

Restructuring Object-Oriented Distributed Systems: An Impact-Driven Approach

Dan C. Cosma

Radu Marinescu

LOOSE Research Group

“Politehnica” University of Timișoara, Romania

danc@cs.upt.ro, radum@cs.upt.ro

Abstract

Distributed applications address the needs of heterogeneous networks of human activities, therefore they must consist of balanced, relatively autonomous inter-communicating components. Unfortunately, real-world systems do often not follow these requirements, or their architecture gets degraded in time by many ad-hoc changes. Consequently, these systems have to go frequently through the costly and oftentimes manually-performed operation of restructuring services and their inter-connections. This paper introduces a restructuring technique for distributed systems that supports an architect in performing a cost-aware analysis of various restructuring scenarios. The technique enables the engineer/architect to explore and evaluate the impact of the restructuring process at a fine-grained level, by providing both a forecast of the restructuring outcome, and the projected cost of the process itself. The approach described is mainly based on a suite of novel coupling metrics, a new algorithm that builds the forecast of the restructured architecture, and a formula for computing the cost. The approach was successfully applied on a commercial distributed framework implemented using RMI and in this paper we summarize the findings and the practical experience. Although the technique was developed for the specific case of distributed systems, we are confident that it can be easily generalized to a larger class of applications.

1 Introduction

Distributed object-oriented applications differ from their ‘locally-acting’ counterparts in that they are usually designed to fit the necessities of *heterogenous* networks of human activities. They must execute diverse tasks at geographically distinct locations, and the environment they run in often imposes important limitations on their ability of communicating with each other. At the same time,

the organizations they serve need to *integrate* their workflow in the wide dispersion of their worksites, therefore the distributed systems become complex, and they have to be made 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 paper 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 involves a process that allows the architect to experiment with different scenarios of fine-grained restructuring, and evaluate their impact in the target system.

The approach defines criteria to single out the classes that may represent an independent functionality within the code, and uses them as starting points for an algorithm that computes the probable layout of the system after extracting that particular functionality. Along with the projected layout, the techniques also provide means for assessing the cost of the redesign, while the insight gained through the process helps enhancing the understanding of the system.

We have designed the approach to address the necessities specific to distributed applications, and we have applied it in the context of systems built in Java and using Remote Method Invocation as the means of communicating over the

network. Nevertheless, we believe the issues we are raising in this paper can be generalized to other types of systems as well, and the process can be adapted to a larger class of software applications.

The paper is organized as follows: Section 2 discusses the design-related considerations for distributed systems, Section 3 presents our restructuring approach, Section 4 describes a case study, Section 5 presents the related work, and Section 6 draws the conclusions.

2 Design Quality of Distributed Systems

2.1 Well-Designed Services

Distributed applications ([17, 16]) are often structured as a set of services, that are available remotely via technology-specific mechanisms. Other components (clients) are provided means for discovering and connecting to the available services, and use them in accordance to their needs.

The design must provide the functionalities available remotely so that it both minimizes the network communication, and makes 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. We believe a service should be designed so that it is manageable by both the developer, and the user. The choice of implementing a smaller service will make it 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 an architecture that provides a certain symmetry regarding the importance of the various nodes. It consists of equally important remote components that balance their functionalities and minimize their inter-dependencies, especially by avoiding creating nodes that encapsulate features needed by the entire system. Centralized features will almost always become communication bottlenecks when the system is deployed;

moreover, in case they fail, it will significantly impact the entire system.

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 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 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.

2.2 Causes of Poorly-Designed Services

Real-life systems are definitely not optimal [4, 7], and this may happen because of at least the following causes:

Simplistic approaches. The systems can be poorly designed from the start when the developers don't properly analyze all the possibilities for creating a balanced architecture. Many applications are simple, client-server, systems that consist of a single, centralized server and one or several clients using its services. The services themselves are gathered at the server location for no other reasons than the simplicity and apparent swiftness of doing so.

False platform constraints. A somewhat related case is caused by incorrect perceptions of the architectural constraints the communication infrastructure imposes on the system. For example, many systems are deployed within third party application servers which provide specific functionalities, e.g. web-based user interaction, transaction management or persistency. While the application servers themselves do not impose architectural constraints that prevent a flexible design, many developers assume the correct way is to deploy all the main functionality at the precise location of the application server, avoiding an arguably more complex approach that distributes the system features over the network. This case is often encountered when a team of developers assimilates a new technology and aims to apply the knowledge in too short a period of time;

Evolution. As any software system, distributed applications can evolve in time, due to change requests arrived at different points in the system's life, or due to other causes, such as ad-hoc modifications and bug fixes. As a result, the evolved architecture may reach a point where it does not follow anymore the (otherwise good) design principles it was based on at the beginning.

2.3 Restructuring Support

The causes described above are often occurring in real-world systems. Therefore, the need to restructure a dis-

tributed system is probable during the system's life, and approaching this necessity implies two important measures:

- understanding the system, by identifying the distribution-aware features
- manipulating parts of the code (services, features or other groups of inter-related classes) so that to improve the system by restructuring it.

The goal of understanding the system can be achieved by applying various analysis techniques that help the engineer obtain significant insight on the system's characteristics, detailed enough to enable complex activities like those related to the maintenance and evolution. An example of a methodology developed specifically for understanding distributed systems is the one we previously introduced in [3], which assesses the impact the distributed aspect application's design.

The next goal for an approach aiming to support the maintenance process is to provide a flexible way of manipulating the code artifacts as a means for system restructuring. The involved techniques can be described and evaluated from two main perspectives:

- a *characterization of the restructuring results*, which should be in accordance with the objective of producing small and focused decentralized entities,
- 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 the results it provides.

3 The Approach

Our restructuring approach is built around a simple, yet versatile technique: the identification and *extraction* of those sets of the classes that seem to represent distinct functionalities, separable from the rest of the system. The idea is to identify those functional clusters that, once extracted from the system, can form the base for a newer and suppler service, deployable separately if necessary. Thus, the technique addresses the objectives described in Section 2, and provides means to reshape a poorly-designed system into one consisting of smaller, better focused decentralized entities.

In short, the main technique in the approach consists of the following steps:

- identify the classes that seem to represent a separable functionality
- apply an algorithm that extracts that functionality, and computes the resulting shape of the system

The extraction technique can be applied successively, as part of more complex *restructuring scenarios*. That is, the architect can experiment with chains of extractions, until the outcome of the process becomes acceptable as a viable solution of restructuring.

In order to find the best restructuring scenario, the engineer can apply the process to obtain two types of information:

1. **The forecasted layout** of the redesign attempt. This provides an overview of the structural 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 be to extract the respective classes from the existing code.

They can be used both as intermediary results in the iterative exploration of the various extractions, and as the final outcome of the approach, to characterize the future layout of the system restructured using the chosen scenario.

3.1 Selecting Extraction Seeds

The first step in our approach is building a so-called *extraction seed*, which we define as *the set of classes that represent the starting points for the extraction*. To build the seed, the engineer selects several classes that may represent an independent feature within the system, according to certain criteria.

The reasoning which aims to identify the relevant classes that form an extraction seed can be applied at different levels of granularity within the system. The best place to start is to focus the search to the already existing groups of system entities, such as components, services or packages, by considering procedures such as:

- use the existing knowledge on the system, and focus the attention to the already identified groups of classes. For example, analyze the components or modules that appear too large, and identify the classes that seem to provide separable functionalities;
- look at the entire system and select the classes that communicate directly via the network providing remote features;
- analyze the locally-acting groups of classes to find in a similar way representatives of distinct functionalities (e.g. database connectors, user interface handlers, etc.)

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 seeds.

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.

3.1.1 Selection by Service Identification.

The first strategy is based on the premise that in distributed applications, a description of a service (which in RMI is done by creating 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 entities and the related classes influence the structure of the system.

For Java/RMI, this translates to defining an extraction seed 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.

3.1.2 Selection by Functional Clusters.

The second strategy covers a more general case where we need to identify sets of purpose-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 component, a previously extracted service, or even the entire system.

The strategy uses a coupling-related metric to identify the most interesting classes, rather than focusing on the shared purpose of the classes themselves. That is, we define for each class C in group G a metric we call *In-group adequacy (IGA)*, as follows:

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

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 [3].

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

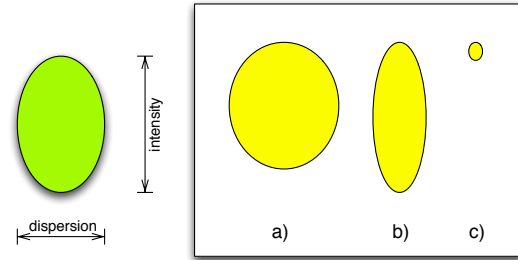


Figure 1. In-group adequacy (*IGA*) patterns

collaboration of the class with the classes in the group, i.e. the higher the chance that the class is better fitted to belong to 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) [8].

Visualization as support for selection The *IGA* metric can be used to discriminate between the system classes, in order to find candidates for the extraction seeds. However, the values are not always sufficient. Another source of information is the actual structure of the dependencies within the group, i.e. which class communicates with which, and what is the shape of the dependency chains within the group.

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 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 dependencies between two classes exist, only one edge is drawn. An example of such visualization is presented in Section 4, Figure 2.

The *In-group adequacy* is visible in the shape and size of the classes: each 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 for analyzing the classes in the group in order to find candidates that can be included in the extraction seed. 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:

General Contributor - classes drawn as large, almost cir-

cular ellipses (Figure 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.

Focused Contributor - classes drawn with the major (vertical) axis significantly larger than the horizontal one (Figure 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 Participant classes, drawn as small ellipses (Figure 1,c) feature low collaboration with the other classes, therefore they might be easy to extract (the cost would be fairly low). On the other hand, their limited collaboration may also suggest they cannot be considered as relevant representatives of features, they probably are only 'accessories' to such features (e.g. interfaces, utility classes, etc.). Considering them in extraction seeds will therefore add little value to the restructuring process.

The consequence of the above observations are that the best candidates for including in an extraction seed 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.

3.2 The Extraction Process

With the extraction seed 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 at the beginning of this section, 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.

3.2.1 Forecasted Layout of the Extraction

The first item is obtained by applying an algorithm that separates the original group by isolating the extraction seed and the closely related classes. We define the metric *Acquaintance with Class Group* between a class C and a group of classes G as follows:

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:

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

In the above formula, C_n is the current class, and K_{n-1} is a neighbor of the current class, at the previously processed level ($n-1$). The sum iterates on all classes accessible as neighbors (the directly coupled classes) from 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, defined as the sum of bidirectional couplings of the class with all the classes in the system [3].

The threshold factor. The algorithm iterates through all the classes in the group (except for those already included in the extraction seed) and calculates for each their acquaintance with the group of classes in the extraction seed. 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 seed 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.

3.2.2 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.

3.3 Remarks

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 seed) 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 seeds 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 seed classes) 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 seed 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 ex-

traction result.

Bottom line, once focused on relatively structured units of code, an extraction-based exploratory analysis can help isolating functionalities at an even finer degree of detail.

4 Case Study

This section presents a case study to show how our the approach can be applied to an object-oriented distributed software application. The analyzed system is FWS, a commercial framework for building and executing Java/RMI workflow applications, developed by a local software company.

A *workflow* is a specified sequence of local or remote activities executed by pieces of software called *agents*. The agents can be local or remote, and are executed conforming to a specification that essentially describes a state machine, each state being able to run more than one agents

FWS consists of two distinct components, deployed in different locations at runtime. One is the *Workflow Engine*, responsible for creating and executing workflow instances, the other is *Agent Engine*, responsible for managing and running the agents, both local and remote.

Preliminary extractions. The existence of only two components suggested that the system functionalities may need further separation in order to transform the system into a more balanced one. In order to assess the need of restructuring and the directions to consider as possible extraction scenarios, we started by looking at each component in turn. We noticed that both of them included more than one remote interfaces, meaning that they each published several remotely-available *services*. In the context we discussed in Section 2, it looked like the several services in each component may be good candidates for extraction in order to obtain an architecture consisting of smaller distributed entities.

Consequently, the first extraction scenario was created focusing on *identifying the distinct services* published by the system and the classes most involved in providing each of them. We used 6 extraction seeds, 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, the algorithm described in Section 3.2 obtained the 6 groups of classes closely related with each seed. For identification, we named them after the remote interface of the respective extraction seed: *Agent-Engine*, *AgentHandler*, *Workflow*, *WorkflowManager*, *MessageQueue*, and *ClientManager*.

The Workflow Group. The most prominent service group, both due to the number of classes (over 40), and

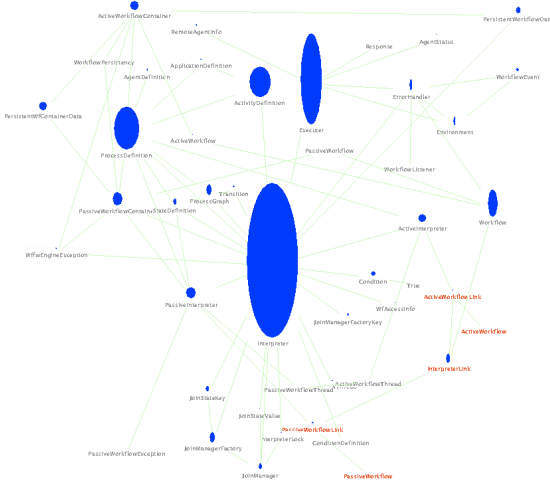


Figure 2. The Workflow service group

also considering its name, was Workflow (Figure 2). We have decided to analyze it further, to assess whether its internal structure facilitates possible extractions of independent functionalities.

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 considered each of these classes as individual extraction seeds (each extraction seed consisting of only one class), and performed independent extractions for each. The extraction costs obtained are shown in Table 1.

	<i>Executer</i>	<i>Interpreter</i>	<i>ProcessDefinition</i>	<i>Workflow</i>	<i>ActivityDefinition</i>
Extraction cost	1.98	1.78	2.60	2.3	2.56

Table 1. Extraction costs

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 high numbers just below 3.0 (when extracting prominent, highly connected classes). We have gathered these figures by experimenting with several extractions in different parts of the code.

Interpretation. The similar values for *ProcessDefinition* and *ActivityDefinition* along with their sharing 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 seed and we 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 seed, 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 *Interpreter*. Design-related data from the company that developed the system confirmed the two classes (*ProcessDefinition*, *ActivityDefinition*) implement the distinct functionality that stores the internal definition of a workflow (as a state machine). 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.

Conclusion. The process of applying successive extractions and experimenting with various extraction seeds 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 seeds 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.

5 Related Work

Understanding and restructuring distributed software is often addressed by the software engineering community.

Li and Tahvildari [9] 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. As ours, their approach uses graph representations to extract the relations between the entities and identify the business services in the applications. Ricca and Tonella [15] propose an interesting ap-

proach aimed at identifying the static Web pages in a site that are best candidates to be restructured as dynamic versions of themselves. The approach supports the migration from entirely static web sites to sites that are built using web application techniques. 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. In the same line of supporting the analysis, maintenance and development of web sites and applications, the authors also use a visualization technique [14] that presents the Web site evolution over the time. Di Lucca et. al. [10] propose a method of decomposing the application into functional units. They use techniques similar to ours when separating the functionality, specifically coupling-based measures, but they have a different goal: their focus is on the particular case of Web applications. Andreopoulos et. al. [1] use clustering to produce decompositions of large software systems. Their approach considers the multi-layered structure of the studied systems, and use both static and dynamic information to improve the clustering technique. Poshyvanyk et. al. [13] use a combination of Latent Semantic Indexing and scenario-based probabilistic ranking of events to identify the features in the source code. Edwards et. al. [5] use dynamic techniques based on causal ordering of events to address the feature location in distributed systems, while [6] employ static analysis and scenario-based dynamic information gathering to capture the features. In [11] the distributed architecture for C/Unix projects is recovered by identifying the modules, their relations, and the way they are shared in the program. The interaction features are identified using a syntactic pattern matching technique. Pinzger et. al. [12] analyze applications built on the COM+ component framework which and build a model suitable for understanding the software. Similar to our case, the constraints imposed by the technology are used to extract the relevant information, by following the major architectural patterns implied by the COM+ framework specifications. Chiricota et. al. [2] separate clusters of related software entities in a graph representation of an application by removing edges that loosely connect components having distinct functionality.

6 Conclusions

The distribution-related nature of distributed applications imposes specific concerns regarding their design. To fully benefit of the distributed aspect, the services must be decentralized, and focused on specific, well-isolated functionalities. As the design of many applications fail to follow these requirements, the need of restructuring often arises.

Our approach at supporting restructuring is based on a technique driven by applying sequences of extraction sce-

narios, in order to separate new functionalities within the system.

Coupling-based measures enhanced by visualization focus the selection of the items that initiate the extraction, characterize the extraction cost, and forecast the layout of the structural changes.

References

- [1] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Multiple layer clustering of large software systems. In *WCRE 2005*, pages 79–88. IEEE CS Press, 2005.
- [2] Y. Chiricota, F. Jourdan, and G. Melancon. Software components capture using graph clustering. In *IWPC 2003*. IEEE CS Press, 2003.
- [3] D. C. Cosma and R. 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.
- [4] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object Oriented Reengineering Patterns*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. Foreword By-Ralph E. Johnson.
- [5] D. Edwards, S. Simmons, and N. Wilde. An approach to feature location in distributed systems. *Journal of Systems and Software*, 79(1):57–68, 2006.
- [6] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, vol. 29, pp. 210–224, Mar., 2003.
- [7] M. C. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2005.
- [8] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.
- [9] S. Li and L. Tahvildari. A service-oriented componentization framework for java software systems. In *WCRE '06*. IEEE CS, 2006.
- [10] G. A. D. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. D. Carlini. Comprehending web applications by a clustering based approach. In *IWPC '02*. IEEE CS Press, 2002.
- [11] N. C. Mendonça and J. Kramer. An approach for recovering distributed system architectures. *Autom. Softw. Eng.*, 8(3-4):311–354, 2001.
- [12] M. Pinzger, J. Oberleitner, and H. Gall. Analyzing and understanding architectural characteristics of COM+ components. In *IWPC '03*. IEEE CS Press, 2003.
- [13] D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.
- [14] F. Ricca and P. Tonella. Visualization of web site history. In *WSE '00*. IEEE CS Press, 2000.
- [15] F. Ricca and P. Tonella. Using clustering to support the migration from static to dynamic web pages. In *IWPC '03*. IEEE CS Press, 2003.
- [16] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2001.
- [17] J. Wu. *Distributed Systems Design*. CRC Press LLC, 1999.