Distributable Features View:
Visualizing the Structural Characteristics of Distributed Software Systems

Dan C. Cosma Radu Marinescu
LOOSE Research Group
"Politehnica" University of Timişoara, Romania
{danc, radum}@cs.upt.ro

Abstract

The software industry is increasingly confronted with the issues of understanding and maintaining a special type of software systems, namely distributed systems. Although these systems are usually implemented in an object-oriented fashion, they raise very specific, and technology-dependent, understandability and quality assessment challenges. This paper presents a visual approach for comprehending the design of distributed software systems, by using technology awareness to isolate the functionally-distinct units within the code, so that the blueprint of the system's distributed behavior can be easily extracted. The approach provides means for observing the system's distributed architecture, visualizing the structure of the functional entities, and understanding their collaboration within the system, while focusing the analysis to the most substantial code fragments that deserve being taken into consideration.

Keywords: distributed systems, reverse engineering, software understanding, visualization

1. Introduction

Nowadays an increasing number of large-scale software systems are designed as distributed applications. Furthermore, a significant number of "classical" systems are redesigned with the aim of becoming distributable. In this context, the issue of understanding and assessing the design quality of such systems becomes increasingly important.

There are various kinds of distributed applications [19]: on one hand, some of them are in fact local applications that were just slightly modified to have a small remote functionality (*e.g.*, reporting the results of the various activities, logging the runtime events on specialized machines, etc); on the other hand, there are applications that were specifically designed with the distributed behavior in mind (*i.e.*, consisting of highly independent and feature-balanced structural components, running on different computers and communicating with each others to follow a common goal).

Analyzing the design of a distributed system must start with the understanding to which of these categories the system belongs, and how aware is it of the remotely available functionalities. Furthermore, the quality of the system's design can be assessed by determining whether the deployment of the various components is the best one, or whether the current design still agrees with the initial goals of the system – as any software, its design may decay in time [13].

As most distributed applications are nowadays written in an object-oriented fashion, we could employ already established analyses, techniques and tools for understanding and evaluating the design of object-oriented systems. These will definitely prove useful when trying to understand the application's core object-oriented structure.

However, there are important design structure and implementation aspects that are specific to distributed systems that can not be extracted by using a general approach. For example, assume that in a JAVA system we encounter the following declaration: `class MyClass implements AnInterface`. From an object-oriented viewpoint, all we can say about is that the software is designed using interfaces. Yet, if we take into account information related to a particular distributed technology (*e.g.*, *Java Remote Method Invocation*, we may notice that `AnInterface` is in fact extending the `java.rmi.Remote` interface. Suddenly `AnInterface` is not just another interface, but part of a core RMI-specific mechanism aimed to define a remotely available service. Therefore, a class that implements this interface is most probably a service provider, and it plays a central role in the distributed behavior of the system.

When trying to understand the design of a distributed systems there are at least three specific questions that we need to answer:

How much of the system is actually distributed? In other words, what is the the ratio between the classes that were specifically designed to act in a distributed context, and those that focus mostly on local activities. This can help an engineer to: (a) easily find the pieces of functionality that are easily deployable on different locations; (b) find out what parts of the application are technology-dependent and, therefore, what are the costs of changing the technology itself.

Which are the main features intended to be distributed? Furthermore, which are the sets of classes that one needs to focus on in order to understand the feature? Answering these questions helps us understand each of the distributed functionalities in isolation, by regarding them as a small set of classes that collaborate to provide a specific feature, and which may be deployed on a separate location.

What is the impact of the distributed features in the source code of the system? Knowing the answer to this question would enable us to find out how is its implementation dispersed among the system classes, and how easy it is to separate the feature from the rest of the system in case of a potential deployment at a different location. The answer is also related to scalability issues, as it helps identifying the classes that will have to change when trying to separate functionalities that have a common background.

All the aforementioned questions reveal the need for a multi-dimensional analysis of the distributed system. In this context, we present in this paper a novel visual approach, namely the DISTRIBUTABLE FEATURES VIEW, for understanding the design specificities of object-oriented distributed system, with a particular emphasis on systems that use RMI as a means for distributed communication. The use of the DISTRIBUTABLE FEATURES VIEW enables us to achieve the following goals:

1. *Capture the distributed nature* of the application, by finding the remote-acting entities within the code, and their main interactions and roles (such as client, server, peer etc.) The goal is to observe the basic structure of these interactions without having to read all the classes in the system, in fact by watching only those entities that describe it at its very core.
2. *Understand the patterns of collaboration* between the distribution-related functionality and the rest of the system.

The rest of the paper is organized as follows: next (Section 2) we provide a brief summary of the main concepts and issues related to distributed communication. In Section 3 we describe in a stepwise manner our analysis approach that in addition to looking at the code, also takes into account the specificities of distributed communication technologies. The DISTRIBUTABLE FEATURES VIEW is presented in Section 4, followed (Section 5) by a presentation of the most significant visual patterns that we identified while applying DISTRIBUTABLE FEATURES VIEW on two case-studies. After a brief discussion about some interesting findings (Section 6), and a presentation of some related work (Section 7) we summarize the conclusions of the paper in Section 8.

2. Communication in a Distributed System

To describe distributed software, we must consider our approach in respect to the general structure of this class of systems. Tanenbaum and van Steen [19] differentiate between three types of distributed software: *distributed operating system*, *network operating system*, and *middleware*, emphasising that the latter complies best with the requirements implied by the goals of distributed computing, by providing transparency, openness, and scalability.

Distributed software is, essentially, a set of arbitrary number of processing elements that run at different locations, interconnected by a communication system [20]. The communication system is an infrastructure built in such way that it hides all the details related to the communication itself, so that the application can focus on its actual goals. We call such an infrastructure the *Communication Mediator*, in short *Mediator*, and see it as a standalone, external actor, interacting directly with the application. Examples of mediators include classic middleware software (as Corba, DCOM), and any other type of software capable of providing means for remote communications (APIs, frameworks, services, etc.). Our current work focuses on using the Remote Method Invocation technology in Java applications, but the approach is designed to be extensible to other infrastructures as well.

The design of a distributed software system is highly dependent on the technology specific to the Mediator, this having a serious impact on how the software entities are written. There are different levels of the impact. For instance, in applications using BSD sockets [18], the constraints are directly visible in the code as patterns involving the creation and manipulation of the communication endpoints (the sockets). The Java-based Remote Method Invocation approach (RMI) [15, 4] involves defining special interfaces extending the provided java.rmi.Remote, that describe the services published remotely. There are cases when the constraints are not obvious in the source code; for example, for Java Message Service (JMS) [12], the Mediator is an external service, and the connection to it can be described dynamically, at runtime.

3. The Approach

In order to address the questions raised in the introduction we necessarily need to identify the design fragments (*i.e.*, classes and interfaces) that are implementing those features of the systems that involve a distributed communication. For this we developed a semi-automated approach supported by a tools infrastructure which is part of the IPLASMA software analysis environment [11] developed our group. While the main focus of this paper is not the detection of the design fragments that build the core of a system's distributable features, we will briefly present the steps of our detection approach (see Figure 1) in order to provide the reader with both the context and the motivation needed for understanding the visualization (*i.e.*, the DISTRIBUTABLE FEATURES VIEW) presented in the next Section.

Step 1: Detecting the Frontier Classes In order to find the distributed behavior of an object-oriented system, we first have to find those classes and interfaces that act as a *frontier* between the system and the *Communication Mediator*(see Section 2). These are the classes that depend directly on the mediator fir sending, receiving, or otherwise remotely manipulating data or events. We call them *frontier classes*. We start our analysis by the frontier classes as *they are the most significant representatives* of the distributed functionality of the system, the best first classes to look for when studying the distributed nature of the system.

In order to identify those *frontier classes* we have to rely on technology-specific programming idioms and coding rules. For the specific case of RMI, we consider as frontier classes (a) all interfaces implementing `java.rmi.Remote`, and (b) all the classes that call the methods of these interfaces.

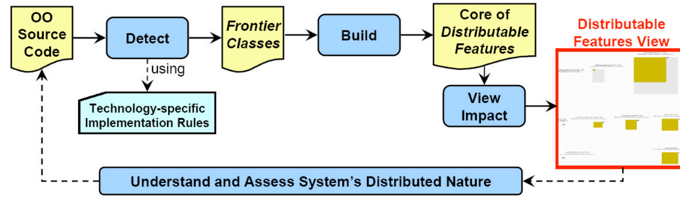

Figure 1. The process of understanding the distributable features of a software system

Step 2: Building the Cores of Distributable Features While knowing the *frontier classes* is essential for identifying the features of the system that are intended to be distributed it is obviously not enough to actually *understand* the distributable features themselves. Therefore, in the second step we want to find the *core of classes* that significantly add to the knowledge about all the distributable features, and then partition this core according to the specific distributable feature that they are most involved with. Consequently, this step involves two sub-steps: *Step 2.1: Build the dependency graph of the core classes involved in all* distributable features. The classes added in this step are mainly those that are "close" the *frontier classes*. To measure the "closeness" of two classes we use as a measure the *direct dependencies* between classes [7] in terms of method calls, attribute references, and inheritance. These classes are relevant as they interact significantly with the *Communication Mediator*, and they manage directly the remotely provided services. *Step 2.2: Identify the distinct* cores of distributable features. In order to identify the distinct cores we need to partition the core built in Step 2.1. In order to do this we employed two types of strategies to eliminate edges from the graph, so that we end with a set of connected components representing the distinct cores of distributable features. The first strategy uses information dependent on the constraints imposed by the mediator technology (*e.g.*, for RMI we eliminate all the generated stub classes). The second strategy uses is based on a modified version of the clustering algorithm used in [3], providing further feature separation by eliminating the *weak edges* in the graph, in terms of subsystem cohesion. The motivation of using this algorithm is that software is usually made of loosely-connected cohesive subunits [3, 10].

Step 3: Understanding the impact of distributable features The first steps of the analysis have basically split the system in two sets of classes:

1. classes contained in the *cores of distributable features*, included in the previous steps.
2. the rest of the classes in the system, as entities that have a more distant relationship with the distributed features of the system.

Our experiments proved that less than 20% of the system classes were designated as belonging to the *cores of distributable features*. While this is very valuable for narrowing the focus of the analysis, we can not ignore the fact that there are a lot of other classes in the application (the second set), and they are definitely there for a reason.

This brings us to a highly important challenge, that we aim to address in this paper *i.e.*, how to understand the relation between the classes contained in the *cores of distributable features* and the (overwhelming) rest of the system's classes?

For each class in the second set, we therefore need to assess:

- the level of collaboration between that class and the distributed part of the system.
- the level of collaboration (acquaintance) to **each** of the identified *distributable features*;'

In order to assess, we need indeed *measures* capable of telling us whether a class is related or not to a certain distributable feature, and what is the strength of this relation. Although these measures are necessary, what we need is a multifaceted assessment of the impact of the distributable features over the whole system. In other words, we must study the *concerted action* of all these measure, the way they actually combine to define the functionality of the classes in the system.

For such a complex assessment, we need more than numbers; we need to *see* all the concurrent aspects in a single picture, so that we can discover the relevant patterns. With this in mind we defined the DISTRIBUTABLE FEATURES VIEW as a means to represent simultaneously all the facets of this complex interaction between the classes directly involved in the various distributed features of the system and the rest of the classes.

4. Visualizing the Distributed Features

For such a complex assessment, we need more than numbers; we need to *see* all the concurrent aspects in a single picture, so that we can discover the relevant patterns. With this in mind we defined the DISTRIBUTABLE FEATURES VIEW as a means to represent simultaneously all the facets of this complex interaction between the classes directly involved in the various distributed features of the system and the rest of the classes.

4.1. The DISTRIBUTABLE FEATURES VIEW

The DISTRIBUTABLE FEATURES VIEW consists of two perspective: *Distributed Architecture* and the *Feature Affiliation* perspectives. While the former summarizes the information about the *cores of distributable features*, the latter is revealing the level and type of communication between the rest of the system's classes and the *cores of distributable features*.

4.1.1. Distributed Architecture Perspective In this perspective each *distributable feature core* – identified at the end of Step 2 of the approach described in the previous section – get a name and are represented as a rectangles of a different color. In Figure 2 we depicted the identified *distributable feature cores*

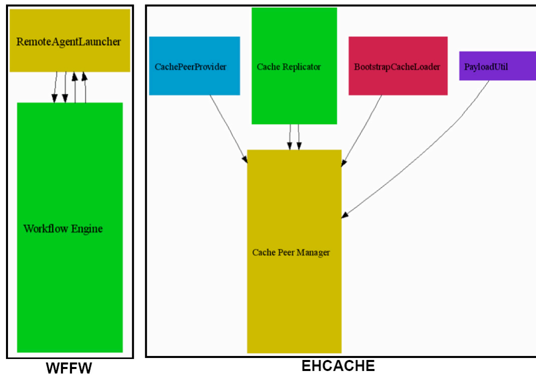

Figure 2. Distributed Architecture Perspective

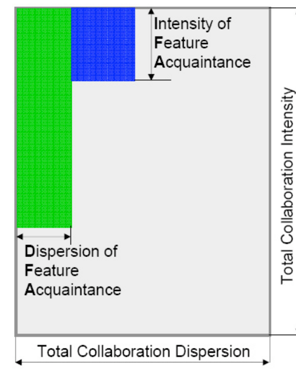

Figure 3. Feature Affiliation Perspective

for the two systems that we used as a case-study (see Section 5.1). While the widths of the rectangles are equal, their height is proportional with the number of classes that build each of the cores. From this point of view, the *Distributed Architecture Perspective* is a simple *polymetric view* [6].

The *remote calls* made by the classes belonging to the different cores are depicted in this visualization using directed arrows. Thus, the *Distributed Architecture Perspective* has a double role:

- It enables us to see both the architectural relations between the various distributable features, as represented by their core classes. This helps us to draw the first conclusions about the distributed architecture of the system. Thus, we can identify client-server dependencies (features calling each other in a single direction), peer-to-peer communication (bidirectional calls), or we can detect more complex interactions, as layered communication, rings of communicating entities, etc.
- It visually associates each distributable feature with a color. This is an essential element for the interpretation of the second perspective, the *Feature Affiliation Perspective*. From this point of view, this first perspective acts as a color legend for the second one.

4.1.2. Feature Affiliation Perspective The second perspective (Figure 3) expresses the impact of the distributable features in the rest of the system, revealing the level and profile of communication between the distributed features of the system and the other classes. This perspective depicts each of the classes that do not belong to a *core of distributable features* as follows:

- a light gray rectangle showing 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)[7].
- Within the light gray rectangle, vertically we display a set of colored rectangles, one for each distributable feature with which the class collaborates with, whereby the color corresponds to one of the colors from the *Distributed Architecture Perspective*. The *height* of each colored rectangle is proportional with the *intensity* of bidirectional coupling (acquaintance) between that class and a particular distributable feature, while the *width* is proportional with the *dispersion* of that coupling.
- 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 a visualization point of view, the *Feature Affiliation Perspective* extends the concept of *polymetric view* [6], by "embedding" within one polymetric rectangle a set of other (correlated) polymetric rectangles. This could be called a *composed polymetric view*, and to the best of our knowledge it was not used before as such.

4.1.3. The Feature Acquaintance Metrics Before closing the description of the DISTRIBUTABLE FEATURES VIEW, we need to add a few explanation about the metrics used to quantify the coupling between a *core of a distributable feature* and another class of the system (called *feature acquaintance class*).

We measure a *degree of acquaintance* of a class with respect to a given distributable feature, as a sum of the direct and indirect couplings between the class and all the classes in the core of that feature. The indirections are computed through chains of classes that act as intermediaries between the current class and the partition. Each indirect coupling is applied a fractionary factor so that the influence of the indirection is getting smaller when at a greater distance from the partition itself.

5. Patterns of *Distributable Features*

In this section we present the results of our experiments on two Java systems that use RMI for implementing the distributed part of the communication within the system. The aim of our experimental study was to empirically detect patterns of how the distributed part of a system impacts the rest of the system. More precisely we approached this experimental study with two concrete questions in mind:

1. Which are the *system-level patterns* that can be noticed by taking a 12,000 ft. look at a DISTRIBUTABLE FEATURES VIEW?

2. Which are the *class-level patterns i.e.*, which are the interesting types of classes that we can notice by looking at a DISTRIBUTABLE FEATURES VIEW?

We considered a pattern to be interesting, and selected it for presentation if it was frequently encountered, and if our code inspection showed that classes with that pattern play a particular role from the point of view of distributable features.

5.1. The Analyzed Systems

Before looking at the visual patterns that we identified during our experimental study, in order to get a better understanding of the patterns presented below let's first take a brief look at the two case-studies. In Figure 4 we summarize the size characteristics of the two systems. Additionally we placed in this table some numbers about the distributable features that we identified using the approach described in Section 3

	# Classes	# Methods	Code Size (KB)	# Distributable Features	# Classes in Distributable Features Core
WFFW	378	2585	~1250	2	35
EHCACHE	93	954	~720	5	16

Figure 4. The case studies in numbers

The WFFW System. This first system is a commercial framework for executing workflows developed by a local software company over several years[1]. A workflow is a sequence of actions that can be performed by agents. Being a framework, the goal of WFFW is to let its users to instantiate it with particular types of agents and activities. Agents can be either local or they can be distributed over a network. WFFW uses RMI for the communication between the workflow engine and the remote agents. One of the main advantages of using this system as a case-study is the fact that we have access to the developers of the system and can thus verify the validity of our findings.

The EHCACHE System. This system is a widely used Java distributed cache for general purpose caching[2]. The distributed aspect of EHCACHE is given by the fact a group of caches can act as a distributed cache, whereby each of the caches is a peer to the others. The central element of the system is the *cache peer* which can communicate via RMI with other peers. In this context, some of the issues addressed by EHCACHE are the replication of caches, peer discovery, peer bootstrapping etc.

5.2. System-Level Patterns

As mentioned earlier, a DISTRIBUTABLE FEATURES VIEW is a visualization of the entire system which reveals the impact of the system's features which were intended to be distributed. Therefore, in a DISTRIBUTABLE FEATURES VIEW the first thing that one can see is the the relative proportion between colored and gray visual elements. This gives us a good view over the global impact of the distributable features in the system. For example, in Figure 5 we see a fragment of the DISTRIBUTABLE FEATURES VIEW of the EHCACHE system[3]. If the colors dominate, then the entire system is strongly oriented towards providing the set of features identified as distributable. If gray is dominating the picture, it means that the system may also be providing other types of features, not directly related with the remote communication.

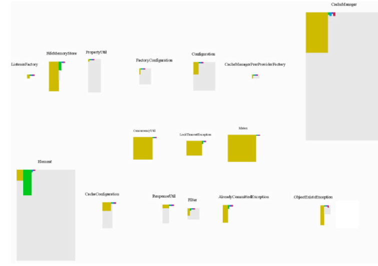

Figure 5. Part of a DISTRIBUTABLE FEATURES VIEW

Analysis of WFFW. A high-level look at the DISTRIBUTABLE FEATURES VIEW of WFFW reveals a lot of gray spots, meaning that there are a lot of features that does not directly relate to the distributable features. There are many classes with a large amount of gray *i.e.*, classes with lots of communication not related to the two distributable features. Furthermore, we identified a region of the DISTRIBUTABLE FEATURES VIEW, with over 70 classes, that don't have any color spots *i.e.*, they are completely unrelated with the distributable features. We found out that those classes are the implementation of a tool for visually editing workflow specifications.

By talking to the developers of WFFW we found out that the system was initially designed considering that the agents will be all local, and that it was later on extended with the feature of working also with remote agents. These remarks fit perfectly with the conclusions of our system-level reading of the DISTRIBUTABLE FEATURES VIEW.

Analysis of EHCACHE. In Figure 5 we display a fragment of the DISTRIBUTABLE FEATURES VIEW for EHCACHE, which is relevant for the general aspect of the view. We noticed in this case much more color than in the case of the WFFW system. Most of the classes in the view are acquaintances of the *Cache Peer Manager* distributable feature, which is at the same time the largest distributable feature in the system. This is explicable, as most of the functionality of EHCACHE is related to the issue of cache management. By analyzing the documentation of EHCACHE we found out that the application was also designed initially as a non-distributed system, but the way it was

1 The actual source-code of WFFW that we analyzed contains not only the framework itself, but also some small (test) application written to exercise the framework

2 http://ehcache.sourceforge.net/

3 Due to the fact that visualization depicts the entire system, and also due to the layout of the view it is unfeasible to present the entire DISTRIBUTABLE FEATURES VIEW for any of the two case-studies

redesigned in version 1.2 transformed EHCACHE from a rather monolithic into a more modular and distributable design (centered around the `CachePeer` concept).

5.3. Class-Level Patterns

The second phase of analyzing a DISTRIBUTABLE FEATURES VIEW is zoom-in and look at the most significant *feature acquaintance classes i.e.*, those acquaintance classes with remarkable traits. We identified three interesting patterns, all based on the ratio between the colored and the gray areas in each acquaintance class (*i.e.*, the level of communication with the distributable features versus the communication with other parts/features of the system).

Figure 6. Examples of *Big Color Spot* pattern

5.3.1. Pattern 1: *Big Color Spot* (Significant Feature Acquaintance) Classes with this pattern, appear in a DISTRIBUTABLE FEATURES VIEW with one (or more) dominant color spots, and thus only a small area is gray (see Figure 6). If only one color is dominant, then the class is very important for understanding the corresponding distributable feature. If several colors are present, in balanced area sizes, then the class is highly probable a *connector class* that offers common functionality for several distributable features. The *Big Color Spot* classes are also important in a restructuring context, as they are the main candidates for redesigning when trying to completely separate the features (*e.g.*, in order to deploy them on different locations).

Analysis of WFFW. In WFFW we identified 5-8 classes with large color spots, all related to the largest distributable feature which is *Workflow Engine* (see Figure 2). The developers of WFFW confirmed that in order to understand the *Workflow Engine* feature, it is enough for an engineer to look at these 5-8 *feature acquaintance classes* in addition to the the core classes of distributable feature. The fact that we found relatively few *feature acquaintance classes* related to the distributable features shows that the functionality of these features is very well located in the system.

Analysis of EHCACHE. Although EHCACHE is significantly smaller than WFFW we found more than 12 *Big Color Spot* classes. Similar to WFFW, in all of them there is a single dominant color *i.e.*, the one corresponding to the largest distributable feature which is *Cache Peer Manager*. These *feature acquaintance classes* have a different trait than those found in WFFW: they are squarish, not very large, and there is almost no gray area. This shows that these *feature acquaintance classes* are more dedicated to the distributable feature, and serve a specific role (that is why they are not very large) which is of interest to many classes in the distributable feature (that is why the aspect is squarish). From this point of view the names of the classes speak for themselves: `Mutex`, `Sync`, `ConcurrencyUtil`.

Figure 7. Two *Big Gray* classes

5.3.2. Pattern 2: *Big Gray* (Non-Distributed Feature Class) A *Big Gray* class (see Figure 7) is one with a lot of communication with other classes in the systems (the large gray area), but with little or no collaboration with the distributable features (very small amount of color, or no color at all). These classes are either implementing another (non-distributed) feature of the system or is a local utility class.

Analysis of WFFW. The most striking cases in WFFW is given by the over 70 classes, – already mentioned in the previous section, – that implement a tool for visually editing workflow specifications. In addition to those we noticed other 6 classes, all of them belonging to other features. For example we found `WorkflowIsPersistent` which deals with making a workflow persistent, or `MyWorkflowListener` which belongs to one the test applications built using the WFFW framework.

Analysis of EHCACHE. In this application there are not more than 5 *Big Gray* classes. The most prominent one (*i.e.*, the one with the largest gray area) is `Cache` which is one of the central classes in the system, modelling very concept of a cache. Of course it is heavily used through the entire system, but it it not directly connected with any of the distributable features, as these features are mainly dealing with the distributed storage of the cached information by means of cache peers. A further example is the `ConfigurationHelper` class which is a class that contributes to a local feature the manages the XML configuration files for EHCACHE.

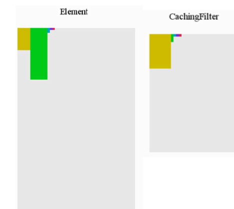

Figure 8. Two *Color Spotted Gray* classes

5.3.3. Pattern 3: *Color Spotted Gray* (Connector Class) h 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 that connects an already identified distributable feature with another non-distributed feature. These classes are particularly interesting, as their analysis helps in understanding the linkage between the two types of functionalities.

Analysis of WFFW. We noticed 5 significant *connector classes* and the discussion with the developers of WFFW confirmed that all of these classes really match the description. We are postponing for now the discussion of our findings for this case, as we will discuss next (Section 6) in a larger context a very interesting case *i.e.*, the `ProcessDefinition` class.

Analysis of EHCACHE. We visually identified 6 such *connector classes* in EHCACHE. The most interesting case is that of the class `Element` (see Figure 8) which represents the data items the system caches. It was the only class detected as having a noticeable relation with the *Cache Replicator* distributable feature (that is, a significant green stripe) as it is the single most important item the Replicator manipulates. Indeed, the purpose of replication is to exchange updated cached items, in fact `Element` instances. It consequently connects the *Cache Replicator* with the basic, non-distributable, caching feature of the system.

6. Discussion

Looking at classes in isolation provide valuable information about their placement in the system. Moreover, the findings related to the classes can be correlated with each other, so that higher-level understanding is achieved. In WFFW, we have detected an interesting relation between several of the classes that we placed in the three patterns described above (Fig. 9). By looking at the names of the classes and at the relations between them, we have seen that the `ProcessDefinition` class (that conforms to the Connector class pattern) actually links two significant features in the system. The `ProcessDefinition` proved to be the internal representation of the workflow that is enacted by the *Workflow Engine*. Classes that play a core role in managing the workflow (`ActiveInterpreter`, `PassiveInterpreter`), detected as significant feature acquaintances, use the `ProcessDefinition` to run the workflow. On the other side, there is a non-distributable feature, represented by the `PDParser` class which is in fact the functionality that parses the actual workflow specification (that we found out it is represented as an XML file) and creates the internal representation in the terms of the `ProcessDefinition` class. As an interesting side note, we have observed the `ActivityDefinition` class, which can be visually placed in both the Big Color Spot and the Color Spotted Gray patterns. By analyzing it, we found that in fact it is also a part of the workflow specification, its multiple instances being contained by the `ProcessDefinition` internal representation. Nevertheless, the class describes the most active part of the workflow, that is, the activities that the workflow starts at each step. Consequently, it is intensely used by the core of the *Workflow Engine*, and it earns its placement as a class that is very important to analyze when understanding the Engine functionality itself.

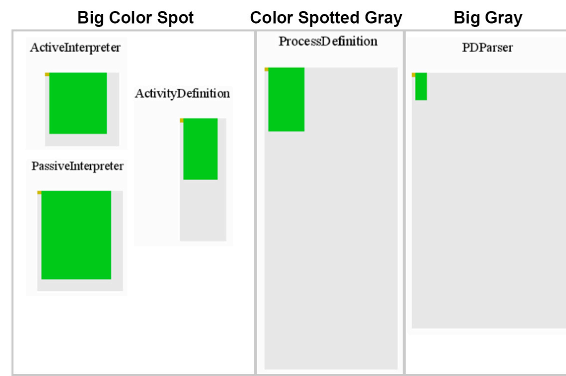

Figure 9. Interesting classes from WFFW

7. Related Work

The issue of understanding distributed software is often being addressed by the software engineering research community. Identifying the structure of the application is central to many approaches, and involves finding the functionally-distinct components or partitions within the code. Often, the constraints imposed by the technology are used as starting points in understanding the software, while clustering is in many cases the core technique for separating the functionalities. Pinzger et. al. [14] extract information from applications based on Microsoft's COM+ component framework to build a model suitable for understanding the software. Similar to our case, their approach is highly aware of the technology-related constraints. It follows the major architectural patterns implied by the COM+ framework specifications to identify the components, analyze meta-data from the type libraries, and extract the information from the configuration and the source code of the system. Li and Tahvildari [8] propose a service-oriented componentization framework for systems written in Java. Graph representations are used to identify the business services in the applications by extracting the relations between the entities. This resembles our approach in separating the features, but their purpose is different: to transform the object-oriented design of the application into a service-oriented one by identifying of top-level and low-level services. A visualization technique for finding the interactions between the components in a RMI application is presented in [2]. The main difference to our approach is that they use dynamic monitoring of the RMI calls to detect the relations between the components; the structure of the components is not taken into consideration. Han et. al. [5] propose a technique that reconstructs the software architecture for Java2 Enterprise Edition Web applications. They use an approach that separate the architecture descriptions into four views to reduce its complexity, thus making the system easier to understand. Di Lucca et. al. [9] also focus on Web applications as an important type of distributed software, and apply a method for decomposing the application into functional units. While refer-

ring only to the particular case of Web applications, their approach involves techniques similar to ours when separating the functionality: a coupling-based measure feeding a clustering algorithm to focus the understanding.

Clustering is also used by Andreopoulos et. al. [1] to produce decompositions of large software systems. They choose an approach that enrich the clustering technique with both static and dynamic information, and take into consideration the often multi-layered structure of the studied systems. Ricca and Tonella [17] employ clustering to facilitate the migration from static to dynamic web pages They use a semi-automatic process that recognizes the structure of an existing Web site, and generates candidate templates that should be involved when dynamically generating the pages through a Web application. In [16], they also use a visualization technique to show the Web site history. Chiricota et. al. [3] propose an efficient clustering algorithm that separates related clusters of software within an application represented as a graph. They focus on detecting weak edges, that is, edges that loosely connect different components that may prove to have distinct functionality. Their idea proved useful in one of the steps involved by our approach.

8. Conclusions

This paper presents a visual approach for understanding distributed software, by identifying the impact of the distributable features in the system. The approach uses visualization to narrow the scope of the analysis, by focusing on the patterns occurring in the DISTRIBUTABLE FEATURES VIEW. The constraints imposed by the communication technology are involved in the separation of the features. Two case studies of systems using Java RMI were conducted to extract the relevant patterns occurring in relation to the distributable features impact in the system.

Our future work will focus on instantiating the approach on additional communication technologies, specifically, we intend to study the particularities of applications built using Web Services. Furthermore, the process will be used to extract additional information from the systems, such as identifying possible deployment scenarios for the detected distributable features.

References

[1] B. Andreopoulos, A. An, V. Tzerpos, and X. Wang. Multiple layer clustering of large software systems. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 79–88, Washington, DC, USA, 2005. IEEE Computer Society.

[2] N. Bawa and S. Ghosh. Visualizing interactions in distributed java applications. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 292, Washington, DC, USA, 2003. IEEE Computer Society.

[3] Y. Chiricota, F. Jourdan, and G. Melancon. Software components capture using graph clustering. In *Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC 2003)*, 2003.

[4] J. Graba. *An Introduction to Network Programming with Java*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[5] M. Han, C. Hofmeister, and R. L. Nord. Reconstructing software architecture for j2ee web applications. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*, page 67, Washington, DC, USA, 2003. IEEE Computer Society.

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

[7] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer Verlag, 2006.

[8] S. Li and L. Tahvildari. A service-oriented componentization framework for java software systems. In *WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 115–124, Washington, DC, USA, 2006. IEEE Computer Society.

[9] 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: Proceedings of the 10th International Workshop on Program Comprehension*, page 261, Washington, DC, USA, 2002. IEEE Computer Society.

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

[11] C. Marinescu, R. Marinescu, P. Mihancea, D. Raţiu, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *Proc. IEEE International Conference on Software Maintenance(Industrial and Tool Volume)*, 2005.

[12] R. Monson-Haefel and D. Chappell. *Java Message Service*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2000.

[13] D. L. Parnas. Software aging. In *Proceedings 16th International Conference on Software Engineering (ICSE '94)*, pages 279–287, Los Alamitos CA, 1994. IEEE Computer Society.

[14] M. Pinzger, J. Oberleitner, and H. Gall. Analyzing and understanding architectural characteristics of com+ components. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 54, Washington, DC, USA, 2003. IEEE Computer Society.

[15] D. Reilly and M. Reilly. *Java: Network Programming and Distributed Computing*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2002.

[16] F. Ricca and P. Tonella. Visualization of web site history. In *WSE 2000: Proceedings of the International Workshop on Web Site Evolution*, Los Alamitos, CA, 2000. IEEE CS Press.

[17] F. Ricca and P. Tonella. Using clustering to support the migration from static to dynamic web pages. In *IWPC '03: Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 207, Washington, DC, USA, 2003. IEEE Computer Society.

[18] W. R. Stevens, , B. Fenner, and A. Rudoff. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 2004.

[19] A. S. Tanenbaum and M. V. Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[20] J. Wu. *Distributed Systems Design*. CRC Press LLC, Boca Raton, Florida, 1999.