REVERSE ENGINEERING OBJECT-ORIENTED DISTRIBUTED SYSTEMS

Teză destinată obţinerii
titlului ştiinţific de doctor inginer
la
Universitatea "Politehnica" din Timişoara
în domeniul ȘTIINȚA CALCULATOARELOR
de către

Ing. Dan-Călin Cosma

Conducător ştiinţific: prof.dr.ing. Ioan Jurca
Referenţi ştiinţifici: prof.dr.ing. Valentin Cristea
prof.dr.ing. Ioan Salomie
prof.dr.ing. Vladimir Creţu

Ziua susţinerii tezei: 18.05.2009

Seriile Teze de doctorat ale UPT sunt:

1. Automatică
2. Chimie
3. Energetică
4. Ingineria Chimică
5. Inginerie Civilă
6. Inginerie Electrică
7. Inginerie Electronică şi Telecomunicaţii
8. Inginerie Industrială
9. Inginerie Mecanică
10. Ştiinţa Calculatoarelor
11. Ştiinţa şi Ingineria Materialelor

Universitatea „Politehnica” din Timişoara a iniţiat seriile de mai sus în scopul diseminării expertizei, cunoştinţelor şi rezultatelor cercetărilor întreprinse în cadrul şcolii doctorale a universităţii. Seriile conţin, potrivit H.B.Ex.S Nr. 14 / 14.07.2006, tezele de doctorat susţinute în universitate începând cu 1 octombrie 2006.

România, 300159 Timişoara, Bd. Republicii 9,
tel. 0256 403823, fax. 0256 403221
e-mail: editura@edipol.upt.ro

"When we think we know something, that's precisely the moment when we should look deeper into the thing."

Frank Herbert

Foreword

The software industry is increasingly facing the issues of understanding and maintaining software systems, and the research community is continuously concerned with designing and developing tools and techniques to address them.

A significant number of the modern, large-scale software systems are designed as distributed applications. There are different types of distributed systems, and while they are usually implemented in an object-oriented fashion, their inherent nature implies particularities that raise very specific, technology-dependent, understandability and quality assessment challenges. Distributed systems are different from their 'locally-concerned' counterparts, and the differences have a dual nature: on one hand, they represent obstacles that limit the applicability of 'classic' understanding approaches, and on the other they constitute valuable sources of information for an approach built specifically for this class of systems. My doctoral thesis, which I present in this book, describes the methodology I developed for understanding object-oriented distributed systems through a reverse engineering process driven by the technological and domain-specific particularities of such systems. The approach starts with the source code of the application, and makes both system-wide and class-level characterizations, capturing an overview of the system's distributed architecture, providing detailed understanding of the system traits, and supporting its restructuring.

There are many people I'd like to thank for the help they gave me during the various stages related to the dissertation. As is customary, this is the point where I can start enumerating their names and the things I'm grateful for, so that they get the recognition their efforts deserve. As I am not very experienced in this kind of writing, I'd like to ask the reader to accept these paragraphs as they are, without assuming the existence of any hidden agenda, and, more important, I must apologize to those whose names I forgot to include.

I express my gratitude to Professor Ioan Jurca, who, besides being the scientific advisor for my doctoral studies, is one of the rare persons that are actually capable of *earning* their collaborators' appreciation and respect, rather than making them feel artificially obligated to do so. I have a very good professional relationship with Prof. Jurca, and I hope our collaboration will continue at least with the same efficiency as today. I thank Prof. Jurca for the continuous encouragement and support during the research and development of this thesis, and for the rewarding discussions we regularly have, on various interesting topics.

I would also like to thank the professors at the Department of Computer and Software Engineering, for their contribution to my development as software engineer, especially to Prof. Ioan Jurca, Prof. Vladimir Creţu, Prof. Horia Ciocârlie, Prof. Ştefan Holban, and Prof. Petru Eles.

Good teachers shape one's future – I will never forget the person that directly influenced my most important choices in high-school, Professor Adrian Niţă. His dedication and involvement in supervising the Informatics Lab (Cercul de Informatică) at the Emanuil Gojdu High School

in Oradea made me discover this wonderful field of Computers and Software, which became both my main hobby and the career choice. I am grateful for this, and I hope I'll eventually find the time to pay the old school a visit.

I also want to thank my friend Adi Takács, for being my best friend since the first year in High School. I only wish we could meet more often in Oradea or elsewhere, to discuss our favorite topics about Computers, Software and Universe.

I'd like to thank the Sava Technologies company in Timișoara for providing me with access to the source code of several software applications developed in-house, which I was able to use as test cases for the various phases of my analisys approach.

I must make a particular note regarding the research group I am part of (LRG), a fine selection of researchers that continuously provided a great environment to work in. I therefore thank my colleagues Cristina Marinescu, Petru Mihancea, and Mihai Balint, and I hope the high quality principles the group follows will continue to influence the development of our local scientific community.

I owe the most special gratitude to my friend and colleague, Radu Marinescu, without whose continuous encouragement and effort the research that lead to this dissertation would not have been possible. His invaluable feedback and selfless involvement was essential, and I will never forget the long and fruitful meetings that helped me both focus my ideas, and put them to good use. Thanks a lot, Radu!

I thank my parents for raising me well, and providing me with the proper perspective on the world and its remarkable features. I hope I'll be able to provide a similar environment to my daughter, too. I thank my sister Diana, for her guidance and for being a good friend. I also thank my dear wife Janina, for her love, support and encouragement, and for always being there for me.

Timișoara *Dan Cosma*
13.05.2009

Dan-Călin Cosma
Reverse Engineering Object-Oriented Distributed Systems

Teze de doctorat ale UPT, Seria 10, Nr. 18, Editura Politehnica, 2009, 154 pagini, 38 figuri, 3 tabele

ISSN: 1842-7707
ISBN: 978-973-625-884-8

Abstract: This dissertation presents the author's original approach in analyzing object-oriented distributed software systems through a process of reverse engineering. It describes a comprehensive methodology to assess the application's characteristics by exploring its structural and architectural traits. The process provides detailed system understanding, at both the system and class level, and presents support for restructuring. The approach is enriched by software visualization, and is supported by a comprehensive tool infrastructure.

Contents

1 Introduction **11**
- 1.1 Background . . . 11
- 1.2 Goals . . . 12
- 1.3 Approach . . . 13
- 1.4 Main Steps . . . 14
- 1.5 The Structure of the Thesis . . . 14

2 Distributed Software Engineering **17**
- 2.1 Reverse Engineering . . . 17
  - 2.1.1 Definition . . . 17
  - 2.1.2 Application . . . 19
- 2.2 Distributed Software Systems . . . 20
  - 2.2.1 Definition . . . 20
  - 2.2.2 Distributed Architectures . . . 22
  - 2.2.3 Communication Technologies . . . 26
  - 2.2.4 Component Deployment . . . 31

3 Analyzing and Understanding Software **33**
- 3.1 Clustering-Driven Architecture Reconstruction . . . 33
  - 3.1.1 Structural-Based Clustering . . . 34
  - 3.1.2 Semantic and Dynamic Clustering . . . 35
- 3.2 Design-Driven Approaches . . . 36
- 3.3 Concept Location . . . 36
- 3.4 Distributed Systems Analysis . . . 37
- 3.5 Visualization . . . 40
  - 3.5.1 Fisheye Views . . . 40
  - 3.5.2 Structural Representations . . . 41
  - 3.5.3 Dynamic and Semantic Sources . . . 41
  - 3.5.4 Metrics-Driven Visualization . . . 42
- 3.6 Reverse Engineering Distributed Software . . . 43

4 Representing Distributed Software for Analysis 47
4.1 Criteria for Understanding Distributed Software Systems . . . 48
4.2 Building Blocks for Capturing the Distributed Nature . . . 50
4.3 A Representation of a Distributed Software System . . . 56
4.3.1 Model Concepts . . . 56
4.3.2 Attributes and Relations . . . 59

5 Core System Analysis 63
5.1 Goals . . . 63
5.2 Initial System Representation . . . 66
5.2.1 Class Dependency Information . . . 68
5.3 Identifying the Frontier . . . 68
5.4 Building the Core . . . 70
5.4.1 The Algorithm for Isolating the Distributable Core . . . 71
5.5 Reviewing the Results . . . 74
5.6 Identifying the Distributable Features . . . 75
5.6.1 Technology-Aware Heuristics . . . 76
5.6.2 Cohesion-Based Clustering . . . 78
5.7 System-Level Understanding . . . 78
5.7.1 Identify the Remote Communication Channels . . . 79
5.8 Visualization - The Distributable Features View . . . 81
5.8.1 The Distributed Architecture Perspective . . . 81

6 Impact of Distribution 83
6.1 Acquaintance Classes and Metrics . . . 83
6.1.1 Bidirectional Coupling Metrics . . . 84
6.1.2 Acquaintance Metrics . . . 85
6.2 Interpretation . . . 88
6.2.1 Involvement in Distribution . . . 89
6.3 Patterns of Acquaintance . . . 90
6.3.1 Significant Feature Acquaintance . . . 90
6.3.2 Local Feature Contributor . . . 92
6.3.3 Connector Class . . . 93
6.4 Visualization . . . 94
6.4.1 The Feature Affiliation Perspective . . . 94
6.4.2 System-level Visual Assessment . . . 96
6.4.3 Visual Patterns of Acquaintance . . . 96

7 Restructuring Support 101
7.1 Criteria for Restructuring . . . 102
7.2 Restructuring the System . . . 103
7.2.1 Extraction Units . . . 104
7.2.2 Visualization as Support for Selection . . . 105
7.2.3 Extraction Process . . . 107
7.3 Improving the Understanding . . . 108

8 Evaluation of the Approach **111**
8.1 Experimental Setup . . . 112
8.2 System-Level Characterization . . . 114
8.2.1 Overview of the Distributed Architecture . . . 114
8.2.2 Characterization of the Distributed Awareness . . . 116
8.2.3 Visualization . . . 117
8.3 Patterns of Acquaintances . . . 118
8.3.1 Significant Feature Acquaintance . . . 121
8.3.2 Connector Class . . . 123
8.3.3 Visualization . . . 124
8.4 Restructuring Support . . . 125

9 Tool Support **129**
9.1 Tool Architecture . . . 129
9.2 System Representation . . . 131
9.3 Core Distributable Features Discovery . . . 131
9.4 System-Level Visualization . . . 132
9.5 The Distributable Features View . . . 132
9.6 Support for Restructuring . . . 133
9.7 Extensibility . . . 137

10 Conclusions **139**
10.1 The Methodology . . . 139
10.2 Conference Publication . . . 140
10.3 Contributions . . . 141
10.4 Future Work . . . 143

Bibliography **144**

List of Figures

2.1 Reverse engineering and the related terms (adapted from Chikofski and Cross II [10]) . . . 18
2.2 Client - Server . . . 23
2.3 Peer-to-peer . . . 24
2.4 Three-tier Architecture . . . 25
2.5 The TCP/IP stack . . . 27
2.6 Remote Method Invocation . . . 28
2.7 Java Message Service . . . 30

4.1 A general schematic of a Distributed System . . . 51
4.2 Application-Mediator interaction . . . 53
4.3 Main model concepts . . . 57
4.4 Model concepts - the service view . . . 59
4.5 Model overview . . . 60

5.1 Methodology for understanding distributed object-oriented systems . . . 65
5.2 The initial dependency graph . . . 67
5.3 Loosely-connected cohesive clusters . . . 79
5.4 Example of Distributed Architecture Perspective . . . 82

6.1 Computing the ADF metric . . . 88
6.2 Significant Acquaintance . . . 91
6.3 Local Feature Contributor . . . 92
6.4 Connector Class . . . 93
6.5 Example of Feature Affiliation . . . 95
6.6 Example *Big Color Spot* class . . . 97
6.7 A *Big Gray* class . . . 98
6.8 A *Color Spotted Gray* class . . . 99

7.1 In-group adequacy . . . 106

8.1 Distributed Architecture Perspective of the case-studies . . . 117
8.2 Overview of FWS . . . 119
8.3 Overview of EHCACHE . . . 120

8.4 Examples of *Big Color Spot* pattern 124
8.5 Two *Big Gray* classes 125
8.6 Two *Color Spotted Gray* classes 126
8.7 The shapes of the classes in the Workflow service group 127

9.1 The architecture of niSiDe 130
9.2 Reviewing core content 132
9.3 System overview 133
9.4 The structure of feature cores 134
9.5 A Distributable Features View 135
9.6 Extraction example 136

1

Introduction

The continuous development of the society, along with the emergence of new technologies and communication infrastructures have paved the way for the wide adoption of a class of software systems that became more and more important in the everyday life: the distributed applications. From programs that provide direct, person-to-person communication via the Internet to complex systems that manage the numerous resources of large organizations dispersed over the world, distributed software has become essential for the functioning and growth of the civilization.

1.1 Background

Distributed applications are basically systems that consist of several, sometimes many, software components, each running at a different geographic location, collaborating to each other to fulfill a common goal. The components are involved in complex interactions, and are interested in both the communication with their remote counterparts, and in providing functionalities that are specific to the local environment they run on. Consequently, they are inherently complex, and their development and maintenance are tasks that imply thorough preparation and elaborate approaches in all the phases related to the systems' life cycle. Moreover, they work in inherently dynamic environments, as they serve communities that evolve themselves. Therefore, to a higher degree than in the case of classic software applications, the requirements they must address change at a fast pace. For example, it is frequent the case when a distributed software system must scale up significantly to serve a suddenly larger or more demanding client base, or when the resources it processes have grown significantly in size or number. Another usual change scenario is when the system must cope with the evolution of the organization it supports, by providing the new features it needs, or by adapting its existing functionality to better fit the new characteristics of the environment.

In an ideal world, these challenges are addressed and overcome by designing the systems so that they easily cope with change, and by ensuring that their architecture is flexible enough to allow for both the easy integration of new functionalities and the modification of the existing ones. While the software engineering community continually develops new techniques to approach this goal, their experience shows that such a system is hard to develop, and it does

not exist in the real world. The reason is that the requirement changes are so diverse, and depend on so many factors – some of them unexpected –, that their prediction is simply not possible to make, at least not in a sufficient degree. This opens the way for software engineers to adopt a more pragmatic, and at the same time cost-effective perspective, that of working on the existing applications, analyze them, and develop strategies to change and improve their design and functionalities. Besides circumventing the quixotic nature of a 'perfect' design, the insight on the existing applications presents several important advantages. First, adapting the old applications to fit new requirements will provide the users with a familiar environment that supports their needs in a way they were accustomed to, rather than forcing them to change their habits to follow the particularities of a new system. Next, a change in an already established application may be sometimes easier and faster to make than developing from scratch a new solution. Moreover, such an enterprise is often cheaper than investing in the initiation of a separate software system, which may add a lot of new and unexpected issues that were not encountered with the original application. Finally, the continuous restructuring and change of operational software applications is a natural part of their life cycle, determined by the correlation of feedback and refinement that is characteristic to all the dynamic and productive long-term enterprises.

The most important part in approaching existing software systems is the exploration that provides knowledge about their properties. This knowledge is not always available per se, even when the software is maintained by the same organization that developed it. Teams of programmers change or move on to different projects, and the information about a system that was developed not so long time ago can be lost to a significant degree. Moreover, a frequent case is when the software must be approached by parties that had nothing to do with its development, as the maintainers are often not working for the same company or department that developed the system. While they may be provided with documentation and initial data about the system, there are numerous instances when this information proves inaccurate or incomplete, as it describes an outdated version or it simply failed to capture all its relevant attributes. Consequently, in order to be able to restructure or transform the application, the main thing engineers need is to understand the system in depth, to extract its characteristics through a dedicated process of analysis.

1.2 Goals

This dissertation presents our approach in analyzing distributed object-oriented software systems through a process of reverse engineering that starts with the application's source code. We define and develop a comprehensive methodology that involves a series of automatic tasks that assess the application's characteristics by exploring its structural and architectural traits. At the same time, the experience of the engineer is held in high regard, by allowing the user interaction at all the critical steps of the pursuit. The procedure is driven by the goal of understanding the main characteristic that provides for the system's distinctiveness, that of being a distributed application. The impact of the distribution-specific functional traits on the system entities is analyzed, so that an in-depth comprehension of the system is achieved. Moreover, the approach integrates support for the basic restructuring of the application by consistently using the same concepts and techniques that produced the system understanding. This way, the task of restructuring comes as a natural continuation of the analysis enterprise,

rather than necessitating additional preparation and effort in its undertaking.

The methodology aims to inspect the software system in an efficient way, focused on the important parts of the application, and to minimize the computational power needed for exploring large systems. Therefore, one of the traits of the approach is that, at a core step in the analysis, it selects only a core set of entities that are representative for the system's distribution-related functionality [21]. The value of this technique is that it helps the efficiency when applying computationally-intensive algorithms, and it allows the engineer to focus on a small set of highly relevant objects in order to inspect them closer, even with a manual approach if necessary.

The approach strives to understand in depth the distributed functionality of a software system without detecting the distributed components in their entirety, and without using deployment information, as this information is usually scarce. This enables the analysis to obtain essential and sufficient knowledge of the system by only looking at the source code of the application.

1.3 Approach

The technology of the software communication infrastructure the application relies on is used by the process of understanding as a very important source of information about the system. As we need to assess the system's distribution-related functionality, the points where the application communicates over the network are the parts of the system that provide the first relevant knowledge. Considering the communication technology helps the identification of these parts of the application, as they are directly influenced by the constraints the technology usually imposes. As we will show during this work, the constraints are often detectable as source code patterns, and can provide us with methods of isolating the relevant entities. While built to be extensible to other types of technologies, our approach is applied on Java RMI applications.

One of the main concepts we define in this dissertation deal with the impact of the distribution-related *features* on the structure of the system. Our previous experience with distributed systems [18, 17, 106] made us interested in the importance of the features provided by a distributed system for shaping the application's characteristics. In the early stages of our research, we have explored the way the features (as a concept) can be used as basic blocks for building the software [16], and how they can be employed to design complex behaviors (as code migration). The author of this dissertation has also supervised a diploma project that developed a prototype framework as a proof of concept [7]. In the research we present in this thesis, we have approached the importance of the distributed features from an entirely new perspective: they are the elements that provide essential knowledge about a distributed system. The way the system entities participate in providing distributed features is indicative of the fundamental nature of the system.

The feature participation assessments, as well as the other types of system evaluations are done using a set of measurements based on software metrics we developed specifically for this purpose [21]. The numerical representations are based on calculating the strength of dependencies between the application classes, and the relations these dependencies reveal when considering the distributed features.

The restructuring support included in our approach is based on a technique that allows the engineer to experiment with scenarios of structural modification on the different parts of the system. The scenarios can be performed and their impact measured and evaluated, in an iterative and repetitive process that provides a basis for the selection of the best projected restructuring solution.

The outcome of the analysis approach is the understanding of the main system properties, to a degree that provides the engineer with extensive control on the analyzed system. It includes assessments related to the distributed architecture, the structure of the distribution-aware functionalities in the system, and the relation between the system entities and the distributed and local features.

To support the understanding and facilitate the identification of the various patterns that provide knowledge about the application, the entire analysis approach (including the restructuring support) is enriched with a set of software visualization techniques [20]. They present the attributes of the system entities so that the important patterns to be identified visually, as a method for driving the process of selecting their relevant traits.

The methodology is supported in all its phases by a complete tool support [19], that we developed and used when evaluating the approach. The tool infrastructure provides all the automatic tasks implied by the various activities related to the analysis, while always allowing the user to evaluate and validate the intermediary results, as well as to control the parameters of the algorithms that are employed during the assessment of the distributed application's characteristics.

1.4 Main Steps

The methodology we describe in the following chapters consists of several parts, driven by a set of specific goals:

- Define a model for distributed object-oriented software applications tailored for the goal of software understanding;
- Extract the core distribution-related functionality of the system and isolate the distinct distributed features in the application;
- Provide a view on the system's distributed architecture;
- Assess the impact of the distribution aspect on the system's overall design;
- Understand the patterns of collaboration between the system entities (classes) in respect to the distributed and local functionality;
- Provide support for restructuring the application.

1.5 The Structure of the Thesis

The next chapters in this thesis are organized as follows: Chapter 2 presents the **main concepts** involved in reverse engineering distributed software, and Chapter 3 analyzes the **state of the art** in the field.

As a prerequisite for our methodology, in Chapter 4 we introduce **a representation for object oriented distributed systems** that we have built to support the system understanding, as well as the considerations that stay at its base.

The actual **methodology, its evaluation through case studies, and the tool infrastructure** we have developed are described in the next five chapters. They should be regarded as a whole, as they are steps that describe a single, unitary approach. Chapter 5 describes the details involved in the core system analysis, including the isolation of the system's distributed features, and the architecture-related assessments. Chapter 6 presents the approach that evaluates the impact of distribution on the system's features, and the extraction of the relevant patterns of collaboration, while Chapter 7 presents the restructuring support provided by the methodology. Chapter 8 describes the methodology evaluation we have conducted through case studies, and Chapter 9 presents the tool support we have developed.

The presentation ends in Chapter 10 with a set of **conclusions and the list of contributions** of our work to the field of reverse engineering distributed software applications. The same chapter analyzes the **potential future developments** that will enrich and continue our work.

2

Distributed Software Engineering

As presented in the Introduction, this dissertation describes our approach in analyzing distributed object-oriented software systems. For this purpose, we define a methodology of reverse engineering tailored to this particularly important class of systems, by emphasizing the fact that they present particularities strongly influencing any analysis that targets system understanding.

This chapter presents the basic concepts related to the field of reverse engineering software systems, then analyzes several aspects that are specific to the distributed applications. The distribution-related characteristics, highly dependent on the communication technology, can be used by the analysis to extract the relevant knowledge that drives the process of understanding by providing helpful hints on the parts of the system directly responsible for the distributed functionality. Moreover, a synthesis of the different types of architectures specific to this field is important when interpreting the architectural information recovered by the reverse engineering process, by placing it in the correct context that describes the system's distribution awareness.

2.1 Reverse Engineering

2.1.1 Definition

The Merriam-Webster Collegiate Dictionary [77] defines the verb "reverse engineer" as *"to disassemble and examine or analyze in detail (as a product or device) to discover the concepts involved in manufacture usually in order to produce something similar"*.

The definition has definite origins in hardware, and relates to the more 'traditional' industries such as manufacturing or electronics. Reverse engineering is used since the beginnings of technology, both to improve the products of the same company, or to understand the inner workings of those made by an external, even competing party.

Software reverse engineering is a relatively newer concern, and it basically refers to two types of enterprises:

1. Disassembling the binary form of a software application to understand how it works, what features does it provide, and so on. One of the most famous cases of reverse engineering binary code was the disassembly of the IBM PC BIOS (about 1983) to develop

hardware clones of the machines that basically started the era of personal computers. The knowledge gathered through reverse engineering was employed in building a detailed specification of software features and behavior. The specification was then used in an independent development process that produced a BIOS program from scratch, making it legal to include in the IBM PC compatible machines.

2. Analyzing the source code of a software system in order to understand it. It may imply activities such as the recovery of its architecture, the application redocumentation, and so on, and it is strongly related to the fields of software maintenance and reengineering [10].

We consider the above distinction to point out the different emphases used by the two approaches. The first one is arguably closer in concept to the original meaning of the term, as its main concern is to disassemble a finished product and to understand its internals in order to replicate or improve. It inherently implies that the target object belongs to a different party, and the emphasis is placed on the process that transforms the machine-specific executable format to a representation (such as a program) that is analyzable by a human engineer. On the other hand, the second approach considers that the program itself is already available, thus placing the context of reverse engineering closer to the aspects related to the maintenance and evolution of software.

For the purpose of this dissertation, we are using the latter interpretation when we address the issue of reverse engineering. We are interested in the techniques, processes and methodologies that aim to extract as much information as possible about a software system by analyzing its source code, which is considered to be available in its entirety to the analysts.

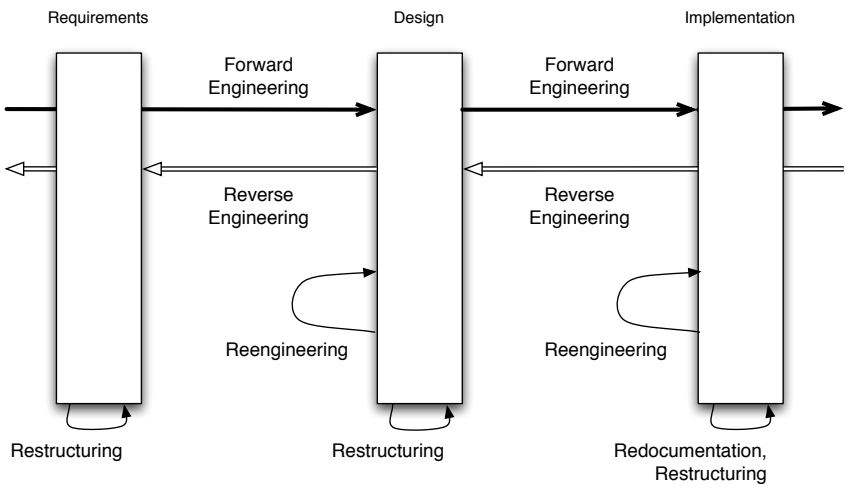

Figure 2.1: Reverse engineering and the related terms (adapted from Chikofski and Cross II [10])

Regarding the terminology related to the field of reverse engineering, we adhere to the widely-accepted taxonomy introduced by Chikofski and Cross II in [10]. They define the main

terms involved, as follows (Figure 2.1):

1. **Forward engineering.** "The traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."

2. **Reverse engineering.** "The process that analyzes a subject system to

   - identify the system's components and their interrelationships, and
   - create representations of the system in another form or at a higher level of abstraction"

3. **Restructuring.** "The transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior."

4. **Reengineering.** "The examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

2.1.2 Application

The various approaches to reverse engineering use different types of techniques in order to extract information about the system. Some may be purely static, by only analyzing the software in its inactive form, while others may make extensive use of dynamic information, by analyzing the runtime interactions between the system components, or by approaching the running software as a black box in order to detect its functional traits.

In our case, we adopted a static approach, and the entire methodology we present is based on analyzing the system as it is, without the need to run it in the process. This has a few advantages. First, it does not have to rely on particular tools or environments that capture and trace events at runtime, thus making it platform-independent. Moreover, it can be applied to systems that cannot be started in execution for various reasons such as the unavailability of the needed libraries or runtime environments, the lack of the necessary processing power to dynamically analyze complex systems, or even because the system itself is missing auxiliary items needed when running (such as configuration data, deployment information or appropriate production environments).

Scenarios for reverse engineering are very frequently encountered in modern software development -related contexts. The need to understand a system whose source code is readily available is justified by various reasons, related to the development process or to the inherently complex nature of modern software applications. It may be useful both to external parties, and to the original developers of the respective software system, and may be caused by several issues, such as the following:

- Legacy software. This is the case when a software system is inherited, is received from a different company, or the team that originally developed it is no longer available. The system needs to be understood so that modifications that improve its performance or add new features are performed, or in order to build new systems with the same specifications.

- Outdated documentation. While the team of developers may be (largely) the same as when the software was first built, the development process failed to document the software so as to describe it accurately and to date. There are frequent cases when the documentation is heavily outdated, and the many changes make the software too complex for the developers to track its status.

- Inaccurate documentation. This is a variation of the above case, when the documentation of the software proves inadvertently ambiguous or inaccurate, and may present misleading facts about the system's architecture, behavior or purpose. A reverse engineering process is in this case necessary to correct the mistakes, and to synchronize the documentation with the actual state of the software.

- Insufficient documentation. This is the case when the documentation, while accurate and to date, is insufficient to understand particular aspects of the system. The respective aspects may have been outside of the original documentation's focus for various reasons, but became nevertheless important at an ulterior stage in the software evolution. A reverse engineering process is needed to build new or different views on the software's characteristics, so that the unaddressed issues are covered.

- Design changes. This case is directly related to the issue of software aging [85], where the current design of the application no longer meets the initial system goals. A reverse engineering approach would help the developers or maintainers to identify the problems that made the system change in time and the places where the modifications occurred, so that the necessary corrections are made.

The outcome of the reverse engineering process varies with the approaches, and depends on the reasons that made it needed in the first place. The most frequent result is a representation that describes, at a certain degree of detail, the architecture of the application. The process may also generate new views on the system's high-level structure, and can provide insights into the lower-level details of the interdependencies between the system entities. The roles of the different entities can also be assessed by an analysis, and the features provided by the application can be discovered. Other types of results include updates to the existing documentation, system specifications that can be used in a new development, and so on.

The process of reverse engineering can be conducted by different parties, depending on its purpose: the original developers, the engineers that must maintain the system, various types of auditors, or programmers that must continue developing the system without being involved from the start. During the presentation in this thesis, we will use the term *engineer* to describe any human actor that may be involved at a stage or another in the process.

2.2 Distributed Software Systems

2.2.1 Definition

Referring to a larger context that includes both hardware and software, A.S. Tanenbaum defines a distributed system as *"a collection of independent computers that appears to its users as a single, coherent system"*[108]. The various components of the system are dispersed

at different geographical locations, and act together to fulfill the same goals. The intercomunicating hardware nodes are connected to each other by a large variety of communication infrastructures, and many of the architectures implied are heterogenous in nature.

The continuous evolution of the society lewd to the current state of human activities, where organizations, companies or even communities depend on distributed infrastructures to communicate, collaborate and work. The wide variety of interdependencies between the components of our civilization, and their impact in creating and maintaining highly integrated networks of inter-related activities need to be supported by equally complex and developed technological solutions. Therefore, modern distributed systems could not serve their purpose without advanced and elaborate software applications to provide the diverse functionalities specific to each of the different types of problems addressed by the distributed contexts. The distributed software systems are not simple conduits for the casual communication between parties, they must be designed to address the complex issues of intermediating and coordinating the distributed interaction.

A distributed *software* system can be defined and represented as *a set of an arbitrary number of processing elements running at different locations, interconnected by a communication system* [114]. The processing elements are relatively independent software components that run on different hardware nodes and communicate to each other so that the design requirements are met. The communication is done via an infrastructure, implemented itself in software, that provides services for transporting data between the remote locations, coordinating asynchronous events, managing the concurrent interactions between the system components, and so on. The infrastructure can be represented by various layers of software, some being distributed systems themselves: services implemented in the operating systems, specialized middleware such as object request brokers [103, 94, 39], message-centered communication providers [82], application servers that provide distribution services ([81]), etc.

The different software nodes usually act as independent entities. That is, an important part of their activity is concerned with providing local features, such as interacting with the user or performing tasks that process local resources. The communication with the other components over the network is done only at the moments it is needed to maintain the overall system functionality, and many distributed components tend to minimize the communication as it usually implies significant costs.

Nevertheless, there are systems that consist of components that heavily communicate to each other during the entire running order of the application. They are either deployed on infrastructures able to provide high bandwidth and low latency for the data transfer, or the communication is done in short bursts that do not overload the network.

The aspects related to the communication and the system-wide inter-dependencies between the application entities are specific to its *distributed architecture*. The distributed architecture of the system is a view on a system's structural characteristics that present the relations that are established over the communication infrastructure. It specifies the communication channels, the roles of the components that communicate remotely, and it usually ignores the other architectural traits that were defined at the design time.

Consequently, extracting the distributed architecture of a system is not always the same as the general architecture recovery enterprises that treat the application without considering its distributed nature. Indeed, applying such a process to a distributed software system ends by providing architectural views on the system, by analyzing the various types of dependencies

between the system entities. However, as the process did not analyze the nature of these dependencies in relation to the distribution-aware function of the application, it may miss the most significant trait of the application, the fact that it is distributed and the way the distribution impacts its design.

For example, if we were to analyze a simple chat system from a point of view that is not interested in its distributed aspect, the analysis will find that the application is one heavily concerned with providing a way for the user to enter text data via an interactive window, storing the text content when the user chooses to, checking the spelling of the phrases as they are typed, and managing the user/password information for authentication purposes. Dependencies on other instances of the chat component may be detected, however, they will not tell us anything about the fact that the communication is done remotely and the main concern of the application is the sending and receiving the messages, rather than storing them locally. In this respect, the application wouldn't seem that different from a rather simple text editor.

Understanding the importance of the application's nature – as distributed, in our case – leads to the analysis of two core aspects of this class of systems: their type of distributed architecture, and the technology that is used for communication.

We will analyze these characteristics in the following subsections , emphasizing the main issues that are involved, and the most common realizations of the architectural and technological traits.

2.2.2 Distributed Architectures

There are several types (patterns) of distributed architectures that are adopted by most applications that are made of components communicating over the network. A single application can be designed as following exactly one of these 'classic' architectures, or – most frequently – it employs a combination of the basic types, components having different architectural roles according to the purposes of the system.

2.2.2.1 Client-Server

The most common, widely-used distributed architecture is the client-server one (Figure 2.2). It implies the existence of two types of components:

- **Server.** Usually large, this component is designed to provide a set of software *services* that are available to the remote parties. The services are described in a manner dependent on the particular communication technology, and may be more or less related (functionally) to each other. This type of component implies a significant amount of *centralization* in the system, as it is the single place that implements complex functionality.

- **Client.** This component is traditionally designed as lightweight, and its main purpose is to use the services provided by a server. By doing so, it employs a usually limited functionality that processes the data received from its counterpart, and in most cases it presents it to the user.

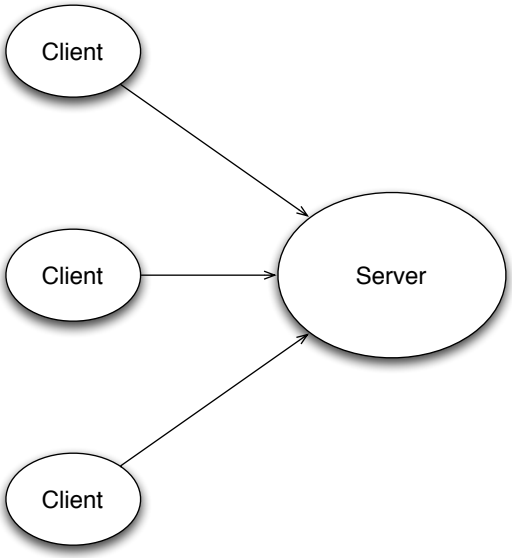

Figure 2.2: Client - Server

In a client-server architecture, the usual deployment scenario implies a single server (or few ones) and a comparatively higher number of clients. The centralization specific to this architecture can create several problems at the runtime, such as the following:

- The communication to the single server can become too intense, especially when the number of clients is large. This may cause high traffic to the server's network node, which in turn may create service availability problems.

- The server may act as a bottleneck in the system that limits the system's performance and usability. Depending on the server's efficiency in providing the services, and on the computational power provided by the hardware and software environment it relies on, this may become a significant problem, especially when the system must be scaled up to serve large numbers of clients

- The limited number of servers (usually only one) may present problems for the system availability. As the entire application functionality depends on the functioning of the server, a software or hardware issue that stops it from working effectively renders the entire application inoperable.

- The maintenance of a centralized server can be difficult, especially when it provides many, loosely-related functionalities. Servers can become too big and too complex, and the restructuring or reengineering needs that may arise at a point in the system's lifetime can imply significant engineering problems.

Modern distributed application developments tend to overcome these problems by adopting other architectures, by distributing the services to several independent servers, or by designing the servers to be easily scaled or split into several entities if necessary.

2.2.2.2 Peer-to-Peer

In this architecture (Figure 2.3), the components of the distributed system have equally important roles. Each component (a *peer*) can provide services for others to use, and, at the same time can use the services provided by the others. In a way, a peer acts successively as server and client, according to the system needs at a stage or the other in its functioning.

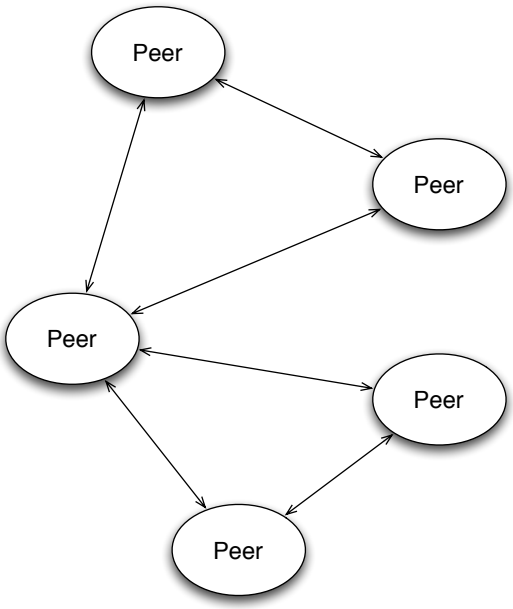

Figure 2.3: Peer-to-peer

The fundamental trait of this architecture is that the components are balanced, and there usually is no centralized point in the system. This effectively counteracts the disadvantages of the server-client architecture, by providing a way of designing highly scalable and flexible distributed software applications.

However, the architecture has an important problem, and it is related to its very basic principle of balancing the components. The fact that there is no central point, may create difficulties in the process of the component discovery at the runtime. That is, the system must implement strategies that make the already running components of the application reachable by the newer ones. This is done by including the component deployment information in each node of the application, or – more frequently – by using a server-based functionality to register and locate the running components. The latter method affects in a degree the balanced nature

of the system, but as it is involved only on a single, very specific point in the component's lifetime, it is usually accepted as a minor compromise.

2.2.2.3 Three-Tier

This architecture (Figure 2.4) describes the application as consisting of three logically-distinct sets of functionalities. Each tier consists of system entities with a very specific role, and the architecture is an extension of the server-client architecture, as the dependencies between the tiers are unidirectional. That is, the third tier acts as server for the second one, while the second provides services to the entities in the first tier.

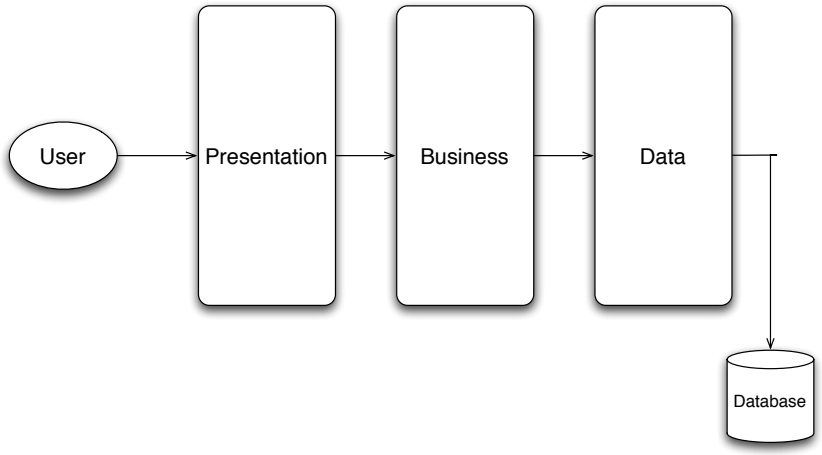

Figure 2.4: Three-tier Architecture

The architecture defines three tiers. In the software-specific context, they can also be referred as functionally-distinct software *layers*:

1. **Presentation.** The entities in this layer are responsible for the functionality that interacts with the user. They are lightweight software components that present data to the user (via an interface) and allow the user control and data validation.

2. **Logic or Business.** This layer is responsible for the implementation of the main functionality of the application, such as the algorithms that process data, and provide the features requested by the user.

3. **Data.** The entities in this layer are responsible for managing the persistent data objects manipulated by the business layer. It usually implies a database that stores the information, and consists of the software entities that model the stored data (as object-oriented data abstractions, for instance).

2.2.2.4 Multiple Tiers

This is an extension of the three-tier architecture, and refers to the case where several layers are involved in providing the software functionality, as more that three logical functionalities are considered necessary for the design.

The most frequent implementation is when the user services are provided by more than one application. For example, this is the case when a web application forwards the user requests to a three-tiered enterprise application, specialized to provide the requested feature.

2.2.2.5 Other architectures

The design of distributed software applications is not limited to the above types of architectures. The designers may use a combination of the "mainstream" approaches, and can also consider the specific issues required by the system's functionality. For example, if a system must consist of several equally important peers, at a point in its functioning, there may occur the need of electing a component as temporary leader. For this purpose, the requirements of the chosen leader election algorithm have to be taken into consideration, and the components can be – for instance – designed as connected in a synchronous ring, to fulfill the respective algorithm's constraints [65].

Moreover, components can be connected to each other in more complex ways, such as trees or even generalized graphs of inter-communicating entities that act as clients and servers to each other at the same time.

2.2.3 Communication Technologies

In order to communicate remotely, distributed applications depend on specialized infrastructures that provide the lower-level functionality of sending or receiving data over the network, in a synchronous or asynchronous manner, depending to the system requirements.

The infrastructures distributed software systems use for communicating are important in that they directly influence the way the distributed architecture is implemented. Some technologies imply a specific architecture, while others allow several or even unlimited architectural choices. More importantly, the communication technology usually imposes a set of specific constrains on the development of the application, that are frequently in the form of specific constructs in the applications source code. This makes the technology-related information highly relevant in an analysis, for detecting the distribution-related concerns within a software system.

We describe briefly the most frequently used communication technologies, and their main characteristics.

2.2.3.1 Protocol stacks

The protocol stacks [107] describe facilities included in the modern operating systems that provide support for network communication to the application level.

The design of a protocol stack is dependent on a chosen reference model that represents the software architecture of the parts of the operating system providing the network services. A stack describes several layers of abstraction, each introducing protocols that deal with a

particular problem related to the data transmission and reception. The most frequent case of such a design is the ubiquitous *TCP/IP stack*, present in virtually all current operating systems capable of providing network connectivity.

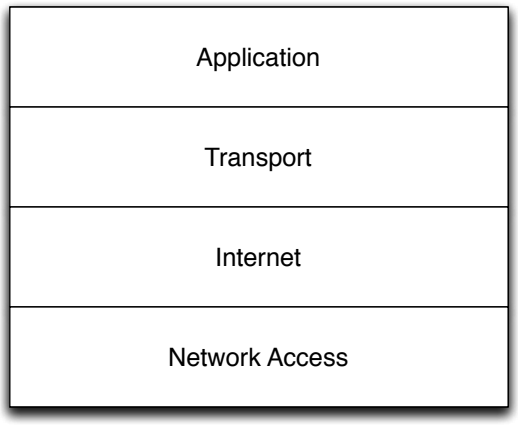

Figure 2.5: The TCP/IP stack

The TCP/IP stack (Figure 2.5) is based on a respective TCP/IP layered model. It consists on four layers, with the following roles:

- **The Network Access Layer** deals with all the details concerning the transmission of the data provided by the superior layer, its encoding and the transmission of datagrams to a remote host.
- **The Internet Layer** represents the network as a set of interconnected sub-networks, and deals with the routing of data from a host to another. This layer defines the *IP address* that identifies the network interfaces and is used when routing the information.
- **The Transport Layer** defines protocols that deal with the higher-level concerns in transmitting data, such as maintaining the communication channels, error control, fragmentation and sequence of data arrival. There are two specific protocols at this layer, one being connection-oriented (TCP), and one connectionless (UDP).
- **The Application Layer** describes protocols used by the applications, such as FTP (File Transfer Protocol), HTTP (Hypertext Transfer Protocol), SMTP (Simple Mail Transfer Protocol), and so on.

The TCP/IP stack is accessible from the application through primitives specific to the operating system and to the programming language. Such a primitive is the one known as the *socket*, with is widely used implementation as a *BSD Socket*. In UNIX, sockets are implemented (from the application's perspective) as descriptors that can be created and used by the programs. The language-specific libraries provide specific functions that create and manipulate these primitives, which in turn call the correspondent operating system services.

As the functions or methods that are called by programs when creating and configuring the sockets are very specific, an analysis can easily detect the lines of code that access the sockets. This way, an approach is provided with a valuable mean for isolating the code fragments that are responsible with the network communication, thus being directly related to the distributed aspect in the application.

2.2.3.2 Remote procedures or methods

This technology places itself at a higher level of abstraction than the direct usage of sockets. The purpose is to provide the programmers with communication-related services that are used in programs in the a natural way, specific to the programming language.

The chosen strategy is to allow the program to call procedures remotely, that is procedures that reside on a program running at a different location. The most representative case is the *Remote Procedure Calls* technology [105] specific to the UNIX [104, 98, 15] environments. Similar mechanisms are employed by more complex middleware systems, such as CORBA [90, 39].

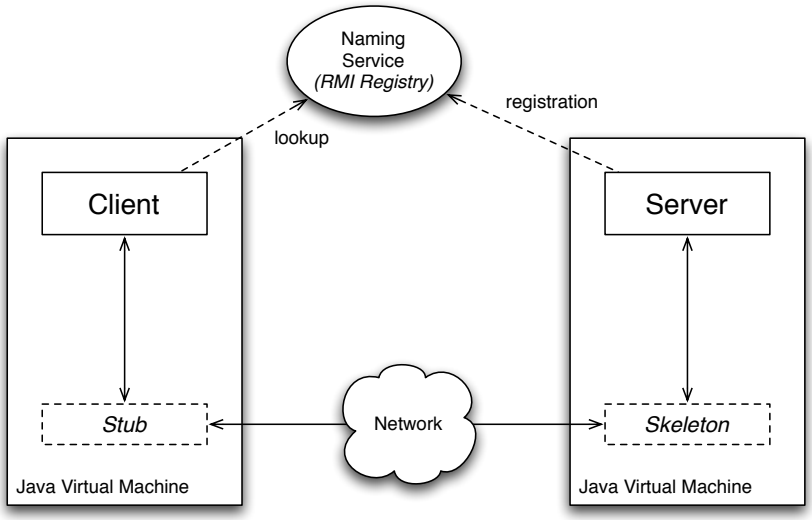

Figure 2.6: Remote Method Invocation

In the object-oriented context, this technology translates in invoking methods of objects instantiated at a remote location. One of the most representatives technologies is the *Remote Method Invocation (RMI)*, specific to the Java environment [94, 39, 18]. In RMI (Figure 2.6), an application that wants to communicate via the network must follow a set of specific requirements related to the implementation:

- The methods that need to be made available remotely must be gathered in special Java interfaces called *remote interfaces*. Each remote interface must extend the

`java.rmi.Remote` interface provided by the standard Java packages, as a marker for the methods' network-aware functionality.

- The actual implementations of the methods are done in server classes that implement the remote interfaces, either directly, or via a hierarchy of interface extensions. The server classes must either inherit a RMI-specific provided class (called `UnicastRemoteObject`), or explicitly call a static method in that class that exports its functionality to the network.

- The client classes use the server functionality by referring the remote interfaces. They do not have direct dependencies with the server classes; instead, the Java virtual machine-specific implementation will hide the actual communication and provide the clients with *remote references* to the servers. In the earlier versions of Java, this implied the automatic generation of a set of infrastructure classes called *stubs*, and *skeletons* which represented the actual channels of communication between the parties.

- The server classes can register themselves to a naming service, so that they are discoverable by the clients. Both actions imply specific calls of RMI-provided methods.

The usage of RMI in applications is detectable by an automatic analysis by isolating the specific code constructs it implies. For instance, the server classes can be isolated as being the classes that implement remote interfaces, which in turn are easily identifiable as extending `java.rmi.Remote`. Clients are the entities that directly refer the remote interfaces, and the network-aware actions are the calls to the methods these interfaces describe.

2.2.3.3 Messaging systems

Services for communicating via the network can be also provided by systems that may be implemented as applications on their own. An example of such applications are the message-oriented service providers, which are complex systems dealing with the creation, manipulation, and persistence of application-defined messages. Applications can connect to the messaging systems and use them as intermediaries that send or receive messages; components of the application can send a message, and other components may receive them, all details of the actual sending or receiving being dealt with by the infrastructure.

An example of messaging infrastructure is *Java Message Service (JMS* [83]. As it is, JMS is not an actual system, but a specification of messaging systems that vendors may implement in order to provide standardized message-oriented communication in the Java environment. A JMS application consists of the following system actors (Figure 2.7) :

- **Clients**, that are the entities that create, receive and send messages;

- **Messages**, used in communication, with a content defined by the application;

- **JMS Provider**, the actual implementation of the messaging system.

The JMS messaging infrastructure must provide the following set of communication-related primitives, that can be used by the applications depending on the type of messaging they need:

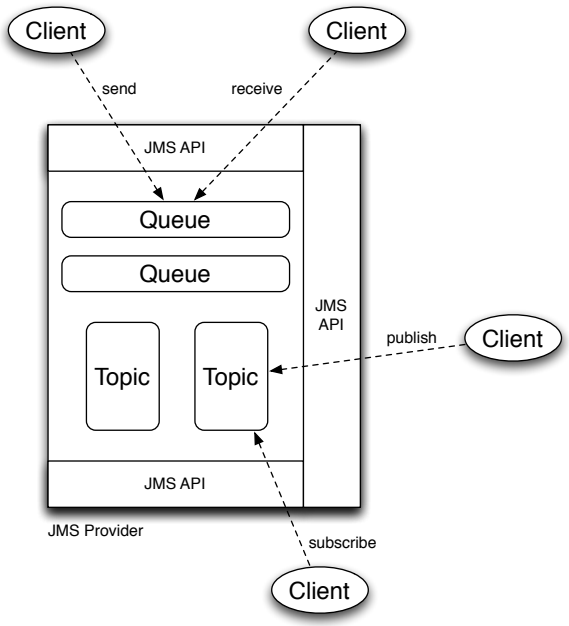

Figure 2.7: Java Message Service

- A **Connection Factory**, which is used to create the connection between the client application and the messaging system
- A set of **JMS Destinations**, representing the resources the clients access, and that deal with the message manipulation. JMS defines two types of destinations, as support for the two major paradigms in message-oriented systems:
  - *Point-to-Point*, represented with message queues available for the clients to enter or extract messages;
  - *Publish-Subscribe*, represented by the so-called *Topic* destinations, that can be used by clients for publishing messages or subscribing to various message types.

The usage of JMS is visible in the source code through the specific references to the JMS destinations, and the method calls that send or receive, publish or subscribe to messages. Again, an analysis can make use of these constructs to identify the parts of the application that are interested in the distribution-related system functionality.

2.2.3.4 Application Servers

A different class of technologies is represented by the *Application Servers*. An application server is a software environment that manages the application by providing it with a set of specific, high-level services. The application is deployed *inside* the application server, and is

heavily dependent on it. The entire development of the application must follow strict rules in both its design and implementation, and the deployment-specific configurations.

A very popular platform that employs an application server is *Java2 Enterprise Edition (J2EE)*, with its *Enterprise Java Beans* technology [9]. The applications that use this environment must be built using the three-tier architecture, and the technology provides means for creating the different types of entities, and services for distribution, persistence or transactions. EJB defines the following types of application entities, called *enterprise beans*:

- **Entity beans**, placed at the data layer, representing the entities that model in an object-oriented approach the data stored in a database;
- **Session beans**, places at the business layer, implement the logic functionality in the system. They can be of two types:
  - *Stateful*, that are able to maintain their internal state between the client calls,
  - *Stateless*, that do not maintain the state and are consequently lightweight in comparison with their counterparts;
- **Message-driven beans**, specialized entities able to subscribe and react to messages in a JMS environment, so that asynchronous application behavior can be implemented.

The clients at the presentation layer are normal Java applications (classes) that connect using specific techniques to the enterprise beans. The constraints the technology imposes on the application are very strong, and a complex analysis can be developed to detect various characteristics related to the system, including the distributed functionalities, and the various architectural traits. As significant parts of the structural information are available through specific, uniform descriptors, important data can be extracted automatically, thus encouraging the development of a flexible and feasible reverse engineering process.

2.2.4 Component Deployment

One of the important aspects that must be taken into consideration when designing or analyzing a distributed software system is the information that specifies the way the different components of the system are dispersed over the network. The *deployment information* is sometimes vital to understand the core characteristics of the system, as it is directly related with its distributed architecture. The deployment information is specified in various ways, and it is highly dependent on the communication technology the system relies on. The information can specify one or more of the following attributes of the distributed application:

- The network-specific address of each system component. This is a specification that shows which parts of the system (modules, packages, sets of classes) are deployed on which hardware nodes in the network.
- The relation between the component deployed in a node and the other local entities that may be related to the system, such as libraries, databases, application containers and so on.

- The dependencies between a component and its other remote counterparts that belong to the distributed application. Sometimes, configuration information specifies which are the addresses of the remote parties, or which are the component's "neighbors" (nodes to communicate with) in the particular distributed architecture of the application.

As it depends on the concepts specific to the communication technology (addresses, ports, service names, etc.), the deployment data is different from an application to another. It can also include parts that are dependent on the application design, by containing data that refers to the particular system entities or their runtime environment (such as component or service names, name service identification information, and so on). Moreover, the presence of the deployment information is not mandatory for a system design, as there are many cases when the deployment is done differently from an installation instance to another, to fit the particular needs of the users. Consequently, the deployment information, while important to understand the system, cannot be an item to rely on when building an approach for analyzing distributed software systems. Therefore, a reverse engineering process must be developed so that even when the deployment data is not present, the system understanding is achieved.

The approach we have developed and present in this dissertation addresses this problem by implementing a technique for extracting the system's distributed architecture characteristics without relying on the deployment information, and using only the information that is always available, the source code of the application.

3

Analyzing and Understanding Software

The issue of analyzing software systems is often being addressed by the software engineering research community. Identifying the structure of the application is central to many approaches, and usually involves finding the functionally-distinct components or partitions within the code.

This chapter discusses the state of the art in the field of reverse engineering software systems through a selection of the approaches showing the most interesting traits in gathering the information that provides system understanding. The selection is driven by assessing the core techniques applied by the different researchers, and the ways the existing methodologies make use of the information sources that provide knowledge about the system.

The main concern in any approach that aims to understand a software system by analyzing its source code is to capture the architectural traits in the application. Systems are usually large, and the main architectural components must be detected in order to focus the analysis, and to isolate the different types of functionalities. The field of software architecture reconstruction [89] is therefore highly important, and the approaches vary with the techniques used for decomposition.

3.1 Clustering-Driven Architecture Reconstruction

The vast majority of approaches that target the reconstruction of software architectures use at one stage or another in the process a specifically-built version of clustering techniques. A comprehensive survey and classification of clustering techniques and algorithms is presented by Wiggerts [113].

Systems are usually represented as graphs, and the detection of the clusters drives the process of separating the groups of system entities (source files, modules, classes, etc.) that are related to each other according to the specific criterion the approach considers important for its purpose. The separation is done with the main goal of partitioning the system so that the engineer that needs to understand it be able to focus only on some of the partitions when analyzing the system. Many approaches have the clustering technique at their core, and the

main focus is on identifying the best similarity measures that are used for grouping the entities together, and on improving the performance of the clustering algorithms.

3.1.1 Structural-Based Clustering

The majority of approaches view the software system as a graph of entities linked together by their relative structural interdependencies. For example, a system can be modeled as a set of vertices, each vertex representing a source file, while the edges are the usage dependencies between the functions or methods in the respective file. The system graph is built using manual or automatic techniques, then a clustering algorithm is applied to separate the system into several groups of entities.

R. Koschke [52, 53] proposes a unification of 23 techniques based on clustering to produce a classification of component recovery approaches. Moreover, he presents a semi-automatic method of analysis aimed to overcome the issue of the insufficient detection quality of such techniques. The method works incrementally [51], and combines improved versions of the techniques, integrated through an intermediate representation [55]. The techniques are run successively and are validated by the user. The work also includes a metric-based technique called *Similarity Clustering*, which is evaluated along with other techniques, using an evaluation scheme that measures the recall and precision of the analyzed component recovery techniques. Moreover, in [54] Koschke and Eisenbarth propose a framework for experimentally evaluating clustering techniques.

Clustering is also used by Andreopoulos et. al. [1] to produce decompositions of large software systems. They choose an approach that enriches a previously developed clustering algorithm [2] by considering both static and dynamic information. The approach takes into consideration the multi-layered structure of the analyzed applications, which is regarded as a very frequent case of modern software systems. The dynamic information is used by the algorithm which associates weights with the dependencies between the application files and incorporates them into the clustering.

Chiricota et. al. [11] propose an efficient clustering algorithm that separates related clusters of software within an application represented as a graph. The method is based on computing the strength of an edge in a graph by only considering the neighborhood of the two vertices.

The main assumption the approach is built on refers to the fact that software systems are usually organized in cohesive clusters of entities that are loosely connected to each other. The assumption is present in extensive research [68, 69, 3, 79, 80, 63], and it captures an important property of the software systems.

The goal of the algorithm in [11] is to detect the *weak edges* in the graph representing the software system.

The weak edges are, in fact, the connections that link the cohesive groups of entities, and their detection is of significant importance when isolating them from one another. The algorithm defines the strength measure by considering the edges that originate in the two vertices of a given edge. The metric is calculated using the ratio of the number of paths (cycles) of length three or four that exist between the two vertices, in relation to the maximal number of such paths.

The algorithm is applied to all the edges in the graph, and the ones with the lower

strength values are considered as weak, thus as possible links between different clusters of software entities.

This algorithm proved useful in our approach, where we adapted it to help the detection of code-level dependencies that artificially linked groups of classes that represented distinct functionalities in the target object-oriented distributed applications. By 'artificial', we mean those dependencies that were not related to the interactions relevant to our approach, being rather trivial in comparison. An example of such a dependency is the relation two components may have with a same, utilitarian, library: it links them together (they both need, for instance, to convert weight units), but it does not expresses a main architectural trait.

Brian S. Mitchell [78] proposes an approach to extract architectural information from the source code via a clustering-driven process. The system is represented as a Module Dependency Graph (MDG), and a series of search-based algorithms are applied to detect the functional clusters within the code. The approach works by randomly generating partitions of the graph and calculating fitness functions for them, evaluating the quality of the respective generated cluster. The clustering is supported by meta-heuristic search algorithms so that the quality of the generated partitions be high. Hill-climbing and genetic algorithms are involved. The result of the process is a set of clusters that represent subsystems in the analyzed application.

3.1.2 Semantic and Dynamic Clustering

Despite the fact that many approaches consider the structural relations established between system entities, this doesn't always provide all the necessary data about the system. For this reason, researchers sometimes look for alternate sources of information, and tackle the problem of recovering the architecture from different perspectives.

Kuhn et al. [56] focus on a type of information often ignored by software analyses, the semantics inherently present in the code artifacts such as comments or identifiers. They use *Latent Semantic Indexing* [22, 67, 70, 71] to analyze the spread of the relevant terms through the source code, and use clustering to group the entities that use similar terms. The approach is complemented by structural analysis, so that the technique can be applied to applications at different levels (to classes, methods, etc.).

Bauer and Trifu [4] address the issue in clustering-based decomposing of software systems where the clustering techniques are exclusively based on assessing syntactic dependencies rather than considering higher-level semantic data. They propose a method of gathering information about the architectural traits of the application by searching for architectural clues that in turn collaborate to suggest architectural patterns in the system. The high-level information obtained this way is therefore used to calculate a similarity measure which is applied in a clustering algorithm that produces the decomposition.

Xiao and Tzerpos [116] approach the problem of software clustering from a different perspective. Instead of considering statical information about the system, they perform the clustering using as a similarity measure the *dynamic dependencies* between system entities. After applying the approach on a large open source application, they conclude at the end of the paper that the consideration of dynamic information when performing the clustering is an interesting research topic, and can provide useful information about the systems.

3.2 Design-Driven Approaches

Some approaches focus on analyzing the system architecture in relation to the known traits of the system. They assume some external information about the system does exist, and it refers to the various roles of the design elements in the system (such as architecture requirements, functional roles of the system entities, etc.)

Schmerl et al. [102] address the issue of determining whether an application's actual architecture is the same as it was originally designed. They aim to provide an approach applicable to a large class of systems, that dynamically discovers the architecture of the applications by analyzing them at the runtime. For this purpose, they develop a framework that facilitates the mapping between the implementation styles to the architectural styles. The mapping is used at runtime and is able to help detecting patterns that show whether the various actions of the system represent "architecturally significant" activities, *i.e.*, provide information from which architectural knowledge can be extracted. They have developed a tool called *DiscoTect* [117], that feeds on captured and filtered running events and produces architectural information to build incrementally the recovered architecture.

Deursen et al. [111] synthesize their experience by presenting the *Symphony* process of evaluating the actual system design (and the impact of its prospective future evolution) by reconstructing the architecture. The approach is based on designing the viewpoints that are needed for understanding a particular system, and extracting the corresponding views from the system itself. The process is iterative, and involves concept analysis extracted from discussions with the interested parties, mapping the hypothesized views to the system-extracted ones, and gathering and interpreting the information from the actual system by applying established techniques.

Christl et al. [12] improve the *Reflexion Model technique* [84] of mapping source-level models to specified or hypothesized high-level representations of the systems. The goal is achieved by enriching the method with automatic clustering techniques to support the user in the mapping process. The result is a semi-automatic process that provides better results than the original method.

Jakobac et al. [45, 44] propose a user-guided approach that analyzes software systems by separating the architectural concerns to facilitate understanding. The concerns in the system are observed from two perspectives: the *purpose view* assessing the *processing*, *data*, and *connection* roles of the architectural elements [87], and the *usage view* that describes what parts of the system are shared and which are exclusive to individual entities. The analysis uses the available sets of clues specific to the application domain to label the different types of entities, and propagate the labeling to the related parts of the system to support the understanding of the application's architecture.

3.3 Concept Location

Extensive work related with the domain of software comprehension has been done in the field of *concept location* [92].

Eisenbarth et al. [31, 6] propose an approach that aims for the identification of selected features in the source code of a software system. They only focus on the features that are considered relevant for the system analyst, and start the identification by describing and

performing the *scenarios* that make the respective feature manifest itself. The scenarios are runtime sequences performed by the user to invoke a particular needed feature. The approach uses concept analysis [35] and both static and dynamic analysis to create *feature-unit maps* that describe which system entities implement a set of features considered important.

Salah et al. [100] describe an approach for comprehending large software systems based on dynamic analysis. The process extracts different views of the system (use case, module interaction and class interaction) to support the location of features within the application. The analysis is done by performing on the system a set of scenarios extracted from the use cases, and analyzing the execution traces. The approach is different from Eisenbarth's in that the features are identified during the user's interaction with the system, therefore eliminating the need to end the program after each scenario. Moreover, the approach is capable to analyze larger software applications, and generates different types of views [101, 99].

Edwards et. al. [28] use dynamic techniques based on causal ordering of events to address the feature location in distributed software systems, while Poshyvanyk et. al. [91] use a combination of Latent Semantic Indexing and scenario-based probabilistic ranking of events to identify the features in the source code.

3.4 Distributed Systems Analysis

This section gathers several approaches specifically built to analyze distributed software that we consider representative for the issues we discuss in this dissertation. The approaches analyze different types of systems, and combine various reverse engineering techniques and sources of information to provide system understanding or specific architecture recovery.

Mendonça and Kramer propose in [74, 75, 76] an approach for recovering distributed applications. They apply the work to systems built as C/C++ projects in the UNIX environment, and analyze them starting with the source code.

The approach is static, and combines several techniques for architecture recovery, focusing on identifying the executable components and their potential interaction at runtime.

The first technique is called *component module classification* and aims to find which are the compilation modules that make each of the executable components in the system, and classifies them as exclusive or shared, depending on the number of executables that use them.

The next technique in the approach is called *syntactic pattern matching*, built for specifying and executing queries on a syntactic representation of the source code, in order to identify the typical interactions between the system components. The patterns are described using a library of Prolog predicates, and are applied on an abstract syntax tree representing a program. Usages of the queries include the identification of basic program-specific patterns, such as assignment expressions and subroutine calls, along with any combination that describe a more complex code pattern. For example, the authors present the description and usage of specific queries that provide identification for the socket creation patterns in C, at both the client and server side, patterns for shell invocation or process creation, and so on.

The third technique, *structural reachability analysis* is used to determine which components use the various runtime features encapsulated in the shared modules in an application. For this purpose it builds an activation graph that models the activation units in the system (functions, methods, etc.) and the relations between them (as being the function calls or method invocations), and computes the transitive closures of the graph .

Pinzger et al. [88] propose an approach that extracts information from three-tiered distributed software applications based on the COM/COM+ component framework.

The goal of their techniques is to use the extracted data about the system so that the understanding of the respective application is supported. They have developed a semi-automatic approach that investigates the architectural characteristics of the analyzed system by understanding the attributed of the COM+ components. They build a model that describes the various encountered aspects, such as the persistence, security, transactions and error handling. Moreover, they analyze the dependencies that are established between the components, so that they extract the relevant information.

The abstract model is built using several sources related to the target application. They use the IDL definitions in the source code to understand the descriptions of the components, and search for the particular COM+ statements that indicate architectural characteristics, such as those that are specific to handling transactions. The process uses the COM+ API to extract the type library information that contains data about the interfaces provided by the various components, thus avoiding the need to parse IDL files that, as the authors remark, are not always available for analysis. Moreover they use the COM+ registry data to estimate the architectural characteristics specific to the deployment of the components. To identify the components in the application they start with the presentation layer, that represents the client to the rest of the application. By analyzing its code, and searching for instantiation statements, they determine all COM+ components it uses. After assessing the interface definitions from the meta-data stored in the type library, the configuration information is processed to extract data about the transaction semantics and security settings specific to the deployment of the application.

The approach is similar to our case, especially as it is highly aware of the technology-related constraints, by . following the major architectural patterns implied by the COM+ framework, and using them to detect the information that provides the system understanding.

Li and Tahvildari [60] propose a service-oriented componentization framework for systems written in Java. The purpose of their approach is to process an existing Java software system, and transform it into a service-oriented one to support component reuse. The developed framework supports the identification of the business services that exist in the application, and transforms each identified service in a separate, independent software component. The initial application's architecture is thereby transformed by the approach to become service-oriented, that can be used and deployed as such.

The process consists of several phases: the architecture recovery, the service identification, the component generation and the actual system transformation. The approach models the system as a graph by representing the classes and interfaces as vertices, and labeling them with their name and package information. The edges represent the relations between the classes or interfaces, such as inheritance, realization, aggregation, association, usage, and composition.

Regarding the system's architecture, they distinguish between the so-called *top-level services*, as those that are not used internally by other services, and the low-level services that comprise of functionalities that collaborate to form a top-level service.

The process uses a combination of top-down and bottom-up procedures to identify the services. The top-down technique heavily relies on applying graph processing to identify the various inter-related system components. The bottom-up approach describes an algorithm that iteratively agglomerates the atomic services that are highly related to each other to

obtain at each iteration a higher granularity service. The user is presented each intermediary result, and can decide the termination or the continuation of the process. The final stages encapsulate the resulting top-level services and their related low-level ones into self-contained components that follow a service-oriented architecture.

As it uses graph processing, the approach resembles to some degree our methodology in what regards the separation of the distinct features. Nevertheless, their purpose is different *i.e.*, to transform the object-oriented design of the application into a service-oriented one, and it does not focus on the distributed aspects in the system.

Han et al. [40] describe an approach that aims to reconstruct the software architecture for J2EE web applications. They use the Siemens Four Views approach [43, 13] to separate the architecture into four views: conceptual, module, execution and code architecture.

The approach consists of an iterative process that analyzes the views in a non-sequential order. The first step considers the code architecture and execution views to extract basic understanding about the structure of the system, such as the directory structure of the system files, their probable functionality, the runtime dependencies and so on. The next step describes the module view by understanding the static relationships between the entities in the application. They consider all the classes, JSP files and other source entities, and the basic dependencies between them. The next phases of the process focus on the conceptual view to modify the already extracted information about the modules. Then, a second pass on the execution and code views adds the new information regarding the mapping between the modules and the entities in these views.

The approach focuses in particular on the dependencies between the different entities in the application. The authors distinguish between the usage dependencies (such as method invocations) and the other relations, described as "knows" dependencies (instantiations of classes without calling the methods of the respective class, objects that are received just to be passed to another entity, etc.). The approach is entirely manual, though the possibilities of automating the process are analyzed in detail.

Di Lucca et al. [62] analyze the limitations in web applications comprehension approaches that render the applications in visual representations. According to the authors, the respective techniques are incomplete, as the size of the modern web applications is usually too big for the representation to be manageable by the user. The proposed solution uses a set of clustering techniques, along with a coupling-based measure to produce a hierarchy of clustering. The metric is specifically built considering the target system type (web applications), and therefore it is based on intuitive assessments on the characteristics of the respective system. Thus, the typology and the topology aspects of the dependencies are considered, and the manner in which they produce different types of coupling between the components is analyzed. They weigh the importance of each type of connection in the strength of collaborations between components, so that the metric to ensure a proper capture of the domain-specific traits.

Like the metric, the clustering algorithm is specifically built for the web applications field. It is based on an agglomerative approach, that starts with individual items (such as web pages) and groups them in clusters, iteratively. Once applied, the algorithm provides a hierarchy of clustering, each level in the hierarchy containing a set of system entity clusters. The selection of the appropriate level in the hierarchy that provides the most relevant set of clusters is based on specific quality measurements. The paper defines a *quality of a clustering* metric that is used for this purpose: the clustering that exhibits the maximum value for this metric

is considered the best candidate partition of the application components. Consequently, the value is used to find the cut-height in the already obtained hierarchy of clustering for the optimum set of identified clusters.

Ricca and Tonella [96] propose an interesting approach aimed at identifying the static Web pages in a site that are best candidates to be transformed in dynamic versions of themselves. The approach supports the migration from entirely static web sites to sites that are built using web application techniques.

The process identifies the common structure of the pages, by isolating the parts that are variable, that change from one instance to another while the rest of the page remains the same. The differences are cataloged and entered into a database, and a special script generates automatically the migrated pages.

The main technique uses clustering to group the similar pages in the web site. The clustering is based on a similarity metric that tries to identify pages which share the same template, and may be candidates for transformation in a dynamic counterpart.

The algorithm is agglomerative, and it is based on the Johnson's algorithm described in [113]. It produces a hierarchy of clustering that is then processed by a technique that increments the cut-height starting from the level of individual pages (height 0), and stops when the current level contains a cluster having a number of pages greater or equal to a specified threshold value. The respective set of clusters (at the corresponding height in the hierarchy) is then selected as the entities to be transformed in dynamic pages.

The actual transformation consists of several phases, such as the extraction of the templates, and the generation of the dynamic information to be inserted in a database. The process is semiautomated, and the user interaction is specifically important when recognizing the templates in the pages that were clustered at the different levels in the hierarchy, and for refining the final templates and the information that is stored in the database.

In the same line of supporting the analysis, maintenance and development of web sites and applications, the authors also use a visualization technique [95] that presents the Web site evolution over time.

3.5 Visualization

Analyzing software systems is a complex enterprise and often implies large amounts of data that must be processed by the analyst. The information related to the various system entities, at the several levels of abstraction the approaches focus on is usually multifaceted, thus the assessment of the attributes is not an easy task. To help the analyst, many methodologies make use of software visualization to represent the critical attributes as intuitive, easy to handle, views on the system. Research in this domain focuses on visually capturing the most significant data, while avoiding the cluttering and allowing the user to detect the relevant patterns that serve the analysis purpose.

3.5.1 Fisheye Views

One of the first visualization techniques in software was the *fisheye views* introduced by Furnas [33]. The idea is to assign degrees of importance or relevance to the entities in the system (such as the line codes) and visualize them accordingly. The entities that have higher

importance are emphasized and shown with higher level of detail, while the ones that are not relevant to the particular context are represented with smaller sizes, or even ignored. This way, the structure of the program can be assessed in an easier way, by concentrating to the main aspects, while also seeing (with lesser detail) the rest of the system.

Storey and Muller [66] propose a visualization that uses fisheye views to document architectural diagrams and design patterns on several abstraction levels. The approach represents software systems as nested graphs and emphasizes in a fisheye view the regions of interest. This way, the details of a certain part of the graph can be easily viewed, while the overall structure of the system is still visible.

Turetken et al. [110] propose the application of fisheye views to the field of software design and analysis, especially by integrating the technique in Computer-Aided System Engineering (CASE) Tools. The different degrees of details are used in their approach to focus the attention of the engineer in both process and data modeling.

Jakobsen and Horbæk [46] analyze fisheye view visualizations applied on single source code files in Java programs, and assess the applications to other text-only data. The views focus on the selected portions of code, while presenting an overview that still shows the structure of the respective file and its general outline. Moreover, a second view shows only the lines of code that are computed as relevant with a *degree of interest* function, while the others are represented with very small fonts that make them unreadable. For example, the function may select only method signatures, and hide the actual implementations in order to obtain an overview of the respective class. The authors conducted an experiment where 16 participants used the fisheye views and compared them with the normal, linear ones in Eclipse.

3.5.2 Structural Representations

Structural patterns and relations in the systems are assessed in various ways by the researchers. Ducasse and Lanza [27] introduce a visualization of object-oriented systems that targets to depict the characteristics of multiple methods at the same time. They develop the *microprints*, pixel-based visual representations that show different aspects that may be of interest when assessing the methods: the state access, the relations derived from method invocation and the control flow.

Eick et al. [30] introduce a visualization technique that depicts various statistics about the lines of code in a format that allows quick identification of interesting patterns. Each line in a source code is drawn as a thin segment in a different color, according to the value of the chosen statistic, for example, the lines most recently changed are drawn in red, and those least recently modified in blue. The tool called *Seesoft* visualizes about 20 sources files at the same time, providing the user with an useful overview at a single glance.

Other interesting structural-related views in literature are the graph-based representations discussed in [86, 14], and the design pattern visualizations in [57] or [50].

3.5.3 Dynamic and Semantic Sources

The dynamic perspective on the system is also helped by visualization. Hill et al. [41, 42] introduce a method that facilitates the visualization of the runtime object structure of object-oriented applications. They developed the *ownership trees* that extract the encapsulation

structure from the runtime object graph. A runtime object A "owns" another object B if and only if all paths from the root (main) object to B include A. That is, were A to be deleted, B would be unreachable in the runtime system, and removed by the garbage collection mechanisms. The view facilitates the understanding as it presents in a intuitive manner the inherent relations within the running system, which also facilitates the program debugging.

Jerding et al. [47] analyze and visualize the execution of object oriented systems by capturing the runtime interactions in order to understand the dynamic behavior of the system. The visualization tools allow the user to browse the event traces in real time, and view the message interaction patterns that are detected. The views are complemented by a visualization of the source code, used as reference when assessing the system behavior. An approach that uses both static and dynamic information when visualizing software systems is presented in [97].

A visualization technique for finding the interactions between the components in a RMI application is presented in [5]. They use dynamic monitoring of the RMI calls to detect the relations between the components, and the structure of the components is not taken into consideration, as only the remote interactions are analyzed by this technique. The visualization is depicted as a sequence diagram, and the basing interactions between the RMI components is shown.

The data for the visualization can also be extracted from the semantics in the analyzed system, such as in the approach proposed by Lungu et al. [64]. They present a visualization that explores in an interactive manner the clusters of classes that share the same terms. They use the approach based on Latent Semantic Indexing in [56] for driving the clustering, and the visualization consists of three perspectives: *Exploration*, *Map*, *Detail*.

3.5.4 Metrics-Driven Visualization

Software visualization is often used in conjunction with software metrics to produce characterizations of the various attributed of the system entities at different levels of abstraction. Termeer et al. [109] present an approach that enrich UML diagrams with metrics visualization. The views are highly customizable, and the metrics are applicable both at the system level and to individual entities, such as elements or relations.

An interesting approach is presented by Wettel and Lanza [112], who develop a visualization that represents the attributes of the system components in tridimensional perspectives that resembles cities. Different metrics are used on each dimension, so that entities (classes) are drawn as parallelepipeds, each one representing a "building" in the "city" (the system). The different heights of the "buildings" and the area they occupy are indicatives of the entity properties.

Lanza and Ducasse [58] introduce the concept of *Polymetric Views*, a visualization technique using software metrics. They represent up to 5 metrics on the same node representing a single entity in the system. The measured values are depicted in the node size (width and height), node color, and node position (in a bidimensional coordinate system). The multi-faceted representation allows for easy visual recognition and identification of several characteristics of the entities at the same time.

Polymetric Views are used to capture runtime information [25], to assess the structure and interactions of the packages in an object-oriented system [26], or to visualize the evolution in

time of the class hierarchies [37, 38]. The view is enriched with a third dimension in [115].

In our work, we define a particular case of Polymetric Views when we represent system properties (Chapter 5) and use an extension of the concept when we represent the class participations to the different functionalities in the system (Chapter 6).

3.6 Reverse Engineering Distributed Software

The approaches in the literature cover various cases of software systems and use a wide set of techniques to extract knowledge about the applications. However, we have found that there are several areas that are not fully covered by the current state of the art, especially when considering the analysis of distributed software applications.

First of all, the majority of approaches **apply to a too large class of systems**, thus missing the benefits in considering the specificities of the various classes of applications. This is more evident in the case of distributed systems, where the very distributed nature is not consistently taken into consideration as a relevant source of information.

In our opinion, when analyzing distributed software, the knowledge related to the application domain is essential for driving the process of understanding. Specifically, we believe that the main information that should be taken into consideration is the *technology* the software is built on, the one related to the communication infrastructure used for establishing connections between the system components. Technology should be the main concern that drives the process of extracting knowledge from distributed systems, as it should also represent a main concern in the process of their design.

The choice of technology strongly influences the architecture and the lower-level structure of a distributed application. When architects choose a communication technology they inherently limit the possibilities of developing the application, in that the technology constrains the development with a set of specific, well-defined rules in designing and writing the application. The developers must use the services provided by the communication infrastructure in a given, technology-specific way, they must follow strict rules related to the system architecture (such as describing and implementing the service descriptions in the way requested by the technology), and they must closely follow the specific development steps the technology imposes.

The various types of constrains provide the effort of understanding such systems with valuable information, because generally they impact the code in constructs visible as detectable patterns. Moreover, the technology-related specificities of the architecture provide information that can be used to help the identification of the functional units within the system. For example, in an EJB application, the very type of the component (visible directly from the inheritance declaration of the main class) provides information about the role it plays in the system: it is concerned with the persistent storage of data (the entity beans), it defines the functionalities that perform business processing (the session beans, that offer further information when detected as stateful or stateless), or deals with asynchronous events (the message-driven beans). In RMI, the existence of a remote call provides information about a direct dependency in the system, established over the network to a service provided by the system. This is important knowledge as this particular type of dependency is highly relevant for understanding the distributed nature of the application; the technology-related information

clearly distinguishes this dependency from the usual class-to-class dependencies that exist in any object-oriented system and do not provide equally important data.

The information about the type of the technology used in distributed applications is one that can be easily obtained in most of the cases. The choice of communication infrastructures is not excessively wide, as most of them are based on standardized network or application protocols. Moreover, the knowledge is usually implicitly available in the system's basic description. For instance, in a software company (or any other organization qualified to start an understanding process) it is easy to know that the application they analyze was built using CORBA or RMI, because most probably this was one of the main traits that distinguished it from other applications built by the same company.

Therefore, this kind of knowledge does not present a problem for the analysis, at least it implies an effort highly rewarded by the gains.

Another type of concern is that many generic approaches for reverse engineering software systems are **computationally intensive**, as they are applied on the entire system without differentiating between the basic system entities. For example, clustering techniques are usually based on graph partitioning algorithms, which are known as being NP-hard problems [36]. While the researchers aim to address this problem by optimizations or application of heuristic methods, the computational effort is still significant when applied to large applications.

Especially when assessing the structure of a well-defined class of applications (as the distributed ones), we believe that any approach should start by considering the basic nature of the system, and defining a goal regarding the data to be extracted. The analysis should place value on all the available information about the technological aspects known about the application and about the nature of the items the approach is interested in. In the case of distributed systems, knowing their nature can focus the search in an efficient manner, and allows the process to concentrate on extracting the very core functionalities that define the system, rather than generically detecting functional aspects, without considering whether they are all of the same relevance to the system understanding effort.

In short, the analysis should direct its efforts on understanding the distribution-aware functionalities, and the entire process of understanding must be driven by this purpose, by conceptually isolating the most relevant partitions in the code in respect to the main system nature, that of being distributed.

Another problem is that many fully automatic approaches (such as clustering-based techniques) **do not detect the partitions that define the architecture correctly** [52]. Moreover, **system entities cannot always be placed in clearly delimited partitions**, each representing a distinct functionality. Basic entities, such as classes, often participate to more than one feature, therefore a clear-cut partitioning or clustering scheme does not accurately capture the characteristics of the application.

Consequently, the entities should be specifically analyzed so that their participation to the different functionalities is clearly assessed. Aside from the fact that this would complement the partition techniques by pointing out the places where they failed to capture the system traits, a detailed analysis of the participation of system classes to the various functionalities would provide knowledge at a finer grain in the system. Moreover, it will also address the important issue that systems usually include significant numbers of entities that are *shared* between more than one functionality, rather than being participants to a single feature.

Finally, a common characteristic of the reverse engineering techniques is that they **usually**

focus on a single, large goal, such as clustering the application in a set of subsystems, and leaves the rest of the work to the engineer. That is, they do not offer support for the other activities related to the maintenance or evolution, at least not in a manner that is consistent with the analysis itself. As the system understanding is usually followed by a process of changing the structure of the system in a way or another, we believe that a comprehensive approach should be enriched with at least basic restructuring support, which uses the same concepts and works with the same system-related assumptions as the process of understanding.

The next chapters of this dissertation present our approach to reverse engineering the distributed object-oriented software systems, driven by the considerations we made above. The targets of the methodology we developed are, consequently:

1. Use the information related to the technology the system is built on to capture the distributed nature of the application;
2. Isolate the core system entities responsible for the distributed functionality, thus focusing the computationally-intensive tasks on a minimum amount of relevant information, so that the detection of the distributed functionalities is efficient;
3. Assess the participation of the system entities to the various detected distribution-related functionalities to both complement the functionality detection and understand the collaborations in the system;
4. Provide support for application restructuring by applying the knowledge gained about the system and consistently working with the same concepts and techniques that were used in the understanding process.

The methodology is designed to be supported as much as possible by automatic tools (which we have also developed), while benefitting from the experience of the engineer at the particular stages of the approach where it is essential for attaining optimum results. Nevertheless, the interaction with the user is kept in reasonable limits, to minimize the effort the analysts must make in order to understand the characteristics of the target applications.

4

Representing Distributed Software for Analysis

Approaching the goal of understanding distributed systems can be done only by analyzing the actual issues that must be studied by the engineers. The process has to start by defining the actual need of understanding by clearly delimiting the goals, and must continue by expressing them in a way that enables the research to produce a detailed description of the analysis techniques. As in any scientific field, we need to make use of a representation of the system which, while being only a simplification of the real world, provides all the necessary means for delimiting, measuring, and analyzing the relevant characteristics of the system. The representation must define and describe all the structural or functional units that provide the necessary views on the studied aspects, and enable easy and precise observations on the attributes of interest.

In itself, a system representation is of little use without an approach that also considers the actions that must be done to actually understand the system. Therefore, any model must be enriched by describing a method or methodology of analysis which processes the model entities in an ordered, consistent, and repeatable manner. For this purpose, building the model must consider these issues, and it must support the development of the needed methodology.

This chapter is concerned with two main goals:

- finding the criteria that describe the necessity of understanding a distributed software system, specifically the elements related to distribution and remote interaction between components

- defining and describing a representation of distributed systems that serves the process of understanding.

4.1 Criteria for Understanding Distributed Software Systems

Distributed software systems are the natural outcome of the continuous and complex process of evolution software applications witnessed over the time. They represent the industry's way of answering the very important requirement of the modern society of integrating in an efficient yet transparent manner the diverse, inter-related and geographically-distributed day-to-day human activities. At their first beginnings, distributed software systems targeted specific, isolated, problems like simple remote communication between parties or remote transfer of (limited) digital resources, being mainly complementary pieces of software for the already established applications, enriching them with a single main feature: the ability to act as inter-related separate components running on different computers. Nevertheless, this single characteristic represented a serious paradigm shift, and the heterogenous, loosely-related independent applications rapidly evolved into complex, specifically designed, distribution-aware *systems*. This evolution promoted the distributed nature of a system to the rank of being a veritable frontier that separated this class of applications from the more 'traditional', locally-acting ones. Indeed, the issues implied by the distribution have large implications in designing and maintaining these systems, and there are many cases when formerly established techniques are insufficient for both their developing and analyzing.

On the other hand, no distributed application is purely so – that is, the purposes it serves are more frequently a mix of local and distribution-aware concerns, and its behavior is influenced by the both aspects. Moreover, fairly large parts of the applications are designed without directly being interested in the distributed concerns, especially when complex frameworks or infrastructures are used, specifically-targeted for providing communication-related services. Consequently, the task of analyzing and understanding such an application must balance the techniques it utilizes, in that it must make the most of the already established, 'traditional', analyzes, while emphasizing the inherent value of the characteristics that make the application distributed.

With these considerations in mind, we must asses which are the main needs an engineer has when aiming to understand a distributed system. As a prerequisite for this assessment, we start with a set of assumptions that outline the scope of our concerns, and delimits the types of activities we are supporting. For the purpose of this thesis, we make three such main assumptions about the target system:

- The software system is available to the engineer at the source code level;
- The original developers of the system are not available for questioning, or at least they are not able to provide all the required information; the documentation is also not enough for fully understanding the system;
- The system must be understood so that it can be maintained, restructured, or further developed.

As can be easily seen, the above assumptions describe in a fairly accurate degree a pattern often occurring in the software industry: legacy or long-term developed applications that are still needed, while the original team of developers are either not available anymore, or work on other assignments.

Further on, we must describe the nature of the system itself, and - for our purposes - it has the following characteristics:

- it is object-oriented
- it has an important distribution-aware functionality that actually defines the very nature/purpose of the application

As these two attributes further delimit the scope, they have two important implications on the structure and focus of the techniques that must be involved in the effort of analyzing and understanding the system:

- the approach must rely on existing object-oriented analysis techniques to capture the general characteristics of the system
- it must focus on capturing and understanding the distribution-related concerns, the characteristics that can never be fully covered by an object-oriented-only perspective on the system.

The latter implication is one that describes the actual relevance of the approach, by bringing into focus the aspect where 'traditional' techniques fail to provide useful information. We will enter a short divagation to look into this matter closely.

Distributed applications are built for various purposes and needs, but for most of them the distribution-related context actually defines the application's nature, and makes them very different from their 'classic', locally-acting counterparts. A good example to consider is the case of an electronic mail delivery system. Such applications are widely used, and their utility is beyond argument. Let's assume we look at a mailing system and try to understand its structure without being aware or even considering the possibility that it has a pronounced distribution-related functionality. We can read the source code, analyze the internal relation between the composing entities, and isolate the main functionalities of the system, such as the most intensive tasks it performs on the (local) resources, and the main activities related to storing and processing data. We can do it by ignoring the nature of the libraries or infrastructures the application uses, thus treating the providers of remote communication (like middleware or operating system services) as any other utility the system happen to use. From this perspective, we can arrive at a very probable conclusion: the system is very much concerned in storing text and related binary files in organized databases, and it has advanced features for managing the users that can access and manipulate those files. If we continue to close the eyes to the distributed aspect, we will definitely miss the most important functionality the system actually provides, that of sending and receiving to/from remote locations user e-mails.

The main conclusion we can draw from this assessment is that the understanding of the specific distribution-aware properties of the system is essential for the accurate understanding of the system's actual functionality. Consequently, a representation of a distributed system will have to describe these properties by adopting a perspective that facilitates the valid exploration of the system's distribution and communication-related functionality.

The distribution awareness of a system is best expressed by its interface with the communication medium, by the actions it does by using or providing resources over the network. When describing the functionalities a distributed component of a system provides for remote

parties, we usually refer them as *services* available or published by the respective piece of software. Different components provide and/or use services at the same time, and the correlation between the usage scenarios form the actual distributed footprint of the larger application. The communication itself plays a particularly significant role, as it represents the connexions between the various remote parties. In virtually all modern applications, the communication is not a task accomplished entirely by the application software. Instead, specialized software infrastructures are used, that virtualize and manage in an efficient way the data transmission, so that the application focuses on its main goals rather than on the details regarding the remote sending or receiving information.

To capture the distributed footprint of an application in a representation usable for detailed analysis, we believe that an approach must provide means for describing the following main aspects:

- the set of functionalities (services) the entities in the system provide as available for remote locations;
- the remote communication between parties, at a level of abstraction that allows for a good delimitation between the functionalities that belong to the studied system and those provided by external infrastructures;
- the relation between the distribution-related parts of the system, and the ones that only address local concerns.

The latter aspect has to be understood not as a clear-cut delimitation between "distributed" and "local" entities, as in most systems such a straightforward approach is hardly realistic or useful for that matter. Entities in the system may have a lower or higher degree of interest in the distributed activities, and 'pure' functionalities are rarely present, if any. Thus, the representation must provide a way of describing the degree of involvement in one or the other types of functionality, enabling the analysis to asses the various roles of the system entities.

Finally, a description of a distributed system must allow for the integration in the analysis of the issues concerning the particular type of technology the application relies on. Different technologies – mainly related to the network communication – are used when building distributed software, and the problem is that they usually have a great impact on the application's design and functionality. A representation must be as general as possible, thus cannot specifically be built for a particular technology: in fact, it must be created at the highest level of generality available so that it is applicable regardless of the particular technological constraints.

4.2 Building Blocks for Capturing the Distributed Nature of Software

Distributed software systems are essentially made of an arbitrary number of processing elements that run at different locations and are interconnected over a network [114]. Figure 4.1 presents a general schematic of such a system, as a basic view on the involved entities.

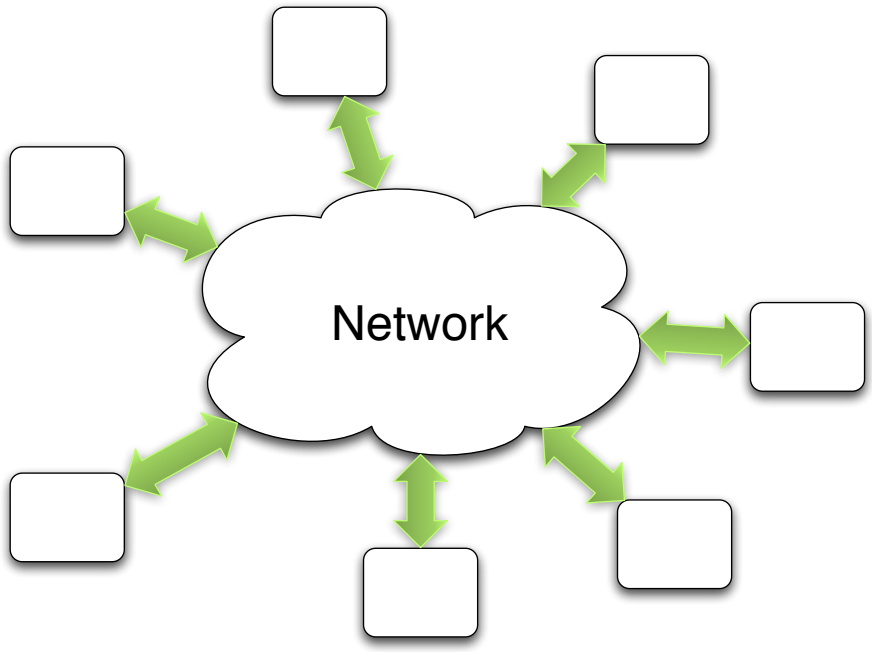

Figure 4.1: A general schematic of a Distributed System

In most cases, the remote transfer of information is handled by a special communication system, usually external to the distributed application it supports, and often being a fairly complex distributed system itself. Its purpose is to hide the details of transferring data between the remote locations, and to provide higher-level primitives that describe the data formats and control the communication-related activities. It acts as an actual infrastructure the system is built on, and it plays a very important role in both the design of the application, and on its distributed behavior. The presence of the communication infrastructure influences the analysis of a distributed system at least because of its following attributes:

- It may consist of distributed components itself, some of them running at the same locations as the application entities. When processing the application's source code, the analysis approach must differentiate between the entities that belong to the analyzed system, and those that are part of the infrastructure
- It effectively defines the language the application uses when communicating remotely. The infrastructure provides a set of primitives (like sockets, remote object stubs, message queues, etc.) and a number of rules for manipulating them. This imposes certain constraints on the application, and the analysis can make use of the information related to their nature to better detect or understand the interactions between the components of the application and the communication providers. The reason is that the communication-related interactions describe the very core of the distributed nature of the system, which is the main focus of the analysis itself.

One important aspect here is that the communication infrastructure is not necessarily an entirely independent entity. There may exist several layers of software between the application core and the network, some of them implemented 'in house' by the developers, some being third party products employed by the application, and others being mainstream, widely used and general-purpose communication infrastructures (e.g. middleware, language-specific or operating system services). The important task in this respect is to draw the line that delimits the relevant application-specific system entities from those that only address the communication. By 'relevant' we refer to those system parts that actually represent (and work for) the application's goals, those that implement the design use cases and provide the specific functionality that justified the development of the application.

Because of these concerns, our representation of distributed systems clearly separates the 'relevant' application from the communication infrastructure. We use the term *Communication Mediator* (or, in short, *Mediator*) to designate all the entities, third party or not, that concur to the basic tasks of just sending and receiving data (however complex these tasks may be), and are not involved in the actual system-specific functionalities. As an important component in our model we define the *Application-Mediator Frontier*, as the imaginary line that separates the two sets of entities: on one side will reside only the code fragments belonging to the application, and on the other the ones that implement the communication. Drawing this line is not always easy, and it may be highly related to the task of defining the scope of the analysis. The engineer may experiment with drawing the frontier at different levels in the application's layered model (if such a model exists) to include the parts of the system that are more involved in what is considered the application-specific functionality, rather than belonging to the part that implements the 'communication infrastructure'.

In most cases, though, applications use either general purpose communication infrastructures, or rely on easily separable libraries or framework instantiations that deal with the communication details. They may be independent applications, services included in the operating system, and may imply constructs specific to the programming languages themselves. Examples of widely used infrastructures are CORBA, Java/RMI, BSD sockets, Java JMS providers, etc. All define clear and documented constraints on the application, and their parts (as libraries or binary components) are easily identifiable when looking at the system. Considering their relatively limited number and high degree of usage in the industry lead us ascertain the fact that they allow for an easy delimitation of the relevant application entities in most instances of analyzing systems in order to understand their distribution-related functionalities.

Figure 4.2 shows the relation between the communication mediator and the application components, as a refinement of the previous picture we have drawn. The Mediator entities are the smaller, gray boxes. They can reside near the application components, at their particular geographical location, to act as proxies in the communication, or can be distributed over the network, interacting to provide the data transfer services. Due to the higher level of detail, the communication channels are becoming more visible, and the application-mediator frontier is easily identifiable.

The above diagram is, however, insufficient for describing the analyzed distributed application under the requirements we have already established in the previous section. The internal structure of the Communication Mediator is not always of interest for the analysis, and in fact can often be an unnecessary level of detail that only hinders the process. Specifically, we are interested in the structure of the application entities, rather than the details

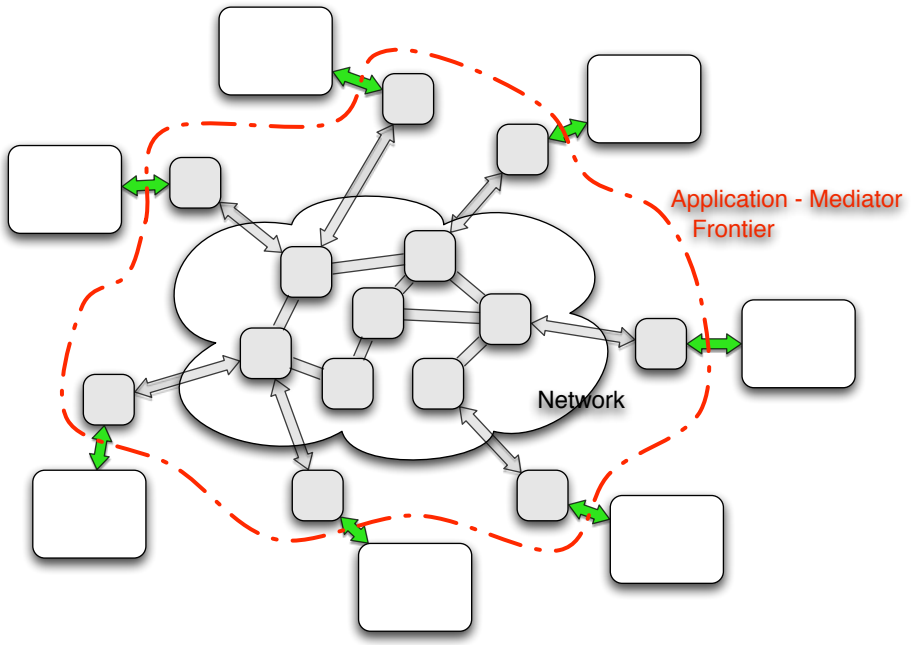

Figure 4.2: Application-Mediator interaction

regarding the basic communication, and the above picture does not reveal much about it. We must, consequently, further refine the model, so that it provides a more accurately targeted information.

Moreover, when analyzing an application starting from its source code, we have a very particular and somewhat limited view on the system, which remarkably excludes, in many instances, a very important aspect of the distribution: the information describing the actual deployment of the system components over the network. This information is usually scarce, and may reside in various places in the system. Some independent components may have their code written in separate compilation units (packages, for instance), but this is far from the norm, as they may at least share utilities, common feature providers and other entities that will easily break this rule. On some systems, the deployment information may be specifically written down in dedicated descriptors (as is the case of Enterprise Java Beans applications), but these cases are very particular and cannot be generalized to become the foundations for a model suitable for a larger range of applications. On other systems, there is effectively no deployment information per se, the task of assigning parts of the system on the various nodes belonging exclusively to the engineer that performs the installation.

Bottom line, the representation must be built so that it follows the realities, rather than limit the approach to the more straightforward cases where the deployment information is readily available. For this purpose, our approach starts with a specifically-targeted, intentional assertion in this matter: *we are certain that we can gather a deep enough knowledge about the system, and can understand it in an accurate and sufficient degree even without using*

the deployment information at all. Further more, we theorize that, while definitely useful, *the deployment of the components is far from being the main issue in understanding the functionality of the distributed system*. This assertion is meant to lead us to a methodology of analyzing distributed software that works well using the information that is realistically available in the most frequent industry-specific scenarios, that is, when the only usable artifacts are the programming fragments as encountered in a large, undiscriminating, source code repository.

Not being able to use the deployment data, we need to assess what is the information that we do have, in correlation with the representation depicted in Figure 4.2. Regarding the nature of the system, a relevant information at this point is that the application is object-oriented, and it is distributed. Therefore, we have two main items of interest:

- A set of classes, browsable at the source-code level;
- An orientation towards network communication in the system.

The latter information is highly relevant, and must be explored in detail, as it is directly related to the main goal of understanding the distributed aspects in the application. As noted above, distributed applications are usually built over a communication infrastructure (the Mediator) that deals with all the details of sending and receiving data over the network. Because this is an important aspect that directly influenced the design, in most of the cases the nature of the Mediator and the technology it implies are known or can be discovered without a significant effort. The number of 'popular' infrastructures or communications technologies is low enough to limit the search for this information even if it were not available. Usually, though, the engineers just know that they analyze an application that – for example – communicates through sockets, or one that is built using RMI or CORBA, and so on. This is valuable information for a very simple reason: it provides us with the tools of actually finding the frontier between the application and the Mediator, and the relation between it and the application classes.

Every technology for network communication imposes a set of rules or constraints the applications must follow in order to use its features. In many cases, these constraints are directly visible in the application's source code, and can be relatively easily detected when needed. For example, BSD sockets imply calling specific functions available in a provided library, in a quasi-standard order when establishing a TCP connection. Sending and receiving data is done using specific, identifiable library calls, so that the classes that deal with these tasks are easy to isolate. In Java RMI or in CORBA, specific interfaces must be written to describe services (see Chapter 2), and the classes that implement them can be identified looking at language-specific constructs in the code. Of course, there are cases where the constraints are not fully identifiable through the source code, one example being the applications using Java JMS-compliant message services. In the JMS case, components use an external provider to publish or send messagas, and a part of the interaction is described dynamically, at runtime. Nevertheless, there still exist elements that influence the code, so the case is definitely not lost.

The constraints specific to the technology are useful when determining the relation between the system classes and the Mediator. In our approach, we use these constraints to identify

the *frontier classes*, which we define as those classes or interfaces[1] in the system that either directly use Mediator services or they are built to follow Mediator-specific rules in order to export system services or otherwise interface with the Mediator facilities. For example, in RMI, we consider as frontier:

- all interfaces that extend `java.rmi.Remote`, because they effectively represent the service declarations, and are built so that they follow the RMI-specific rules for this purpose
- all the classes that call methods of such interfaces – they are the parts of the system directly involved in remote communication.

Not all the classes in a system concern themselves in a relevant degree with the distributed aspect of the application. Moreover, our experience and studies have found that for each system, there is a relatively small set of classes that have a significantly higher distribution-related role than the others. They usually represent the kernel functionality of the provided remote services, or form the main entities that use remote services as clients. We believe that the identification of this core of functionality is very important to understand the distributed nature of the system, for two main reasons:

- it is small in terms of number of classes, therefore it is easier to understand
- it concentrates the distributed functionality, being highly relevant when trying to understand the distributed nature of the system

The distributed software system representation that we are building must include the core entities as elements of the model, as the main concepts the analysis is involved with. However, we cannot consider them without relating to the goals of the analysis, so that we have to follow them trying to understand how they describe the actual distributed functionalities of the software system. With this issue in mind, two essential assertions can be made:

- systems usually do not provide a single distributed function or feature, therefore there exist, in fact, several *cores* of classes with important distributed functionality, rather than a single core;
- the purpose of the core classes are directly related to the features the system provides or uses remotely.

The second case is very important, as the aim of the approach is to understand the distributed functionalities of the system. For this purpose, we define, through a structural approach, an important concept of our methodology: the *distributable feature* .

A distributable feature is a group of classes classes that either

- implement or contribute to a distributed functionality provided as sets of remotely accessible services, or
- are themselves users of remote services, working together to implement the same functionality.

[1] For simplicity, we only use the term 'frontier classes' although at some points we also include Java interfaces in this category

These features use or depend on the communication infrastructure to comply to their design goals, and they are either deployed at distinct locations, or are good candidates for such deployment. An important observation here is that *the feature is not a distributed component* deployed at a certain location in the network. A component may consist of several distributable features, and moreover, the same distributed functionality may be provided by several components. In fact, this observation is related to one of our main goals of the methodology, related to the one we described when considering the deployment information: *we aim to detect the relevant distributed characteristics of the system without targeting a thorough component identification*. Instead, we detect distributable features, and analyze the system in respect to them. As this dissertation will show, this approach *allows us to make good assessments without using other information about the system than the source code*.

As noted above, we are interested in isolating the core distributed functionality to focus the analysis. Adding to this purpose the concept of distributable features, we can identify within the code, a set of *cores of distributable features* , as smaller sets of highly distribution-aware classes, that represent the main parts of each distributable feature in the system.

The rest of the classes in the system provide more or less distribution-aware functionality, depending on their design. In our methodology, we see them as having a measurable degree of involvement in providing the distributed functionality, by having various degrees of *acquaintance* with the system distributable features. We call these classes *feature acquaintance classes* , and we analyze their characteristics separately.

4.3 A Representation of a Distributed Software System

Synthesizing the aspects discussed in the previous section, we can define a model that represents distributed software systems, built from a structural-centric point of view, intended to describe the aspects of the system that provide a good understanding of its distribution-related functionalities.

4.3.1 Model Concepts

To describe a distributed object-oriented software system, the model defines and uses a set of concepts that delimit the relevant entities. They are discussed in the following paragraphs.

System. The entity describing the entire software system.

Communication Mediator or, in short, *Mediator*. The infrastructure that provides the means of communicating between remote locations, providing methods for sending, receiving, and otherwise manipulating information over the network. It may consist of operating system services, frameworks, middleware, in-house or third party applications or libraries, and so on. The technology it implies and the constraints imposed by it must be identifiable and manageable, so that the relation between the system and the Mediator can be characterized in the analysis.

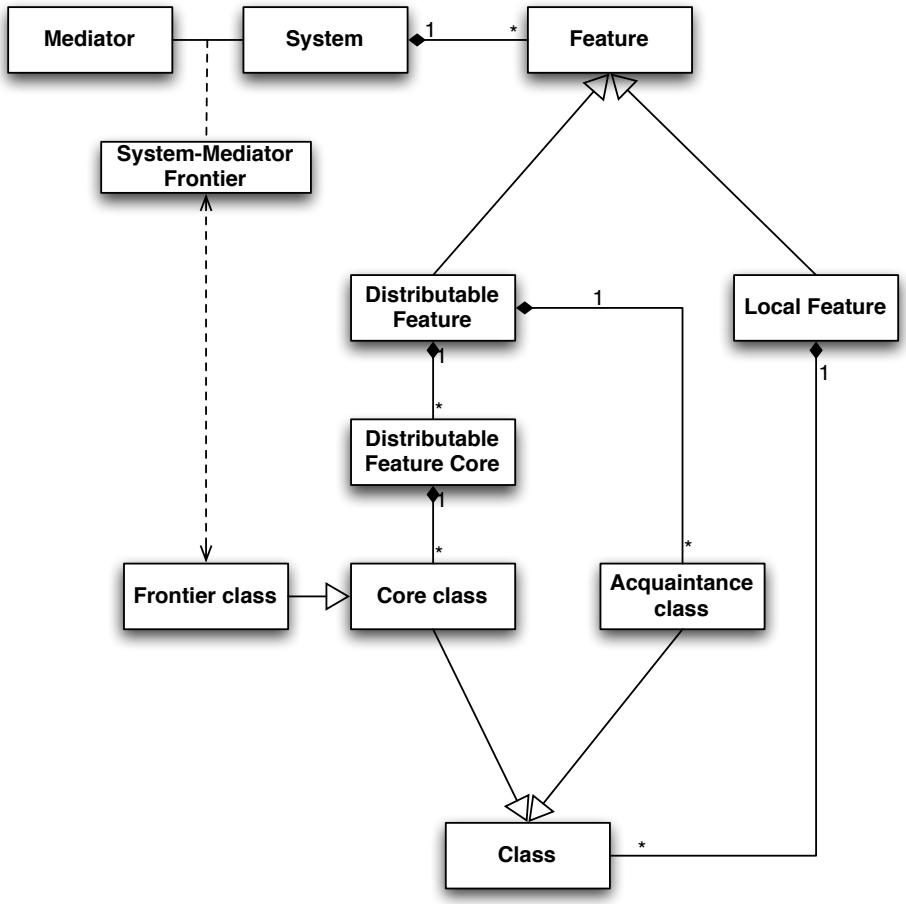

Figure 4.3: Main model concepts

System-Mediator Frontier. The imaginary line that separates the classes belonging to the analyzed system from the entities specific to the communication infrastructure. It is used as a means of identifying and characterizing the particular system classes that directly and consistently act as receivers , senders, subscribers, publishers, etc. of information via the network.

Feature. A *feature* is a part of the system that provides an identifiable functionality in the system. Features can be provided for other system entities to use, or may represent aspects the systems is concerned with when working to fulfill is design goals.

Distributable Feature. A feature to which the distribution-related functionality is central. It may be made of one or more *services* provided for other features or system entities, or may

represent a client functionality for other system or external features. It relies on communicating with remote entities to fulfill its goals, and therefore it contains classes acting at the System-Mediator frontier.

Service. A specific, closely-related set of functions, grouped in a single, design-specific, unit that describes a partial functionality provided remotely by a Distributable Feature. A Distributable Feature can contain one or more service descriptions, and their classes can indiscriminately participate in providing one or more such services. Depending on the technology and the programming language, service descriptions can sometimes be identified in the source code as interfaces or similar language-specific constructs. In these cases, the service descriptions are also considered *Frontier Classes*.

Distributable Feature Core. A minimal subset of the distributable feature classes that concentrate enough distribution-aware functionality so that they can be used to identify the distributable features within the system and characterize their main interactions. All distributable feature cores form the *distributable core* of the system.

Class. A class in the system.

Frontier Class. A system class that directly acts at the frontier with the Communication Mediator by describing, providing or using remote services. It either represents a definition of a service – therefore complying to technology-specific requirements –, or is a class that uses the Mediator to send or receive data over the network, to generate or be informed of remote events, or to otherwise manipulate remote data during the system runtime.

Core Class. A class belonging to a Distributable Feature Core. It cannot belong to two such features at the same time, but it may be involved in providing one or more services that work for providing the same feature.

Acquaintance Class. A system class that does not belong to the Distributable Feature Core. Its main attribute is the *Feature Acquaintance* which measures the degree in which it participates to one or more of the distributable features in the system. Its involvement decides whether it is actually a part of a distributable feature or is only concerned with local functionalities. Such a class may be involved with more than one distributable features, and it can have both distribution-aware and local concerns. This concept models the majority of the classes in a real-world system, where the entities that exclusively provide a single, distribution-related, functionality are relatively few.

Figure 4.3 shows the concepts discussed in this section, and highlights the relations between them. For clarity, the Service-related concern is drawn in a separate view (Figure 4.4).

Figure 4.5 presents an overview of the model, by exemplifying how a distributed system can be represented in an analysis approach. There are several remarks that must be made at this point:

- The distributable feature *cores* are disjunct entities, having no shared classes between them or with other model entities. While they may be connected to each other through

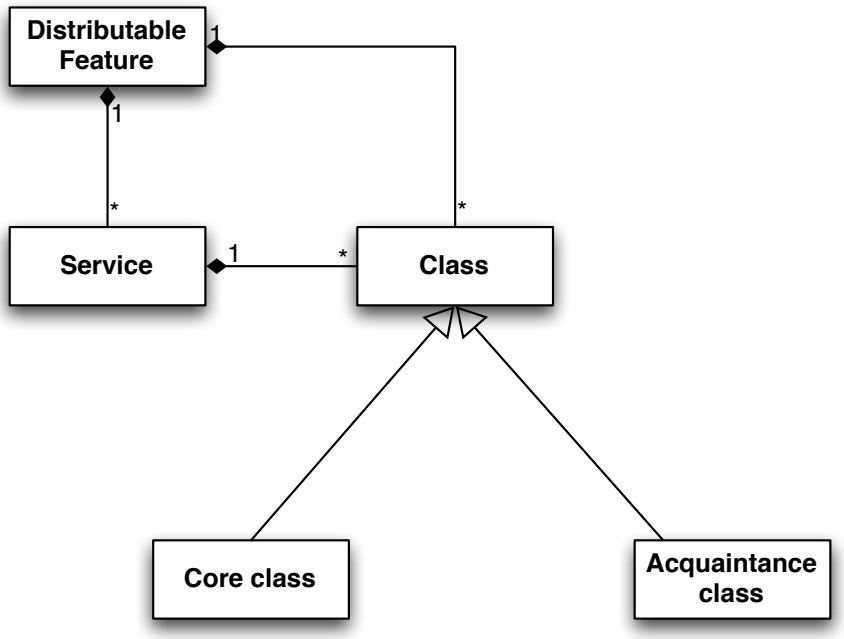

Figure 4.4: Model concepts - the service view

relations of mutual dependency, these connections must be loose enough to be safe to ignore when analyzing the characteristics of the feature cores in isolation.

- On the other hand, the distributable features themselves *can* share classes, both between them and with the local features provided by the system. In fact, in real-world systems, the shared classes are usually numerous, and the same class can participate in many features at the same time, regardless of their distribution-related characteristic. Consequently, the distinction between the *core* and the larger feature around it becomes a significant one. The core can represent the information that uniquely identifies a feature, and therefore can be utilized with a higher degree of success when characterizing the respective feature and its relations with the rest of the system.
- The local features may include classes that also participate to the distributed aspect. However, they may contain classes entirely separated from the distribution-related functionality. The ratio of predominance of the less involved classes in the system is a very interesting indicator that shows the degree in which the application was actually built for distribution, rather than being mainly focused on local activities.

4.3.2 Attributes and Relations

The model for understanding object-oriented distributed systems can be augmented with a set of attributes and relations related to the concepts described above. They cover the

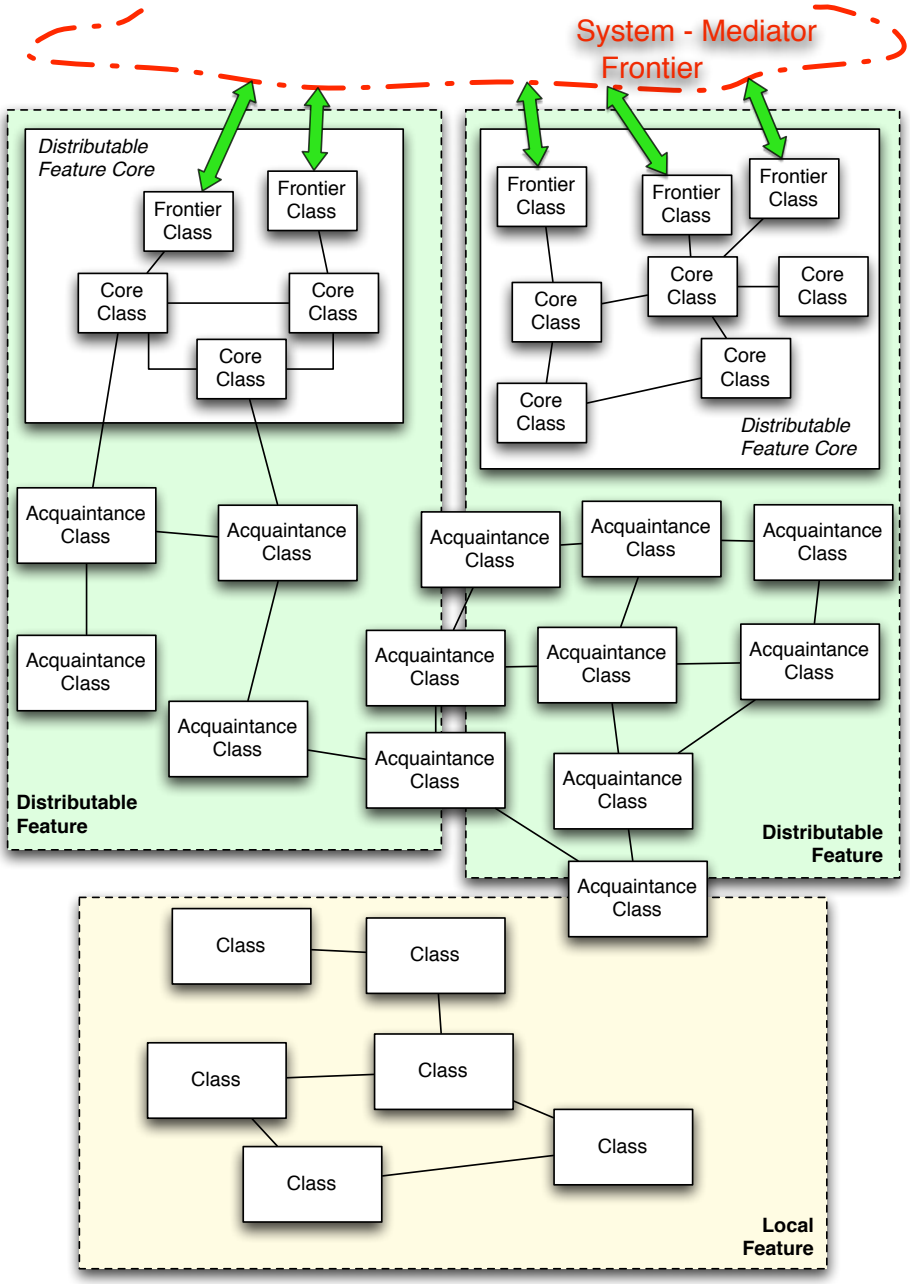

Figure 4.5: Model overview

particular characteristics of the model entities that are considered relevant for understanding the distributed functionality and the structural concerns that need to be identified within the system.

We define the following relations between system entities:

- **Remote Dependency.** Describes the relation established between entities that cooperate over the network. The ends of the relation can be distributable features, services, or distributable feature cores.
- **Acquaintance.** Represents an internal (in-system) dependency established between two entities that work together in a higher or lesser degree. It applies to relations between classes and distributable feature cores, classes and generic class groups, classes and services, and to the relations between arbitrary groups of classes in the system.

For the purpose of characterizing different aspects of the model entities, we define a set of generic attributes. During our approach, they will translate in numerical values based on software metrics, either calculated directly, or by involving specific algorithms. The attributes are described as follows:

- **Entity size.** Characterizes the relative extent of the entities, so that they can be compared. In our methodology, it is applied to the distributable feature cores.
- **Coupling.** Characterizes the strength of dependency between two arbitrary entities of the system. When classes are involved, the attribute refers to the different dimensions of the coupling-related measurements, and it is mainly used to calculate the degree of acquaintance. The attribute is also applicable to the relation between a class and groups of classes, including that between a class and a distributable feature core.
- **Degree of acquaintance.** This is a calculated attributed, based on *coupling*, meant to characterize the specific relation identified as *acquaintance* above. It is particularly useful to characterize the relations that show the involvement of the acquaintance classes in the various distributable features, or those that show the classes' participation in the provided services.
- **Distribution awareness.** This is an attribute that characterizes the importance the distributed aspect has in the design of the system. Low distribution awareness can suggest a system that wasn't actually or properly built as distributed, and which is is mainly a locally acting application that was augmented with a few distribution-aware features that are not highly significant in its functionality.

5

Core System Analysis

As the previous chapters have shown, software engineers are often confronted with the problem of understanding software applications they have to manage without relying on the often insufficient system documentation, and without having access to the system developers. The process of understanding is not an easy one, especially when dealing with large and complex projects. The engineers must rely on fairly complex tools and methodologies that are developed to support the process of extracting meaningful information from the source code, which must be tailored to fit the necessities occurring from the particularities of the analyzed system.

Distributed software applications are very demanding in this respect. They are not easy to understand because of their very structural particularity, as they consist of separate components dispersed over the network which take part in fairly complex interactions in order to fulfill the design goals. Moreover, they rely on communication infrastructures that heavily influence their design, and impose patterns of collaboration not usually encountered in 'classic' software applications. Supporting the understanding of distributed software must be done with both these issues in mind. The methodologies and tools must provide specific means for extracting the distribution-specific particularities, while being aware of the many important ways in which the communication technology influences the application's design and functionality.

This chapter presents the first part of the methodology we have developed to address these issues, by describing the core system analysis that provides the main items of understanding regarding the analyzed application. The next four chapters describe the rest of the methodology, evaluate it, and introduce the tool we developed as part of the process. All five chapters should be regarded as a whole, as they are steps that describe a single, unitary approach.

5.1 Goals

The purpose of our methodology is supporting the understanding for distributed software applications. Synthesizing the criteria discussed in the previous chapter, the minimal characteristics the target systems must have in order to be analyzable with our approach are:

- They are object-oriented applications;
- They are available to the engineers at the source code level;

- The information about the communication technology or platform the system relies on is readily available.

In particular, we have applied the techniques presented in this dissertation on distributed Java applications which use Remote Method Invocation as the means of communicating over the network. Nevertheless, the methodology is designed to be as general as possible so that it is adaptable to any object-oriented language and to most of the communication infrastructures mainstream applications currently use.

The approach follows the considerations brought up by the previous chapter, and aims to build a representation of the particular analyzed systems in the terms of the model introduced there.

Analyzing a system with this approach focuses on the application's distribution-aware characteristics, and provide a set of essential items of understanding:

- Finding how important the distributed aspect is in the application;
- Identifying the parts of the system intended to be distributable, and the respective distributable features;
- Revealing the main interaction patterns between the identified distributable features;
- Assessing the impact of distribution in the entire system, by measuring it at the class level (involving the concept of feature acquaintance classes);
- Providing support for restructuring by a structural, extraction-driven technique.

The entire process aims at minimizing the effort the engineer must employ when tackling the system, by providing a set of techniques that are easily automated, and by supporting the necessary tool infrastructure in this respect. Moreover, the main steps of the methodology focus the engineer's attention to minimal sets of entities to analyze directly, so that the human intervention is limited to only the really important aspects of the analysis approach, those that involve well-targeted, high-level process-related decisions. The approach is structural-based, and aims to extract all the possible information on the system by in-depth looking at its source code.

In order to further support the understanding, the methodology is enriched with a set of *visualizations* which dramatically improve the engineer's insight on the structural characteristics of the application and on the complex interaction patterns, while also simplifying the process of system restructuring.

The approach consists of several steps, presented in detail in this dissertation, each step designed to address a specific concern of the understanding process (Figure 5.1).

Step 0. Start with the source code. This is not an actual step of the methodology, but rather an observation regarding the perspective which the process takes when considering the system. Object-oriented applications can be represented in various ways, but at this point, we are interested in a particular type of view on the system: the application is made of a set of classes, which relate to one another by calling methods, referring attributes and so on. In other words, we are interested in a relatively simple model that represents the dependencies

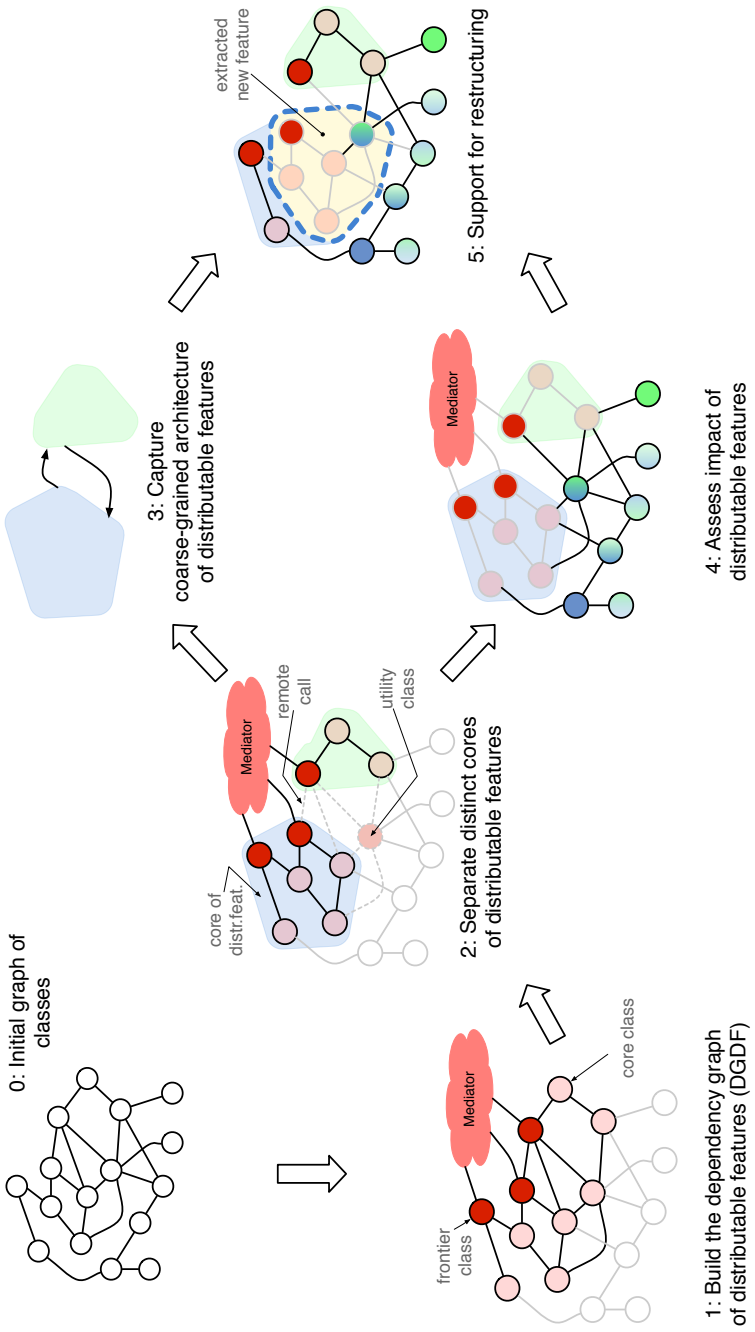

Figure 5.1: Methodology for understanding distributed object-oriented systems

between the application's classes and interfaces. The particular nature of the dependencies we are interested in, and the way they are used to extract the needed structural insight is described in detail in the following sections.

Step 1. Build the dependency graph of distributable features. This step focuses on finding within the code a minimal set of classes that represent to a higher degree the distribution-related functionality. It relies on using the established class-to-class dependencies, and building a core graph that represents the target set.

Step 2. Separate distinct cores of distributable features. The core graph of classes being identified and built, is consequently partitioned into a set of clusters that are relatively independent on each other, in that they provide different functionalities within the system. This step uses both technology-aware heuristics, and cohesion-related clustering techniques to separate the groups of classes, and present the outcome for the engineer to review.

Step 3. Capture coarse-grained architecture of distributable features. As the main cores of distributed functionalities were identified at this point, the approach provides the first assessments regarding the overall qualities of the system, by creating an overview of the distribution-related architecture of the system.

Step 4. Assess impact of distributable features. At this step, the focus of the analysis moves on the rest of the classes in the system, that were not processed by the previous phases. As the previous assessments specifically and intentionally targeted a small number of representative classes, the entities involved at this point are in fact the vast majority of the classes in the system. They are analyzed by looking at their relation with the cores of distributable features, and their participation to providing these features is measured. Moreover, their level of involvement in *non-distributable* functionalities is taken into consideration so that the importance of the distribution is assessed both at the system and at the class levels.

Step 5. Support for restructuring. While definitely a very important goal in itself, understanding a system is often a part of a larger concern, that may involve complex redesigns of the target application. In the case of distributed systems, the most frequent scenarios of redesign imply the restructuring of the code so that it better meets the requirements for a balanced, decentralized deployment. Consequently, very often, engineers need support for extracting parts of the system that must be deployed at different locations, and must assess the impact of such an enterprise. This step in our methodology proposes an approach to restructuring that simplifies the process of extraction, and provides direct feedback by estimating the impact of the projected modifications in the system.

5.2 Initial System Representation

As noted in the previous section, the initial representation of the object-oriented applications that the approach relies on describes the system mainly as a set of classes that depend on

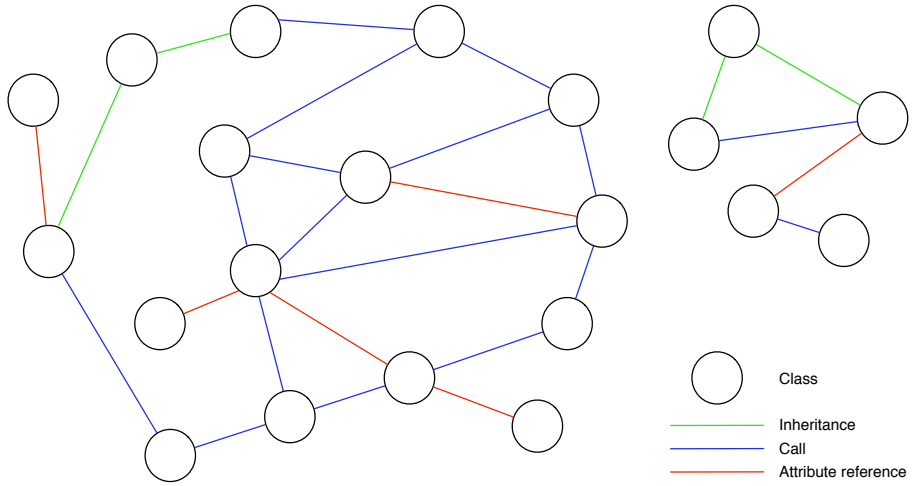

Figure 5.2: The initial dependency graph

each other. This section deals with the details regarding this representation, and the reasons behind them.

First of all, we have to analyze what are the most basic information artifacts that we always have when reading a source code, and, at the same time, we unmistakably need for extracting the relevant information.

The goal of the methodology is finding information about *functionalities* related to the distribution and remote communications. Functionalities imply sets of activities performed by system entities to accomplish specific goals. While executing activities, the entities (in our case system classes) must *collaborate* with each other, make use of each others' built-in capabilities in a controlled, pre-designed manner. Different functionalities imply different design goals, and lead to different types of collaborations between different sets of capabilities. Therefore, distinct features are usually provided by distinct groups of entities (classes), that collaborate with each other in a higher degree than they collaborate with other classes in the system. In other words, distinct functionalities are provided by distinct, highly cohesive clusters of classes. This is a very important particularity of systems, and the software engineering field is often concerned with its consequences [11, 69].

In our case, as we are interested in object-oriented software systems, we are interested in the object-orientation-specific dependencies between the application classes that define in an accurate manner the way they collaborate for achieving the design goal. For our purpose, we take into consideration the following dependencies:

- method calls,
- attribute references,
- inheritance.

Another aspect is that we are interested in knowing the *strength* of collaboration between classes, rather that the *direction* in which they depend on each other. The reason is that the

main concern of the approach is finding which classes are related to which when working for a common goal, rather that the details of the particular flow of activities necessary to achieve the goal itself.

Therefore, when analyzing a dependency between a class A and a class B, we do it by looking at both sides – that is, by looking both at the way class A depends on class B, and the way B depends on A. This *bidirectional dependency* is also assessed from a quantitative point of view, to accurately characterize the strength of the relation.

5.2.1 Class Dependency Information

As a consequence, the first requirement of the methodology we present in this dissertation is that the analyzed system should be represented initially in a way that allows building an undirected graph of class dependencies (Figure 5.2), with the following structure :

- The vertices are the classes in the system
- An edge exists between two vertices if at least one of the following dependencies are established between the respective classes:
  a) one class calls at least a method of the other one
  b) one class refer at least one attribute of the other one
  c) an inheritance relation is established between the two classes

This does not imply that at this point the graph itself has to be effectively built by the engineer, it rather specifies a precondition for the first steps of the approach. The involved algorithms will use this information, and graphs will be built only when necessary.

Because the source code of the application is available to the engineer, this requirement is not unrealistic, instead it is one that can easily be fulfilled when analyzing object-oriented systems. All the dependency information is inherently present in the OO language-specific constructs, and therefore can be extracted when necessary. Moreover, the actual process of extraction of the class dependencies is a feature present in most current software engineering support infrastructures and tools. In our case, the *Memoria* model [93, 72] employed by the *iPlasma* environment directly addresses these issues, and the infrastructure provides methods for direct interrogation of the basic object-oriented structure of the system.

We must note that the above requirement does not imply that the information regarding the direction of the dependencies will be ignored for the entire approach. Further more, this information will be used in selected parts of the analysis process, and at the respective points it will be regarded as complementary data about the system, besides the dependency graph(s).

Once the above requirement is fulfilled, the first steps of the methodology can be applied.

5.3 Identifying the Frontier

The main goal of the approach is to find relevant information about the system's distributed functionality, while minimizing the effort the of engineer during the analysis. The first step of the process is concerned with identifying a minimal set of classes that concentrate most of

the distribution-related functionality, in other words, acting as representatives for the system's distributable features.

At the initial point, there is not much information about the system. All we have is a set of classes available as source code, from which we can extract dependency information as discussed in the previous section. Therefore, the process needs to find additional data, and this information must be closely related to the goal of understanding the system's distributed behavior. To be able do do this, we must identify the best possible starting points that inherently provide data about the system's involvement in communication over the network. In our methodology, these points are represented by the *frontier classes*, that directly act at the frontier between the system and the Communication Mediator, as defined in Chapter 4.

We define two categories of *frontier classes*:

- **Frontier actors.** They are classes that directly send or receive information over the network. They can either be active entities – they produce or specifically request data – or can be passive – they are listeners for events or asynchronous receivers of information
- **Service descriptions.** This category includes all classes or interfaces that follow the rules specific to the Communication Mediator to specify sets of remotely-available functionalities grouped in specifically-designed services. They are very important, as they specifically provide essential information about the distributed functionality of the system, available to be used both by external clients, and by the other components of the system itself.

As they directly depend on the mediator to send, receive, or otherwise remotely manipulate data or events, the relation of the *frontier classes* with the distributed functionality is beyond question. Therefore, the reason we start with the frontier classes is valid: they are the most significant representatives of the communication-related system functionality, the best first classes to look for when studying the distributed nature of the system.

The interaction between these classes and the mediator is highly dependent on the technology used in communication, because they must be directly aware of all services provided by the mediator, and comply to all the constraints implied by these services. As already stated, the particularities of the constraints imply specific manners of coding the frontier classes, by applying the technology-related patterns that allow the interaction to be completed properly. This an important aspect, as it represents the main tool that enables us to search the source code for the specific patterns and therefore accurately identify the frontier classes in the application.

The active frontier actors can be identified by analyzing the code, and looking for occurrences of Mediator-specific calls that send or receive data over the network. The passive actor classes can be isolated by knowing and identifying within the code the constructs specific to the particular mechanisms provided by the Mediator which facilitate the registration of classes as listeners or otherwise declare them as consumers of data or events.

In the case of Java Remote Method Invocation (RMI), to have both services and service clients directly represented, we mark as frontier entities the following entities in the application:

1. **All Java interfaces that extend** `java.rmi.Remote`. In this particular technology, the respective interfaces represent the actual descriptions of the services published by the system components for other components or systems to use over the network. The RMI

terminology identifies them as *remote interfaces*. We use them as the representatives for the system entities that provide the services, that is entities that communicate directly over the network at the request of the clients.

2. **All the classes in the system that call methods of the remote interfaces** in the application. These are the classes that communicate directly over the network by using the services published by other parts of the same system. This is an important aspect, as it provides, besides frontier entities, initial information regarding the distribution-related interactions that are established within the components of the analyzed application, therefore can form the base for consequent architecture-related assessments.

At this point we must make an important remark: the assumptions on which we base our technique rely strictly on generally applicable rules (related to distributed communication) that fit both well-structured programs, and poorly designed systems. Indeed, at any point in the approach, we avoid making assumptions regarding the particular aspects of the system's design, about the patterns that were used or the architectural decisions that were made. This holds true for our entire methodology, and thus, the feasibility of the approach is not dependent on the quality of the system's design, as we limit its assumptions to elementary rules and patterns that are enforced by technology (*e.g.*, RMI) specificities, and which must be implemented by all applications.

5.4 Building the Core

Finding the frontier classes is an essential step, but they are not enough to capture and understand the actual functionality of the system's distributable features. Therefore, the next step in our methodology focuses on finding a set of additional classes in the application that add significant knowledge about the distribution-related functionality of the system. Moreover, the cardinality of this set should be as small as possible, so that it can be analyzed in detail by the engineer. In other words, we try to identify the very core of the distribution-aware nature of the system, to select from the great number of classes only those that provide the most valuable information.

The best place to find these entities are in the immediate *vicinity* of the *frontier classes*, as they are most likely strongly involved in the activities related to the remote communication over the network, and therefore provide us with significant knowledge about the distributed structure of the system.

This is the point where the considerations we made in Section 5.2 regarding the dependencies between system classes come to value. In the system's dependency graph, classes are connected by edges that show their collaboration: they call each other directly, refer each other's attributes, or are linked by inheritance relations. We can use the dependency graph as the framework that defines the relation of 'neighborhood' between classes, in terms of structural dependence, in fact in terms of their inherent collaboration for fulfilling the design goals of the system. Classes will be 'neighbors' to one another if they are directly linked by an edge in the graph, and they are 'close' if the distance between them is relatively low.

To capture this structural trait, we defined a technique that starts with the already identified frontier classes and builds around them a structure we call the *Dependency Graph of*

Distributable Features (DGDF). In this graph, each class is a vertex, and the edges represent the direct dependencies between classes, as discussed in Section 5.2. As the distributed communication is usually *bidirectional* we are interested only in the fact that the two classes are related with each other, not in the precise direction of their collaboration; therefore the DGDF will be *undirected*. An edge will be created in the graph between any two vertices (*i.e.*, classes) if at least a dependency exists from one class to the other. For this purpose, we take into account three types of structural dependencies: (i) method calls; (ii) attributes accesses; and (iii) inheritance relations.

The DGDF is built *iteratively* and *incrementally* starting with the frontier classes as initial vertices, through an algorithm that we describe in the following paragraphs.

5.4.1 The Algorithm for Isolating the Distributable Core

5.4.1.1 Prerequisites.

The algorithm tries to detect the minimal dependency graph between the classes that encapsulate the core distribution-aware functionality of the system. The algorithm is described in terms of its implementation in an object-oriented language, and it includes the considerations that are specific for analyzing applications built using Java RMI.

Besides the concepts introduced in Chapter 4, there are a set of notions that must be defined before describing the algorithm:

- **Build strategy.** Implements the approach used at each step for finding the set of classes that will be processed at the next step. In our implementation, a build strategy consists of two utility classes, the first for the initial step of the algorithm, the second for the rest of the steps. A build strategy class provides methods that return *Memoria model entities related to the current set, one method for each type of relevant relation. For instance, the* `UsersOfAllMembers` *class (one of our build strategies) has a method called* `methodRelated()` *that returns all methods calling (methods of) the current entity.*
- **Build Rule.** *A specification (realized in the form of a class) of the sequence of build strategies that are to be applied for building the graph. The build rule is instantiated by the concrete algorithm, i.e. by a class derived from the class* `AbstractGraphBuilder`. *Example:* `RMIBuildRule` *adds, in order, the following two strategies:* `UsersOfAllMembers`, `AllUsedMembers` *and is instantiated by the* `RMISystemGraphGenerator`.

The algorithm is parametrized with a specified maximum depth of search, a user-configurable item that limits the search for relevant classes.

In our implementation, the core algorithm is encapsulated by the abstract `AbstractGraphBuilder` class. The actual, technology-aware, algorithms are implemented in a class derived from it. For the case of Java RMI applications, the class we implemented is called `RMISystemGraphGenerator`.

5.4.1.2 Generating the Dependency Graph.

This step tries to build a graph that shows a set of dependencies between the system classes. Relations as "calls method", "method called by" and "ancestor of" will be used as represen-

tatives.

General description. The algorithm starts with a set of classes as root entities and builds the graph. The root classes include, but are not limited to, the frontier classes already identified.

Selecting the Root Classes. This is a **technology dependent** step, i.e. it is specific to a technology-related instantiation of the algorithm.

For RMI, the selection of root classes is done by the algorithm as follows:

a) selects all interfaces extending of `java.rmi.Remote`, as well as the classes that refer them, and marks them as FRONTIER;

b) selects all classes that implement remote interfaces;

c) the union of these two sets is returned as the initial set.

Building the graph. This is a **generic** part, i.e. it does not depend on the technology. The sequence of actions that build the core graph of classes directly related with the distributed functionality of the system is described in Algorithm 5.4.1

5.4.1.3 Implementing the Build Strategies

As hinted by the algorithm description, the build strategies are facilities that are able to generate, for each class that was added to the core dependency graph, a set of knowledge items that can be used to determine the next set of classes to be considered by the algorithm. This is a very flexible approach, as it provides means for easily adapting the algorithm to various criteria in selecting candidate classes.

In the instantiation of our algorithm that we specifically used when conducting the test cases, the strategies we have implemented provided two types of information:

- method-related items, such as methods that are related to the current class, belonging to system classes
- attribute-related items, such as entities accessing attributes of the current class

Different strategies generate different sets of results, depending on their purpose. As previously noted, we have used two such strategies when we applied the approach to Java RMI systems:

- `UsersOfAllMembers` which generated at each step:
  - all methods calling the current class;
  - all methods accessing attributes of the current class.
- `AllUsedMembers` which generated at each step all the methods called by the current class, except those of the class itself

Algorithm 5.4.1: BUILDCOREGRAPH(root class set)

```
main
  for each class in root class set
    do { mark as ROOT
         call GENERATESUBGRAPH(class)

procedure GENERATESUBGRAPH(class)
  for each build rule
    do { for each strategy in the rule
         do call BUILDDEPENDENCYGRAPH(rule, strategy, class)

procedure BUILDDEPENDENCYGRAPH(rule, strategy, class)
  if specified maximum depth was reached
    then return

  use the current strategy to select the next set of "interesting entities".
  The "interesting entities" are those that provide useful information about
  the next set of classes to be added to the graph, such as methods
  called by the current class, methods calling methods of the current class,
  or attributes referred by the current class.
  The strategy class is selected depending on the current depth in
  the graph: in the beginning the first class in the pair is used as information
  provider, then the second. A set of entities (as methods or attributes)
  is returned by the information provider's methods.

  compute the set S1 of classes resulting from the entities obtained above.
  S1 contains the classes related (in terms of dependency) with the
  current class: classes containing the called/caller methods and the
  referred attributes. This covers the called-by/caller-of and attribute-related
  dependencies.

  compute the set S2 of classes that are ancestors to the current class

  for each class in S1, S2
    do call BUILDDEPENDENCYGRAPH(rule, strategy, class)
```

To exemplify the degree of flexibility gained by this approach, we should add that, at some point in our experiments we have also defined a strategy called `UsersOfMembersByFieldType` which provided the following entities:

- all methods in other classes that accessed attributes of the current class, with the attribute type specified as an argument;

- all methods in other classes that called methods of the current class which in turn accessed a subset of its own attributes identified by type.

5.5 Reviewing the Results

The graph generated at this point includes only a part of the classes in the system. This is exactly as required by the considerations we made above, as the classes we obtain are the most important ones in what concerns the identification of distributed behavior. In our experiments on real-world applications, less than 20% of the system classes were designated as belonging to the cores of distributable features, which is very valuable for narrowing the focus of the analysis.

During each iteration, the algorithm identifies the new classes that are linked by dependencies (*i.e.*, method calls, attribute accesses, inheritance relations) with the classes that were already added to the graph.

The process continues until a specified depth of search has been reached. While the search depth can be set by the user, we found that a depth value of 6 to 8 is appropriate for building a DGDF that includes only classes that are close enough to the detected frontier. This minimizes the chances to include classes that are only slightly involved in the distributed parts of functionality.

Besides specifying the search depth for the above algorithm, at this step in the analysis the engineer is given an additional opportunity to improve the relevance of the classes belonging to the DGDF. In our approach, the engineer can fine-tune the automatic construction of the graph, by manually adding and removing classes to the dependency graph. This way the engineer can control the core set of classes that represent the main distributed functionality of the system.

The decision of adding new classes to the core highly depends on the engineer's experience, and can be based on various assessments made by looking at the set generated by the Algorithm 5.4.1 and the neighboring classes in the system dependency graph:

- classes that are directly linked with entities already included in the core, and are not connected with any other class in the dependency graph;
- classes that have many connections with the generated set, and significantly fewer connections with other classes;
- classes that have names resembling or suggesting logical connections to the names of the classes that were included in the core;
- classes that implement an interface already included in the set, and the algorithm happened to miss (because of an improperly tuned search depth);
- classes that suggest (by name) that are directly related to the distributed aspect, but were ignored by the algorithm for various reasons.

To remove classes, the engineer can consider one or more of the following criteria:

- classes that are loosely connected with the other classes included in the core, while exhibiting stronger connections with other classes in the system dependency graph;

- classes that have names suggesting different purposes than for the majority of classes in the core. This criterium should be applied with care, as these classes may actually represent different distribution-related functionalities, rather than classes inadvertently included in the core set;
- classes that strongly suggest (by name) functionalities that are not related to the distributed aspect;
- classes that seem to be general-purpose utilities, such as string manipulators, format converters, etc. Keeping them adds no significant information regarding the distribution-related functionality, as they are built to be neutral entities, usable in many different circumstances.

Nevertheless, we noticed during our case studies that even without any human intervention we usually end with a relatively small number of classes, most of them highly relevant to our purpose, and which encapsulate sufficient knowledge about the basic distribution-related functionality of the system. While different applications may need different levels of user intervention at this step, the algorithm provides a reliable starting point for a successful analysis.

5.6 Identifying the Distributable Features

Once the core set of classes related to the distribution-aware functionality was isolated, our methodology takes a step further in representing the system through the model we have introduced in Chapter 4. Its goal is the identification of the distributable features, so that the distinct aspects of the distributed functionality of the system are extracted and analyzed separately.

The approach is based on the observation that, in general, distinct functionalities within a system are represented by loosely coupled units of system entities [69], that can be isolated with properly tuned semi-automated techniques.

Considering this observation by representing the system as a dependency graph, we can see an object-oriented application as being made of a set of loosely- connected clusters of classes, each cluster representing a different feature provided by the system. In order to extract these features, we need a technique that is able to separate the functional units, by using all the information available about the system to detect, separate and analyze the relevant clusters.

In our approach, we are already at the point where we separated the core distributed functionality from the rest of the system classes, We continue the analysis by applying the above considerations to separate this core into a set of functionally-distinct clusters of classes. To do this we apply the following method:

- We analyze the graph obtained at the previous step, and use a set of heuristic rules to detect the edges that link separate functionalities.
- we then temporarily eliminate these edges from the graph, and
- apply a general-purpose graph algorithm to isolate the set of connected components in the modified graph.

- Each connected component will represent a candidate for a core of distinct functionality. As the processed graph was already the one that represented the system's distributed nature, the connected component will in fact represent a core of a distributable feature in the system.

- The result is presented to the engineer for review.

The essential component of the technique is the identification of the edges or vertices that are to be eliminated so that we remain with a set of connected components in the graph. In other words, the goal is to eliminate all entities that prevent us from easily identifying the functional units that provide distinct features.

In our opinion, we can obtain valuable information in this respect by a careful analysis of all the available data describing the inherent nature of the system. This is especially true in the case of distributed software, where the constraints the Communicator Mediator imposes on the application have direct and identifiable consequences on the source code. Knowing the particularities of the system can provide useful data about the types of dependencies established between classes, so that we are able to find which of these dependencies can be safely ignored (removed) for the purpose if isolating loosely-connected functionalities.

This approach, aware of the system's distribution-related particularities would be able to pave the way for applying a more general technique (applicable to other classes of systems as well) able to separate the loosely coupled units. Considering this aspect, our approach uses two types of rules that are applied on the system in order to detect and eliminate edges from the graph:

- rules derived from analyzing the specificities of the technology the application is built on, specifically the technology of the Communication Mediator,

- graph-oriented rules, built for extracting the loosely-connected cohesive clusters of classes in an object-oriented system.

5.6.1 Technology-Aware Heuristics

Knowing the constraints the Mediator-specific technology imposes on the application's source code enables us to create a set of rules that identify several types of dependencies between the classes which are not central to providing a distributable feature. The edges in the graph representing these dependencies can be consequently temporarily removed so that the identification of the loosely-coupled cohesive functionally-distinct clusters is easier.

For the case of distributed, object-oriented software applications that use Java RMI as the communication infrastructure, we have made several observations that enabled us create and apply a set of heuristic rules for this purpose.

5.6.1.1 Remote calls.

In Java RMI, the descriptions of services provided over the network are encapsulated in remote interfaces, that is Java interfaces that extend `java.rmi.Remote`. Consequently, calling a method of a remote interface usually means that the caller class belongs to a component that

is located at a different location, and thus it has a good chance of belonging to a different functional unit.

The first heuristic we apply temporarily eliminates all edges in the graph that represent remote calls. Note that, even if the respective call refers to an interface in the same functional cluster, this rule does not diminish the chances of correctly identifying it: to belong to a single functionality, the classes must be related to each other in several additional ways, so that eliminating a single edge will not make a big difference. Moreover, this case is very rare, as it is not feasible to use RMI to call methods that are already available locally.

On the other hand, the case where the two classes linked by a remote call (through an interface) belong to different, functionally-distinct, components is greatly served by this approach, as most probably, their dependency in the graph is only represented by such remote calls. Eliminating them will effectively decouple the two functionalities from each other.

5.6.1.2 Stub classes.

Most of the RMI applications include with their source code the automatically generated stub classes specific for this technology. The stub classes are the entities that are responsible with providing the communication channels that implement the remote calls, therefore can be considered a part of the communication infrastructure, rather than components of the application. Their existence not only fails to add useful information when trying to understand the system, but they artificially raise the number of classes the engineer has to review when trying to analyze the system.

Our approach eliminates all the vertices corresponding to stub classes from the dependency graph, along with the edges that connect them to the rest of the system.

5.6.1.3 Utility classes.

Virtually every application makes use of classes that are not specifically built for the purposes of the system and are rather general-purpose utilities that can be used in many different types of applications. This category may include classes that perform conversions between different formats for representing information, can provide features for the manipulation of strings, can provide means for accessing proprietary or open format archives, etc.

Moreover, the application itself may define specific classes providing features that are not central to the goals of the system, but are nevertheless used by several different units of the code. When trying to separate functionalities, these classes may artificially link different functional clusters, for the simple reason that more than one functionality happened to need the respective features.

Our approach defines a rule that detects some of the utility classes, and removes them along with edges that link them with the rest of the system. The heuristic that detects the utility classes is presented in Algorithm 5.6.1.

We must note that, in this particular case, the technique makes use of the information regarding the direction of the dependency between the classes. Thus, the source of the edge is the class that has a dependency, and the target of the edge points to the class it depends on. Although this information is not directly available in the undirected dependency graph, it is easily extracted from the source code.

The heuristic basically eliminates all the classes in the system that only have incoming edges and are neither interfaces, nor classes that other classes inherit. The main assumption for this heuristic is that classes that never use (by calling methods or accessing attributes) other classes are either the base for an inheritance hierarchy, or represent utilities. This is true for most of the cases, and we found that the rule had provided good results in helping decoupling functionally unrelated system clusters.

Algorithm 5.6.1: isUtilityVertex(current vertex)

```
main
{ entity ← the class corresponding to the current vertex
  if entity is an interface
    then return ( false )
  if there are classes derived (inheriting) from entity
    then return ( false )
  for each edges connected with the vertex
    do { if edge source vertex is the current vertex
         then return ( false )
  return ( true )
```

5.6.2 Cohesion-Based Clustering

To eliminate further edges that connect the functionally-distinct units in the code, we have applied a technique for graph processing based on the clustering algorithm described in [11].

The algorithm, which we adapted to our dependency graph, calculates for each edge a measure that characterizes the "strength" with which it connects the two vertices. The approach considers the density of the existing edges in the neighborhood of the two vertices, so that an edge is considered 'weak' when it is likely to connect two otherwise loosely connected clusters (see Figure 5.3).

To control the algorithm, the engineer is given the opportunity to specify and tune a threshold value for this measure. Edges with the strength under this threshold are eliminated from the graph, while the rest are left untouched.

5.7 System-Level Understanding

Applying the two categories of rules described above transforms the dependency graph representing the core distributed functionality. At this point, we are prepared to make the first valuable assessments about the analyzed application, as the methodology is ready to provide the first items of system understanding.

For this purpose, the next step of our approach continues the processing by applying a general-purpose graph-related algorithm that detects maximally connected components. As

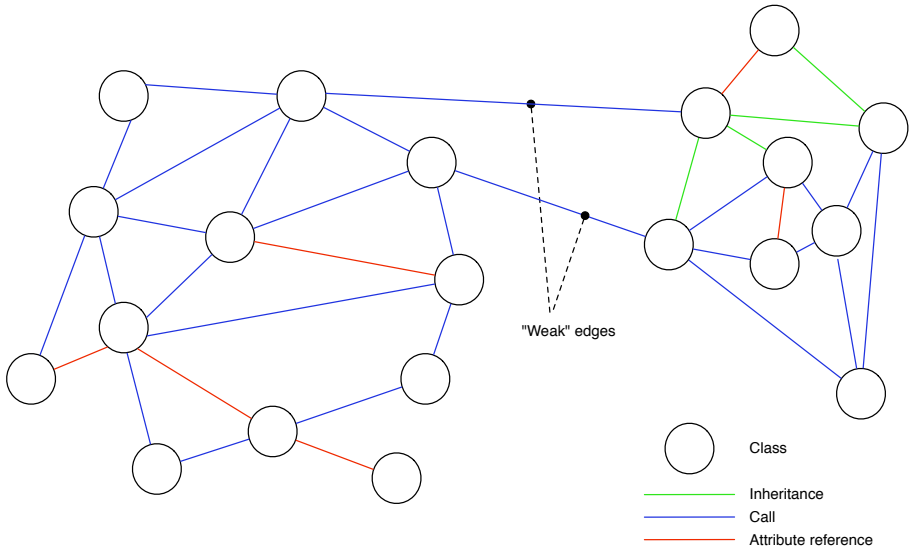

Figure 5.3: Loosely-connected cohesive clusters

the edges removed by the previous steps were specifically selected aiming the separation of distinct functionalities, the set of connected components represent, in fact, the set of separated functional units. Because the graph contained the most representative classes for the system's distributed nature, the clustering provides us with the set of functional cores for the distributable features in the application.

Consequently, the analysis so far has provided us with the following:

- a relatively small set of classes representing the core distributed functionality of the system
- a number of partitions in the above set, each representing the a distributable feature the system provides

To ensure the validity of these findings, the engineer can analyze the outcome of the above processing. In both our methodology and the implemented tool infrastructure, the result can be modified at will by adding or removing classes to clusters.

5.7.1 Identify the Remote Communication Channels

One of the most important aspects when doing static analysis on a software application is analyzing the interactions established between the various entities in the system. While some of the system attributes can be gathered looking at the entities in isolation, the actual understanding requires an in-depth assessment of the way the components work with each other to meet their design goal. For this purpose, the analysis must necessarily focus on identifying the relevant dependencies between the studied entities, especially the direct entity-to-entity relations.

In the context of distributed applications, the most important dependencies are those that provide information about the distribution-aware characteristics of the system. Consequently, this requirement translates as the need of identifying the remote communication channels between the various parts of the system, the data paths established via the Mediator that link them over the network. This information is highly relevant, as it can provide the analysis with *a first preview of the system's actual distributed architecture*.

The entities that we have already identified are the distributable features in the analyzed system. A very important observation at this point is that the distributable features are *not* the direct expression of the various distributed *components* as they are deployed when installing the application in a network. Instead, they are functional units directly related to the distributed functionality, and they may or may not be deployed at distinct locations at runtime. The analysis so far has not used any information regarding the deployment, and, as we stated in Chapter 4, we are confident that such information is not vital to actually understand the system.

The communication channels that are established over the network, however, have everything to do with the component-to-component relations specific to the analyzed application, because their very existence in the system points to a direct need of communication between components deployed at different locations. Therefore, identifying the remote-established channels in the context of our approach does not only provide us with just another type of dependency within the system, it represents the next important information we need to find about the distributed architecture of the system.

Another important observation is that, as we place the core of our approach in being aware of the relation between the system and the entity we have identified as the Communication Mediator, we are in the position of finding the communication channels without actually needing the information about the component deployment. This is because, as we have stated before, we can benefit from a inherent characteristic of this relation: the Communication Mediator always defines a set of rules that translate in a set of constraints imposed on the design of the application. These constraints are often identifiable as technology-specific constructs in the code, therefore we can use them directly in the static analysis we employ on the respective system.

The direct consequence of these two observations is that the distributable features and the communication channels represent two knowledge items that together provide much more relevant information than studied in isolation:

- on one hand, the distributable features represent distinct functionalities *directly related with the distributed aspect* in the system
- on the other hand, the communication channels reflect the *over-the-network dependencies between remotely-deployed components* of the system

The natural conclusion is that while combining the two types of information does not necessarily guarantee the extraction of deployment information, it effectively provides us with information of a much greater value: *the remotely-established dependencies between the distribution-aware system entities*, that is, effectively, *the architectural footprint of the distributed system* itself.

Because the components communicate remotely to each other, finding the communication channels that connect them is a highly technology-dependent step, depending on the type of the communication mediator involved.

For applications using Java RMI for communicating over the network, we make use of their specific way of describing remote services. As previously noted, this involves creating Java interfaces that extend `java.rmi.Remote`. The very existence of remote interfaces in the application's code is the expression of the need to publish a set of services available for remote locations. Therefore, any reference to such an interface usually indicates a communication between two entities deployed at different locations. For each pair of distributable feature cores already identified, the communication channel between them is consequently made of all references to remote interfaces that link one feature with the other.

The different types of dependencies that represent communication channels can be used to reveal different types of distributed architectures [108] in the analyzed system. For instance, we can detect a client-server dependency by observing features that call each other in a single direction. Peer-to-peer communication will be represented by bidirectional dependencies between distributable features, and this approach would also allow us to observe more complex interactions, as layered communication, rings of communicating entities, and so on.

5.8 Visualization - The Distributable Features View

One of the most important aspects when understanding structures is representing them by the means of visual elements. Architectures, plans, abstract models, and any other conceivable arrangements of inter-related entities are better communicated and understood when represented through a well-drawn picture.

This is the reason our approach in understanding a widely-used category of systems makes an important point in providing a set of relevant *software visualization* [24] techniques specifically built for analyzing distributed object-oriented systems. The visualizations we have developed were first presented in [20], and complement the main analysis techniques our methodology provides.

In accordance with our chosen terminology, we introduce the *Distributable Features View* as a visual means to representing all the concurrent structural-related interaction facets we find relevant for understanding distributed systems. It consists of two perspectives, the first one is detailed below, and the other one will be introduced later in our presentation.

5.8.1 The Distributed Architecture Perspective

In this perspective, we are concerned with representing the knowledge we have gained by identifying the distributable features and the communication channels they are linked with each other via the Mediator infrastructure, by drawing the first preview of the distributed architecture of the system.

In Figure 5.4 we have exemplified the perspective by drawing the distributable feature cores we have identified in some of our case studies. The features are represented as rectangles of the same width, their height being proportional to their size in terms of number of classes. From this point of view, the Distributed Architecture Perspective is a simple case of *polymetric view* [58].

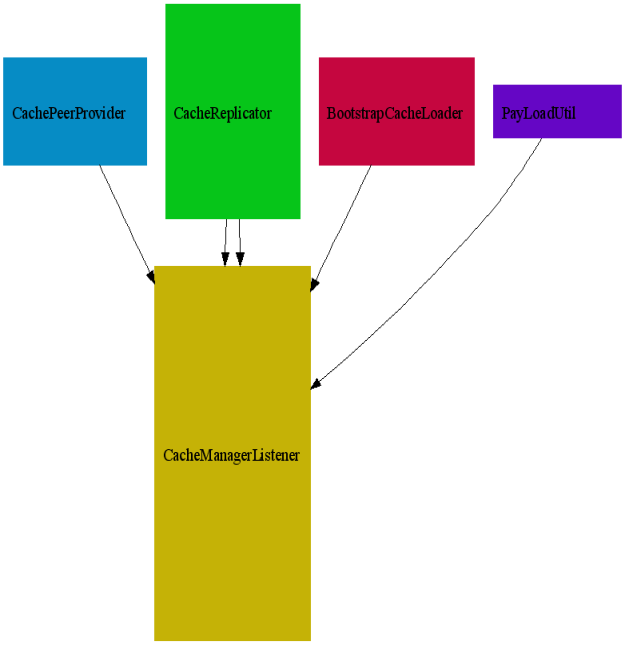

Figure 5.4: Example of Distributed Architecture Perspective

The communication channels between distributable features (in this case the remote calls made by the classes belonging to the different cores) are depicted in this visualization using directed arrows.

In the context of vizualizing the system attributes to improve the understanding, the *Distributed Architecture Perspective* has a double role:

1. It enables the engineer to actually *see* the architectural relations between the various distributable features, as represented by their core classes. This helps drawing the first conclusions about the distributed architecture of the system to identify client-server dependencies (features calling each other in a single direction), peer-to-peer communication (bidirectional calls), and so on.

2. It visually associates each distributable feature with a color. This is an essential element for the interpretation of the further visualizations (which we discuss later), where this 'color coding' will be used as a key.

The tool infrastructure we provide for the understanding process automatically generates the visualizations as soon as the necessary information was gathered from analyzing the application's source code.

6

Impact of Distribution

The steps of the methodology described in the previous chapter focused the attention of the research to a relatively small set of classes that represented the core of the distributed functionality of the application. The process was very useful in capturing the main system attributes related to the distribution-related concerns, by greatly reducing the effort of the engineer and provided a first view on the system's distributed architecture.

Nevertheless, the vast majority of the application's classes were not yet taken into consideration, and an approach that aims to understand the system would never be complete without analyzing them, too. This chapter describes the techniques we have employed to understand the positions these classes take in respect to the already gathered knowledge about the system. Specifically, we analyze them in relation to the identified distributable feature cores, so that we understand in a better degree the application's relation with the distribution-related tasks.

For this purpose, we analyze for each class outside the previously processed core its involvement in providing the already discovered distributable features. We define and use a set of metrics that characterize this relation and interpret them so that we gain a deeper understanding of the system. The approach is complemented by visualization, which facilitates the capturing of the most interesting patterns of collaboration that occur within the system.

6.1 Acquaintance Classes and Metrics

In order to do so, we have to measure the degree in which a particular class participates in providing a certain distributable feature. We call this involvement *affiliation to the distributable feature's goals*, and the class itself becomes an *acquaintance of the distributable feature* . A class is 'better acquainted' to a feature if it plays a more important role in providing it, while classes 'less acquainted' are mainly involved in other activities, distributable or otherwise.

When measuring, we are interested in two main characteristics of the target class:

1. The degree of affiliation of the class to the general distribution-related functionality in the system;

2. The level of involvement of the respective class in providing each of the distributable features identified in the previous steps of the analysis.

The first coordinate provides information about the overall importance of distribution in the application's design, while the second focuses on capturing the details regarding the way each of the distribution-related functionalities are provided by the system.

6.1.1 Bidirectional Coupling Metrics

The first measurement that we need when assessing the above characteristics of the system is directly related with the way individual classes are related to each other. We have already used an important property in this respect when we built the core graph of distributable features, the direct dependency between classes in terms of coupling.

For building the set of metrics that will help us extract further attributes of the system, we define two coupling-related measures at the class level. We calculate the coupling on two coordinates: the coupling *intensity*, which assesses the strength of interaction between the classes in terms of the *number of collaborations* that make the dependency, and coupling *dispersion* which quantifies the *number of collaborators* within the classes [59]. On the other hand, as mentioned earlier, we are interested in both aspects of coupling (bidirectionally), namely the *import* and *export* ones [8]. We introduce two metrics:

The *Bidirectional Coupling*(BC) between class A and B is the pair.

$$BC(A,B) = \begin{cases} BC_I(A,B), & \text{the coupling intensity} \\ BC_D(A,B), & \text{the coupling dispersion} \end{cases}$$

The coupling intensity is defined as:

$$\begin{aligned} BC_I(A,B) \;=\; & Calls_I(A,B) + Acc_I(A,B) \\ & + Calls_I(B,A) + Acc_I(B,A) \\ & + Inh(A,B) \end{aligned}$$

where

$Calls_I(X,Y) =$ the total number of method invocations from methods in class X to methods in class Y,

$Acc_I(X,Y) =$ the total number of accesses from methods in class X to attributes in class Y, and

$Inh(A,B)$ is defined as

$$Inh(A,B) = \begin{cases} 1, & \text{when A inherits from B or B inherits from A} \\ 0, & \text{otherwise} \end{cases}$$

The coupling dispersion is:

$$\begin{aligned} BC_D(A,B) &= Calls_D(A,B) + Acc_D(A,B) \\ &+ Calls_D(B,A) + Acc_D(B,A) \\ &+ Inh(A,B) \end{aligned}$$

where

$Calls_D(X,Y) =$ the number of *distinct* method invocations from class X to methods in class Y. If a call to the same method is made several times between the two classes, it is only counted once,

$Acc_D(X,Y) =$ the total number of *distinct* accesses from methods in class X to attributes in class Y. If an access to the same attribute is made several times between the two classes, it is only counted once , and

$Inh(A,B)$ is the same as above.

The *Total Bidirectional Coupling* (TBC) of a class A is

$$TBC(A) = \sum_{All\ \ K \in System} BC(A,K)$$

where K is a class in the system to which A is directly coupled. This metric measures basically the degree in which the class directly collaborates with all the classes in the system. As above, it can be computed on two coordinates, so that we have $TBC_I(A)$ (the total coupling intensity), and $TBC_D(A)$ (total coupling dispersion.

6.1.2 Acquaintance Metrics

The coupling measures can be used for assessing the collaboration between a class and a distributable feature, for instance by adding all the BC values between a certain class and all the classes in the cluster representing the distributable feature core. However, in real-world applications it is not enough to calculate the simple class-to-class dependencies, as the relation between a distributable feature and the other classes involve more complex collaborations. Classes that do not belong to the cores may be directly coupled with the classes that form the core of a particular feature, but they may as well collaborate with the distributable feature core via intermediary classes. Therefore, our measurements must take into consideration the *indirection levels* between classes, so that the assessment is applicable to all the classes in the system.

6.1.2.1 Class-to-Feature Acquaintances

In order to measure the strength of collaboration (acquaintance) between a class C and a core of distributable feature F, we define a metric that we call *Acquaintance with Distributable Feature* (ADF). Its definition has two parts as follows:

Direct Acquaintance. If class C is directly coupled (has a direct dependency) with at least one class K that belongs to F we consider that class a *direct acquaintance* of the distributable feature F. We calculate the ADF value for such a class as:

$$ADF_{direct}(C, F) = \sum_{classKinF} BC(C, K)$$

In other words, for direct acquaintances the ADF metric is the sum of the intensity of all bidirectional couplings (BC) between the acquaintance class and the classes belonging to the distributable feature core. For example, in Figure 6.1 the middle class in column labeled *0* is directly coupled with both classes of the distributable feature core and therefore its ADF value is 9 = 3+6

Indirect Acquaintance. Classes that are not directly coupled (do not have direct dependencies) with classes in the distributable feature core are called *indirect acquaintances*.

We say that an indirect acquaintance class belongs to the *n*-th *indirection level* when, in the dependency graph, there are $n-1$ classes between it and the closest class belonging to the distributable feature core. For such a class, the ADF metric is computed relative to the ADF values of the classes belonging to the *(n-1)*-th indirection level, the one that is closer to the distributable feature.

The metric is defined as follows:

$$ADF_{indirect}(C_n, F) = \sum_{classK_{n-1}} ADF(K_{n-1}, F) \cdot \frac{BC(C_n, K_{n-1})}{TBC(C_n)}$$

We compute the *Acquaintance with Distributable Feature* between each class K and a given distributable feature F using an iterative calculation that starts with the classes that are directly coupled with the respective distributable feature core, then considers the rest of the classes in the system. The procedure is presented in Algorithm 6.1.1.

The values of the Acquaintance metric for the classes that are not directly coupled with the given distributable feature core (F), are calculated by taking into consideration their dependency on the classes that were processed at the previous steps in the algorithm.

A weight is applied to each acquaintance values taken from the neighbors, which is a subunitary number that is calculated as the ratio between the coupling of the class with its neighbor and the total coupling of the class with the entire system ($\frac{BC(C,K_{n-1})}{TBC(C)}$). Thus, we ensured that the influence of the distributable feature core is getting lower at each step farther from it. The applied factor is in fact the degree in which the dependency on the neighboring class is important in the class' collaboration with the entire system. A low factor means that the class is mostly involved with other classes in the system, rather than with the neighbor that links it with the distributable feature core; therefore, the influence of the distributable feature core via this neighbor is not that important in the class' purposes. Reversely, the more class C is involved in collaborations with classes that are closer to a distributable feature, the more it "receives" from their ADF value.

Algorithm 6.1.1: COMPUTEADF(C, F)

```
main
  currentSet ← all classes directly coupled
     with the feature core
  for each class K ∈ currentSet
    do ADF(K, F) = ADF_direct(K, F)
  while true
    do { prevSet ← currentSet
         currentSet ← all classes directly coupled with the
           classes in prevSet
         if currentSet is void
           then break
         for each class K ∈ currentSet
           do { ADF(K, F) = 0
                for each class L ∈ prevSet
                  do { if L neighborOf K
                       then ADF(K, F) = ADF(K, F)
                                        + BC(K,L)/TBC(K) · ADF(L, F) }
```

For example, in Figure 6.1, the *ADF* value for the upper class in the n-th indirection level is computed as follows:

$$ADF = 4 \cdot \frac{2}{8} + 8 \cdot \frac{1}{8} = 2$$

Note that we assumed that all the classes it depends on are visible in the picture and therefore the *Total Bidirectional Coupling (TBC)* is 8 = 2+1+2+3.

6.1.2.2 System-wide acquaintance measurement

Another metric we defined so that we characterize the classes involvement in the system's functionalities is the *Total Acquaintance with Distributable Features (TADF)*. It is calculated as the sum of all *ADF* values for the respective class:

$$TADF(C_n) = \sum_{i=1}^{no.offeatures} ADF(C_n, F_i)$$

The *Total Acquaintance with Distributable Features* of a class is a metric that measures the degree in which the respective class is involved in *all* the distributable features. This means,

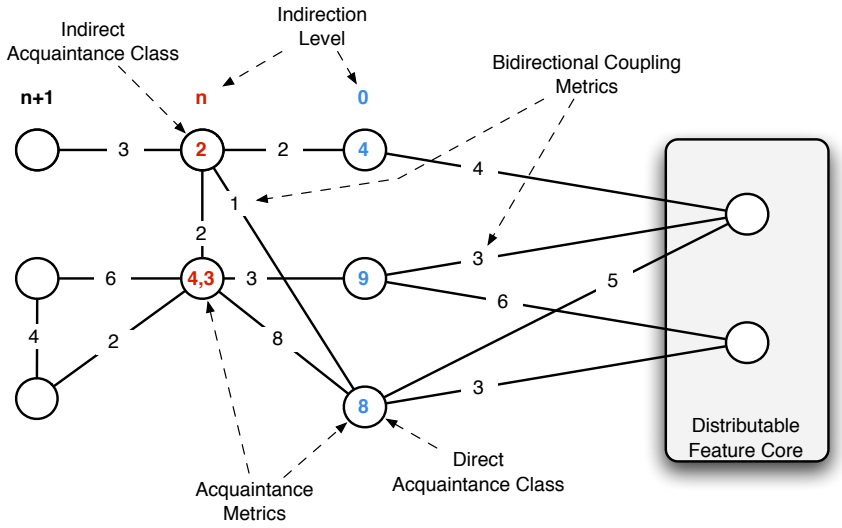

Figure 6.1: Computing the ADF metric

effectively, that we have a way of quantifying the contribution each class has in providing the distribution-aware functionalities of the system. Consequently, we can consider the *TADF* values of the classes in the entire system to obtain information about the overall involvement of the application in the distributed activities. For this purpose we define and use an *Average TADF*, as follows:

$$AverageTADF(System) = \frac{\sum_{K \in System} TADF(K)}{NOC(System)}$$

where $NOC(System)$ is the number of classes in the application.

High values for the Average TADF are specific to systems that are mainly concerned with distribution-related activities, while low values show that the system is mainly involved with local tasks.

6.2 Interpretation

When analyzing the classes outside the distributable feature cores, the metrics we introduced in the previous sections provide means for a detailed characterization of the system's distribution awareness.

These assessments can be described in form of *detection strategies* [73], in fact metrics-based rules for

- drawing conclusions regarding the properties of the system classes or of the application itself which are relevant to the study

- describing patterns that identify design fragments with a correlated set of quantifiable properties.

One key element in writing such detection strategies is to find the adequate thresholds for the metrics involved in the detection strategy, and to use these thresholds to classify or discriminate between the target entities. Choosing the threshold values is a well-known metrics issue, and it is mostly done based on experimental evidence [61, 59]. For this reason, we decided to use *threshold identifiers* for the detection strategies described next, rather than raw numbers, as these identifiers do better encapsulate semantics than particular threshold values, and are less volatile. Based on experimental evidence, the threshold identifiers will be associated with actual numbers later in this dissertation when we will describe the case studies .

In this section, we will interpret the metrics defined above and describe a set of detection strategies that quantify this interpretation.

6.2.1 Involvement in Distribution

At the class level, the *Acquaintance with Distributable Feature* metric shows for each class its involvement in providing a distributed feature. In our approach, we measure all the possible *ADF* values for the classes outside the core, that is, for all the identified distributable features. We can consequently observe the various degrees in which a class participates to the distributable features, and find patterns of involvement characteristic to the analyzed system.

The *Total Bidirectional Coupling* metric shows the collaboration of a class with the entire system. We can use this measure in combination with the *TADF* values to understand what is the proportion of the class' involvement in distribution-related tasks over the local-concerned ones. We can thus identify more classes that are specifically built for distribution, besides those that we have already isolated as belonging to the core. As a consequence, we can consider these newly detected classes as part of the distributable features themselves. Moreover, we can find the classes that mainly or even exclusively participate to local features, so that we can isolate the parts of the system that are not concerned with the distribution.

6.2.1.1 System Distributed Awareness

The *Average TADF* is an important metric through which we can make system-wide characterizations. Furthermore, the extent of involvement of an application in distribution-aware activities can provide useful information regarding the way the respective distributed system was designed.

For example, if a system claims it was specifically built to be decentralized and distributed but its average TADF is low, we can suspect that the respective system either doesn't follow its claims, or it has some distribution-related design flaws.

We use the *Average TADF* to make assessments about the *Distributed awareness* of the system, and our first detection strategy classifies the entire application as:

- distribution-aware when the average *TADF* is high
- locally-concerned when the average *TADF* is low

The 'high' and 'low' identifiers are assigned values depending on the size of the application.

6.3 Patterns of Acquaintance

Up to this point, our analysis approach has identified the main distribution-related functionalities, by isolating their core entities. We are already able to make assessments on their nature, and we can also assess the interactions between them via the network. Further, by applying the Average TADF measure, we have gained the capability of measuring the overall distribution awareness of the system, the importance the distribution plays in its design goals. With these important items of understanding in mind, we can now proceed to analyze at a finer grain the interactions that the system entities are part of in respect to providing the distributable aspect of the application.The metrics presented above were specifically built for this purpose, and it is time to put them to good use.

Our goal is to understand how the system classes are involved in the distribution functionality, and how separated the already identified distributable features really are in the system's code. Moreover, we need to assess the relation of the distribtion-aware parts of the system with those that are only concerned with local activities, and thus to be able to identify the classes that do not depend on the distributed aspect.

For this purpose, we analyze each system class in turn, apply the acquaintance metrics we defined, and assess its collaboration with the different types of functionalities in the application. The class involvement is different from case to case, as some can be built to contribute to a single functionality, while others can be used by more than one features in the system. In order to understand them, we need some sort of classification of the different types of involvement, so that we can label the classes in rapport to their utility in the system. We consequently need to identify the most relevant *patterns* of collaboration within a distributed system, and the analysis needs to find how the system classes meet these patterns.

This section presents the patterns related to the analysis of distributed software systems that we have identified as relevant for making important assessments about the distribution-related functionality. We define them in terms of detection strategies made at the class level, that can be applied by the analysis in order to discriminate between classes. In accordance to the metrics they are built on, we call these strategies *patterns of acquaintance* and use them to quantify the class involvement in providing distributable features.

The patterns of acquaintance are all metrics-based, and focus on the relative proportions between the total acquaintance and the total coupling metric for a class. This technique provides us with a clear view on the ration between a class' involvement in a particular aspect, and its total collaboration in providing the features of the system, thus making possible accurate assessments on its importance for the studied aspect.

The two main metrics we use at the class level are:

- the *Total Acquaintance with Distributable Feature* (TADF) and
- the *Total Bidirectional Coupling* (TBC), as defined in Section 6.1.

6.3.1 Significant Feature Acquaintance

One of the most important aspects when studying the classes that are outside the core of distributable features is understanding their involvement in the distributed functionality, specifically, identifying those classes that are strongly related to the distributed aspect. They are highly

relevant when an analysis aims to understand where a line can be drawn that delimits the distributed part of the system from the local one.

This provides the engineer with the information that allows for decisions related to the system's evolution in respect to its distributed nature. For instance, it can indicate which classes are most affected when the communication-related technology changes, and which classes are to be restructured when the importance of the distribution-related system activities needs to be reevaluated.

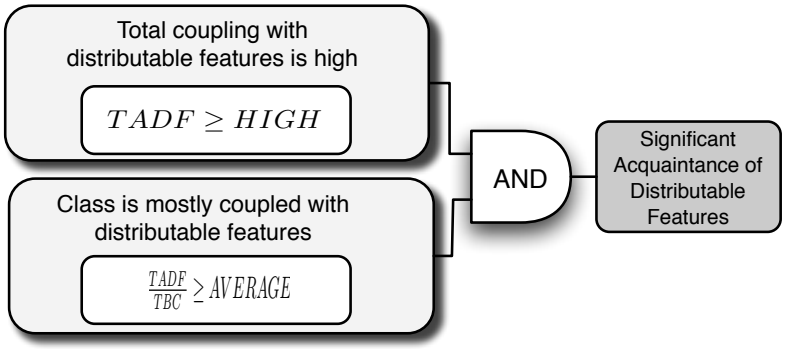

Figure 6.2: Significant Acquaintance

The classes most involved in distribution are those that work together in a higher degree to provide distributable features, and their activity outside this concern is limited. In terms of our metrics, this translates in a high ratio between the total acquaintance with the distributable features (*TADF*) and the total bidirectional coupling of the respective class (*TBC*). Indeed, the *TBC* shows how involved the class is in the overall system activities (distributed or otherwise), while *TADF* shows the involvement in the distribution-related features. Nevertheless, this is not sufficient to make a correct assessment. In any application there may be many classes that follow this pattern (have a low $TADF/TBC$ ratio), but their actual contribution is so small that they can't actually be considered important. This is why our detection strategy for this case also considers the size of the distribution-related contribution when selecting the classes. We call the classes that have a high contribution with the distributed functionality *Significant Feature Acquaintances*, and the strategy that detects them is presented in Figure 6.2.

A class that conforms to this pattern has a substantial collaboration with the distributable feature(s), as most of its dependencies are related to the distributed part . At this point, we can analyze the classes at a deeper level, and can distinguish two subcategories that fit this description:

- Classes that have a high *TADF* because of their high acquaintance with a *single* distributable feature. In other words, one *ADF* value for the class is very high, while the others are minimal, or even zero. This kind of classes are basically built for providing

that single feature, therefore they are highly relevant for understanding that distributable feature itself.

- Classes that have a high *TADF* because of balanced collaboration with two or more distributable features. These classes are not exclusive to a single functionality, they rather provide services for several distribution-related tasks. They may link the features together, and they can be classified as a special category of *distribution connector classes*. As their *TADF* is high, these classes are important in a restructuring context, qualifying as the main candidates for redesigning when trying to completely separate the features (*e.g.*, in order to deploy them on different locations).

6.3.2 Local Feature Contributor

The other case of entities that must be isolated when assessing the ratio between the distributed and local functionality is that of classes that are very little involved in the distribution-related activities. Many distributed systems contain an important number of such classes, and their presence can be an indication of various traits of the application.

Locally-concerned classes and their activities can be necessary to support the distributed functionalities, by, for instance, logging the system runtime events, managing the authorization processes for users, archiving data or providing user interfaces. On the other hand, there are applications where classes that act only locally represent the vast majority of classes in the system, and the distributed aspect is the one that plays only a supporting role. Identifying these classes and assessing their characteristics in rapport with the distributed functionalities is consequently very important for an insightful analysis.

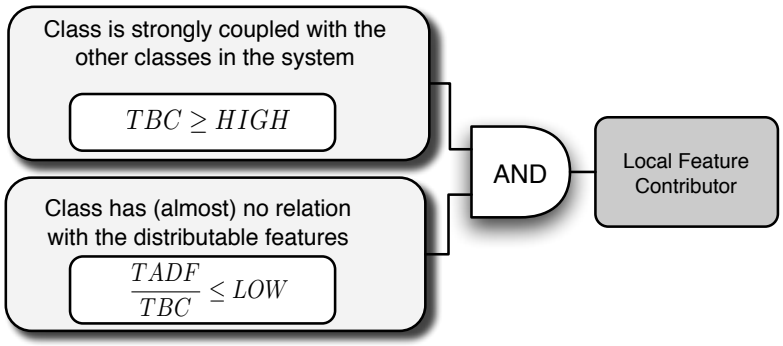

Figure 6.3: Local Feature Contributor

Unlike the significant feature acquaintances, the collaboration of these classes with the distributable features is low, while their other collaborations occupy their entire activities. Using the metrics we defined in Section 6.1, this means the ratio between the total acquaintance with the distributable features (*TADF*) and the total bidirectional coupling of the respective class (*TBC*) is very low. Moreover, to eliminate classes that have a small overall involvement

in the system (which, as in the previous case, would make their study of little relevance), we must also consider their absolute *TBC* value when selecting the relevant cases.

We call the classes that have significant involvement in local features *Local Feature Contributors*, and the strategy that detects them in the application is presented in Figure 6.3.

The classes that fit this pattern can be of two types:

- they implement one of the system's non-distributed feature;
- they are local utility classes that the local or distributable features use.

6.3.3 Connector Class

Not all classes in a distributed software system can be placed with a high certainty in the category of distribution-concerned entities or in the locally-acting one. More often than not, there are classes that actually place themselves at the frontier between the local functionalities and the distribution-aware concern of the system.

There is seldom a clear separation between the two aspects, and it is natural that the collaboration between the two parts is usually a strong one, as the overall goals of the subsystems are the same. Consequently, the classes that link the two types of functionality become highly relevant for understanding the system, as they can provide indications on how the distributed and non-distributed parts work together for achieving the design goals.

We call this type of entities *connector classes*, and we have built a strategy that detects them among the classes in a distributed software system.

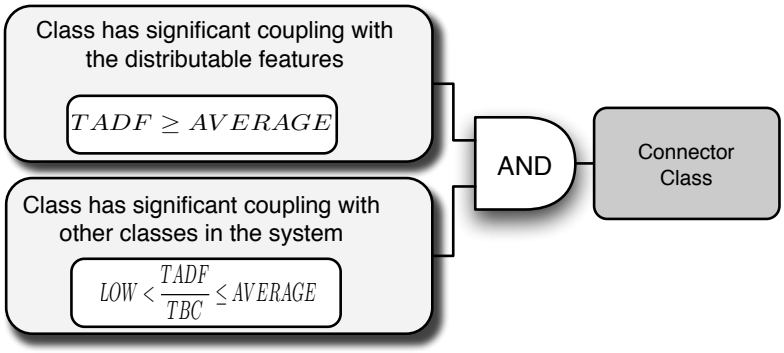

Figure 6.4: Connector Class

Ideally, classes that link the local and distributed functionalities, would have an equal involvement in both of them. In terms of the metrics we defined in Section 6.1, their total acquaintance with the distributable features (*TADF*) over the total bidirectional coupling of the respective class (*TBC*) would be equal or very close to to 0.5.

However, we must analyze the class involvement in respect to the other classes in the system, and therefore the characterizations must take into consideration the overall involve-

ment of the other classes, too. Therefore, we consider that all the classes that have a low to average *TADF*/*TBC* ratio can be candidates for inclusion in the connector class category.

The next filter is based on the fact that we are interested in the linkage between distributed features and the local ones, with a particular emphasis on the distribution. Moreover, as in the previous cases, we need to avoid considering classes with small involvement, as they cannot add relevant enough information to our knowledge.

Therefore, we take into consideration only the classes that have an over-the-average *TADF*. In combination with the other criterium, our strategy of detecting connector classes addresses *those classes that have a significant amount of collaboration with the distributed part of the application, while most of its collaboration is still with the other classes in the system*. The strategy is described in Figure 6.4, and applying it helps us understand the main linkages between the two parts of the application.

6.4 Visualization

Calculating the values implied by the metrics, and applying the detection strategies above on the system classes provide the analysis with comprehensive data regarding the collaborations within the application, and the ratio of importance of the distributed functionality over the non-distributed one.

Nevertheless, the software understanding field often necessitates insights that can be only obtained by relying on the engineer's experience and direct involvement. As processing large amounts of raw data is usually difficult for a human being, an approach that aims to improve the efficiency of the understanding enterprise must provide means for presenting the data to the user in a way that facilitates easy identification of the studied system traits.

In this respect, our methodology places the human at the heart of the process, by providing a set of visual representations of the system characteristics. Similar to the parts of the methodology described in Chapter 5, we believe that the identification of the various patterns of collaboration is greatly facilitated by using software visualization techniques.

Using software visualization, the approach synthesizes the large amounts of gathered data into a set of diagrams that present the collaborations between the system classes and the distributable features, while capturing the total bidirectional coupling traits of the classes. The characteristics are presented simultaneously, so that the engineer is able to see at a single glance all the cooperating aspects that describe the system.

The specific visualization perspective we have defined enables the engineer to easily recognize the categories of classes identified by the pattern detection strategies described in the previous section. The visualization is called *Feature Affiliation Perspective*, and it it is part of the Distributable Features View we introduced in Chapter 5.

6.4.1 The Feature Affiliation Perspective

This perspective (Figure 6.5) uses visual elements to show the impact the distributable features have on the classes that were left outside the core graph by the first phases of the analysis. It is designed to reveal to the engineer the level and profile of the collaboration between the classes and the distributed features of the system, so that the category the class belongs to

is easily identified. At the same time, the total collaboration for each class is shown, so that the collaborations can be put in perspective.

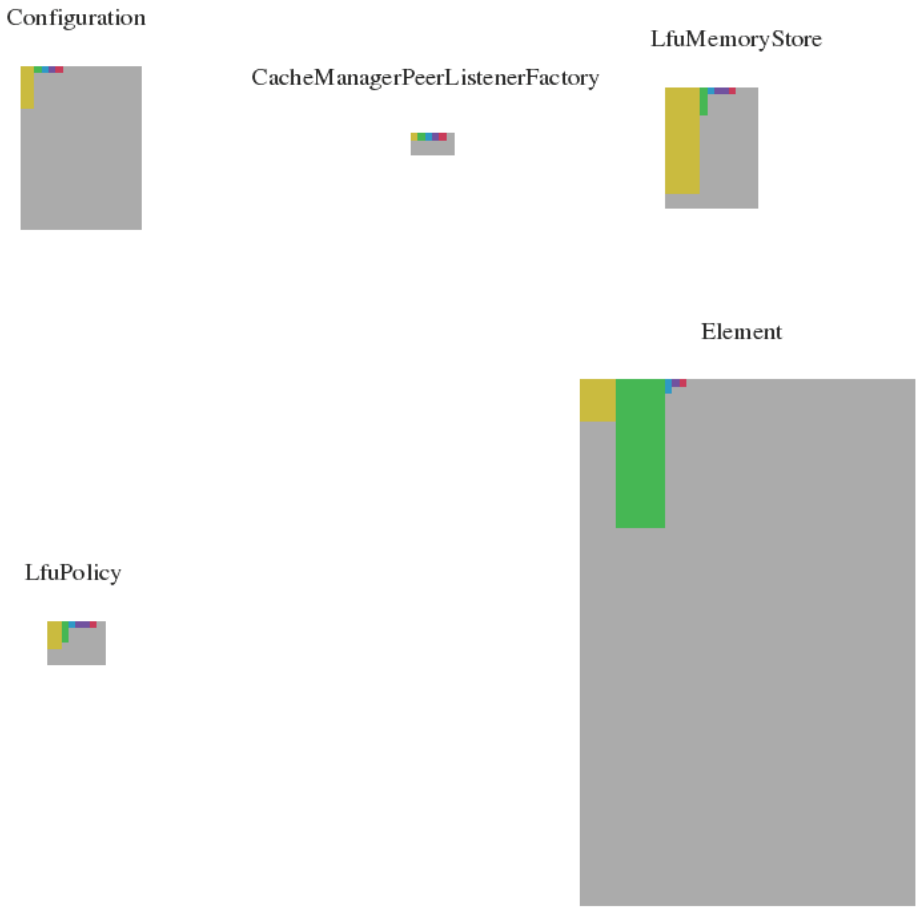

Figure 6.5: Example of Feature Affiliation

Each of the classes that do not belong to a *core of distributable features* is drawn as follows:

- *Total collaboration.* A light gray rectangle shows the total bidirectional coupling of the class with the other classes in the system. The *height* of the rectangle is proportional with the *intensity* of the coupling (number of collaborations), while the *width* is proportional with the *dispersion* of coupling (*i.e.*, number of collaborators)[59].

- *Distribution-related collaboration.* Within the light gray rectangle, we display a set of colored rectangles, one for each distributable feature which the class collaborates

with. The color of the rectangles corresponds to one of the colors from the *Distributed Architecture Perspective* we described in Chapter 5.

The *height* of each colored rectangle is proportional with the *intensity* of the Acquaintance with Distributable Feature metric value calculated between that class and a particular distributable feature. The *width* is proportional with the *dispersion* of that same metric.

The colored bars are placed in the upper-left corner of the gray rectangle, so that we can visually ascertain the ratio between the colored and gray areas, actually seeing the importance the different distributable features have in the functionality of the respective class.

From the point of view related to the software visualization field, the *Feature Affiliation Perspective* effectively extends the concept of *polymetric view* introduced by [58], by "embedding" within one polymetric rectangle a set of other (correlated) polymetric rectangles. We call this *Composed Polymetric View*, and to the best of our knowledge it was not used before as such.

Figure 6.5 presents an example of the Affiliation Perspective for several classes in one of our case studies.

6.4.2 System-level Visual Assessment

In Section 6.2.1 we identified a strategy that assesses the system distributed awareness by calculating the *Average TADF*. Systems having a high value for this metric are strongly involved in distribution-related activities, while application with a low *Average TADF* are mostly concerned with local tasks.

Using visualization, the engineer is able to assess these traits without measuring the system directly, by simply looking at the diagram we build with the *Feature Affiliation Perspective*. The picture contains all the classes in the system, with their collaborations depicted as described above. The engineer only has to look for the overall presence of color in the diagram, and can make a rapid evaluation of the system's distributed awareness:

- a visualized system where color is visible in many places and is relatively dominant over the grey areas is one that has a high involvement in distribution-related tasks;
- a diagram containing mainly areas where gray is visible is concerned with local activities, and its distribution awareness is low.

Most application fall somewhere in between the above extremes, and the visual evaluation is very helpful in assessing the global relation between distribution-aware and locally-acting system entities.

6.4.3 Visual Patterns of Acquaintance

When applied to the system classes, the visualization technique provides means for easy identification of the collaboration patterns we have introduced in the previous section. The engineer can identify the relevant classes that belong to one category or another by simply looking at the diagram, and by seeing the colored patterns for each drawn class.

Figure 6.6: Example *Big Color Spot* class

For each of the detection strategies related to the collaboration traits, we have identified a visual pattern that the engineer can use as reference when looking at the visualized system classes. The patterns represent the direct interpretation of the visualization technique, and are presented below.

6.4.3.1 Pattern 1: *Big Color Spot*

This pattern corresponds to the detection strategy that identifies the Significant Feature Acquaintance classes.

Classes with this pattern, appear in a Distributable Features View as having one or more dominant color spots, so that only a small gray area is visible (Figure 6.6). This fits the definition of the detection strategy, which considered only high values for the *TADF* metric (corresponding to the colored spots in the view), and higher than average values for the *TADF*/*TBC* ratio (visible as little gray in the image).

As when describing the detection strategy we can identify the two types of significant acquaintance classes:

- Classes where only one color is dominant, are the significant acquaintances for a single distributable feature, as one *ADF* value is much higher that the other *ADF* values for the respective class, which makes a single color dominantly visible.
- Classes where two or more colors are visible in sufficiently large areas to be easily spotted represent the classes participating in more than one distributable feature. They have at least two relatively balanced *ADF* values, and qualify for the category we previously called *distribution connector classes*.

6.4.3.2 Pattern 2: *Big Gray*

This pattern corresponds to the detection strategy that identifies the Local Feature Contributor classes in the system.

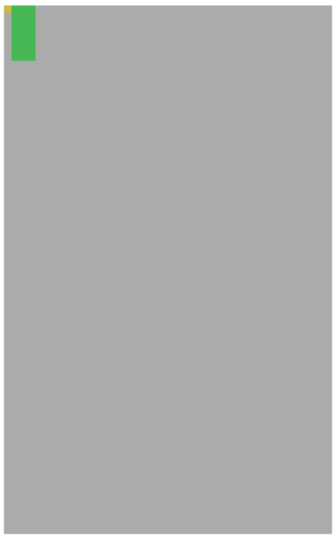

Figure 6.7: A *Big Gray* class

A *Big Gray* class (Figure 6.7) can be observed in the visualization as one that presents a large gray area, while the colored areas are small in comparison. The large grey area is present because their total bidirectional coupling with system classes (TBC) is high, therefore they have significant participation in the system's functionalities, yet most of these functionalities are local. The trait is confirmed by the fact that the colored areas are comparatively small (or even non-existent), therefore the *TADF*/*TBC* ratio is low.

6.4.3.3 Pattern 3: *Color Spotted Gray*

This visual pattern corresponds to the detection strategy responsible for finding the classes that act as connectors between the distributed and non-distributed functionalities.

The visualized classes in this category are shown as dominantly gray rectangles, yet significant color spots are also visible. Figure 6.8 shows an example of such classes. The ratio between the colored and gray areas can be put in perspective with the other classes in the system, so that the connector functionalities are identified in relation with the overall system traits.

The dominant gray areas fit the description in the detection strategy where the *TADF*/*TBC* ratio is considered below average. The colored spots are the expression of classes having higher than average Total Acquaintance with Distributable Features, which is again in accordance with the specifications of the detection strategy.

The view has the advantage that the engineer can easily identify the connector classes, and, moreover, by looking at the color spots he or she can *see* without further analyzing the code which are the particular distributable features that depend in a higher degree on the

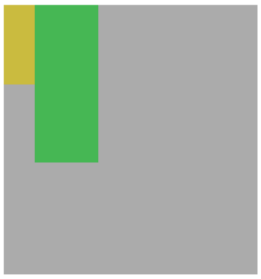

Figure 6.8: A *Color Spotted Gray* class

non-distributed functionalities of the system.

7

Restructuring Support

The previous steps of our methodology defined a process that starts with the source code of an existing distributed application, and uses the information about the communication technology to extract valuable information about the system. The approach isolates the separate distribution-aware functionalities, offers an architectural overview of their remote interactions, and provides means to thoroughly understand the involvement of every system class in providing the respective functionalities. Complemented by a visualization technique and a software tool that automates most of the tasks, the process enables engineers to significantly add to their understanding of the analyzed systems.

Software systems understanding is a very important task for an engineer, often being a goal in itself, as the knowledge items gained through detailed analyses represent valuable tools for long-term controlling and maintenance of the applications. Software applications are in a continuous process of evolution, therefore their internal structure tends to degrade in time. A methodology for understanding software can consequently be enriched by providing at set of techniques that enable the engineer to reason about the restructuring needs, and support the process of redesign.

We believe that, when addressing this issue, an approach should concentrate on finding simple yet highly relevant means of manipulating the code when trying to change its structure. They must be relevant to the domain describing the target systems so that the approach fits the most important issues that occur within the respective application class, and they must be simple in order to keep the interaction between the engineer and the code artifacts at an easy, manageable level.

Distributed object-oriented applications differ from their 'locally-acting' counterparts in that they are designed to fit the necessities of (usually) heterogenous networks of human activities, systems of tasks executed at geographically distinct locations. Organizations need to integrate their workflow in the wide dispersion of their worksites, and this usually requires a complex distributed system consisting of balanced, sometimes relatively autonomous, software nodes working together for the same goal.

Unfortunately, the real-world applications are, in many cases, far from such a scenario. Many 'distributed' applications consist of several or many instances of the same (or few) lightweight client(s), connecting from different locations to a single, oversized server, that

centralizes the entire workflow. The server provides all the features needed by the network, even though most locations may actually need only a subset of these services, those that specifically address their local concerns. Such a system is unpractical, and will not be able to function efficiently for long, as it will not cope with the situations like the growing of the organization it serves, or the evolution of the features it must provide. Consequently, at some point in its future, the system will have to be restructured, particularly by *extracting* the individual services and (possibly) deploying them at or near the locations that need them most.

This chapter presents our approach to restructuring distributed object-oriented applications by focusing on the nature of the restructuring needs that often arise during the system's lifetime. It uses an extraction-based technique which isolates inter-related pieces of software, providing means to both assess the layout and the cost of the redesign, and to enhance the understanding of the system.

Extraction of services and, generally, of groups of interrelated software entities is the basic method we use in our approach to restructuring distributed software. At the point the restructuring step occurs in our methodology, a lot of valuable knowledge about the system was already gained by the engineer, therefore the extraction itself begins with a lot more than qualified guesses about the 'what', 'where' and 'how' concerns of restructuring.

7.1 Criteria for Restructuring

Regardless of the technology involved, a distributed application must be designed so that it serves efficiently its purpose. If structured as a set of services, it must provide them so that they both minimize the network communication, and make intelligent use of the geographical distribution of the components. Most commercial distributed applications manage the resources and workflow of real-life distributed organizations, such as companies having multiple branches, or individual sites that need to integrate their work. We argue that, in order to be efficient, the services in such a system must be:

- **Small.** Regarding the actual size of the service, we believe it should consist of as few classes as possible, so that it is manageable both by the developer, and the user. A smaller service will be easier to maintain, and easier to configure in the production environment, for instance by deploying it at a different location without affecting the overall system functionality.

- **Focused.** In what concerns the application's goals, a *focused* service is one that, regardless of its size, limits its functionality to a specific, accurately targeted purpose, rather than combining several, loosely-related, features that may or may not be needed by all its clients. Keeping the purpose in sharp focus during the entire life cycle of the services will provide the users of the system with a great degree of flexibility in assigning (and changing) the functional roles of the different locations the organization is involved in.

- **Decentralized.** By *decentralized* we refer to services that balance their functionalities and minimize their inter-dependencies, rather than encapsulating features needed by the entire system. A centralized feature needed throughout the system will almost always

become a communication bottleneck when the system is deployed, and the dependance of the other components on the centralized activities will hurt the system's performance, flexibility and usability.

Designing the services under the above three coordinates will help create a system that is well-distributed, i.e. it has an optimum services/locations ratio. Small and focused services deployed at the locations they are needed most will ensure better scalability, by providing easier methods of extending the system when the demands grow. Adding decentralization to the mix will help avoid situations when a lot of the components connect to the same location while their actual purposes are very different (heterogenous) and would be better served by several custom services. Moreover, the three qualities above will provide a highly maintainable and evolvable system, because their adoption minimizes the in-system dependencies.

Real-life systems are definitely not optimal [23, 32], and this may happen because of at least two reasons:

- they are *poorly designed* from the start – many applications are simple, client-server, architectures that consist of a single, centralized server and one or several clients using its services. The services themselves are gathered together at the server location for no other reason than the simplicity of doing so. Many systems are deployed in application servers, that provide a specific functionality needed by the system: web-based user interaction, transaction management or persistency. While the application servers themselves do not impose architectural constrains that prevent a flexible design, it is simply easier to deploy all the main functionality at their precise location, avoiding an arguably more complex approach that distributes the system features over the network.
- they *evolved* in time, due to change requests arrived at different points in the system's life, and the current structure does not follow anymore the (otherwise good) design principles it started with.

The need to restructure a distributed system will arise whenever the conditions above are met, and approaching this necessity implies two important phases:

- understanding the system, by mainly identifying the distribution-aware features and their relation to the available services;
- manipulating parts of the code (services, features and/or other groups of inter-related classes) so that the accurate focus and size of the remote acting entities is achieved.

Our approach addresses these concerns, by providing both a methodology and a tool to tackle the issues of understanding and restructuring distributed software applications.

7.2 Restructuring the System

Applying a process that analyzes the software can help the engineer gain a significant understanding of the system, detailed enough to enable complex activities like those related to the maintenance and evolution. The next step is to provide a flexible way of manipulating the code artifacts as a means for system restructuring. This goal can be viewed from two perspectives:

- a *characterization of the restructuring* **results**, which should isolate suppler (more flexible) entities, easier to maintain and/or redistribute over the network,

- *a characterization of the restructuring* **process**, which should provide easy ways to try different restructuring scenarios and to select the best one as the outcome.

The first perspective is definitely the most important one, and our approach follows it while also considering the issues implied by the second one. This way, we achieve a fair, natural, degree of consistency between the process itself and its projected outcome.

Our restructuring approach is built around a simple, yet versatile technique: the *extraction* of those subsets of the system classes that may or will form the core for a distinct, suppler system feature.

In order to find the best restructuring scenario, the engineer can apply the process to obtain two types of work items that can be used both as intermediary results in iterative exploration of the various possible versions of restructuring, and as the final outcome of the approach, to characterize the future layout of the chosen scenario:

1. **The forecasted layout** of the redesign or (as an intermediary result) the forecasted layout of the extraction/redesign attempt. This provides an overview of the changes that will most probably be necessary after extracting a specified set of classes from the system.

2. **The cost** of the redesign or extraction attempt, a numerical score that shows how difficult will it be to extract the respective classes from the existing code.

7.2.1 Extraction Units

The first step in our approach is building an *extraction unit*, which we define as the set of classes that represent the starting points for the extraction, the classes that definitely need to be extracted in order to delimit an independent feature within the system.

The reasoning which includes classes in extraction units is applicable to different sets of system entities, as follows:

- analyze the already isolated entities (such as the distributable features) and find classes that seem to be working together in a tighter manner than the other ones;

- look at the system from a strictly service-oriented perspective and identify the frontier classes that *seem* to provide different features than other frontier classes in the same group;

- apply the same approach of finding functional clusters to the locally-acting groups of classes.

In other words, we have to be able apply the same type of reasonings at any level of detail, from looking at small sets of individual classes to approaching the entire system – the one thing we would need in any of these cases is a way to narrow our search for interesting cases to be selected as members of extraction units.

7.2.1.1 Choosing Classes for the Extraction Units

We support this need by focusing the extraction process to two versions of class selection strategies that, in our opinion cover the most interesting cases in the context of object-oriented distributed systems.

Strategy 1. Service identification. This strategy is based on the premise that in distributed applications, a description of a service (made in RMI through a remote interface) usually refers to an independent feature the system provides. As many components that run on a location publish more than one service description, it is very interesting to see how the extraction of each such description and the related classes influence the structure of the system. For Java/RMI, this translates to defining an extraction unit that consists of one remote interface and the classes that implement it. In this particular case, when a class implementing a remote interface is in turn the base for an individual hierarchy, we include the hierarchy too, as the classes in the hierarchy are certainly directly involved in providing the service.

Strategy 2. Identification of functional clusters. This strategy covers a more general case where we need to identify sets of semantically-related classes that seem to act together more closely than the other classes in the group. By 'group' we mean any set of classes we try to split/reconfigure at a certain point in our redesign, such as a distributable feature, a previously extracted service, or even the entire system. The strategy does not focus on the semantics themselves, but uses a coupling-related metric to identify the most interesting classes: we define for each class C in group G the *In-group adequacy* metric *(IGA)*:

$$IGA(C, G) = \sum_{class\ K \in G} BC(C, K)$$

where *BC* is the bidirectional coupling between two classes: the total number of method invocations and attribute accesses occurring between them, to which we add 1 if the classes share an inheritance relationship (Section 6.1).

Being a coupling-based measurement, *IGA* provides information about the quality of collaboration between a class and the target group. The higher the value, the stronger the collaboration of the class with the classes in the group, i.e. the higher chance that the class is better fitted for the respective group (hence the metric name).

For all coupling-based metrics in our approach, we calculate both dimensions of coupling: the *intensity* of the coupling (number of collaborations), and the *dispersion* of coupling (*i.e.*, number of collaborators) [59].

7.2.2 Visualization as Support for Selection

To support quick and easy identification of the most interesting classes, we defined and used a visualization of the *IGA* for all classes in a group (Figure 7.1). For this purpose, we calculate two *IGA* values, one for the intensity, and one for the dispersion. The visualization draws the dependency graph of the group, each vertex being a class and edges representing bidirectional dependencies between them. If more than one dependency between two classes exist, only one edge is drawn. The *In-group adequacy* is visible in the shape and size of the classes: each

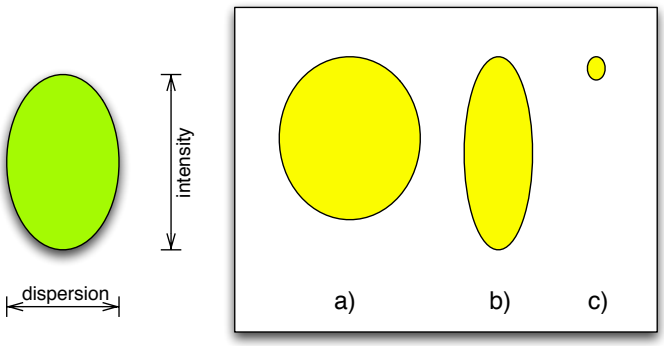

Figure 7.1: In-group adequacy

node is an ellipse, the size of the vertical axis being proportional to the coupling intensity, and that of the horizontal one with the coupling dispersion. As intensity is always larger or equal than dispersion, all classes will have a vertical major axis.

The *IGA* values can be used to analyze the classes in the group in order to find candidates that can be included in the extraction unit. The most visible classes are those that the visualization draws larger, as their *IGA* values are high. The interpretation of the shapes we encountered in our case studies is the following:

- *Large Round Shape* - classes drawn as large, almost circular ellipses (Figure 7.1,a): both intensity and dispersion *IGA* values are high, meaning that the respective class cooperates intensely with many entities (classes) in the group. Classes falling in this category are tightly linked with the group, so that probably extracting them will imply a high cost. At the same time, they might represent a complex feature, and if several such classes are found in different regions of the graph, it may be an indication that several distinct features may occur in the group.

- *Large Elongated Ellipse* - classes drawn with the major (vertical) axis significantly larger than the horizontal one (Figure 7.1,b) – the *IGA* intensity is high while the dispersion is low: the class collaborates intensely but only with few classes. This may describe a localized feature, as the respective class does not need most of the other classes in the group. Classes in this category therefore become good candidates for extraction.

- *Small Ellipse* - classes drawn as small nodes in the graph (Figure 7.1,c) – while their collaboration with the other classes is low and thus they might be easy to extract, the small degree of collaboration also may suggest they are not interesting representatives of features, they probably are only 'accessories' to such features (such as interfaces, utility classes, etc.)

The consequence of the above observations are that the best candidates for including in an extraction unit are the classes in the first two categories. They definitely represent interesting cases, and they are worth looking at in detail when trying to isolate an independent, extractable feature within a given group of classes.

7.2.3 Extraction Process

With the extraction unit prepared, the actual process of extraction can start. Extraction can be done at any step of the redesign, both as a means of quantifying the projected final results, and as a part of a trial-and-error approach which explores possible scenarios of separating parts of the code.

As stated above, the outcome of the extraction consists of two items of interest: a preview of the post-extraction structure of the original group of classes, and a number quantifying the extraction cost.

7.2.3.1 Forecasted Layout of the Extraction

The first item is obtained by applying an algorithm that separates the original group by isolating the extraction unit and the closely related classes. To this purpose, we use a variation of the *Acquaintance with Distributable Feature* metric we have introduced in Section 6.1. The *Acquaintance with Class Group* between a class C and a group of classes G is therefore defined as:

a) If Class C is directly coupled with one or more classes in group G:

$$ACG(C, G) = \sum_{class\ K \in G} BC(C, K)$$

b) If class C is indirectly coupled with the classes, and it sits at the n-th indirection level against the group (there are $n-1$ classes between C and the closest class in the group), the metric is calculated iteratively, using the indirection levels:

$$ACG(C_n, G) = \sum_{class\ K_{n-1}} ACG(K_{n-1}, G) \cdot \frac{BC(C_n, K_{n-1})}{TBC(C_n)}$$

c) If class C is completely disconnected from all classes in the group:

$$ACG(C, G) = 0$$

BC is the bidirectional coupling between two classes, and TBC is the total bidirectional coupling for a class (see Section 6.1).

The algorithm iterates through all the classes in the group (except for those already included in the extraction unit) and calculates for each their acquaintance with the group of classes in the extraction unit. The user can specify a threshold factor t which is applied to perform the actual extraction: all classes for which

$$ACG(C, G) > t * avg(ACG(G))$$

are gathered with the extraction unit classes and are separated from the rest of the group. $avg(ACG(G))$ is the average value of all the calculated acquaintances. We only use the intensity side of the coupling measure, as it better expresses the *strength* of dependence between the entities.

7.2.3.2 Extraction cost

The *extraction cost* is a measure that characterizes the effort that would be needed when performing the actual extraction. Given that the algorithm provides two sets of classes, S_1 and S_2, we consider the cost is proportional with the degree these sets are actually linked together in the current system: the larger the number of calls or references between the classes in the two sets, the higher the cost to actually separate them. We define the extraction cost as follows:

$$EC = \sum_{C_1 \in S_1,\ C_2 \in S_2} \frac{BC(C_1, C_2)}{NOC_s}$$

where NOC_s is the number of classes in the entire system.

The extraction cost is particularly useful when trying different possible scenarios of extraction, in an iterative, exploratory, manner. It provides the engineer with a useful way of distinguishing between paths that may lead to cost-effective restructuring scenarios and those that represent 'dead ends' because the cost becomes too high.

The actual values for the extraction cost vary from system to system, but are consistent (comparable) within the same application. When assessing the costs, the engineer must first conduct a set of preliminary extractions to determine the range of the costs throughout the system. Low values will be obtained by extracting loosely connected individual classes, and high costs are specific to classes that have many dependencies (i.e., they are the origin of many edges in the graph), with medium to high *IGA* values.

7.3 Improving the Understanding

The extraction process described above can be used not only to support the system redesign as a tool, but also to improve the knowledge one has about the analyzed system. Isolating clusters of related entities by starting with a few classes (the extraction unit) can prove an useful tool to capture functional aspects in the system that were not obvious from the start.

The analysis can include steps that focus on interesting classes in the system, suggested by the visualization or by other techniques, and find which are the entities that are closely linked with them, and how isolated the set is from the other classes in the system. The same group of classes can present different types of "interesting classes", and the approach can use each of them separately as extraction units to get various views on their impact in the group: different sets of classes will be generated for each extraction case, and analyzing their layout will help identify both isolated sub-functions (clusters that follow only one or few of the extraction units) and general-purpose functionalities (the intersection between the generated sets).

The information received this way can prove useful to identify new dependencies between some "interesting classes". For example, two or more may exhibit similar behavior when extracting: they cluster around them virtually the same classes. They have a good chance of being related to each other functionally, so that the next step should group them in a single extraction unit and restart the extraction process. Iteratively, this approach can identify functionally-related sets of classes that were not identified as such by the previous steps of the analysis. The granularity of the iterations is subject of experimentation: the threshold

factor for extraction can be tuned to determine the inclusion of less or more classes in each extraction result.

Bottom line, once focused on relatively structured units of code (such as the distributable features), an extraction-based exploratory analysis can help isolating functionalities at an even finer degree of detail.

8

Evaluation of the Approach

The previous chapters presented the methodology we developed in order to support the process of understanding distributed object-oriented systems. The approach is based on a process of reverse engineering existing software applications which starts with the source code and analyses them so that the aspects related to the main system functionalities are extracted. The methodology provides a set of integrated techniques for isolating the distributable features in the system and for assessing the participation of classes in providing them. Moreover the remote dependencies between distributable features are detected so that the interactions specific to the distributed nature of the system can be assessed. The process uses a set of detection strategies to identify the most relevant traits of the system and to understand the patterns of collaboration between the classes. Moreover, the aspect of distributed software restructuring is supported by a technique that enables the engineer to both change the system's layout at different levels of detail, and to better understand the interactions within the application. The approach is metric-based and uses software visualization techniques as support for an in-depth understanding of the system.

The goals of the methodology are to provide, by means of structural analysis the following:

1. System-level assessments regarding
   (a) The importance of the distributed aspect in the system
   (b) The layout of the distributed architecture
2. Class-level assessments concerning
   (a) The collaborations with the distribution-related functionalities
   (b) The relation with the non-distributed part
3. Techniques for supporting system restructuring by assessing the dependencies between groups of classes and facilitating the extraction of separate functionalities

To evaluate the methodology and the techniques it implies we have applied the process to real-world distributed software applications provided by sources in the industry, and to systems available as open-source projects. The applications were designed and built using Java and

RMI as the communication infrastructure, and were of medium to large size in terms of number of classes.

This chapter presents the case studies we conducted and analyzes the results of applying the methodology described in the previous chapters. The goals of the evaluation were:

- To reproduce the real conditions that occur in industry when the need of software understanding arises;
- To use only the source code as input for the analysis;
- To assess the findings by confronting them to the available information sources about the systems.

8.1 Experimental Setup

During the development of our methodology we have analyzed a number of 5 software projects.

All projects were available at the source code level, and were analyzed in similar conditions, as follows:

- The source code was used as they were provided, without any kind of tampering. This simulates the cases when projects need to be understood without knowing anything about their structure;
- The analysis was made without referring to any sources with additional information on the systems (as documentation or developers). This simulates a frequent case when the engineers that need to understand the system cannot contact the original developers, and when the project documentation is insufficient;
- All the steps of the methodology were followed in order;
- We have combined the detection strategies and the visualization techniques to improve the understanding;
- After extracting each item of understanding, we have confronted the results with the external sources (documentation, developers). This step was needed to assess the validity of the approach.

Table 8.1 shows the size attributes for each of the analyzed projects and presents the main system-level metric values we have calculated for each.

CAROL (Common Architecture for RMI ObjectWeb Layer) [1] is a library that allows applications to use different RMI implementations, making Java server applications independent of RMI implementations. It is a medium-size project, and it is widely used as infrastructure for developing applications. JOTM (A Java Open Transaction Manager) [2] is a distributed transaction manager that implements a set of standard application interfaces for Java. It is also medium-sized open-source Java project. As JOTM was built using the CAROL library we decided to analyze them together. The application we have called SPRC is a small student

[1] http://carol.objectweb.org/
[2] http://jotm.objectweb.org/

Name	*NOC*	*NOC in Distr. Feat. Cores*	*Distr. Feat.*	*Sum of TBC*	*Average TADF*
CAROL	155	28	4	1762	14
JOTM+CAROL	218	65	9	4437	29
SPRC	24	9	3	57	4
EHCACHE	93	16	5	717	9
FWS	362	35	2	1051	3

Table 8.1: The case-studies in numbers

project we included because its size allowed us to manually inspect the entire application and thus assess our techniques with more precision.

This chapter presents in detail the findings related to two projects we consider highly relevant for the purpose of exemplifying the our analysis approach. They are relevant because they have distinct domains of application, and they present significant particularities as far as the distributable features and the overall system functionality are concerned:

FWS is a code name we have given a commercial framework for building and executing workflow systems. It was developed by a local software company in Timișoara, and we had access to the entire source code of the framework. Moreover, the actual source-code of FWS that we analyzed contains not only the framework itself, but also some small test applications written to exercise the framework, which provided us with a very interesting situation to analyze.

In the FWS terminology, a workflow is a sequence of local or remote activities executed by pieces of software called *agents*.

The framework's user is a developer (or a team of developers) who instantiates it by writing a set of Java agents that meet the necessities of the particular workflow that is to be enacted. The workflow itself is specified as a state machine whose description is stored in a specific XML file. The agents can be configured to run all on the same machine, or they can be distributed over the network. The communication between agents and the engine that supervises the workflow is done via RMI.

This case study brought an important advantage to our evaluation approach: we had direct access to the developers, thus we were able to verify in detail the validity of our findings.

The EHCACHE system is a widely used, open-source, Java distributed cache for general purpose caching [3]. Its distributed aspect is directly related to the fact that the caching can be done by a group of separate, and distinctly deployed *cache peers* which act together over the network as a distributed cache system. The communication related to the data exchanges between the peers, is done remotely through RMI.

The documentation for the system was available, and it included design-related data which allowed us to verify the assessments of the methodology. Moreover, the project proved to be an interesting case of peer-to-peer technology, which both differentiated it from the 'mainstream'

[3] http://ehcache.sourceforge.net/

applications and provided us with the opportunity of applying our techniques on an interesting type of distributed architecture.

8.2 System-Level Characterization

By analyzing the characteristics of the application, our methodology provides two types of information at the system level:

- An *Overview of the Distributed Architecture*, by identifying the cores of distributable features and analyzing the dependencies between them that constitute the remote communication channels;
- A *Characterization of the Distributed Awareness* of the system by assessing the importance of distribution on the system's design. This is accomplished by measuring the involvement of the classes in the distribution-related activities, based on the *Average TADF* value. As the *TADF* metric expresses the strength of the relation the class has with *all* the distributable features, the average *TADF* tells us about the overall importance of the distributed aspect in the system.

8.2.1 Overview of the Distributed Architecture

8.2.1.1 FWS

By applying the core isolation and clustering techniques, the automated process detected for FWS two distinct cores of distributable features. The first core is made of 28 classes, of which 5 were identified as frontier classes. One of these 5 is a class that calls methods in a remote interface situated in the other detected core. The rest of the frontier entities are remote interfaces themselves, that provide a set of functionalities to remote actors.

We have analyzed the dependency graph representing the first core, and looked at the names of the classes. Doing so, we have found a set of several entities that qualified as interesting classes because their name seemed to suggest the functionality of the respective distributable feature.

This set included remote interfaces (`Interpreter` `WorkflowManager`), and classes like `WorkflowContainer`, `ActiveWorkflow` *etc.*. Their name suggested that the feature dealt with the control of the workflows. We have found that the name of the frontier class that called methods of remote interfaces in the other core was `Executer`. Looking at its dependencies in the graph, we have also noticed that it was directly linked by an edge with the `Interpreter`. This information consolidated our impression that the feature was concerned with executing workflow tasks.

We have contacted the developers of FWS, and they confirmed that the identified classes are indeed the central part of a component of the system called *Workflow Engine*. It represents the main functionality that deals with creating and executing of the different instances of workflows configured by the users.

The second core of distributable feature consisted of only 7 classes and interfaces. The main entities were two remote interfaces called `AgentEngine` and `AgentHandler`. The core also contained a class called `AgentEngineImpl` which was linked by an edge in the dependency

graph to another class, called `Agent`. Our assumption was that the corresponding distributable feature is concerned with the tasks of handling (running) the user agents that are part of the workflow. When we referred to the developers, they confirmed again our assumption. The classes we discovered were the main part of a component named *Agent Engine* that creates and manages the agents running on different machines than the one that hosted the Workflow Engine.

The next step was to analyze the communication channels between the two feature cores. The channels were automatically detected by our tool as direct RMI remote calls. We have identified four communication channels. Two of them consisted in calls from the `Executer` class to the remote interfaces in the second core, while the other two were calls made by a frontier class in the second distributable feature core to remote interfaces belonging to the first one.

This bidirectional dependency strongly suggested that the distributable features established a *peer-to-peer* relation, acting in turns as client and server one to the other. This assumption was confirmed by the developers of the framework: the *Workflow Engine* contacted the *Agent Engine* when the creation of a new agent was needed in order to be run at a remote location. On the other hand, the *Agent Engine* component communicated with the *Workflow Engine* each time the status of the running agents needed to be reported back.

8.2.1.2 EHCACHE

For this system, the core detection and separation approach detected a number of 5 distinct cores of distributable features.

We have called the largest core *Cache Peer Manager* because it seemed to be built around a remote interface (`CachePeer`). It consisted of a total of 7 classes, from which we noticed the `CacheManagerPeerListener`, `RMICachePeer`, and `EhCache`.

A very interesting aspect that effectively surprised us as we didn't know anything about the system beforehand was that `CachePeer` was *the only* remote interface in the entire system. The other identified cores didn't contain remote interfaces, the only frontier entities in their case being classes that referred this singular interface. The other distributable feature cores consequently suggested that they represent different functionalities related to each other.

For example, one of those cores contained class names suggesting involvement in replicating caches over the network (`CacheReplicator`, `RMISynchronousCacheReplicator`, *etc.*). Following the names, we called the respective core *Replicator*.

Another distributable feature core, that we called *Bootstrap* was obviously dealing with initializing the cache, as it contained clases called (`BootstrapCacheLoader`, `RMIBootstrapCacheLoader`). A different core contained a class called `CacheManagerPeerProvider`, and we assumed that it was probably related closely to the listeners in the first core (we called this `Peer Provider`).

The fact that only one remote interface was present, complemented by the above observations consolidated our perception that the discovered features were not parts of *distinct* remote components, but functional aspects of a *single* structural, communication-aware, entity.

To confirm our findings we consulted the online documentation of the system [29]. It showed that the distributed caching aspect involved several different tasks, such as peer discovery, replication, the capability of bootstrapping from remote caches. All the tasks were

executed by a set of several *identical* components, running on different network locations, and communicating with each other through the `CachePeer` interface.

The documentation allowed us to link the functional units we have discovered with the real design-specific functionalities in the system, We confirmed the role *Cache Peer Manager* and *Peer Provider* distributable feature cores identified by our approach, which proved to be features involved in the peer discovery and management, as distinct parts of code that can each communicate remotely for this task. We successfully confirmed the accuracy of two more distributable features cores (namely *Replicator* and *Bootstrap*) by reading the document describing the architecture of EHCACHE.

Although we were unable to also link the fifth core our algorithms identified to a documented feature of the application, the approach appeared to be again useful for correctly recovering an overview of the distributed architecture of EHCACHE.

8.2.2 Characterization of the Distributed Awareness

8.2.2.1 FWS

As our methodology specifies, in order to assess the overall importance of distribution in the system's design, we need to calculate the *Average TADF* metric.

For the FWS system, the *Average TADF* is 3, which is rather small compared to other systems we have processed (see Table 8.1).

This suggests that there is a significant amount of functionality in FWS that is definitely not directly related with the identified distributable features, therefore it is not involved in providing distribution-related functionalities.

Particularily, we have looked at the classes with very low *TADF* values, and identified a group of almost 80 classes that had a *TADF* of 0. This shown that the respective entities were totally unrelated with any of the identified distributable features in the system. By looking deeper at the group and considering their names, we have seen elements that suggested a functionality related with an event-driven user interface. By referring to the developers, our assumption was confirmed. The classes were all part of a utility for creating and editing XML workflow specifications, which indeed heavily relies on user interaction.

When we asked about the seemingly low distribution awareness, the developers told us that the system was initially designed for executing only local tasks, and only later a it was adapted to run remotely-available agents. This explains the low *Average TADF*, because most of the activities the system was involved in, were still local, on the *Workflow Engine* side.

8.2.2.2 EHCACHE

For this system, the calculated *Average TADF* value was 9, significantly higher than for FWS.

Assessing the values for individual classes, we noticed that most of them are acquaintances of the *Cache Peer Manager* distributable feature. At the same time, we noted that this feature is the largest distributable feature in the system, which did not surprise us, as most of the functionality of EHCACHE was related to the issue of cache management. Moreover, by reading the design documentation of this system we found out that since version 1.2 the EHCACHE underwent a significant redesign that transformed it from a monolithic into a modular and distributable system, having the concept of cache peers at the core of its architecture.

The higher *TADF* value (*i.e.*, high distributed awareness in the system) was consequently verified, because a high value is typical to applications that were specifically designed (or redesigned) with the distributed functionality in mind.

8.2.3 Visualization

The process of evaluating the methodology made extensive used of the visualization techniques we have defined, so that we would be able to assess how they help the effort of system understanding.

8.2.3.1 Overview of the Distributed Architecture

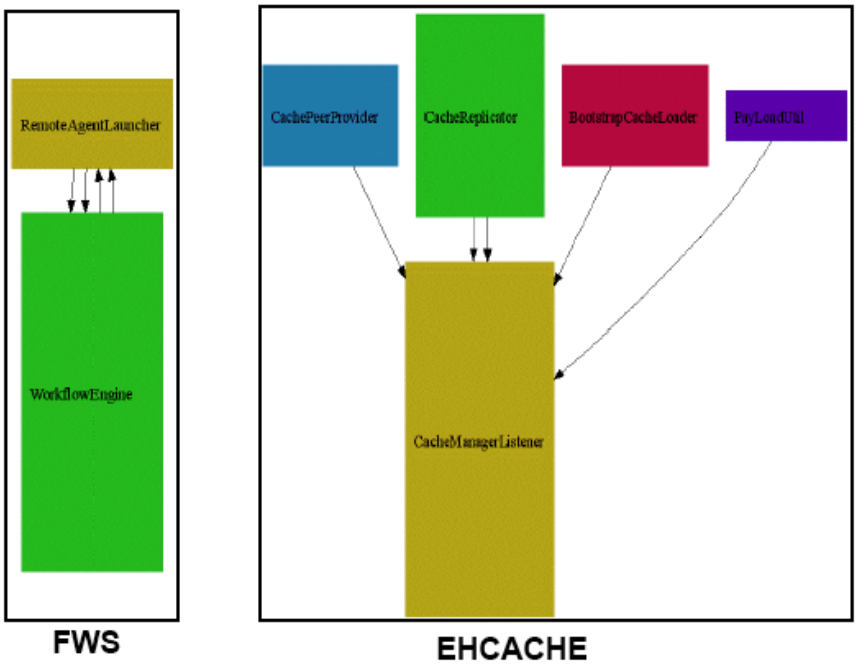

Figure 8.1: Distributed Architecture Perspective of the case-studies

For the part that captured the distributed architecture overview, we found that the visualization was useful to easily understand the dependencies between the classes. This way, the architectural traits (as the peer-to-peer relations) were directly visible in the picture and helped us make the assessments we have discussed above.

Figure 8.1 presents the architectural previews for the two case studies we discuss in this chapter, specifically the corresponding Distributed Architecture Perspective parts of the Distributable Features View.

8.2.3.2 Characterization of the Distributed Awareness

As mentioned earlier, the DISTRIBUTABLE FEATURES VIEW is a visualization of the entire system that captures the impact of the system's distributable features, the functionalities which were intended to be distributed. The dependencies on the distribution are shown in color, while the overall collaboration of classes with the system is shown as gray rectangles. The grey rectangles are overlapped by the colored ones, so that for a class with a mostly distributed functionality, the grey area is less visible.

Therefore, the engineer can look at a DISTRIBUTABLE FEATURES VIEW and *see* is the relative proportion between colored and gray visual elements. If the colors dominate, then the entire system is strongly oriented towards providing the distribution-related functionalities. I, on the other hand, gray is dominating the picture, the system may also be providing other types of features, that are not directly related with the remote communication.

8.2.3.3 Analysis of FWS

A high-level look at the DISTRIBUTABLE FEATURES VIEW diagram for FWS showed a lot of gray spots. This means that a lot of features in the system do not directly relate to the distributed functionalities.

Moreover, we have seen that there were many classes drawn with large amount of gray, classes having significant collaboration that was not related to the two distributable features.

The visualization was the technique that actually helped us to rapidly identify the almost 80 classes we have discussed above, that represented the strictly local, user interface -based, functionality.The respective classes didn't have color spots at all, which showed clearly their exclusively locally oriented nature.

8.2.3.4 Analysis of EHCACHE

When looking at the DISTRIBUTABLE FEATURES VIEW for EHCACHE, we noticed significantly more color than in the case of FWS.

Looking at the color, we have seen that most of the classes were acquaintances of the *Cache Peer Manager* distributable feature, which we have also seen that was the largest one in the system. Consequently, we were able to draw the conclusion that the respective feature is the most important one in the system.

The visualization provided us with means for easy identification of the inherent distributed nature of EHCACHE, which was also shown by the metric-based assessment above and confirmed by the system documentation.

8.3 Patterns of Acquaintances

This step of the evaluation is responsible for assessing the validity of the collaboration patterns and the related detection strategies we have introduced in Section 6.3.

For this purpose, we have applied the strategies to the analyzed case studies separately, and analyzed the classes that were filtered by them.

As discussed in Section 6.2, the detection strategies were described in terms of threshold identifiers, rather than actual numbers. The process of associating them with raw values is

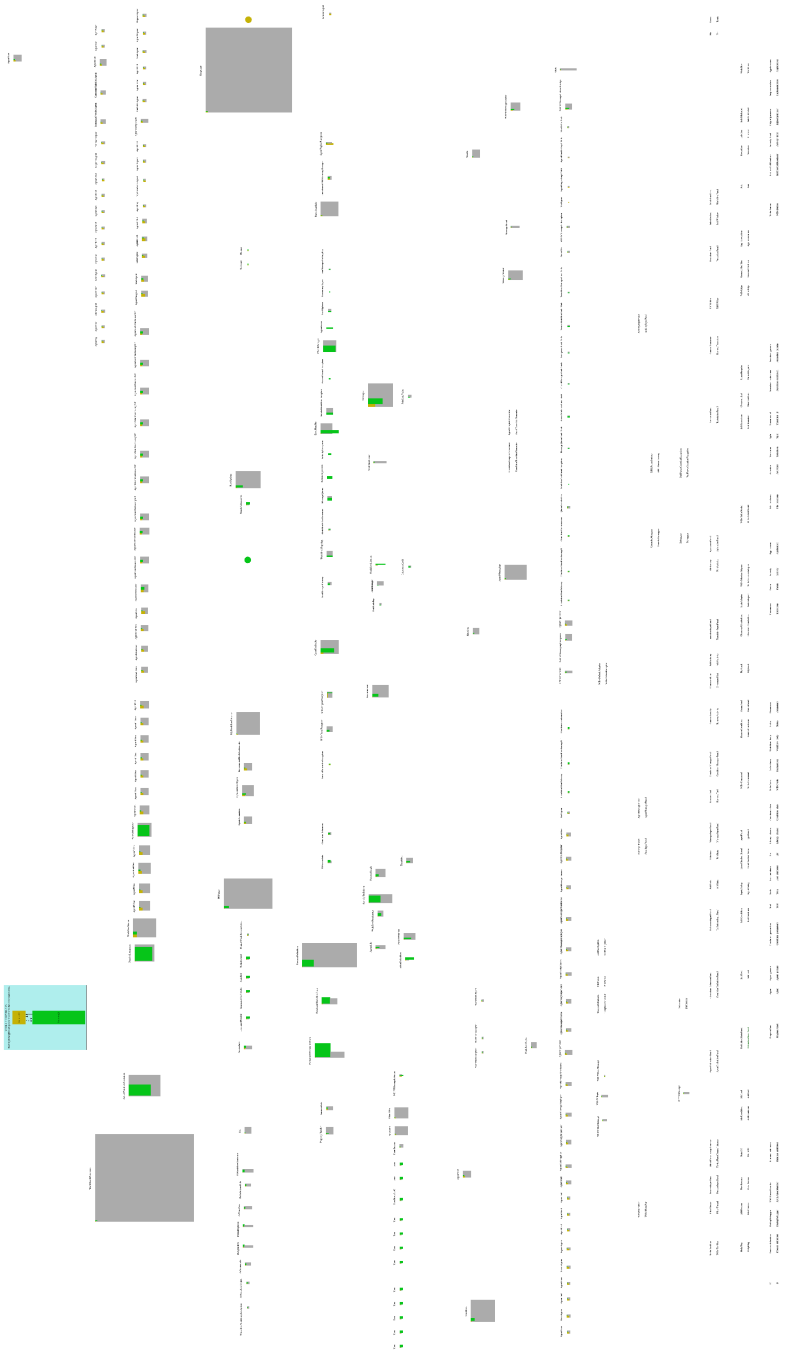

Figure 8.2: Overview of FWS

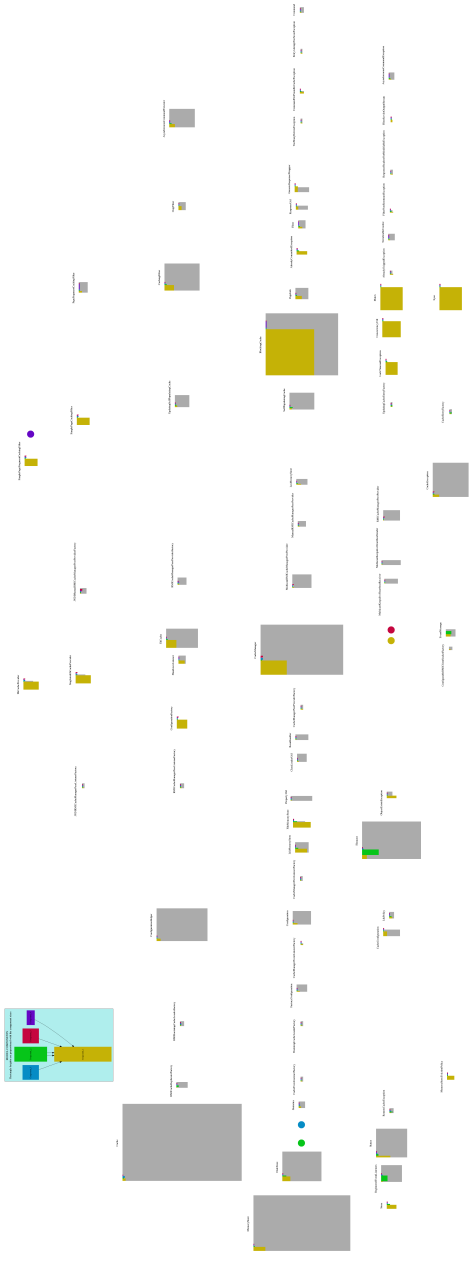

Figure 8.3: Overview of EHCACHE

Metric	*Threshold label*	*Threshold value*
TADF	HIGH	15
	AVERAGE	8
TBC	HIGH	20
$\frac{TADF}{TBC}$	AVERAGE	$1/2$
	LOW	$1/8$

Table 8.2: Experimental threshold values

dependent on the class of the analyzed systems, and done through experiments [61, 59]. For the case of the applications we analyzed, we have chosen, based on our repeated observations, the values in Table 8.2.

At this point, we need to note that we are confident that the threshold values we have chosen can be used for the analysis of similar systems, that is Java RMI projects with 100-400 classes. They can be tuned further if the need arises, especially if analyzing systems significantly larger than the ones we analyzed, yet we believe that the thresholds for such systems will vary only slightly. The reason is that our class-level metrics measure *coupling* between classes, and class-to-class coupling is not a relation expected to increase proportional with the number of classes.

We present next the patterns of collaborations we identified in the two case studies we focus on, and show the way visualization helped in isolating the most interesting entities.

8.3.1 Significant Feature Acquaintance

This pattern is specific to the classes that have a significant participation to the distributed functionality in the system. They are highly relevant for understanding the distributable features, and often must be taken into consideration at the same level with the classes that were already included by the approach in the distributable feature cores.

8.3.1.1 Analysis of FWS

By applying this detection strategy in the case of FWS we have found 5 such classes, all related to the distributable feature we have identified as the largest, the one we named `Workflow Engine`. They were called - `ActiveInterpreter`, `PassiveInterpreter`, `PersistentWfContainerData`, `ActiveWorkflowContainer`, `ProcessDefinition`. At a closer analysis, we have found that the `ActiveInterpreter`, and the `PassiveInterpreter` were related, and their names also linked them to several classes in the core, which confirmed they were indeed close to the respective feature.

When asking the developers of FWS, they confirmed that the 5 classes, along with those in the identified distributable feature core are enough to understand the main properties of the `Workflow Engine` feature. The fact that we found relatively few such classes related to the distributable features showed that the functionality of these features is very well located in the system.

8.3.1.2 Analysis of EHCACHE

The EHCACHE system is significantly smaller than FWS. Nevertheless, when searching for significant feature acquaintances, we found more than 12 such classes. As with FWS, they were all related to the core corresponding to the largest distributable feature in the system (*Cache Peer Manager*).

Nevertheless, these feature acquaintance classes were different from those found in FWS, as the values were smaller while the *TADF / TBC* was very close to 1. This showed that these respective classes are particularly dedicated to the distributable feature, and therefore serve a specific role. From this point of view, the names of the classes were also relevant: `Mutex`, `Sync`, `ConcurrencyUtil`.

8.3.1.3 Local Feature Contributor

The classes in this category are classes that are mostly concerned with the local functionality of the system. While the focus of our approach is to assess mainly the distribution-related traits, these classes are very important in understanding the actual characteristics of the non-distributed functions of the system, thus putting the distribution-related parts in perspective. They can show whether the non-distributed part of the system is only ancillary, or it actually represents the main functionality of the application.

8.3.1.4 Analysis of FWS

As noted above, we have found in the FWS system a set of almost 80 classes implementing a tool for visually editing workflow specifications. This was the most striking case of local feature contributors, that actually had nothing to do with the distributed functionality.

This case has also shown a very interesting side effect of our approach, highly appropriate when trying to understand a system. The tool for editing workflow specification was not actually a part of the framework that users extended in order to build workflow systems. It was rather an additional tool, that helped them in the process of specifying the activities in the workflow, a task that could also be done without using the tool at all, by directly editing an XML file.

Our approach managed to immediately isolate and categorize the classes in the tool as an extreme case of local contributors (thus helping us see their real significance), without being provided any information that the respective set of classes was not a part of the main system. Without our approach, the large number of the respective classes could have posed real difficulties in understanding the system, distracting the engineer from analyzing the really important classes.

In addition to the classes in the tool the detection strategy identified a set of other 6 classes, all of them belonging to other features. For example we found the class `WorkflowIsPersistent` which deals with making a workflow persistent on the file system, and the class `MyWorkflow-Listener` which belongs to one of the test applications, thus it is not part of the FWS framework.

8.3.1.5 Analysis of EHCACHE

In the case of EHCACHE , we identified at most 5 classes that were *local feature contributors*.

The most prominent case, having the highest value for the *TBC* metric was `Cache`. This is one of the central classes in the system, modeling the very concept of a cache, therefore it is only natural that it is heavily used through the entire system. Nevertheless, this class is not directly connected with any of the distributable features, because these distribution-related activities in this system features mainly deal with making the caching distributed between the cache peers, they do not need to be concerned in the actual details of implementing the cache and storing the data locally.

Another example is the `ConfigurationHelper`, a class that contributes to a local feature responsible for managing the configuration files, which has nothing to do with the distribution-related aspect.

8.3.2 Connector Class

This strategy identifies the classes having a significant amount of collaboration with the distributed part of the system, but most of their collaboration is with the other classes in the system. They represent the classes that connect the distributed and local functionalities within the application.

8.3.2.1 Analysis of FWS

Applying this strategy we have identified 5 significant *connector classes* in FWS.

The feedback from the developers of FWS confirmed that all of these classes do indeed fit this category. The most interesting case was the `ProcessDefinition` class. The value of the *TADF* metric in its case was 15, and the *TADF*/*TBC* ratio was 0.2.

The next step was to analyze its relation with the classes directly connected with it in the dependency graph. We have looked at the names of these classes, and found out that the `ProcessDefinition` class links two significant features in the system. The class models in fact the internal representation of the workflow the system uses when executing the activities, in short, the specification of states and transitions. The classes responsible for running the workflow, (`ActiveInterpreter`, `PassiveInterpreter`), that we already identified as being significant feature acquaintances, intensely use (and depend on) this class. On the other side, it is used by a class (`PDParser`) that showed little involvement in the distributable features, and represents a local functionality that parses the XML files that actually store the XML workflow description, and use them to create the internal representation.

8.3.2.2 Analysis of EHCACHE

In EHCACHE we have detected 6 *connector classes*.

The most interesting case is that of the class `Element`. This class represents the data entities that form the information cached by the system. Looking at the ADF values for this class in respect to each of the identified distributable features, and also considering the other similar dependencies in the system, we found that this is the only class having a noticeable relation with the *Cache Replicator* distributable feature. This is natural, as it is in fact the single most important item the Replicator manipulates.

Indeed, the purpose of replication is to exchange updated cached items, in other words instances of the `Element` class. It came natural that the class consequently connects the *Cache Replicator* feature with the basic, non-distributed, caching functionality of the system.

8.3.3 Visualization

The visualizations characteristic for the acquaintance patterns were very helpful as they allowed us to easily and rapidly identify the classes that presented interesting traits. In fact, the assessments presented above when discussing the application of the strategies were effectively driven by visualization. The visual elements focused our attention because the interesting classes were distinguishable among the others in several ways: featuring high color content, being shown as mostly gray shapes, or having balanced grey/colored areas.

8.3.3.1 Pattern 1: *Big Color Spot*

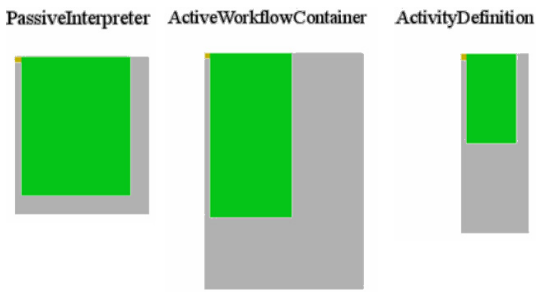

Figure 8.4: Examples of *Big Color Spot* pattern

In FWS (Figure 8.4) we identified the 5 classes with large color spots. They were very visible and the color indicated their exclusive relation with the largest distributable feature (*Workflow Engine*).

For all of the EHCACHE, classes there is again a single dominant color *i.e.*, the one corresponding to the *Cache Peer Manager* distributable feature. They differed form the classes shown in the case of FWS, by being squarish, not very large, and almost without gray areas. This was a visual representation of the fact that the classes were strongly related to the respective distributable feature, and served a specific role – the same fact the detection strategies also helped us understand.

8.3.3.2 Pattern 2: *Big Gray*

In FWS, the visualization helped us see literally in an instant the strange-looking classes that had no participation in the distribution-specific activities, and actually implemented the above mentioned visual tool. The other 6 classes that fit this pattern were also directly visible and their large gray areas showed that all of them belonged to other features.

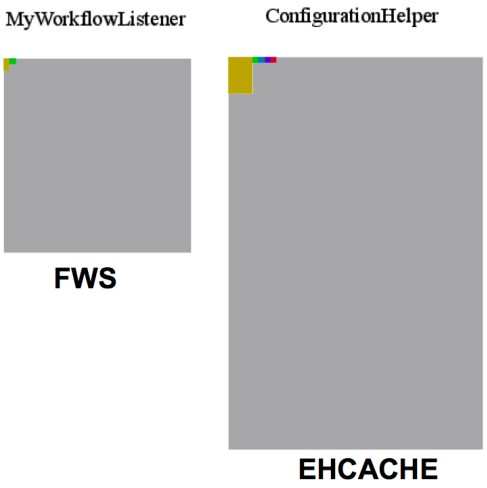

Figure 8.5: Two *Big Gray* classes

In EHCACHE, at most 5 classes qualified as *Big Gray*, as noted in the previous section. The most prominent one was the one with the largest gray area, the `Cache` class we have identified as being one of the central classes in the system, but it is not directly related with the distributed aspect. The `ConfigurationHelper` class which locally manages the XML configuration files for EHCACHE was also easily identified.

Figure 8.5 shows an example of such classes.

8.3.3.3 Pattern 3: *Color Spotted Gray*

This pattern refers to the case where a feature acquaintance class is dominantly gray, but there is also a significant color spot. The interpretation is that the class encapsulates a piece of functionality that connects an already identified distributable feature with another, *non-distributed*, one. These classes are particularly interesting, as their analysis helps in understanding the linkage between the two types of functionalities.

In FWS there were 5 significant *connector classes*, matching the detection strategy discussed above. In EHCACHE we visually identified 6 such classes, the most prominent case being that of the `Element` class (Figure 8.6) the class that connects the *Cache Replicator* with the basic, non-distributable, caching feature of the system.

8.4 Restructuring Support

This section presents a case study that shows the results our extraction-based approach can provide when applied to a distributed software application. We have chosen the FWS system as the most interesting case, because the interactions we have discovered within the code are useful in showing how our approach is applied. Moreover, as the extraction of inter-related

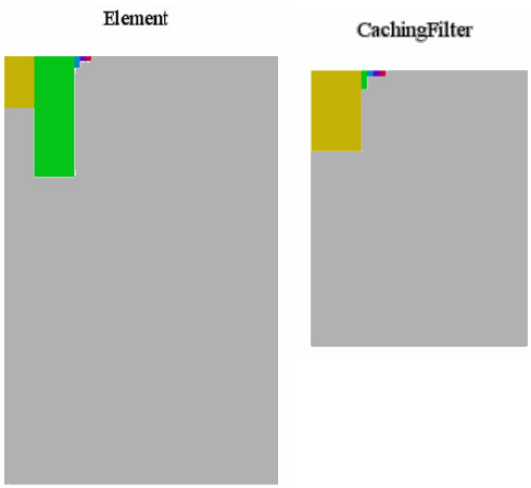

Figure 8.6: Two *Color Spotted Gray* classes

classes needed confirmation from as authoritative sources as possible, the fact that we had access to the developers, was in this case vital to verify the validity of our findings.

As previously discussed, the approach identified two distributable features: the *Workflow Engine*, responsible for creating and executing workflow instances, and the *Agent Engine*, responsible for managing and running the agents.

To find out possible scenarios of restructuring, we started by looking at each distributable feature and found that they both included more than one remote interface, meaning that they published several remotely-available services. Consequently, a first extraction scenario was created focusing on finding the distinct *services* published by the system and the classes most involved in providing each of them. It consisted of 6 extraction units, each of them containing a single remote interface and the hierarchy of classes that implemented it. By applying an extraction threshold factor of 1.3, we have obtained the 6 groups of classes closely related with each unit. For identification, they were named after the remote interface of the respective extraction unit: *AgentEngine, AgentHandler, Workflow, WorkflowManager, MessageQueue*, and *ClientManager*. The most prominent (in terms of number of classes – over 40 –, and also considering its name) was Workflow (Figure 8.7).

By looking at the visualization for this particular group, we have identified five large entities, the classes called *Executer, Interpreter, ProcessDefinition, Workflow* and *ActivityDefinition*. To understand their dependencies, we first used each of them as individual extraction units, performed the extractions, and evaluated the costs. The extraction costs obtained for each operation are shown in Table 8.3.

At this point we must note that the extraction costs throughout the system varied in our study from values as low as 0.1 (when extracting an individual class connected with only one other class) to the highest numbers just below 3.0 (when extracting prominent, highly connected classes).

The similar values for *ProcessDefinition* and *ActivityDefinition* along with their sharing

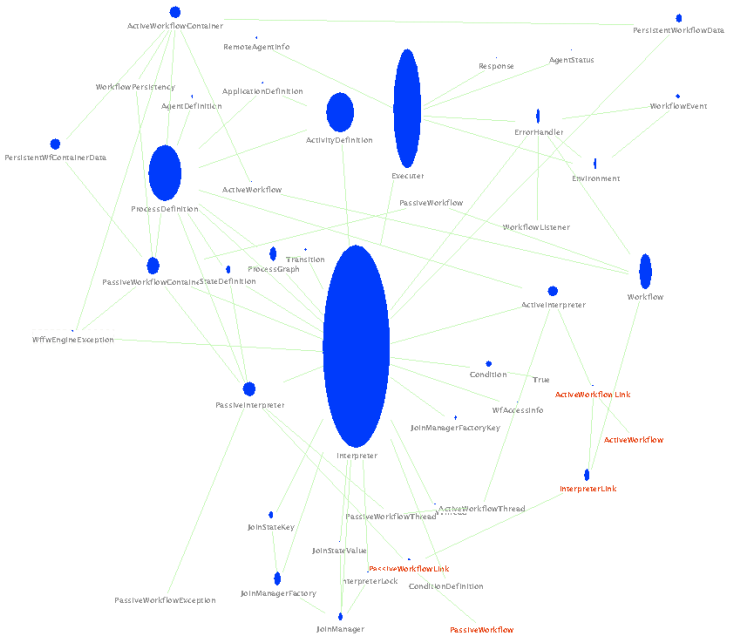

Figure 8.7: The shapes of the classes in the Workflow service group

	Executer	*Interpreter*	*ProcessDefinition*	*Workflow*	*ActivityDefinition*
Extraction cost	1.98	1.78	2.60	2.3	2.56

Table 8.3: Extraction costs

of an edge in the dependency graph, suggested that they are related to each other. Indeed, they both proved to be parts of the internal representation of workflows the system creates at runtime, thus were both designed for the same purpose. When they were included in the same extraction unit and we re-performed the extraction, the cost remained very much the same (2.58) which means they are still relatively hard to separate from the system. We managed to reduce the cost to 1.71 by including the *Interpreter* in their extraction unit, because we noticed that it was included in the set of classes around the two after the extraction. The conclusion is that the three classes (and the ones that clustered around them at the last extraction) may represent a distinct feature, but the cost of extracting it from the group is not very low.

Making similar explorations regarding the other entities we found that the *ProcessDefinition* and *ActivityDefinition* couple is related in a similar degree with the *Executer* entity – they can be extracted together, but the cost is high. This suggested that they represent an important common feature used by both the *Executer* and the *Interpeter*. Design-related data from the company that developed the system confirmed that the two classes (*ProcessDefinition*, *ActivityDefinition*) implement a distinct functionality that stores the internal definition

of a workflow. The *Interpreter* controls the transitions between states, and the *Executer* starts the activities corresponding to each state. As the *Executer* and *Interpreter* entities had different functions, the developers agreed that they could be theoretically extracted as independent features, and confirmed that the high dependency on the *ProcessDefinition* and *ActivityDefinition* (and related) classes would make the extraction expensive, but possible.

Applying successive extractions proved to be a versatile tool for understanding the details of the system, and to propose restructuring scenarios. The identification of relevant classes to be included in the extraction units was quick, and the costs consistently characterized the redesign effort. The engineer's experience proved to be a significant factor, and it made a difference when selecting scenarios that lead to a better system understanding. Nevertheless, the visualization greatly assisted this effort, both by drawing the *IGA* characteristics, and by presenting the graphs of the different class groups before and after the extraction.

9

Tool Support

Any process of software analysis must be supported by a tools infrastructure that enables engineers to efficiently measure and interpret the system's characteristics. Tools must automate all the tasks involved by the analysis and provide the user with instruments that allows for the tuning and controlling the entire process.

This chapter presents the tool we have developed to support the methodology we have described in the previous chapters. The tool follows all the steps in the methodology, and implements all the implied mechanisms that support the understanding and restructuring of object-oriented distributed software systems.

The tool is called *niSiDe* [19], a loose interpretation of its name being **n***on* **i***nvasive* **S***tructural* **i***nsight on* **D***istributed* **e***nvironments*. It consists of an extensible platform that integrates all the algorithms and automated tasks involved at the various steps of the analysis approach. It provides a user interface for interacting with the engineer when necessary, and implements all the visualizations that are part of the process. As our approach aimed to extend the mainstream object-oriented analysis to distributed applications, *niSiDe* was integrated in the iPlasma software analysis environment [72], developed by our group.

9.1 Tool Architecture

The architecture of the tool environment is presented in Figure 9.1. The system consists of the following functional modules:

- The *General Processing Unit* is responsible for the core processes related to the analysis approach. It reads a *Memoria* [72] model of the target application provided by the iPlasma environment which is created by parsing the source code of the system. The module implements all the generic (*i.e.*, technology-independent) parts of the algorithms involved in the discovery of the core distributed functionality and the identification of the system distributable features. It is also the place where all the metric-related computations and algorithms are implemented, and the detection strategies are applied.
- The *Technology-Specific Processing Unit* is the part that is built specifically for a particular type of technology of the Communication Mediator. It contains the implementation

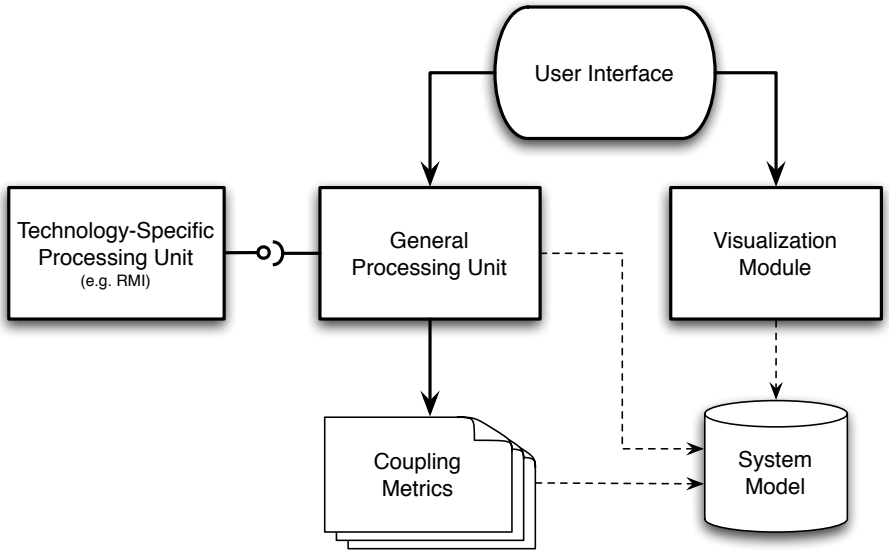

Figure 9.1: The architecture of niSiDe

of the algorithms that detect the frontier classes, the heuristics related to the separation of the distinct cores of distributable features and all the other technology-specific processes. The architecture of *niSiDe* facilitates the integration of as many technology-specific units as necessary, in order to provide easy extensibility. In the context of this dissertation, we have fully implemented the module that deals with applications that use Java RMI as the communication infrastructure.

- The *User Interface* module implements and controls the graphical user interface that interacts with the engineer. It presents the option for tuning the various algorithms and for reviewing and modifying their outcome, and shows the results of the different phases of the approach.

- The *Visualization Module* is responsible for generating and providing all the visualizations that are part of the methodology, both as direct representations presented to the user via the user interface, or as diagrams exported in a standard format that can be later processed by visualization-specific third party tools.

To use the tool, the engineers have to open the *Insider* program part of the iPlasma environment, load the project they are interested in analyzing, and start *niSiDe* at the system level. Once the parameters for the tool were specified, it starts analyzing the system by considering it an RMI distributed application.

The tool follows the steps of the methodology in order, and stops at each point that necessitates user interaction. The results are presented to the user, and after the engineer reviews and possibly modifies the involved parameters, the automatic process continues.

9.2 System Representation

To support the analyses, *niSiDe* creates and uses the internal representation of the system, consisting on the dependency graph of the classes we have discussed in Section 5.2. Vertices represent classes, and edges model class-to-class dependencies.

The graph-related functionality is built using JGraphT [49], a comprehensive graph manipulation Java library. The representation of the system as a graph is built in parallel with the *Memoria* model loaded in the memory, with graph vertices referring extensively annotated classes in the iPlasma-specific model. To adapt the *Memoria* model to our approach, we have extended it as follows:

- we have defined a new system-level entity that models the concept of distributable feature;
- we have annotated the classes with information regarding the distribution-related and analysis-specific aspects. This includes:
  - marking the frontier entities,
  - specifying the root (start) entities for the various algorithms,
  - marking the service representatives for the extraction-driven restructuring,
  - marking the special relationships in the graph, such as particular inheritance relations,
  - storing for each class the measurements related to the various types of acquaintance, including their acquaintance with the discovered distributable feature cores.

9.3 Core Distributable Features Discovery

The first phase of the analysis is concerned with identifying the distributable feature cores, i.e. the main units representing the distribution-aware functionality.

To achieve this goal, the tool creates the initial internal representation as a partial dependency graph. It starts by using the technology-specific rules to detect the classes acting at the frontier with the Communication Mediator, then applies the steps in the methodology that create the entire core dependency graph, consisting of the classes that are most related to the distribution-aware functionality. The next step uses both generic algorithms and technology-specific heuristics to separate the distinct cores of distributable features, which will be used by the rest of the approach.

The detected clusters of classes are presented to the user by a graphical interface. The engineer can, at this point, modify the structure of the distributable feature cores by moving classes from one core to another, in order to improve the outcome of the automated processing (Figure 9.2). As noted when we presented the details of the methodology, the algorithm proved to separate fairly well the distinct functionalities; nevertheless, the user interaction is essential at this step, at least as a mean to validate the results.

After the user confirms the results, the system is further analyzed by enacting the algorithms responsible with the detection of the remote communication channels between the distributable feature cores, so that the first system-level characterizations can be made.

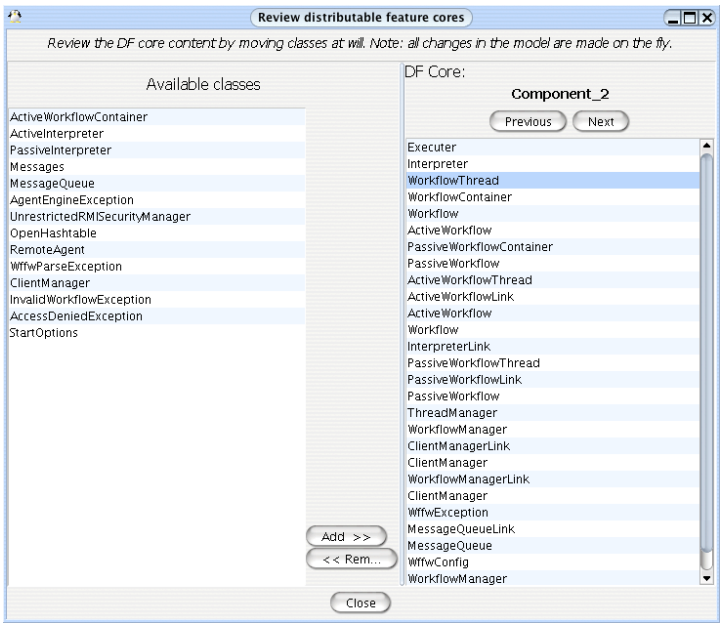

Figure 9.2: Reviewing core content

9.4 System-Level Visualization

The results of the first steps of the analysis are visualized as described in the previous chapters, by drawing two types of diagrams:

- a diagram of the dependencies established between the distributable features, and their relative size in the system (see Figure 9.3);
- a set of pictures showing each feature core's structure, depicting the contained classes, and their inter-dependencies (Figure 9.4).

The visualizations are generated by *niSiDe* in two ways:

- they are exported as *.dot* files, a widely-used graph description text format usable by the popular GraphViz visualization project [34]
- the parts that involve runtime user interaction or reviewing are drawn using the JGraph graph visualization library, along with its JGraphLayout counterpart [48].

9.5 The Distributable Features View

After identifying the distributable feature cores, *niSiDe* follows the next steps of the methodology, and assesses the relation between each distributable feature and the rest of the classes

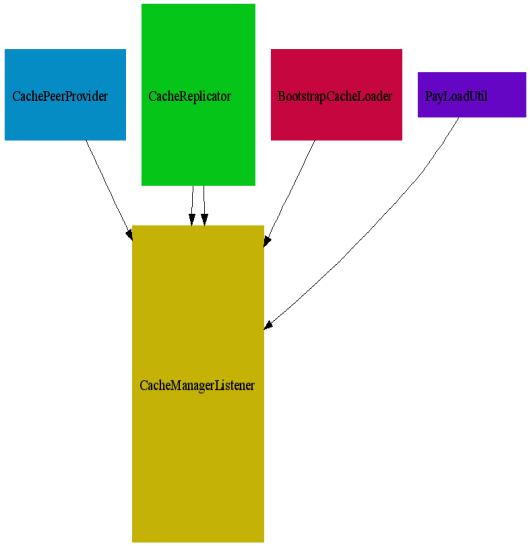

Figure 9.3: System overview

in the system. For this purpose, the tool builds the full dependency graph of the system, containing all the classes as nodes and their dependencies as edges. It then employs the algorithm described in Section 6.1 and calculates the *feature acquaintance* for each class in respect to each identified distributable feature in the system.

The class-distributable feature acquaintance values are attached to the internal representation (through a specific entity annotation mechanism), and are used to build the Feature Affiliation Perspective visualization defined in Section 6.4 as a diagram showing the degree in which each class is involved with each distributable feature (Figure 9.5). The DISTRIBUTABLE FEATURES VIEW is useful when trying to find the most interesting occurring patterns of involvement, and stays at the base of the conclusions we have drawn for the test cases we conducted.

9.6 Support for Restructuring

As presented in Chapter 7 the methodology we have developed addresses the aspect of system maintenance and evolution by providing support for restructuring the system when such a need arises. The approach uses an extraction-driven technique that focuses on identifying and manipulating the code units so that the engineer is able to assess which parts could be extracted as individually-separable functionalities. To implement the technique at the tool level, *niSiDe* makes extensive use of the JGraph library, which is used for drawing the interactive diagrams that show the various parts of the system, each represented as a dependency subgraph of a certain group of classes.

The tool gives the user the opportunity of analyzing the respective group and selecting

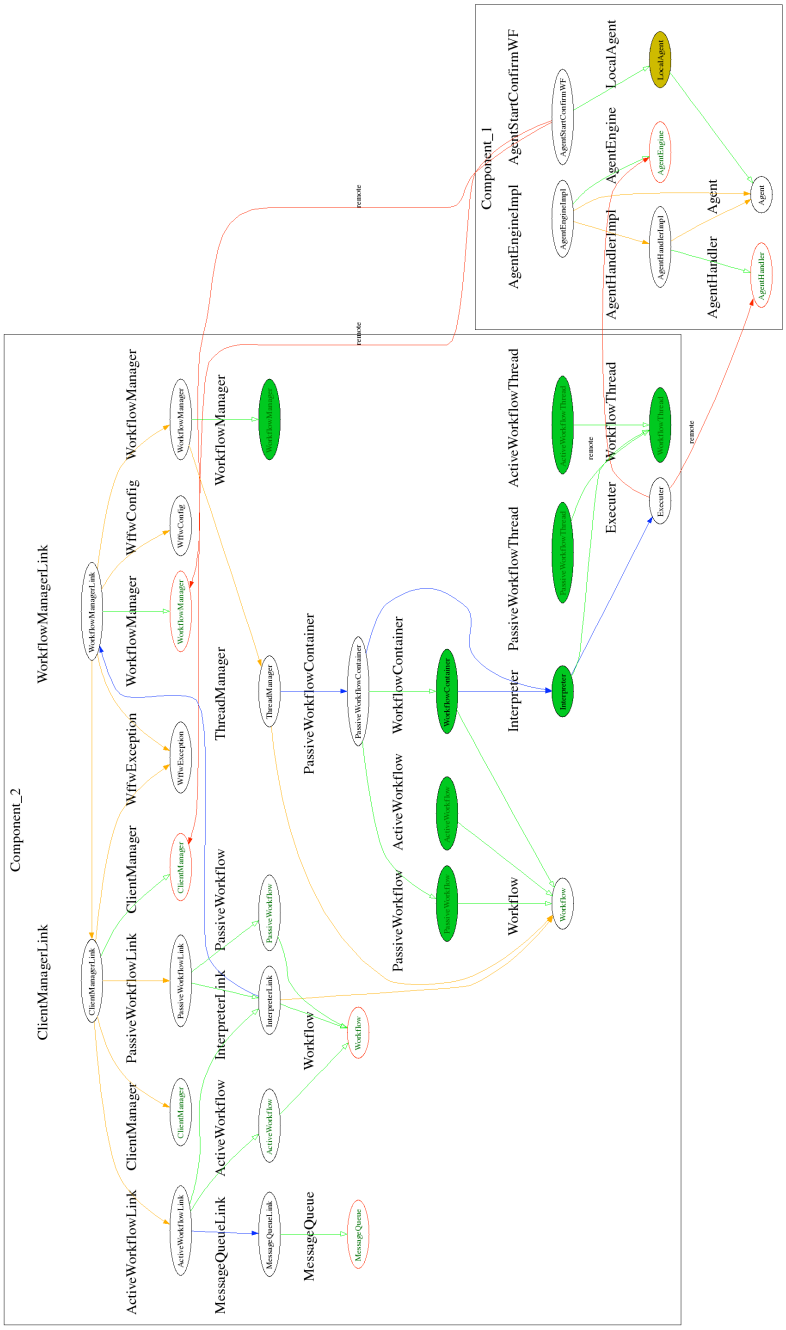

Figure 9.4: The structure of feature cores

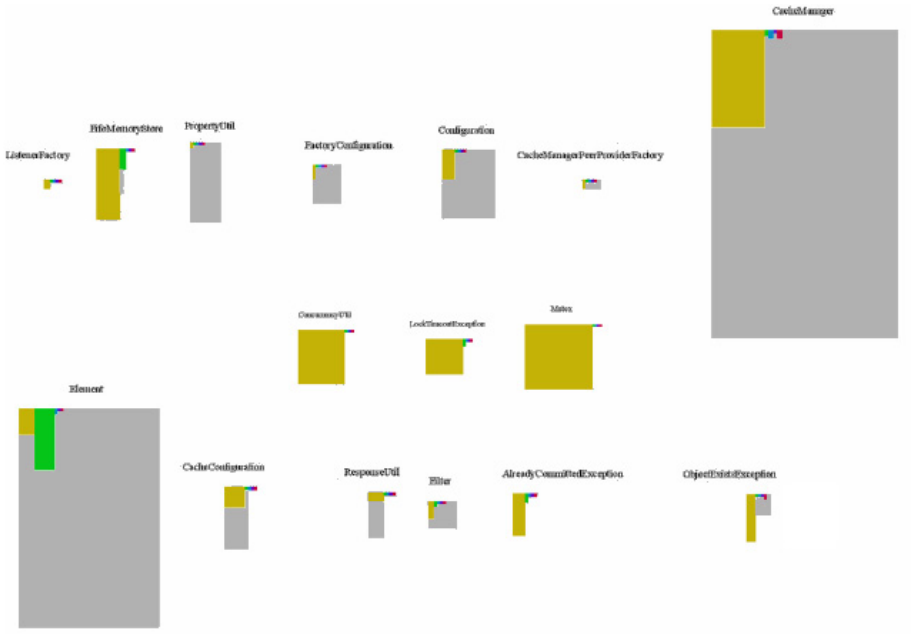

Figure 9.5: A DISTRIBUTABLE FEATURES VIEW

the classes of interest to create an *extraction unit*. The tool then employs the algorithms that identifies which classes will be automatically extracted with the extraction unit and calculates the costs. The result is presented in a new window containing two class dependency graph diagrams, one for the classes that were extracted along with the extraction unit, and one for the rest of the classes. The *extraction cost* is shown, to inform the user about the estimated effort that will be necessary when actually restructuring the respective group by actually performing the separation. The resulting groups are shown in windows with the same characteristics and interaction features as the original group of classes, so that the user can perform subsequent extractions, as long as they are necessary.

As presented in Chapter 7, the user is assisted in the process of selecting the classes that are to be included in an extraction unit, by visualization. *niSiDe* applies for each class the coupling-based metric that calculates its degree of collaboration with the other classes in the group, and presents the results visually drawing the vertices with different shapes and sizes. Any set of classes can be selected by the user as parts of the extraction unit, and the user can consequently analyze various scenarios of extraction.

When the restructuring user interface starts, *niSiDe* provides the engineer with a default set of initial class groups:

- the entire system, as a comprehensive dependency graph;
- the set of all distributable feature cores detected in the system, each presented in a separate, manipulable diagram;

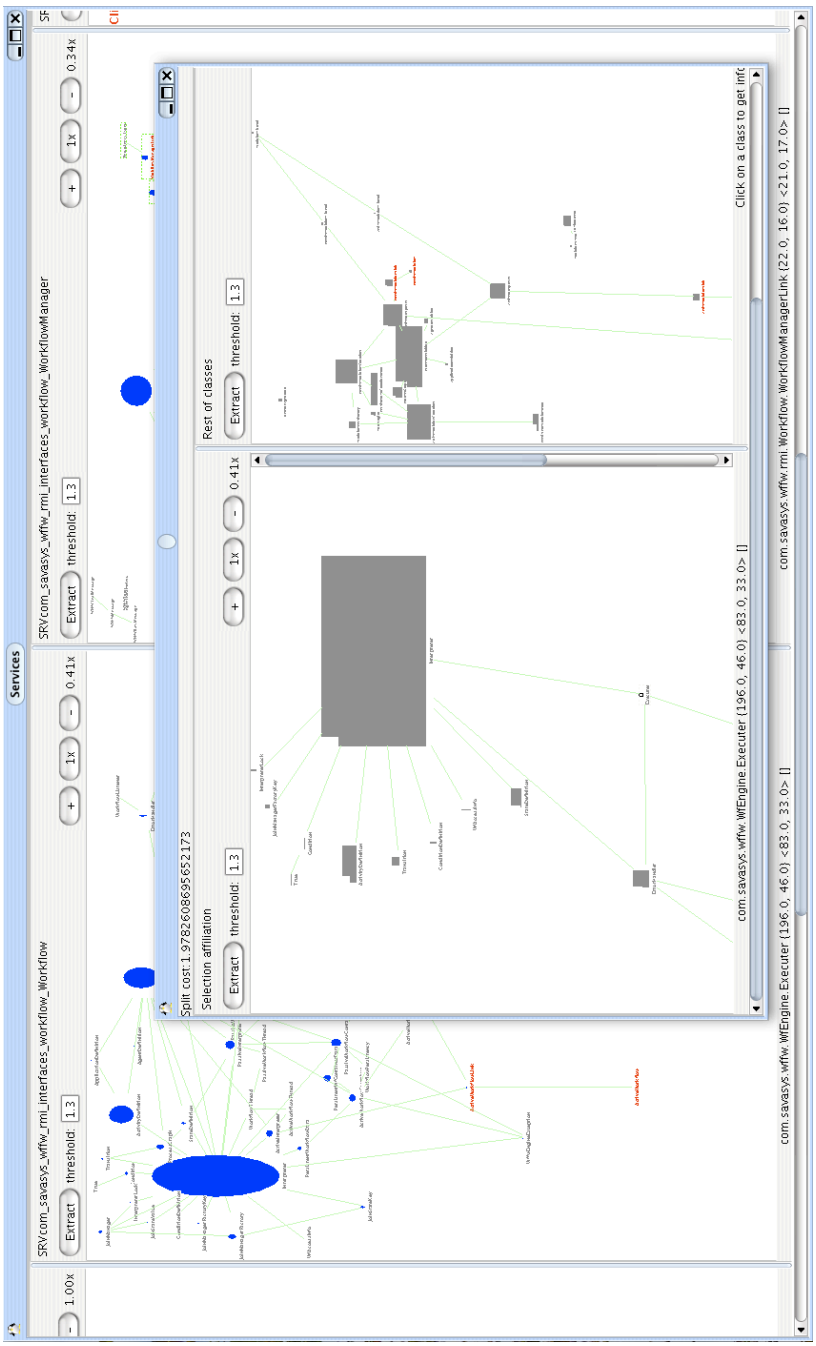

Figure 9.6: Extraction example

- all services detected in the system.

All the above groups are presented by the tool as starting configurations for performing extraction scenarios, and the engineer can analyze their structure and perform as many subsequent extractions as needed.

Figure 9.6 shows an extraction result (in the foreground window), and a part of the initial group, visible in the left side of the background window.

The functionality of providing restructuring support specific to *niSiDe* proved to be a versatile and useful tool both for evaluating different extraction scenarios, and for understanding the isolated functionalities within the code at a finer grain than with the previous steps of the approach.

9.7 Extensibility

As pointed out throughout the description of the methodology and the tool, the current version of *niSiDe* is built for analyzing Java RMI systems. Nevertheless, it was designed to be easily extended for the purpose of analyzing applications based on other communication technologies.

All aspects that are dependent on a particular communication mediator are designed so that other types of technology-dependent aspects can be plugged in when necessary. Moreover, the vast majority of tasks representing the main methodology-related system functionality (graph processing, the abstract algorithms, the visualizations) are designed as independent from the technologically-specific details.

10

Conclusions

This dissertation presented our approach in analyzing distributed object-oriented systems, using a process of reverse engineering that inspects an existing application. The analysis starts by inspecting the source code, and builds an internal representation of the software as a dependency graph of classes. The representation is used by the subsequent steps to extract the most important application characteristics that provide essential system understanding.

10.1 The Methodology

The methodology builds, step by step, a representation of the distributed system that presents it from the perspective that describes its distributed nature. The main concept in this model is the *distributable feature*, and the main relations describe the participation of the system entities in providing the features, in the form of *feature acquaintances*.

The distributable features are discovered through a process that focuses the analysis to a small part of the code, that contains the core distribution-related functionality. The core is detected by using the important clues provided by the technology of the Communication Mediator (the communication infrastructure), and starts with assessing the interactions specific to the *System-Mediator frontier*.

Applying a set of rules, partially extracted as heuristics dependent on the particular technology, the core of distributed functionality is separated in a set of *cores of distributable features* that are used as representatives for the main distribution-related distinct functionalities provided by the software system. This provides the approach with the first important result, in the form of a characterization of the system's distributed architecture that describes the distribution-related functionalities and the relations established remotely between them.

The distribution-aware features are identified without relying on the existence of deployment information regarding the application. This way, a wider selection of distributed applications can be analyzed in order to extract the significant items of understanding the methodology provides. Moreover, the entire process is applied without making assumptions on the quality of the system's design or on the structural or architectural patterns that were used, to make the approach applicable to both well-designed and poorly-designed software applications.

The next step switches the focus on the rest of classes in the system (that form the majority of the system entities), and evaluates their relation to the already identified features. A set of coupling-based software measurements are employed, and the numerical values that result are interpreted in the context of the classes' participation in providing the features.

The approach centers its attention on describing and identifying the main relevant patterns of collaboration that can occur in an application, and which describe the actual impact of the several distribution-aware features in the design of the system. The impact is evaluated both at the system level, thus characterizing the *system distribution awareness*, and at the finer grain of the class level, by assessing each entity's *acquaintance with distributable features*.

The final step of the approach provides support for system restructuring by facilitating the experimentation with different restructuring scenarios, through a process driven by the extraction of inter-related clusters of classes. The engineer can conduct extraction scenarios in any part of the system, and the methodology provides means for evaluating the projected outcome of the structural modification, and the costs the change implies. In order to maintain consistency with the rest of the approach, the restructuring support uses the same concepts and similar techniques with the parts that focused on system understanding. Moreover, experimentation with extraction scenarios can provide the engineer with an additional way of adding to the knowledge about the system, through assessing the classes' inter-dependencies from a different perspective.

10.2 Conference Publication

The main aspects covered by this thesis were recently published as conference papers, both for communicating the results of our research, and to receive valuable feedback from the software engineering community.

The methodology for understanding object-oriented distributed systems was described in the 2008 paper [21] published at one of the major conferences in our field, *12th European Conference on Software Maintenance and Reengineering (CSMR 2008), Athens, Greece*.

The visualization techniques we introduced, as well as the visual patterns and their interpretation were presented in a paper [20] at one of the main conferences focusing on software visualization: the *4th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007), in Alberta, Canada*.

The tool infrastructure and its applicability was described in the paper [19] we published at *10th International Symposium on Symbolic and Numeric Algorithms for ScientiÞc Computing (SYNASC 2008) Timisoara, Romania*.

The first two papers are **ISI-quoted**, and also indexed by **other major databases** (Inspec, IEEExplore, etc.), while the latter was published in the SYNASC post-proceedings volume (**IEEE Proceedings**), and, according to the conference organizers, may also be indexed by ISI.

We have also published several other conference papers and two books that are related with our field of research, as referenced throughout this thesis.

10.3 Contributions

The major contributions of this thesis to the field of reverse engineering software systems are presented as follows.

A methodology for understanding object-oriented distributed systems. The overall contribution of the thesis is a methodology that is **novel and comprehensive**, built to provide *consistent and sustained* support for each and all the needed steps in an analysis that understands a distributed system through reverse engineering, up to, and including the point in which it provides means for system restructuring. While it sometimes makes use of some existing techniques, such as the algorithm that eliminates 'weak edges' as a secondary step in separating the features, the major processes we have presented are entirely developed by us, and are new.

As a confirmation of this claim, when we have presented the core of our approach [21] at one of the major software reengineering conferences in our field[1], we have received very positive reviews, and the fact that we introduced our work as novel was unchallenged by the reviewers. Moreover, one of the three reviewers characterized the technique as *"quite original and useful to the software practitioner"*, while another reviewer summarized the paper as *"a different than usual approach in understanding distributed object-oriented systems by isolating the 'purely distributed features' and examining their impacts on the overall system."*. The latter also characterized the approach as "technically solid" and remarked that the content we presented was actually "too long for a conference paper".

A model for object-oriented distributed systems. The representation of an object-oriented distributed system is **specifically built for the goal of understanding and for the context of this methodology**. It captures all the system characteristics that provide valuable knowledge about the target class of applications, while using a small yet representative set of specific concepts.

The concept of Distributable Feature. This is one of the main concepts in the thesis, and it is designed as a means of characterizing the system's distributed functionality, *without depending on isolating actual distributed components or deployment units*. The overlapping nature of the various distributable features in a system allows for a more flexible approach, which is able to better fit the real-life applications, where the concerns related to the various functionalities are not always clearly delimited.

The concept of Class Acquaintance and the Patterns of Acquaintance. This concept characterizes the classes' collaboration with selected features or even with groups of entities. By analyzing each class' relation with the distributable features, and identifying the pattern it conforms to, the assessments can provide valuable insights on the importance of the distributed aspect in the application.

[1] 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), Athens, Greece

The concept of System-Mediator Frontier. In the context of assessing the core interactions, this concept is introduced and used as a highly relevant starting point for capturing the distributed aspects of the application. This way, it provides essential information about the system, and ensures an accurate focus for the reverse engineering process.

The Distributable Features View. The novel visualization technique we developed is designed as a key tool for both capturing the interesting patterns of collaboration, and for detecting the particular classes that follow these patterns. The two visual perspectives it consists of, the *Distributed Architecture Perspective*, and the *Feature Affiliation Perspective* facilitate both system-wide characterizations, and class-level collaboration-related assessments. The latter introduces a *Composed Polymetric View* as an extension of 'classic' polymetric views, to better capture the visual clues describing the system.

A set of reverse engineering techniques. The techniques we have developed extract the knowledge providing system understanding. We use *a technology-driven focal point* in the analysis that values the clues given by the technology in order to extract complex information, such as the system features. We achieve *the isolation of a core, representative set of classes to focus the techniques* that isolate features, *rather than analyzing the entire application*, so that efficiency and accurate focus on the relevant aspects is obtained. Further, we use of a *combination of technology-related and generally-applicable heuristics* for *separating the functional partitions* within the core.

An *architectural overview* on the application is *extracted without depending on the (usually missing) deployment information, while avoiding to complicate the approach with an aim on detecting the exact shape of the distributed components*. The concept of *distributable features* was sufficient to provide significant knowledge.

Focusing on *the concept of system Distributed Awareness*, extracted from the measured aspects in the application, we accept the fact that system entities cannot always be placed in clearly delimited functional partitions, and develop a *collaboration-centric approach* that *allows for the observation of the overlapping concerns*, such as the participation of a class to multiple features.

We assess *two aspects of the collaboration* of the system classes that provide the understanding related to the system:

- the one that *captures the overall importance* of the distributed aspect in the class' design goals, and
- the one that *characterizes the participation of the class in each features* of the system that are distribution-related.

A set of metrics assessing system characteristics. The metrics we have developed (such as the various acquaintance metrics or the In-Group Adequacy), capture the aspects that provide system understanding. They are used both when applying of the detection strategies, and for driving the visualization techniques.

Restructuring support as a natural extension to the understanding techniques. The specifically-targeted **restructuring support is a part of the methodology**, and provides in a manner consistent with the other steps the means for:

- *assessing the change scenarios*, and
- *achieving additional insight* on the system's functionality.

Comprehensive tool support. The tool infrastructure we have developed addresses all the steps in the methodology, including the evaluation of the system restructuring scenarios. It automates all the tasks implied by our reverse engineering process, while allowing for user interaction.

A targeted literature survey. The survey in Chapter 3 captures the state of the art, driven by the goal of characterizing the techniques that are relevant for the field of reverse engineering software applications, while considering the needs of a comprehensive analysis approach for distributed systems.

10.4 Future Work

While our research described in this dissertation proved to provide a solid approach in reverse engineering object oriented distributed systems, there are still as set of concerns a future development of the methodology can address. The main directions that we can follow in our future work are described below.

- Extend the approach to other distributed technologies. While our methodology was designed from the start to be extensible, it would be interesting to apply the gathered experience to other types of distributed technologies than Java RMI.

  At the time of this writing, the author is already supervising a student diploma project that aims to extend our main techniques to distributed applications built using Web Services;

- The process of identifying the core distributed functionality and the separation of features can be enriched by defining additional heuristic rules, and by possibly improving the existing ones. A larger experimental base would also provide data for building new heuristics;

- The measurements that provide information used for assessing the class-feature participations can be applied for automatically adding in the set representing the core of distributable features the classes with strong involvement on the distributed aspect;

- The support for restructuring can be extended by assessing additional aspects in the projected change scenario, such as automatically calculating the scenarios with minimal cost by identifying and evaluating a set of candidate scenarios;

- The visualization techniques can be extended to capture the characteristics of the projected restructuring scenarios to facilitate their comparison.

Bibliography

[1] Bill Andreopoulos, Aijun An, Vassilios Tzerpos, and Xiaogang Wang. Multiple layer clustering of large software systems. In *WCRE 2005*, pages 79–88. IEEE CS Press, 2005.

[2] Bill Andreopoulos, Aijun An, and Xian Wang. MULIC: Multi-layer increasing coherence clustering of categorical data sets. Technical Report CS-2004-07, Department of Computer Science and Engineering, York University, 2004.

[3] Nicolas Anquetil, Cédric Fourrier, and Timothy C. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE '99: Proceedings of the Sixth Working Conference on Reverse Engineering*, page 235, Washington, DC, USA, 1999. IEEE Computer Society.

[4] Markus Bauer and Mircea Trifu. Architecture-aware adaptive clustering of oo systems. *csmr*, 00:3, 2004.

[5] Nishant Bawa and Sudipto Ghosh. Visualizing interactions in distributed java applications. In *IWPC '03*, page 292. IEEE CS Press, 2003.

[6] Dragan Bojic, Thomas Eisenbarth, Rainer Koschke, Daniel Simon, and Dusan Velasevic. Addendum to "locating features in source code'. *IEEE Transactions on Software Engineering*, 30(2):140, 2004.

[7] Sabina Borlea and Dan Cosma. **A Framework for Feature Migration**. In *Proceedings of the 6th International Conference on Technical Informatics (CONTI 2004), vol 3 (PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE Vol.49 (63) No. 3, ISSN 1224-600X)*, 2004.

[8] 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.

[9] Bill Burke and Richard Monson-Haefel. *Enterprise JavaBeans 3.0 (5th Edition)*. O'Reilly Media, Inc., 2006.

[10] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, January 1990.

[11] Yves Chiricota, Fabien Jourdan, and Guy Melancon. Software components capture using graph clustering. In *IWPC 2003*. IEEE CS Press, 2003.

[12] Andreas Christl, Rainer Koschke, and Margaret-Anne Storey. Equipping the reflexion method with automated clustering. *wcre*, 0:89–98, 2005.

[13] Paul Clements, David Garlan, Reed Little, Robert Nord, and Judith Stafford. Documenting software architectures: Views and beyond, 2002.

[14] Mariano P. Consens and Alberto O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of the 1993 ACM SIGMOD International Conference on Management Data, SIGMOD Record Volume 22, No. 2*, pages 511–516, 1993.

[15] Dan Cosma. **UNIX - Aplicații**. Editura de Vest, Timisoara, Romania, , ISBN 973-36-0338-4, 2001.

[16] Dan Cosma. **Towards Building Feature-Oriented Distributed Systems**. In *Proceedings of the 6th International Conference on Technical Informatics, CONTI 2004, vol 3 (PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE Vol.49 (63) No. 3, ISSN 1224-600X)*, 2004.

[17] Dan Cosma and Sorin Serau. **TraceMaster - A Logging Service for Distributed Systems**. In *4th International Conference on Technical Informatics (CONTI 2000), Timisoara, Romania (Buletinul Stiintific al Universitatii "POLITEHNICA" din Timisoara, seria AUTOMATICA SI CALCULATOARE, Vol. 45(59) No.3, ISSN 1224-600X)*, 2000.

[18] Dan Cosma, Stejarel Veres, and Adrian Petru Mierlutiu. **Aplicații software distribuite**. Editura de Vest, Timisoara, Romania, ISBN 973-36-0377-5, 2003.

[19] Dan C. Cosma. **niSiDe: Interactive Tool for Understanding Distributed Software**. In *10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2008) Timisoara, Romania*, 2008.

[20] Dan C. Cosma and Radu Marinescu. **Distributable Features View: Visualizing the Structural Characteristics of Distributed Software Systems**. In *Proceedings of the 4th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2007)*. IEEE CS, 2007.

[21] Dan C. Cosma and Radu Marinescu. **Understanding the Impact of Distribution in Object-Oriented Distributed Systems Using Structural Program Dependencies**. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008)*. IEEE CS, 2008.

[22] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[23] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. Foreword By-Ralph E. Johnson.

[24] Stephan Diehl, editor. *Software Visualization*. Springer, 2002.

[25] Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of CSMR 2004 (Conference on Software Maintenance and Reengineering)*, pages 309–318, 2004.

[26] 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.

[27] Stéphane Ducasse, Michele Lanza, and Romain Robbes. Multi-level method understanding using microprints. In *Proceedings of Vissoft 2005 (3th IEEE International Workshop on Visualizing Software for Understanding)*, September 2005.

[28] Dennis Edwards, Sharon Simmons, and Norman Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, 2006.

[29] Ehcache single-page online user guide. http://ehcache.sourceforge.net/EhcacheUserGuide.html.

[30] Stephen G. Eick, Joseph L. Steffen, and Sumner Eric E., Jr. Seesoft—a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[31] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering, vol. 29, pp. 210–224, Mar.*, 2003.

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

[33] George W. Furnas. Generalized Fisheye View. In *Proceedings of CHI '86 (Conference on Human Factors in Computing Systems)*, pages 16–23. ACM Press, 1986.

[34] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.

[35] Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.

[36] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Francisco, 1979.

[37] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's Weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 40–49, Los Alamitos CA, 2004. IEEE Computer Society Press.

[38] Tudor Gîrba, Michele Lanza, and Stéphane Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[39] Jan Graba. *An Introduction to Network Programming with Java*. Springer-Verlag New York, Inc., 2006.

[40] Minmin Han, Christine Hofmeister, and Robert L. Nord. Reconstructing software architecture for j2ee web applications. In *WCRE '03*, page 67. IEEE CS Press, 2003.

[41] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS '00*, June 2000.

[42] Trent Hill, James Noble, and John Potter. Visualizing the structure of object-oriented systems. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 191, Washington, DC, USA, 2000. IEEE Computer Society.

[43] Christine Hofmeister, Robert Nord, and Dilip Soni. *Applied software architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[44] Vladimir Jakobac, Alexander Egyed, and Nenad Medvidovic. Improving system understanding via interactive, tailorable, source code analysis. In *In Lecture Notes in Computer Science*, pages 253–268. Springer, 2005.

[45] Vladimir Jakobac, Nenad Medvidovic, and Alexander Egyed. Separating architectural concerns to ease program understanding. In *MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software*, pages 1–5, New York, NY, USA, 2005. ACM.

[46] Mikkel R. Jakobsen and Kasper Hornbæk. Evaluating a fisheye view of source code. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*, pages 377–386, New York, NY, USA, 2006. ACM.

[47] Dean J. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *Proceedings of ICSE '97*, pages 360–370, 1997.

[48] Jgraph visualization and layout. http://www.jgraph.com/.

[49] Jgrapht, a free java graph library. http://jgrapht.sourceforge.net/.

[50] Dong Jing, Yang Sheng, and Zhang Kang. Visualizing design patterns in their applications and compositions. *Software Engineering, IEEE Transactions on*, 33(7):433–453, July 2007.

[51] R. Koschke. An incremental semi-automatic method for component recovery. In *Working Conference on Reverse Engineering*, pages 256–, 1999.

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

[53] Rainer Koschke. Atomic architectural component recovery for program understanding and evolution. In *Proceedings of the International Conference on Software Maintenance*, October 2002.

[54] Rainer Koschke and Thomas Eisenbarth. A framework for experimental evaluation of clustering techniques. In *Proceedings of the International Workshop on Program Comprehension, IWPC'2000*. IEEE, June 2000.

[55] Rainer Koschke, J.-F. Girard, and M. Würthner. An intermediate representation for reverse engineering analyses. In *Proceedings of WCRE '98*, pages 241–251. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.

[56] Adrian Kuhn, Stéphane Ducasse, and Tudor Gîrba. Enriching reverse engineering with semantic clustering. In *Proceedings of Working Conference on Reverse Engineering (WCRE 2005)*, pages 113–122, Los Alamitos CA, November 2005. IEEE Computer Society Press.

[57] Danny B. Lange and Yuichi Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 342–357. ACM Press, 1995.

[58] Michele Lanza and Stéphane Ducasse. Polymetric views–a lightweight visual approach to reverse engineering. *IEEE Trans. on Sw. Eng.*, 29(9), September 2003.

[59] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.

[60] Shimin Li and Ladan Tahvildari. A service-oriented componentization framework for java software systems. In *WCRE '06*. IEEE CS, 2006.

[61] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[62] G. A. Di Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. De Carlini. Comprehending web applications by a clustering based approach. In *IWPC '02*. IEEE CS Press, 2002.

[63] Chung-Horng Lung, Xia Xu, Marzia Zaman, and Anand Srinivasan. Program restructuring using clustering techniques. *J. Syst. Softw.*, 79(9):1261–1279, 2006.

[64] Mircea Lungu, Adrian Kuhn, Tudor Gîrba, and Michele Lanza. Interactive exploration of semantic clusters. In *3rd International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT 2005)*, pages 95–100, 2005.

[65] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, CA, USA, 1997.

[66] C. Best M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[67] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings of the 12th International Conference on Tools with Artificial Intelligences (ICTAI 2000)*, pages 46–53, November 2000.

[68] S. Mancoridis, B. S. Mitchell, and C. Rorres. Using automatic clustering to produce high-level system organizations of source code. In *In Proc. 6th Intl. Workshop on Program Comprehension*, pages 45–53, 1998.

[69] 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 *ICSM '99*. IEEE CS Press, 1999.

[70] Andrian Marcus and Jonathan Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 125–135, May 2003.

[71] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, November 2004.

[72] C. Marinescu, R. Marinescu, P.F. Mihancea, D. Raţiu, and R. Wettel. iPlasma: An Integrated Platform for Quality Assessment of object-oriented design. In *Proc. ICSM '05 (Industrial and Tool Volume)*, 2005.

[73] 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.

[74] N. C. Mendonça and J. Kramer. Developing an approach for the recovery of distributed software architectures. In *IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension*, page 28, Washington, DC, USA, 1998. IEEE Computer Society.

[75] Nabor C. Mendonça. Architecture recovery for distributed systems. In *8th Working Conference on Reverse Engineering (WCRE'01), Workshop on Software Architecture Recovery and Modelling (SWARM'01)*, 2001.

[76] Nabor C. Mendonça and Jeff Kramer. An approach for recovering distributed system architectures. *Autom. Softw. Eng.*, 8(3-4):311–354, 2001.

[77] Merriam-Webster. *Merriam-Webster's Collegiate Dictionary, 11th Edition*. Merriam-Webster, July 2003.

[78] Brian S. Mitchell. A heuristic approach to solving the software clustering problem. *icsm*, 00:285, 2003.

[79] Brian S. Mitchell and Spiros Mancoridis. Comparing the Decompositions Produced by Software Clustering Algorithms using Similarity Measurements. In *Proceedings of ICSM '01 (International Conference on Software Maintenance)*. IEEE Computer Society Press, November 2001.

[80] Brian S. Mitchell, Spiros Mancoridis, and Martin Traverso. Using interconnection style rules to infer software architecture relations. In *Proceedings of the Genetic and Evolutionary Computation Conference*, Seattle, Washington, 2004.

[81] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly, 1999.

[82] Richard Monson-Haefel. *Enterprise JavaBeans*. O'Reilly & Associates, Inc., 2nd edition, 2000.

[83] Richard Monson-Haefel and David Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[84] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[85] David Parnas. Software aging. In *ICSE '94*. IEEE CS Press, 1994.

[86] Mariano P.Consens, Alberto O. Mendelzon, and Arthur G. Ryman. Visualizing and querying software structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 138–156, 1992.

[87] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[88] Martin Pinzger, Johann Oberleitner, and Harald Gall. Analyzing and understanding architectural characteristics of COM+ components. In *IWPC '03*. IEEE CS Press, 2003.

[89] Damien Pollet, Stephane Ducasse, Loic Poyet, Ilham Alloui, Sorana Cimpan, and Herve Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, pages 137–148, Washington, DC, USA, 2007. IEEE Computer Society.

[90] Alan LaMont Pope. *The CORBA reference guide: understanding the Common Object Request Broker Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[91] Denys Poshyvanyk, Yann-Gaël Guéhéneuc, Andrian Marcus, Giuliano Antoniol, and Václav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.

[92] VĞclav Rajlich. The role of concepts in program comprehension. In *in Proceedings of IEEE International Workshop on Program Comprehension (IWPC'02), 2002*, pages 271–278, 2002.

[93] Daniel Ratiu. Memoria: A unified meta-model for Java and C++. Master's thesis, Faculty of Automatics and Computer Science, "Politehnica" University of Timişoara, 2004.

[94] David Reilly and Michael Reilly. *Java: Network Programming and Distributed Computing*. Addison Wesley, 2002.

[95] Filippo Ricca and Paolo Tonella. Visualization of web site history. In *WSE '00*. IEEE CS Press, 2000.

[96] Filippo Ricca and Paolo Tonella. Using clustering to support the migration from static to dynamic web pages. In *IWPC '03*. IEEE CS Press, 2003.

[97] Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings ICSM '99 (International Conference on Software Maintenance)*, pages 13–22. IEEE Computer Society Press, September 1999.

[98] Arnold Robbins. *Unix in a Nutshell, Fourth Edition*. O'Reilly Media, Inc., 2005.

[99] M. Salah, S. Mancoridis, G. Antonio, and M. Di Penta. Towards employing use-cases and dynamic analysis to comprehend mozilla. *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 639–642, Sept. 2005.

[100] M. Salah, S. Mancoridis, G. Antoniol, and M. Di Penta. Scenario-driven dynamic analysis for comprehending large software systems. *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, 00, March 2006.

[101] Maher Salah and Spiros Mancoridis. A hierarchy of dynamic software views: from object-interactions to feature-interactions. In *Proceedings of The 20th IEEE International Conference on Software Maintenance (ICSM 2004)*, 2004.

[102] Bradley Schmerl, Jonathan Aldrich, David Garlan, Rick Kazman, and Hong Yan. Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466, 2006.

[103] Jon Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, 1996.

[104] Richard W. Stevens and Stephen A. Rago. *Advanced Programming in the UNIX(R) Environment (2nd Edition)*. Addison-Wesley Professional, 2005.

[105] W. Richard Stevens. *UNIX network programming, volume 2 (2nd ed.): interprocess communications*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[106] Diana Szentivanyi, Sorin Serau, Dan Cosma, and Ioan Jurca. **DSDAgents - A Java Framework for Developing Mobile Agents Systems**. In *3rd International Conference on Technical Informatics (CONTI 1998), Timisoara, Romania*, 1998.

[107] Andrew Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 2002.

[108] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2001.

[109] M. Termeer, C.F.J. Lange, A. Telea, and M.R.V. Chaudron. Visual exploration of combined architectural and metric information. *vissoft*, 0:11, 2005.

[110] Ozgur Turetken, David Schuff, Ramesh Sharda, and Terence T. Ow. Supporting systems analysis and design through fisheye views. *Commun. ACM*, 47(9):72–77, 2004.

[111] Arie van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. *wicsa*, 0:122, 2004.

[112] Richard Wettel and Michele Lanza. Visualizing software systems as cities. *vissoft*, 0:92–99, 2007.

[113] Theo Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In Ira Baxter, Alex Quilici, and Chris Verhoef, editors, *Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering)*, pages 33–43. IEEE Computer Society Press, 1997.

[114] Jie Wu. *Distributed Systems Design*. CRC Press LLC, 1999.

[115] Christoph Wysseier. CCJun – polymetric views in three-dimensional space. Informatikprojekt, University of Berne, June 2004.

[116] Chenchen Xiao and Vassilios Tzerpos. Software clustering based on dynamic dependencies. *csmr*, 00:124–133, 2005.

[117] Hong Yan, David Garlan, Bradley Schmerl, Jonathan Aldrich, and Rick Kazman. Discotect: A system for discovering architectures from running systems. In *In Proc. 26th International Conference on Software Engineering*, pages 23–28, 2004.