ for more information.
[2] See http://www.cincomsmalltalk.com for more information.

number of packages: 91

number of classes: 9088

number of LOC: 262660

ALCHEMIST

why is it interesting: It's design decisions and implementation is well know to the author of this work.

number of packages: 8

number of classes:

number of LOC:

MOOSE

why is it interesting:

number of packages: 8

number of classes:

number of LOC:

ARGOUML

why is it interesting: ARGOUML is a UML design tool written in Java.

number of packages: 29

number of classes: 1095

number of LOC:

Bibliography

[Abadi and Cardelli, 1996] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

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

[Alpert *et al.*, 1998] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.

[Anquetil and Lethbridge, 1999] Nicolas Anquetil and Timothy Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*, pages 235–255, 150 Louis Pasteur, University of Ottawa, Ottawa, Canada, 1999.

[Basili, 1997] Victor Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[Berard, 1993] Edward V. Berard. *Essays On Object-Oriented Software Engineering*, volume 1. Prentice-Hall, 1993.

[Bergel, 2005] Alexandre Bergel. *Classboxes — Controlling Visibility of Class Extensions*. PhD thesis, University of Berne, November 2005.

[Bieman and Kang, 1995] J.M. Bieman and B.K. Kang. Cohesion and reuse in an object-oriented system. In *Proceedings ACM Symposium on Software Reusability*, April 1995.

[Bieman and Kang, 1998] J.M. Bieman and Byung-Kyoo Kang. Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):111–124, February 1998.

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

[Booch *et al.*, 1998] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1998. ISBN: 0-210-57168-4.

[Briand *et al.*, 1996] Lionel C. Briand, Sandro Morasca, and Victor Basili. Property-based software engineering measurement. *Transactions on Software Engineering*, 22(1):68–86, 1996.

[Briand *et al.*, 1998a] Lionel C. Briand, John W. Daly, Victor Porter, and Jürgen Wüst. A comprehensive empirical validation of product measures for object-oriented systems. Technical Report ISERN-98-07, Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern, Germany, 1998.

[Briand *et al.*, 1998b] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[Briand *et al.*, 1999] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[Bunge, 1974] Mario A. Bunge. *Treatise on basic philosophy*. Dordrecht: Reidel, 1974.

[Chidamber and Kemerer, 1991] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object oriented design. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 197–211, November 1991.

[Chidamber and Kemerer, 1994] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[Chuah and Eick, 1998] Mei C. Chuah and Stephen G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[Corbi, 1989] Thomas A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[Dekel, 2002] Uri Dekel. Applications of concept lattices to code inspection and review. Technical report, Department of Computer Science, Technion, 2002.

[Demeyer *et al.*, 2001] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[Demeyer *et al.*, 2002] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[Dey and Abowd,] Dey and Abowd. Towards a better understanding of context and context-awareness.

[D'Souza and Wills, 1999] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Addison Wesley, 1999.

[Ducasse *et al.*, 2004] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. A top-down program comprehension strategy for packages. Technical Report IAM-04-007, University of Berne, Institut of Applied Mathematics and Computer Sciences, 2004.

[Ducasse *et al.*, 2005a] Stéphane Ducasse, Tudor Gîrba, Michele Lanza, and Serge Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[Ducasse *et al.*, 2005b] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. IEEE Computer Society, 2005.

[Emerson, 1984] Thomas Emerson. A discriminant metric for module cohesion. In *Proceedings of the 7th International Conference on Software Engineering (ICSE)*, 1984.

[Feathers, 2005] Michael C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.

[Fenton and Pfleeger, 1996] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[Fenton *et al.*, 1994] Norman Fenton, Shari Lawrence Pfleeger, and Robert L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, (7):86–95, July 1994.

[Fowler *et al.*, 1999] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Fowler, 1997] Martin Fowler. *Analysis Patterns: Reusable Objects Models*. Addison Wesley, 1997.

[Gîrba, 2005] Tudor Gîrba. *Modeling History to Understand Software Evolution*. PhD thesis, University of Berne, Berne, November 2005.

[Hautus, 2002] E. Hautus. Inmproving Java software through package structure analysis. In *International Conference Software Engineering and Applications*, 2002.

[Henderson-Sellers, 1996] Brian Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[Hitz and Montazeri, 1995] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC '95)*, October 1995.

[Jain *et al.*, 1999] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, September 1999.

[JDepend, 2005] Jdepend, 2005.

[Johnson and Russo, 1991] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report 91-1696, UIUC DCS, May 1991.

[Koschke, 2000] Rainer Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[Lakhotia, 1993] A. Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings 15th ICSE*, pages 35–44, 1993.

[Langelier *et al.*, 2005] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE*, pages 214–223, 2005.

[Lanza and Ducasse, 2003] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, September 2003.

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

[Mancoridis and Mitchell, 1998] Spiros Mancoridis and Brian S. Mitchell. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.

[Mancoridis *et al.*, 1999] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[Marinescu, 2002] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Department of Computer Science, Politehnica University of Timişoara, 2002.

[Marinescu, 2004] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 350–359, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Martin, 1997] Robert C. Martin. Stability, 1997. An article about the interrelationships between large scale modules. Discusses the Stable Dependencies Principle and the Stable Abstractions Principle.

[Martin, 2000] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[McCabe, 1976] T.J. McCabe. A measure of complexity. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[Meyer *et al.*, 2006] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Monrian: An agile visualization framework, 2006. In submission.

[Mezini and Ostermann, 2002] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA 2002*, pages 52–67, November 2002.

[Mili *et al.*, 2002] Hafedh Mili, Ali Mili, Sherif Yacoub, and Edward Andy. *Reuse-Based Software Engineering*. John Wiley and Sons, 2002.

[Mišić, 2001] Vojislav B. Mišić. Cohesion is structural, coherence is functional: Different views, different measures. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS-01)*. IEEE, 2001.

[Morris, 1989] Kenneth Morris. Metrics for object-oriented software development environments. Master's thesis, Sloan School of Management. MIT, 1989.

[Myers, 1978] G. J. Myers. *Composite/Structured Design*. Van Nostrand Reinhold, 1978.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[Pintado and Tsichritzis, 1988] Xavier Pintado and Dennis Tsichritzis. An affinity browser. Active object environments, Centre Universitaire d'Informatique, University of Geneva, June 1988.

[Pintado, 1995] Xavier Pintado. The affinity browser. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 245–272. Prentice-Hall, 1995.

[Pinzger *et al.*, 2005] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[Ponisio and Nierstrasz, 2006a] Laura Ponisio and Oscar Nierstrasz. Using context information to re-architecture a system. In *Proceedings of the 3rd IEEE Software Measurement European Forum 2006 (SMEFS'06)*. IEEE Computer Society, 2006.

[Ponisio and Nierstrasz, 2006b] Laura Ponisio and Oscar Nierstrasz. Using contextual information to assess package cohesion. Technical Report IAM-06-002, University of Berne, Institut of Applied Mathematics and Computer Sciences, 2006.

[Pressman, 1994] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 1994.

[Quatrani, 1998] Terry Quatrani. *Visual Modeling with Rational Rose and UML*. Addison Wesley, 1998.

[Raţiu *et al.*, 2004] Daniel Raţiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 223–232, Los Alamitos CA, 2004. IEEE Computer Society.

[Raţiu, 2003] Daniel Raţiu. Time-based detection strategies. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, September 2003.

[Riel, 1996] Arthur Riel. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA, 1996.

[S. *et al.*, 1987] Kirkpatrick S., Gelatt C. D. Jr., and Vecchi M. P. Optimization by simulated annealing. In *Readings in computer vision: issues, problems, principles, and paradigms*, pages 606–615, 1987.

[S. Patel, 1992] R. Baxter S. Patel, W. Chu. A measure for composite module cohesion. In *Proceedings of the 14th International Conference on Software Engineering*, pages 38–48, 1992.

[Schilit *et al.*, 1994] B Schilit, N Adams, and R Want. Context-aware computing applications. In *First International Workshop on Mobile Computing Systems and Applications*, pages 85–90, 1994.

[Schwanke *et al.*, 1989] Robert W. Schwanke, Rita Z. Altucher, and Michael A. Platoff. Discovering, Visualizing, and Controlling Software Structure. In *Proceedings of International Workshop on Software Specification and Design*, pages 147–150. IEEE Computer Society Press, 1989.

[Schwanke, 1991] Robert W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[Sharble and Cohen, 1993] Robert C. Sharble and Samuel S. Cohen. The object-oriented brewery: a comparison of two object-oriented development methods. *SIGSOFT Softw. Eng. Notes*, 18(2):60–73, 1993.

[Sotograph, 2003] Sotograph, 2003. http://www.software-tomography.com/html/sotograph.htm.

[Stasko and Zhang, 2000] John T. Stasko and Eugene Zhang. Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In *INFOVIS*, pages 57–, 2000.

[Stasko *et al.*, 1998] John T. Stasko, John Domingue, Marc H. Brown, and Blaine A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[Stevens *et al.*, 1974] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[Tzerpos and Holt, 1996] Vassilios Tzerpos and Richard C. Holt. A hybrid process for recovering software architecture. In *CASCON*, page 38, 1996.

[Tzerpos and Holt, 1999] Vassilios Tzerpos and Rick Holt. MoJo: A distance metric for software clusterings. In *Proceedings Working Conference on Reverse Engineering (WCRE 1999)*, pages 187–195, Los Alamitos CA, 1999. IEEE Computer Society Press.

[Tzerpos and Holt, 2000] Vassilios Tzerpos and Richard C. Holt. Acdc: An algorithm for comprehension-driven clustering. In *WCRE*, pages 258–267, 2000.

[VisualWorks, 2003] Cincom Smalltalk, September 2003. http://www.cincom.com/scripts/smalltalk.dll/.

[Weyuker, 1988] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14(9):1357–1365, 1988.

[Wilde and Huitt, 1992] Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[Yourdon, 1979] Edward Yourdon. *Classics in Software Engineering*. Yourdon Press, 1979.

[Zenger, 2002] Matthias Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

[Zuse, 1990] Horst Zuse. *Software Complexity, Measures and Methods*. Walter De Gruyter, 1990.

Curriculum Vitae

Personal Information

Name:	María Laura Ponisio
Nationality:	Argentine
Date of Birth:	September 28, 1971
Place of Birth:	La Plata, Buenos Aires, Argentina

Education

2002 - 2006:	Ph.D. in Computer Science in the Software Composition Group, University of Berne, Switzerland - Subject of the Ph.D. thesis: "Exploiting Client Usage to Manage Program Modularity"
2001 - 2002:	Master of Science in Computer Science at the Vrije Universiteit Brussel, Belgium, in Collaboration with Ecole des Mines de Nantes, France - Subject of the Master thesis: "A Binding-Time Analysis for petitCafé"
1990 - 2001:	School of Computer Science, National University of La Plata, Buenos Aires, Argentina - Subject of the Licentiate thesis: "XML Patterns for the Design of Web Applications"
1985 - 1989:	Secondary School, Colegio Nacional, National University of La Plata, Buenos Aires, Argentina
1978 - 1984:	Primary School, Colegio Normal Nacional Nro. 2, La Plata, Buenos Aires, Argentina