# Managing an Industrial Software Rearchitecting Project With Source Code Labelling

Brice Govin, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, Arnaud Monegier Du Sorbier

# Managing an Industrial Software Rearchitecting Project With Source Code Labelling

Brice Govin[1,3], Nicolas Anquetil[2,3], Anne Etien[3], Stephane Ducasse[2,3], and Arnaud Monegier du Sorbier[1]

[1] Thales Air Systems, Rungis, 94150 Val-de-Marne, France
{*firstName.lastName*}@thalesgroup.com
[2] Synectique, 2 Rue Jacques Prévert, 59650 Villeneuve d'Ascq, France
{*firstName.lastName*}@synectique.eu
[3] Inria Lille – Nord Europe
{*firstName.lastName*}@inria.fr

**Abstract.** Legacy software systems are valuable assets for organisations. From time to time, renewing legacy software system architecture becomes necessary in order to offer them a new future. Rearchitecting a complex legacy software system is a difficult task. It involves understanding and aggregating a large set of data (the entire source code, dependencies, etc.). Understanding a software system is a matter of identifying the concepts that are implemented in the source code and organizing these concepts in a shared logical view of the system (*e.g.* an architectural view). This paper presents the approach used in a real industrial rearchitecting project of a complex legacy software system. We explain how concepts were modelled and mapped to the source code through entities called *tags*. We show how these *tags* were used by engineers and what tools were created to help them.

## 1 Introduction

Legacy software systems represent a significant part of companies' wealth. Maintainability of such systems has become an important goal for companies which own them. Software evolution activities can be classified in two: day to day evolution, where engineers correct bugs, implement new features, or adapt the system to new working conditions; and, much rarer, rearchitecting, where engineers need to completely re-think the architecture of a system because it seems no longer possible to adapt it in small steps.

In our case, the company we are working with wants to restructure a real-time, embedded, critical system. The system is an old (30 years) Ada system in the defence area. The goal is to regain knowledge over the application and to rearchitect the software system to: (1) ease day to day evolution; (2) take advantage of technological improvements that were introduced since the creation of the system.

Rearchitecting a complex legacy software system is a difficult task. Most of the time, complex software systems are too large to be entirely understood by

one person. It involves understanding and aggregating a large set of data (the entire source code, dependencies, etc.). Then understanding a complex software system goes to abstract the software system in an expressive representation. Research has proposed different solutions to extract expressive representation from source code.

We present in this paper how we performed a rearchitecting project. Our solution was based on identifying concepts in source code and materializing them with a special constructs called *tags*, that was used in the industrial rearchitecting of our partner company. Because *tags* are mapped to the source code, we can manipulate, analyze, visualize and compare the *tags* as any other source code entity. We also show how the *tags* were used on the rearchitecting project.

Section 2 presents the context of the reachitecting project. Section 3 presents the existing solution, in literature, for identifying concepts in a software system. Section 4, Section 5 and Section 6 present, respectively, the stage of reverse engineering, re-engineering and forward engineering performed during the project of our partner company. Section 7 presents our solution (*tags*) and show how it was used during the three stages of the project of our partner company. Section 8 sums up this paper and opens on the future work we planned to do to improve our solution.

## 2 Rearchitecting Project Context

A large company called us to follow and help in a complex legacy software rearchitecting project. This project aimed to rethink the software system architecture to include new technologies and new architecture requirements. This section describes the project and the company concerns about the software system.

### 2.1 Software System Description

The rearchitecting project concerns a real-time, embedded, critical software system in the defence area.

The software system is old (30 years), large (500 KLOC) and written in Ada 95. The Ada entities of interest are: packages, tasks, subprograms and instructions. Packages contain entities that can be other packages, tasks, subprograms or instructions. Subprograms contain only subprograms or instructions. Tasks contain subprograms and instructions. Instructions are elementary entities (*e.g.* subprograms calls). The software systems contains 1 494 packages, 13 474 subprograms and 44 tasks.

According to experts interviews and available documentation, engineers found that the software system architecture should be composed of modules and messages. Each module would be responsible for a single, high-level, responsibility and modules should communicate between themselves only by sending messages.

### 2.2 Company's Concerns and Issues about the System

When looking at the characteristics of the existing software system, described in Section 2.1, engineers and experts confronted these characteristics with their client needs. Thus, they identified several issues, that should be answered by the new architecture:

**Modular architecture:** The elements of the existing architecture are not modular enough.

**Standardization of the communication system:** The current communication system is proprietary. It is a costly task for the company to maintain this communication system.

**Data-centric software architecture:** The company wants to have a data-centric architecture instead of the existing treatment-based architecture.

**Separate concerns in architectural elements:** Although the existing architecture is based on layers, dependencies between layers are not done properly.

A prior study in the company led to specific choices to answer each requirements of the new architecture. The company targets a component-based architecture to answer to the modularity issue. The company decided to use Data Distribution Service (DDS)[4] standard as their new communication system, to get rid of their proprietary communication system. DDS was chosen because it is a well known standard and because it is based data sharing rather than on the treatment.

Although new technologies are chosen to rebuild the architecture, the company stated that the new software system must respect exactly the same functional requirements as the legacy one. That implies that the new version of the software system musthave the same behavior and react in the same time as the legacy system. To reach that goal, the company uses the results of the validation tests on the legacy system as their golden standard for comparison.

## 3 State of the Art

The company wanted to identify and materialize the concepts hidden in its legacy software system. Research literature provides several solutions. Concepts materialization are addressed by techniques like Reflexion Model [6] or Intentional Views [5]. Concepts identification is addressed by Formal Concepts Analysis [1] and Feature Location [7]. We explain in more details each of these solutions in the following sections.

### 3.1 Reflexion Model

Reflexion Model aims to extract viewpoint of an architectural model. Viewpoints are strictly subjective and can be used to retrieve physical architecture, logical architecture or even concepts.

[4] http://portals.omg.org/dds/

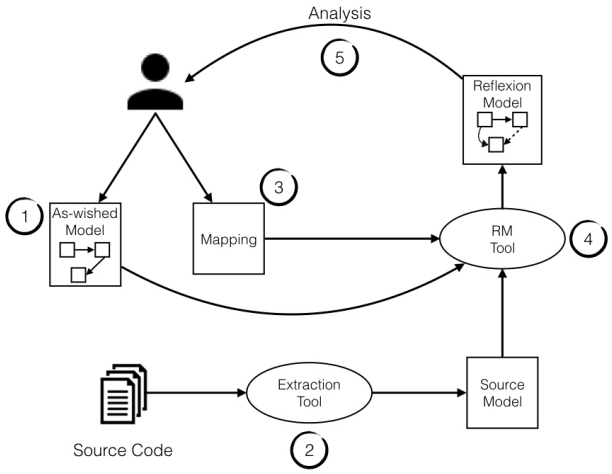


**Fig. 1.** Software Reflexion Model approach from Murphy et al. in [6] (numbers refer to steps of the approach described in the text)

The reflexion modelling technique uses a mapping between a particular high-level model and a model of the source code. This technique is divided into five steps (see Figure 1):

1. The engineers express an architectural *as-wished*(AW) model. Usually, the *AW* model derives from the engineer expertise and the available documentation.
2. The second step consists of extracting structural information from the source code of the system. Structural information can be extracted through statical or dynamical analyses and constitute what is called the *source model* (*e.g.* in Ada, packages or subprograms).
3. In this step, the engineers define a mapping between the *AW* model elements and the structural entities extracted at the previous step.
4. The fourth step uses a tool to compute the reflexion model. Relationships of the Reflexion Model are classified according they are expected in the *AW* model but not found; or expected and found; or not expected but found.
5. In the last step, the engineers analyze the reflexion model . In case the result is not satisfactory, and they can refine either the mapping or the *AW* model.

### 3.2 Intentional Views

Intentional Views was first proposed by Mens *et. al* in [4] as a mean to model concerns that are implemented in source code.

An *intentional view* is a high level concept materialized by a logic predicate that selects the source code entities corresponding to this concept. *Intentional*

*View* can also be refined manually by excluding or including given code entities. The solution proposed by Mens *et. al* is composed of:

***Language model*:** Describes the software entities that will be extracted from the source code (*e.g.* packages, subprograms).

***Intentional view model*:** Contains *intentional views* and the relationships between these views.

Because *intentional views* are explicitly mapped to source code entities, relationships between the views can be automatically derived from the existing relationships between source code entities. Other specific relationships can be retrieved thanks to user-defined predicates that consider the existing relationships between source code entities (*e.g.* **noInstanceVariable** predicate).

### 3.3 Formal Concept Analysis

Formal Concept Analysis (FCA) computes all possible decomposition of objects that have some attributes in common. In this technique, a concept is composed by all the objects that have a specific set of attributes in common. In software systems, objects are generally code entities (*e.g.* in Ada, packages or subprograms or tasks). Attributes are boolean characteristics of the objects, for example for a package, the attributes could be each other packages it imports or the data types it refers to. All possible combination of attributes can lead to a complexity explosion in case of a large software. For example, Siff and Reps experiments in [8] show that from a simple *Perl* application of 26.9 KLOC, giving 189 objects and 32 attributes, FCA computes more than 13 000 concepts. Handling that much concepts is not humanly acceptable, thus concepts have to be handled automatically. However, Bhatti *et al.* state that, automatically handling the complexity of concept lattices is strongly dependent on the context of the software system [9]. Therefore, an algorithm that automatically handles the concepts of software system requires to have a strong understanding of the software system, that the company does not have.

### 3.4 Feature Location

Revelle and Poshyvanyk defined *feature location* as"the activity of identifying the source code elements (i.e., methods) that implement a feature"[7]. In the same paper, they define a *feature* as "a functional requirement that produces an observable behaviour which users can trigger". They classify feature location in three categories that can be combined: (1) Textual Analysis; (2) Dynamical Analysis; (3) Static Analysis.

Textual analysis is strongly dependent on the coding convention deployed in the software system. In the project presented in Section 2.1, one issue is that time and evolution eroded the software system. The coding conventions were not strictly respected during the evolution and maintenance of the software system. Therefore, textual analysis techniques are not applicable in this case.

Dynamical analysis requires to be able to execute the system. However, the test environment of our partner project were not functional and the software system could not be executed in test environment. Moreover, tracing the execution requires to modify the source code to put probe in the software system and one of the requirements was to not modify the executing code.

Static analysis seems to be the better compromise to be used. Although, PDG is adapted to feature location, the size of graph can quickly become overwhelming since we are dealing with complex software system.

## 4 Reverse Engineering Phase

Following the classical "Horseshoe" process [3], the project was decomposed in 3 stages (reverse, re- and forward engineering) that are presented in this section and the 2 following sections. Prior to the task of rearchitecting, the company wanted to first regain the knowledge over its legacy software system. As stated in Section 2.1, interviews with experts and documentation allowed the engineers to idealize a first draft of the existing software architecture. This draft assumed a crude architectural organization in Modules that communicated between themselves through Messages. One goal of the project was to get rid of the proprietary communication infrastructure. Thus engineers tried to recover the modules and messages in the code (*i.e.* map source code elements to the notions of modules and messages).

They started with a manual analysis of the code steered by an old design specification document. In their analysis, they found that modules were represented by Ada packages that contain Ada Tasks with specific naming convention. They identified that packages that should be part of a library since they are used by several modules. They also found that messages were Ada packages that declares a variable with a specific naming convention. Then, they found a specific pattern for message reception and another specific pattern for message emission. Using these two patterns, they identified the messages exchanged between modules.

They materialized the modules and messages in the source code by using *tags*, our solution (see Section 7). Each module is represented by a different *tag* and each message is also represented by a different *tag*. The conventions and patterns they found allowed to semi-automatize the *tagging*[5].

Then, they analysed the existing architecture using the *tags* and their mapping. Engineers discovered that only one package per module was receiving messages and that this specific package was calling the other packages mapped to the modules. Interviews with other experts uncovered the fact that the modules should be composed of one active package and several passive packages. The active package is the only package that should receive messages and trigger the processing of these messages. Hence, the active package is ensuring communication with other modules. The processing of the messages is done in the passive packages that are implementing the business logic of the modules. This analy-

[5] *Tagging* is the fact to map a *tag* to a code entity

sis results in a new representation of the existing architecture, resulting in the simplified meta-model of Figure 2

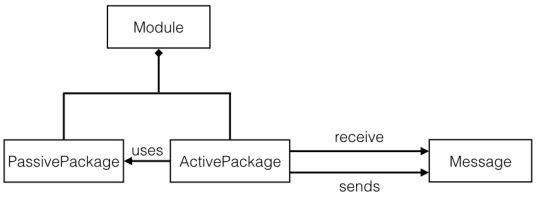


**Fig. 2.** Simplified meta-model of the final software system existing architecture

Using this meta-model plus the convention and the patterns found, engineers dressed a representation of the current architecture of the legacy software system. Figure 3 represents a simplified version of the current architecture of the system. The horizontal square boxes are modules, the circles inside are passive packages

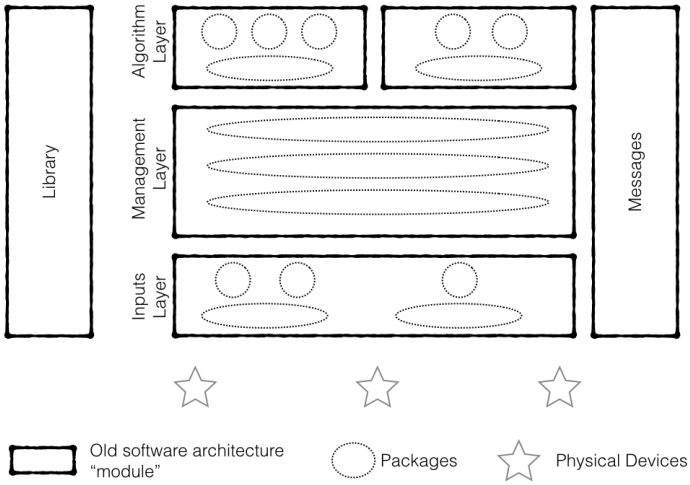


**Fig. 3.** Simplified representation of the existing software architecture

and the ellipses are active packages. Engineers also represented the Physical devices with which the software system is working to highlight the fact that part of the communication is done with these devices.

Finally after tagging about 1 500 packages, engineers identified 44 modules, a Library and over 500 messages. This reverse engineering stage mobilized four engineers during six months. Others engineers were punctually required, for reviews of the extracted architecture or interviews.

## 5 Re-enginering Phase

Before starting the re-engineering phase, domain experts and engineers met to define the new architecture. Section 2.2 stated that the company is aiming at a component-based and data-centric architecture. Domain experts and engineers defined a set of high-level components to be the elements of their new architecture. A simplified representation of the new architecture is given in Figure 4.

The new architecture is composed of three layers: a business layer, a data layer and an interface layer. Both, the business and the interface layer hold Composite Components containing Atomic Components. The data layer represents the new communication system (DDS) that holds the entire data model of the application.

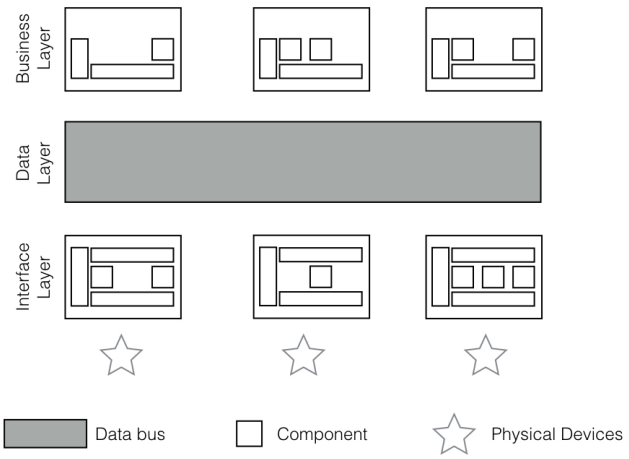


**Fig. 4.** Simplified representation of the target component-based architecture

Engineers studied the source code to map code entities to one of the 14 pre-defined Composite Components, using *tags*. They did this by actually assigning code entities to potential Atomic Components within the Composite

Components. Two tasks were performed concurrently: assign code entities to the Composite Components and, then, decomposing these into a number of Atomic Components. First, engineers tried to allocate entire packages to a given Atomic Component. Yet studying it, they might consider that a part of it (maybe a subprogram) would need to be assigned to another Atomic Component (but always in the same Composite Component); they may sometimes even go down to assigning instructions (always subprogram invocations) to another Atomic Component.

Based on their understanding of the source code, the re-engineers selected a set of packages that were likely to answer the responsibility of a component. The packages imported by each of these initial packages are normally allocated to the same component. Exceptions occurred when an imported package had already been allocated to some other component. In that case, a deeper analysis of the package was performed, by looking at the comments in the source code, to resolve the conflict. In case of doubt, another possibility was available: a library component gathering all the packages that could not be allocated to one single component.

Later, in a second time, subprograms and subprograms invocations were mapped to Atomic Components. Engineers' analysis of the packages content relied on structural information such as dependencies between subprograms, and instructions in order to identify flows of execution. They used *tags* to represent the Atomic Components and mapped subprograms and/or their calls to *tags*.

The re-engineering phase mobilized two engineers during 6 months. As for the reverse-engineering phase, other engineers were punctually required for reviewing the refined architecture.

## 6 Forward Enginering Phase

In accordance with the mapping done during the two last phases, engineers are rewriting the existing code to make the legacy software system work with its new architecture.

At the moment, this project is ongoing, Atomic Components have been identified in all 14 Composite Components. The entire source code has been mapped to components and the rewriting work is almost finished. Engineers are currently running the last step of the V-Cycle, the system verification and validation.

## 7 How Tags Helped

To perform processes described in Section 4 and 5, we implemented a solution that we called the *tags*. We implemented the *tags* as a way to:

- Materialize the comprehension of the system that software engineers gained in a reverse engineering activity (Section 4);
- Navigate a logical view of the system;

– Visualize and analyze the high level view created, and check conformance of the code to this view;
– Compare competing views of the same system.

In Reverse engineering phase (Section 4), engineers used the *tags* to materialize the concepts of modules, active packages, passive packages, modules and the concept of library (an example is depicted in Figure 5. The left tree in Figure

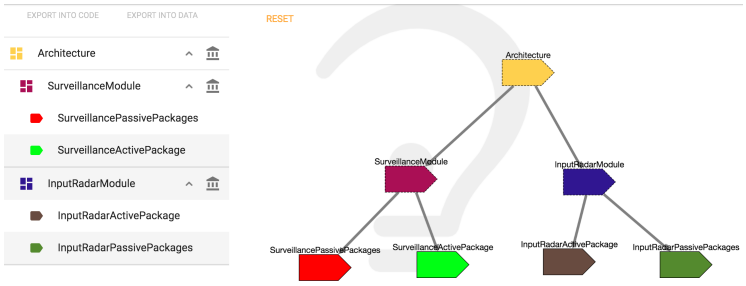


**Fig. 5.** Example of a concept materialization through *tags*

5 represents the concepts organization materialized by the engineers thanks to the *tags*. The right part of the figure is the visualization of this tree organization of *tags*. The top level entity is a *tag* named 'Architecture' and that holds the existing architecture. The middle level entities are *tags* that represent the Modules hidden in the source code and are contained in the 'Architecture' *tag*. The bottom level entities are *tags* that represent the Active Package and Passive Packages of each Module.

Engineers mapped the source code to the materialized concepts through the action of *tagging*. *Tagging* is not restraint to the packages and subprograms, indeed, even the instructions can be tagged if needed. The tagged instructions are considered when analysing and visualizing dependencies between *tags*, *Tagging* is an action that can be partly automated by acting on the code entities resulting from user-defined queries. Engineers can also *tag* or *untag* entities one by one if needed.

Contrary to Intentional Views (Section 3.2), *tags* are integral parts of the model, thus allowing simpler and more natural manipulations of the *tags*. Therefore, engineers can manipulate, navigate, visualize and analyze the *tags*. Figure 6 shows an example of an architecture comparison and a dependency analysis between *tags*. Dependencies between *tags* are derived from the dependencies between the code entities that are mapped to the *tags*. In Figure 6, the left part corresponds to the representation of the physical organization of the software system. The right part corresponds to the existing architecture of the system and thus to the *tags*. These views highlights cross-references between the point of view of the two architectures of the system. They help ensuring the same per-

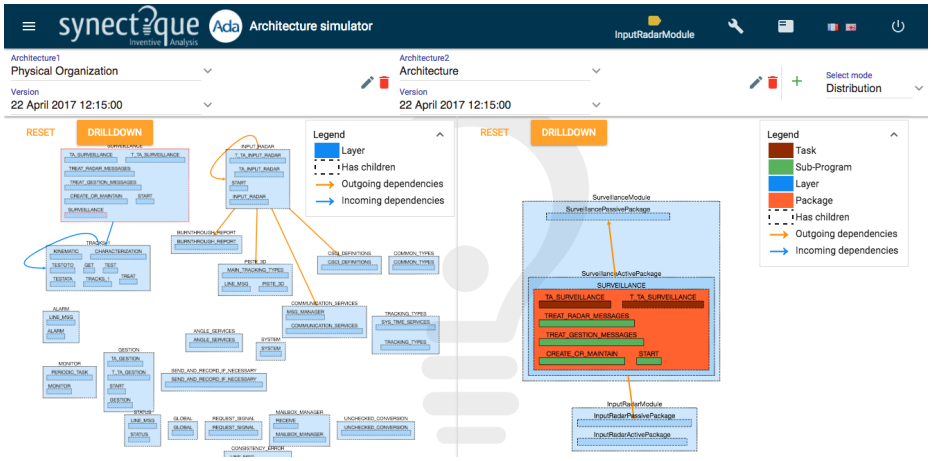


**Fig. 6.** Example of Architecture Comparison and Dependency Analysis

formance and results, by giving to the engineers, a tracability from the legacy architecture to the new architecture, through the source code. Thus, competing views in tags eased tests results comparison between two architecture, as explained in Section 2.2. Figure 6 includes a "Drill Down" button that allows to inspect the content of all the entities represented on the view. This figure depicts a comparison between two architectures, materialized by *tags* but a "simple" mode (top-right part of the figure) also exists and allows to visualize and analyze the dependencies of a single set of *tags*. Engineers used this tool when they needed to analyze the impact of their changes before recoding the entire application (thus before stage in Section 6)

Finally, *tags* analysis is enhanced with a set of metrics for the *tags* (Figure 7). These metrics are derived from the mapping with the code entities and the similar metrics of the code entities. For example, in Figure 7, *SurveillanceModule* has 8 packages that combine 548 lines of codes in total. Metrics were used when engineers reported their work progress during the project. Metrics about number of lines are also important for managers to compare next similar projects.

## 8 Conclusion

A large company called us to follow and help in its legacy software rearchitecting project. The project was split into three stages: reverse engineering, re-engineering and forward engineering. During the two first stages, engineers needed to identify concepts in the source code, to materialize and to analyze these concepts. Most of the existing solutions to the problem of concept identification did not fulfil the requirements the company had for its project. One of the existing solutions, Intentional Views, was a good candidate however, they seemed

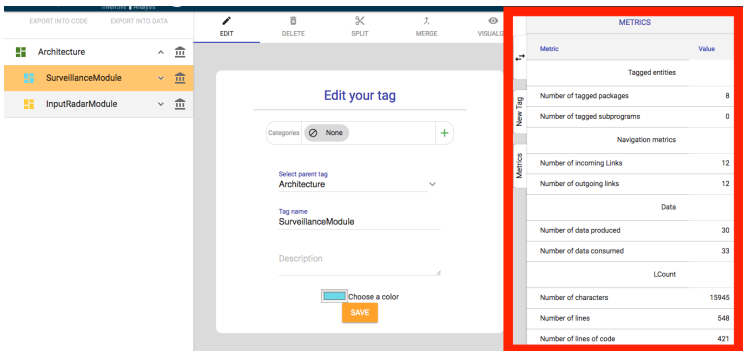


**Fig. 7.** Example of Metric for a Tag (Here SurveillanceModule)

to have been conceived as a tool to monitor evolution of a system. As such, they are primarily defined in intention (thus the name). On the contrary, we were working on a single version of the source code, trying to regain knowledge of it, and preparing for its radical transformation.

We proposed a solution based on *Tags* that allows to materialize and manipulate easily concepts hidden in a source code. We implemented our solution in an environment that helps engineers creating, mapping, manipulating and analyzing the *Tags*. We present how engineers successfully used our solution to perform both the reverse engineering stage and the re-engineering stage of the company's project. The forward engineering stage is in its final phase where the rearchitected software system is being verified and validated. Although we did not address the forward engineering stage, this last stage was steered by the mapping between the code entities and the *Tags* done during the two first stages.

Nonetheless, our solution, in its current state, needs better ways to make the tagging easier for engineers. One of the most important concern during the company's project was to lighten the tagging for engineers. Indeed, engineers had no other hints to *tag* than their expertise about the legacy software system. Therefore, reusing our solution on other projects requires to define more efficient ways to *tag*. We are currently working on this issue and we are already proposing new solutions[2].

## References

1. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer Verlag, 1999.
2. Brice Govin, Nicolas Anquetil, Anne Etien, Arnaud Monegier Du Sorbier, and Stéphane Ducasse. Combinable operators to ease concept identification in legacy software. In Submission.

3. R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models: Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998. ISBN: 0-8186-89-67-6.
4. Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of SEKE 2002*, pages 289–296. ACM Press, 2002.
5. Tom Mens and Michele Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
6. Gail Murphy, David Notkin, and Kevin 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.
7. Meghan Revelle and Denys Poshyvanyk. An exploratory study on assessing feature location techniques. In *Program Comprehension, 2009. ICPC'09. IEEE 17th International Conference on*, pages 218–222. IEEE, 2009.
8. Michael Siff and Thomas Reps. Identifying modules via concept analysis. *Transactions on Software Engineering*, 25(6):749–768, November 1999.
9. Muhammad U.Bhatti, Nicolas Anquetil, Marianne Huchard, and Stéphane Ducasse. A catalog of patterns for concept lattice interpretation in software reengineering. In *Proceedings of the 24th International Conference on Software Engineering & Knowledge Engineering (SEKE 2012)*, pages 118–24, 2012.