Automatic Generation of Abstract Views for Legacy Software Comprehension

Philippe Dugerdil, Julien Repond
Dept. of Information Systems
HEG-Univ. of Applied Sciences of Western Switzerland
7 route de Drize, CH-1227 Geneva, Switzerland
+41 22 388 17 00
philippe.dugerdil@hesge.ch, julien.repond@hesge.ch

ABSTRACT

One of the main motivations for the reverse engineering of software programs is to help with software comprehension. Although several techniques have been presented in the literature to reverse-architect software, the corresponding views usually do not help much. In fact, most of the published techniques recover the architecture of the software by focusing on the abstract properties of the components such as coupling and coherence. We claim that the recovered components should rather represent abstract functional entities whose behavior could be understood independently from the others. Then, an abstract view of the system would represent the interactions between such functional entities. In this paper we present a technique and a tool able to generate abstract sequence diagrams to represent the global working of legacy programs. This shows the main interactions between abstract functional components. When comparing the automatically generated sequence diagrams to the one a developer would produce by hand, we realized that the representation were very close. Our work could then be considered as a first step to the automatic generation of human-understandable abstract views of the working of legacy programs.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement - *Restructuring, reverse engineering, and reengineering.*

General Terms: Design, Experimentation, Algorithm.

Keywords

Reverse-engineering, functional component, software clustering, dynamic analysis, abstract view, sequence diagram.

1. INTRODUCTION

For the last 15 years, legacy software reverse-engineering has become an important field of computer science and an active field of research. It is a known fact that software maintenance is the most expensive activity over the software life cycle [20]. In these costs, the task of software understanding takes the lion's share.

Nowadays a tremendous research activity is spent on techniques to help with software understanding, often targeted at generating views of software system architecture [10]. To give a precise meaning to the word "understanding" in the context of reverse-engineering, we borrow the definition by Biggerstaff et al. [8]: "A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program". Moreover, it is known for long that to "understand" a large software system, which is a critical task in reverse-engineering, it structural aspects (i.e. its architecture) are more important than any single algorithmic component [29]. But the problem with legacy software is that, often, the only reliable source of information is its mere source code which hardly contains clues on higher level components [7]. Therefore it is hard to create representations of the architecture of the legacy software from its mere source code. Besides, we know that software architecture can be represented through many views [10], each targeting a particular purpose.

In this work, we target the construction of an architectural view that could help with understanding the code. We have called it the *functional architecture view* of the system, i.e. the structure of *functional components* and their relationships which implement the high level business function of the software. In this context, a *functional component* is a cohesive set of classes implementing some relevant processing step during the execution of a given scenario. To identify the functional components of a legacy software, we compute clusters of dynamically-correlated classes from the post mortem execution trace of the system. Class clustering has been around in software reverse-engineering for quite some time. Basically, it could be based on static (i.e. source code) or dynamic (i.e. execution trace) analysis techniques or both. For example, Tzerpos and Holt [32] use the identification of patterns of structure in the source code to cluster classes. The latter approach rests on the hypothesis that comprehension is enhanced when familiar structures are discovered in the code. We follow the same initial goal: the identification of clusters that would help with software comprehension. However we are not interested by the "discovery" of structural patterns in the code but by functional clusters of strongly interacting classes. Then we developed a tool to display the functional components on top of the actual structure of the code. Often, these components will span several packages. This is the first step in our reverse-engineering technique. Since we target software understanding, we must show how the functional components (i.e. dynamic clusters) interact to implement the business scenarios (instances of the use-cases) of the system. Then, we developed a tool to automatically generate the sequence diagram of cluster interactions, to show the role they played in the actual working of the legacy system. The idea of generating a sequence diagram from the execution trace of a system is not new. However in any industrial size system, the execution trace usually contains on the order 10^5 to 10^7 events (method calls). Therefore the raw presentation of the corresponding sequence diagram is unusable. A solution would be to work on sophisticated sequence diagram viewers equipped with zooming and navigation features to limit the quantity of information to display at any single time. This is the approach taken by Bennett et al. [6]. But this approach does not help much with the task of separating out important information from the "noise". In our approach, the information to be displayed is limited in principle by the very clustering technique: all interactions internal to functional components are not represented. As a result we obtain a very readable sequence diagram that shows the key interaction relevant to the execution of the scenario. This representation seems to be quite close to what the developer would actually have produced by hand while explaining his system.

The structure of the paper is the following. Paragraph 2 presents the concept of functional component and functional architecture view and its link to the static structure of the code. Paragraph 3 gives a summary of the clustering technique that we implemented. Paragraph 4 discusses execution trace compression techniques and the role played by the clusters in reducing the size of the trace. Paragraph 5 presents the cluster visualization technique (the functional architecture view) and the corresponding sequence diagram. Paragraph 6 presents a case study and Paragraph 7 concludes the paper. Paragraph 8 presents the related work.

2. FUNCTIONAL ARCHITECTURE

The term "Architecture" is somewhat fuzzy since it has many interpretations depending on authors. In software, it has long been acknowledged that architecture must be expressed through many views [10]. But in software understanding, we are not so much interested in recovering an hypothetical design-time architecture of the system. In fact, we should produce a view that helps with software comprehension. Here, software comprehension means being able to understand how the behavior of the system at the business level results from the behavior of its components. It has been known for long that a necessary feature for the behavior of a system to be understandable is for the system to be near-decomposable hierarchically [28]. Basically this means that we should be able to decompose the system in hierarchical subsystems whose *behavior* should be understandable separately from the others. Therefore, the goal for us is to find a way to decompose a system such that the near-decomposability property is exhibited. Any software developer having some experience in debugging legacy software knows that, generally, the package or module structure of the code does not exhibit the near-decomposability property: the understanding of a given package usually requires the understanding of some others. This is not surprising because:

- The architecture of the system has not necessarily been design with the goal to help understanding. In particular it could have been design to satisfy any one of the required quality attribute [4].
- The maintenances on the software system are likely to have spoiled its original architecture over time [5].

Therefore, if the *behavior* of the system is the criterion by which to decompose it:

- The resulting set of functional components will usually span several packages (or any other static grouping of classes or elements).
- Two functional components may share some of their classes.
- The name and the graphical icon used to represent the functional components must be different from the well known syntactical element found in the source code (package, module, unit, component, subsystem, …). This is why we use the term functional component or dynamic *cluster* and designed a specific UML profile for it.

In a sense, a functional component can be seen as a grouping of classes' responsibilities involved in some step of the processing associated to a scenario. Therefore, if the responsibilities of a class are used in several steps of a given scenario we may well end up with two clusters (functional component) sharing the same class. This is again why packages and dynamic cluster (functional components) are not the same concept. This is illustrated in figure 1. The grayed zone represents a functional component (cluster) grouping the responsibilities involved at a given moment in time while executing a scenario.

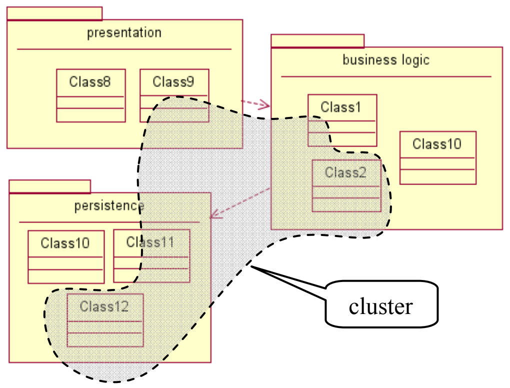


Figure 1: Packages, classes and clusters

Since we are interested by the behavior of the system, we should analyze its working while executing some scenario. But the latter must not be arbitrary. Rather it should represent a real business scenarios i.e. an instance of a real use-case. Although the use-cases of legacy systems are seldom documented, we could rely on the actual users of these systems to recover them. In fact the users are well aware of the business context and business relevance of the tasks performed with the software. Then the users are interviewed to recover the use-cases of the system. In one of our experiments, we video recorded the users while they were performing their job with the system. Then we analyzed the video and designed the use-cases that represented these manipulations. Finally we presented the use-cases to the users for validation and correction.

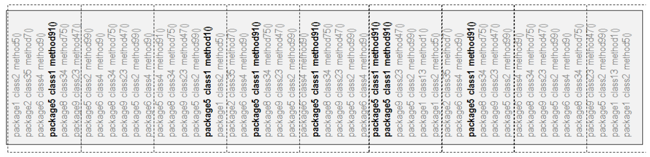
Once we have recovered the use-cases, we must record the working of the system i.e. its execution trace (sequence of method invocations). This is obtained by instrumenting the source code of the legacy system. The inserted statements in the source code will generate information about the executed function in some standard format that can be analyzed later to identify clusters and represent them on different architectural views. The instrumentation is performed using our own tool developed using JavaCC. The steps are the following. First, the legacy source code is parsed to get its syntax tree. Next, the latter is explored to attach the relevant instrumentation code. The resulting code is recompiled and executed following the chosen scenario.

3. SOFTWARE CLUSTERING TECHNIQUE

Many techniques have been proposed in the literature to recover the architecture of legacy software by computing clusters of classes i.e.

sets of classes that are tightly coupled [21]. While most of the published work rest on the static analysis of the code, we focus on dynamic techniques i.e. on execution trace analysis and trace segmentation. Some researchers have tried to analyze the execution trace to identify regular collaboration patterns among classes [16]. However implementation details, in particular the use of polymorphism, could lead to many collaboration variants that should nonetheless be considered the same as far as collaboration is concerned. Moreover, another problem is to decide on the maximum "distance" between two classes in the call sequence to still consider them as close collaborators. In summary if one takes into account all the variants of the interaction patterns and the problem of the distance between the classes to build the clustering algorithm, we end up with a computationally intensive algorithm, similar to finding a subgraph of a graph which is known to be NP complete. Moreover it is important to realize that execution traces of all but trivial programs are generally very large (on the order of 10^5 ~10^7 events). For example, in one of our experiments, the legacy system generated an execution trace with more than 7 millions of events (method calls). A true simplification of the problem arises if we make the hypothesis that the classes which collaborate heavily occur closely in the execution trace. Therefore, an easy way to check for collaborating classes is to split the execution trace in contiguous segments and observe the class presence in each segment (figure 2). In this figure, each vertical string represents a method call. The result of the analysis of the occurrences of each class is presented as a binary occurrence vector for the class. If a class occurs at least once in a segment, the corresponding element in the occurrence vector will be 1, 0 otherwise. As an example the presence of Class1 has been highlighted in the figure, and its corresponding occurrence vector V_{c1} is presented below.

The classes will then be clusterized according to the distance between their occurrence vectors. This is the principle of the technique we proposed to identify the collaborating application classes and that has been presented elsewhere [11][12]. Since we focus on application classes i.e. the classes that the developers of the system wrote themselves, the instrumentation will only target these classes. Therefore the resulting execution trace does not contain calls to the system classes or the classes pertaining to some framework such as Hibernate [18] or Spring [19]. This greatly reduces the number of events to process. To refer to a common terminology in software clustering techniques [3], the *entities* to cluster are the classes and the single *formal feature* used to group classes is the binary occurrence of the classes in each segments. Therefore we compute for each class the feature vector which represents the presence (1) of absence (0) of the class in each segment. This feature, that represents a *sibling link* between the classes, is a kind of shortcut to actually represent a *direct link* [3].



V_{c1} = (1 0 1 1 1 1 0 0 0)

Figure 2: Trace segmentation and feature vector

In other words, we use a measure of similar behavior (occurrence in trace segments) as a proxy for the collaboration between classes when executing some business function. Since the feature is asymmetric (the common absence from a segment is not informative), we use the Jaccard association coefficient to compute the "similarity" (or distance) between classes, which is known to provide the best results [21]:

$$\frac{a}{a+b+c}$$

In this formula, 'a' represents the number of segment in which the two classes simultaneously occur, 'b' the number of segment in which the first class only occurs and 'c' the number of segment in which the second class only occurs. The clustering algorithm we use is non-hierarchical and non agglomerative: we compute the clusters of mutually strongly similar classes. In other words, each pair of classes in a cluster represents closely collaborating classes. To compute the clusters we rely on graph theoretic concepts. Let G(N,E) a graph where the set of nodes N is the set of classes and the set of edges E represents the Jaccard similarity measure between the adjacent nodes. Let G' be the partial subgraph of G in which we only keep the edges where the Jaccard measure is larger than a given threshold T. Then the clusters represent the maximal cliques of G'. The detailed technique has been presented in [13]. Although the maximal clique algorithm is also computationally intensive, the set of classes over which it must be computed is limited: if the threshold T big enough (usually 80%, which means that two related classes occur simultaneously in at least 80% of the segments), then G' takes the shape of a set of small disconnected components. The computation is then performed separately on each of the graph components. This algorithm has the important property that the resulting set of clusters is unique for a given value of T. On the other hand it can produce overlapping clusters as shown in figure 3 where the grayed node is a class belonging to two clusters.

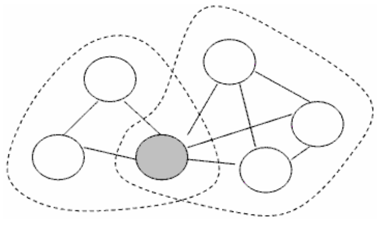


Figure 3: Overlapping clusters

The overlap between clusters comes from our very definition of what constitutes a cluster: a set of mutually strongly interacting classes (functional component). Therefore, it is not enough for a class to have a strong interaction with a single class of a cluster to be associated with this cluster. It must have the same strong interaction with all the classes of the cluster.

4. COMPRESSION & CLUSTERING

One of the ways to represent the working of a software system is to use the UML2's sequence diagram. However the direct display of an execution trace is not an option since the execution trace can contain millions of events. The size of such sequence diagram would be several kilometers! Therefore one must find a way to "compress" the trace to reduce its size, without impacting its usefulness for software comprehension. In the literature, many trace compression techniques have been proposed. An approach used by Hamou-Lhadj and Lethbridge [15] is to remove the calls to classes that represent utilities i.e. convenience classes designed for implementation purpose. The authors advocate that because implementation details are not important for the understanding of the system, utilities could be removed from the execution trace without reducing the comprehensibility of the code. Although the

argument based on implementation details is sensible, it remains to be demonstrated that the classes identified using the utilityhood metrics [15] do identify implementation-only classes and do not contain functional code. In fact, their algorithm is based on the fan-in metrics i.e. the number of static links referencing a given class. Then a ratio is computed between the fan-in and the total number of classes in the system (or component, depending on the scope of the search). All classes whose utilityhood value is bigger than a predefined threshold are considered utilities. However, nothing prevents a class that contains a utility method to contain also some functional code. Besides, the compression ratio using these techniques is not enough to get a resulting execution trace of a displayable size. In a case study, Hamou-Lhadj and Lethbridge removed utility classes calls as well as other implementation details such as methods of inner classes, accessors, constructors/finalizers and the like [15]. The compression ratio was about 30 i.e. there were 30 times less events in the compressed trace that in the original trace. Such compression factor would compress a trace of 6.10^5 events (that we got in the first case study) to about 20.10^3 events. This is still much too big to be displayed as a sequence diagram. It is important to note that the compression technique based on utilities produces some noise reduction but does not address class clustering. Other published compression techniques include the pattern-based trace sequence summarization [16]. There, recurring patterns of calls are identified and written only once in the resulting trace. Later occurrences of the same pattern refer to the first occurrence with the number of repetitions. To identify repetitions, Hamou-Lhadj and Lethbridge applied the common subexpression algorithm borrowed from DNA sequencing techniques [16]. The problem with this approach is that the resulting trace format is difficult to represent as a sequence diagram since there are references and loopback in the compressed format. Moreover, in their case studies Hamou-Lhadj and Lethbridge found compression factor between 11 and 40, which is still insufficient if the target is to display a trace of a million events.
As far as clustering is concerned, the algorithms can be divided in two broad categories: partitional and hierarchical [21]. In architecture recovery, most of the techniques pertain to the hierarchical category because partitional algorithms must start with an initial set of clusters that is supposed to be known in advance, which is usually not the case in architecture recovery. On the other hand, hierarchical algorithm start with the basic elements (for example the classes) and, using some distance metric, try to agglomerate them in clusters. Then, at each step of the algorithm, a distance between each unclustered element and the clusters is computed and the element is associated with the closest cluster. Therefore, noise reduction is important in the agglomerative hierarchical clustering since, depending on the distance metrics used, otherwise unrelated classes may be clustered together, especially if the SLA (Single Linkage Algorithm [21]) clustering algorithm is used. For example, if utility class A would be strongly linked to class B and C, even if B and C are unrelated, the three classes would be part of the same cluster because A would "glue" B and C.
Our approach based on graph-theoretic concepts does not need a preprocessing step of noise reduction. Since the goal is to help with the understanding of legacy code, which is supposed to be badly structured, we do not want to make the hypothesis that some class is an implementation-only class and does not contains functional code. Therefore all classes in the system are considered when computing class clusters. But the very algorithm of maximal clique requires all classes to be strongly linked. Then, even if utility class A would be strongly linked to B and C, these two classes would never be part of the same cluster if they are not closely collaborating. Once the clusters are computed, we display their interactions in a sequence diagram. Since the clustering is based on heavy class interactions, i.e. lots of events, there will be much less events to display if we remove the inner working of the clusters. Figure 4 shows the principle of trace reduction based on clusters. Since (A,B) and (C,D) form two clusters, we only display the interactions between the classes that are member of different clusters. In this case we would then have two interacting functional components.

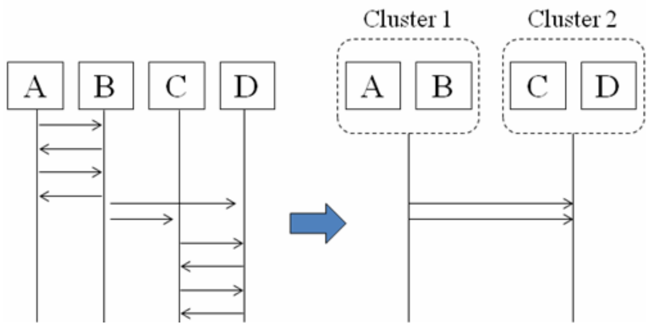


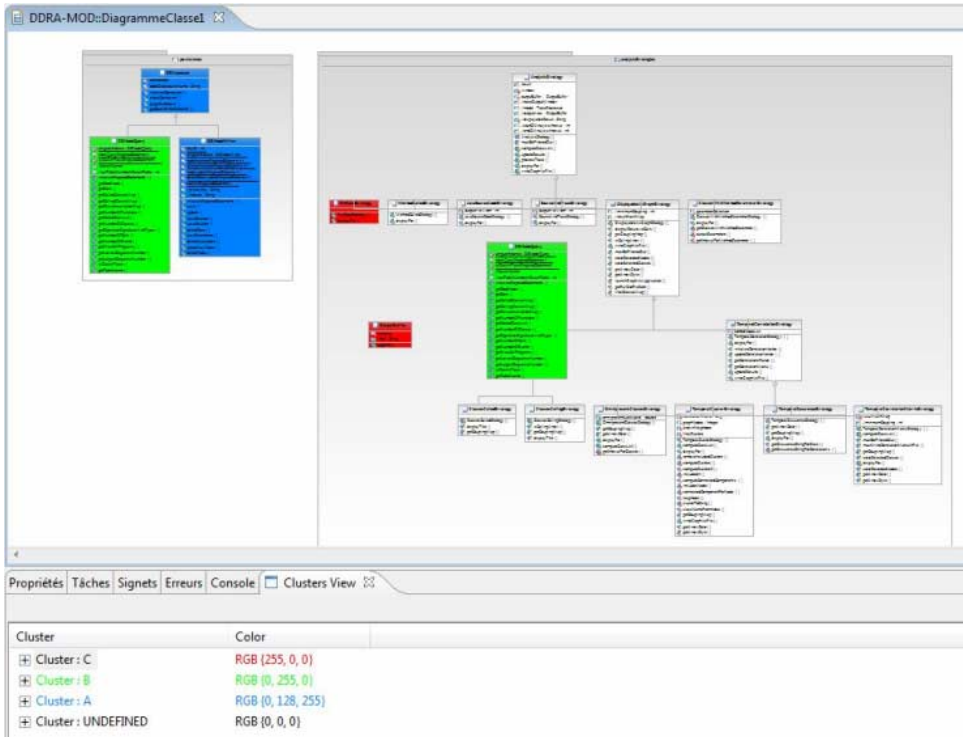
Figure 4: Cluster sequence diagram

5. FUNCTIONAL ARCHITECTURE VIEW

The best support for legacy software understanding is to generate relevant views of the system's architecture. Then, we designed a new architectural view that we called the *Functional Architecture View* of a system with two representations:

1. The functional components (clusters) displayed on top of the package structure of a program.;
2. The interaction of the functional components (clusters) in a sequence diagram.

This new view has been integrated in the IBM's Rational Software Architect (RSA) environment as an Eclipse plugin. The latter computes the clusters and displays the two representations using the tools available in RSA. With these views, a maintenance engineer could highlight the classes of the legacy system that work heavily together to implement some business function and observe their distribution among the packages. Then he could display the way they interact as functional components. The Functional Architecture view was first tested by hand and its relevance assessed with respect to the understanding of the code [12]. Then we decided to automate this technique using the plugin technology of Eclipse [25]. In figure 5 we present the functional components part of the view. In the top pane we see the architecture of the packages and the classes as reconstructed by hand using the Java to UML transformation feature of RSA. In the bottom pane, we see a new tab in called "Cluster View" which displays the list of clusters (functional components) identified in the execution trace. Then, a color can be assigned to each cluster and the classes will be colored according to the cluster they belong to in diagram on the top. With this representation, the maintenance engineer can observe the span of the functional components over the actual static structure of the code.
For a sequence diagram to be useful for system understanding, it must be of a limited size (a few dozen of events at most). Even if the clustering of classes drastically reduces the number of events to be considered, the remaining number of event in industrial size systems is generally still too big to be readable by humans.



Cluster	Color
Cluster : C	RGB (255, 0, 0)
Cluster : B	RGB (0, 255, 0)
Cluster : A	RGB (0, 128, 255)
Cluster : UNDEFINED	RGB (0, 0, 0)

Figure 5: Functional components

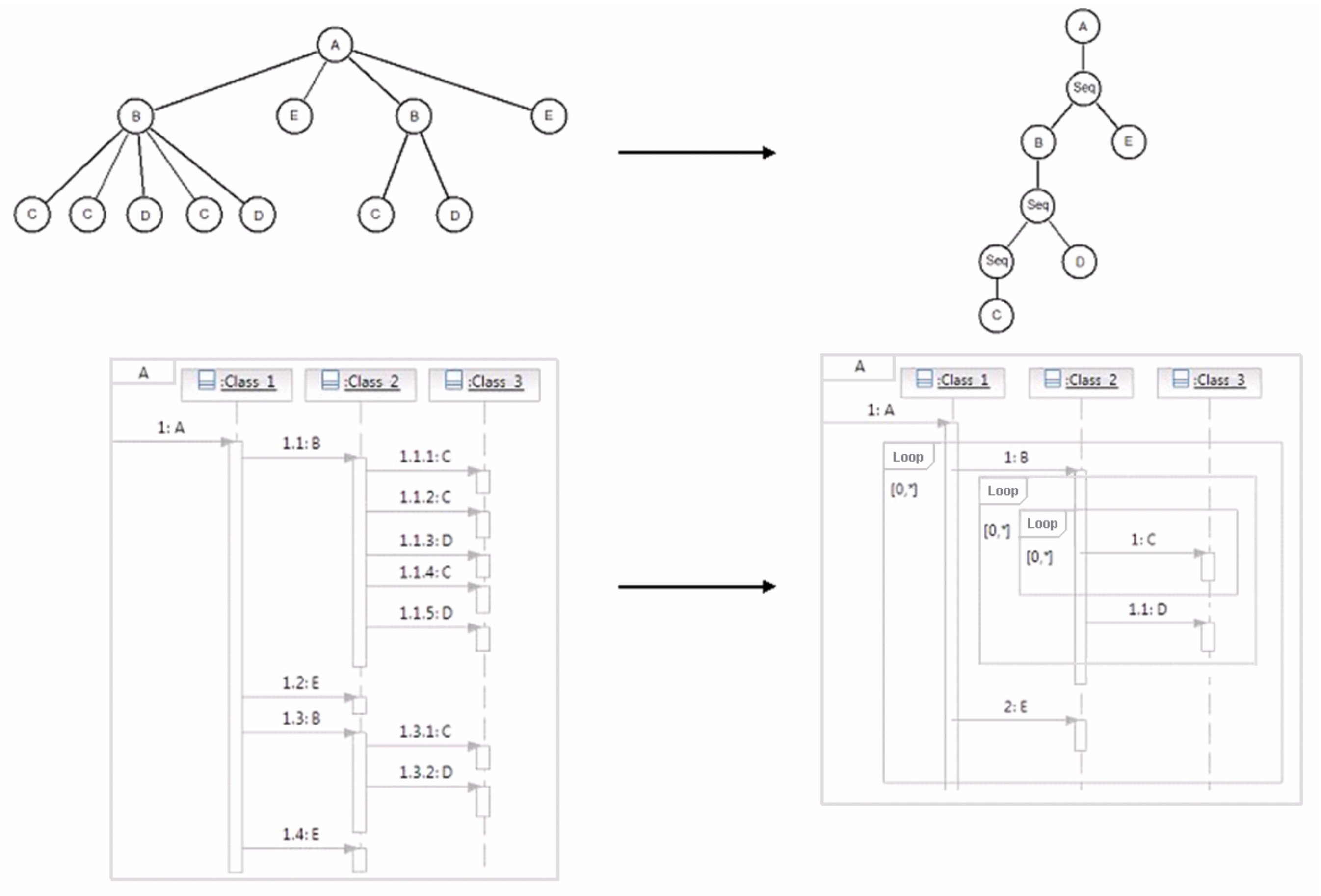


Figure 6: Trace repetitions compression and corresponding UML2 sequence diagrams

Therefore, we applied two extra event compression techniques:

- Removal of accessor methods;
- Compression of contiguous repetitions of similar patterns of events.

The algorithm used to detect event patterns repetitions is similar to what Hamou-Lhadj and Lethbridge proposed to produce the Compact Trace Format (CTF) [14]. This works bottom up (from the leaves to the root). At each level, the contiguous repetition of a pattern in the call tree of an execution trace is detected and replaced by a new SEQ node whose sons are the elements of the pattern. This indicates that the elements are repeated many times. Once a level of nodes is "compressed" we move up to the next level to find new repetitions and so on. This is shown in figure 6. In the left part we present the original call tree and the corresponding sequence diagram. On the right part we present the "compressed" version and the associated sequence diagram. For example, among the sons of the "B" node on the left we find a contiguous repetition of "C" nodes. This becomes a new SEQ node with "C" as its single son. Since a single occurrence of a node can be considered a unique repetition, we discover that the sons of "B" are now: ("sequence of C", "D"), ("sequence of C", "D"). We can then replace it with a new SEQ node for the pattern: ("sequence of C", "D"), as shown in the final call tree on the right. Again, since a single occurrence of a node is matchable against any number of repetitions of the same node, the same compression applies to the sons of "A". The translation to UML2 sequence diagram is easy: the SEQ node becomes a loop fragment and its sons become the sequence that is repeated. However, there is still a problem to solve. As explained in paragraph 3, our clustering algorithm may well generate overlapping clusters. This is when a subset of classes is strongly associated to two different clusters. But the two clusters cannot be merged because the other classes of both clusters are not coupled enough to each other. This is shown in figure 7. However nothing prevents a class outside the two clusters to call the methods of the classes that are located at the intersection of the clusters. This is shown in figure 8 where the class C0 calls a method in cluster A (C1) as well as a method at the intersection of both clusters (C3).

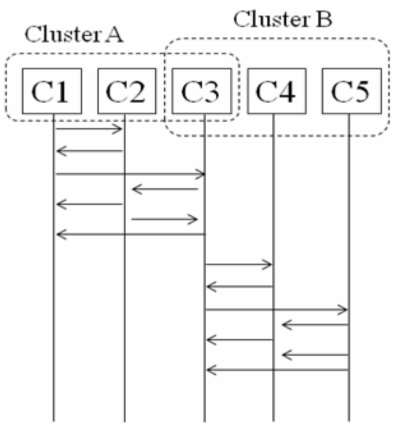


Figure 7: Overlaping clusters

But the latter creates a representation problem: how to display a method call to 2 different entities at the same time in a sequence diagram? To solve this problem we created the concept of "union-cluster" to represent an entity whose contents belong to several clusters (functional component). Therefore, in a sequence diagram, we would represent both the individual clusters and the union clusters. To visually identify the clusters and union clusters in UML2 diagrams, we designed two new stereotypes stored as a new profile. They are represented in figure 9 where the containment relationship is represented using the aggregation association.

Figure 10 presents a sequence diagram displaying the calls to the clusters A and B and to the union cluster between clusters A and B. However, there remains a problem to be solved. What to do with the classes that are not clusterized? Due to the fact that our clustering technique only groups closely interacting classes, there will remain application classes that are not member of any cluster [13]. However our goal is to identify and represent the functional components that are involved in the implementation of the main steps of a scenario. Therefore the interactions between the unclustered classes are not relevant at this granularity level. These classes are then grouped in a single container just to show when ones of these classes interacts with the functional components. This is represented by yet another cluster that we called "Undefined". We could see an example of this in the figures 12 and 13 of the case study.

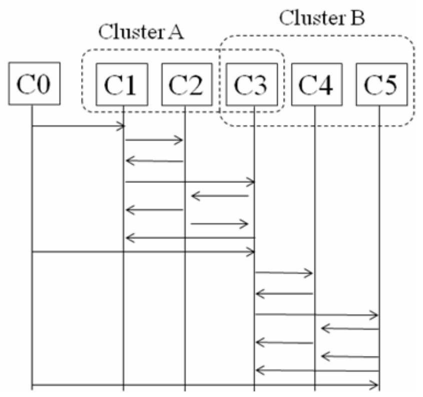


Figure 8: Calls to the class at the intersection of two clusters

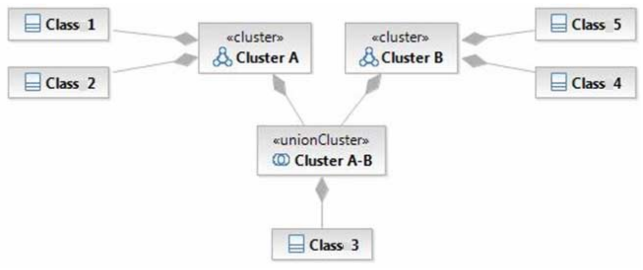


Figure 9: Cluster and "union cluster" stereotype

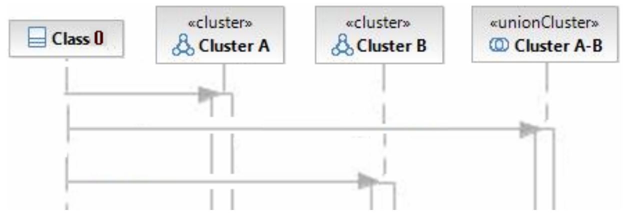


Figure 10: Sequence diagram with a "union cluster"

As for the Eclipse plugin, figure 11 presents its main screen in the IBM's Rational Software Architect environment. This works as follow. The first step of a reverse engineering experiment is to create a new UML modeling project. Then we generate the UML class diagram of the legacy Java system using the standard Java to UML transformation of RSA. The resulting model is then stored in the new project. In parallel, the source code of the legacy system is instrumented, recompiled, executed and the execution trace generated. The latter can now be loaded and analyzed using the trace analysis features of our plugin (whose screen is displayed in figure 11).

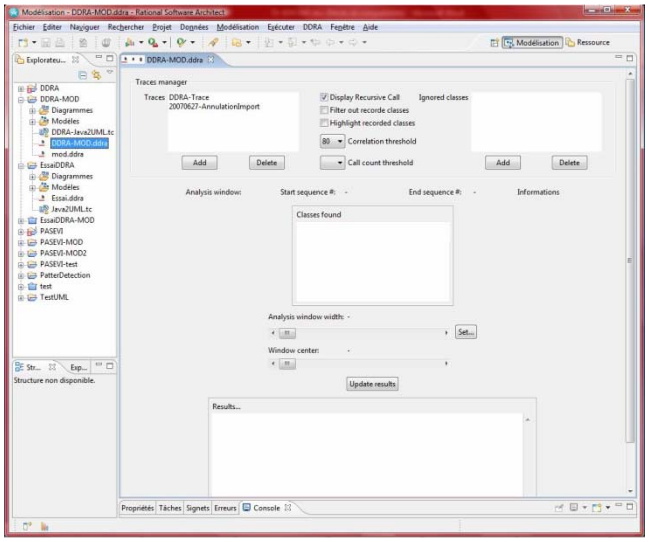

Modélisation - DDRA-MOD.ddra - Rational Software Architect
Fichier Editer Naviguer Rechercher Projet Données Modélisation Exécuter DDRA Fenêtre Aide
Traces manager
Traces DDRA-Trace 20070627-AnnulationImport
Display Recursive Call
Filter out recorde classes
Highlight recorded classes
Correlation threshold
Call count threshold
Ignored classes
Add
Delete
Analysis window:
Start sequence #:
End sequence #:
Informations
Classes found
Analysis window width:
Window center:
Set...
Update results
Results...
Propriétés Tâches Signets Erreurs Console

Figure 11: The plugin's GUI

Once this analysis is completed, the identified clusters are displayed in the project explorer together with the classes they contain as showed in figure 12. Finally we can generate a sequence diagram that would only take into account the discovered clusters as well as the cluster "Undefined" that gathers the remaining, unclustered, classes.

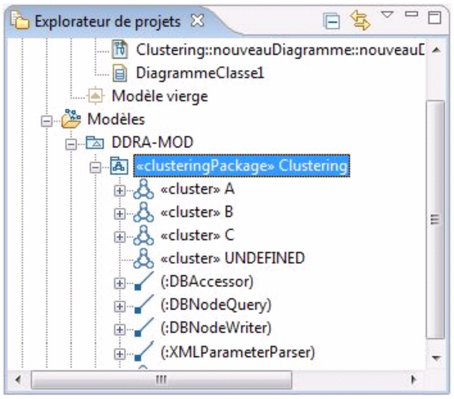


Figure 12: Project explorer with clusters

6. CASE STUDY

Our tool and technique have been applied to the trace analyser itself that we instrumented before running a standard cluster analysis scenario. The execution trace had 365'000 events that we segmented in 600 contiguous segments. The exact number of segment must be computed with respect to the number of classes in the execution trace as explained in [12]. A common factor is 32 (i.e. the number of segments must be 32 times the number of classes in the execution trace). In this example, the clustering algorithm found 3 clusters in the system.

Compression	Remaining events
Clustering	324
Clustering + Accessors removal	166
Clustering + Accessors removal + Repetitive sequences compression	45

Figure 13: Trace compression results

In figure 13, we present the compression factors of the trace using the three steps presented above: the clustering, the removal of accessors and the compression of repetitive sequences of events. Altogether the clustering of the trace and the extra compression steps took about 15 seconds to perform on a standard PC (3Mhz, 3Gb RAM). The clustering alone shows an impressive compression factor of 1000. If one adds the two other compression techniques one gets a compression factor of 8000: from the starting trace of 365'000 events one ends up with only 45 events which is quite manageable.
The question now is: are the remaining 45 events be of any help to understand the working of the system? As expected we found that the remaining event were quite informative and expressed clearly the main steps in the computation of the clusters.

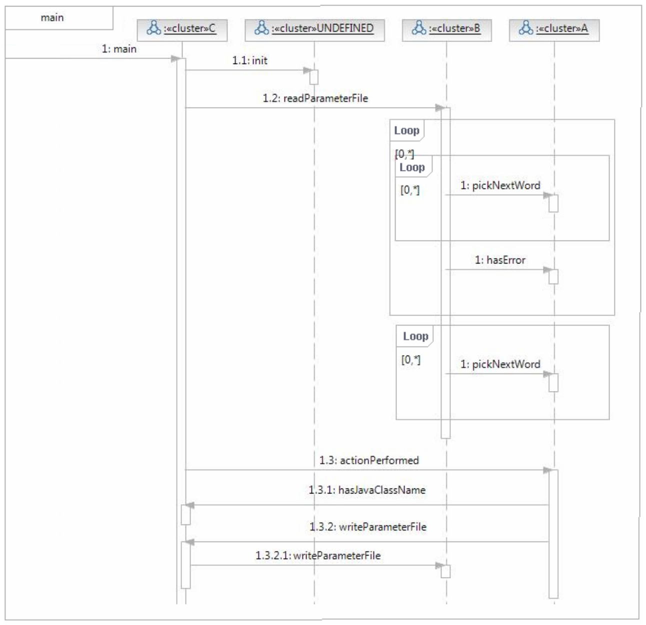


Figure 14: Sequence diagram for the initialization step

For example, figure 14 presents the part of the sequence diagram that represents the initialization of the work. First, all the processing parameters are read from a parameter file. As we can see, the main method of the parameter file reading step are represented in the diagram, especially the "readParameterFile" event and the loop to read the keywords in the file. The "writeParameterFile" event at the end represents the recording of the chosen parameters. Figure 15 presents the part of the sequence diagram associated with the core of the analysis process. The main task is launched by the "processTrace" event. Then the key subtask is to compute the occurrence vector for each class that is represented by the "getOccurenceVectorMap".

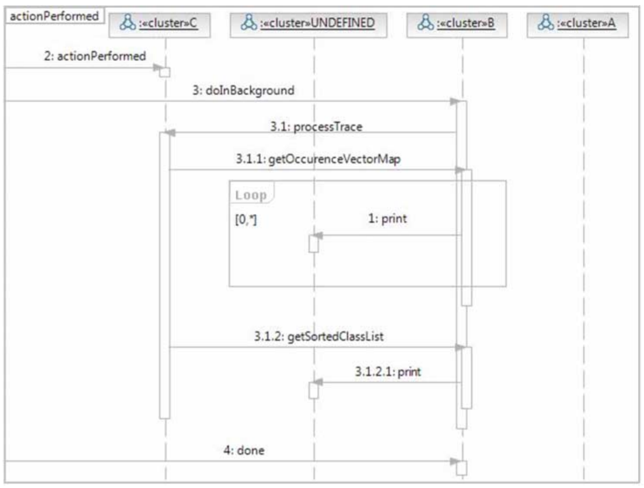


Figure 15: Sequence diagram for the analysis step

Finally the event "getSortedClassList" is used to display the results on the screen. In this example, our "abstraction" technique was able to retain and display the key events in the system. As a second test we applied the technique to an industrial system with 600 classes and whose execution trace had more than 600'000 events. The trace involved 169 classes and the clustering algorithm found 35 clusters. The compression technique ended up with about 200 events. This represents a compression factor of 3000. Although we could still display the resulting sequence diagram in RSA, we cannot show it in this paper since this would take about 8 pages. Again, the resulting event set showed the key working steps of the system.

7. CONCLUSION

In this paper we have showed that our software clustering technique, which is based on the dynamic analysis of the methods called when executing a scenario, allow us to automatically build an "abstract" view of the working of a system. In particular we are able to show that the main steps of a complex processing have been automatically revealed in a sequence diagram. Moreover, the generated sequence diagram seems to be close to the one we would actually draw by hand to explain the working of the system. Although this important feature needs be confirmed by more experiments it is very encouraging. This result has been obtained because of our very dynamic analysis technique coupled with the automatic generation of sequence diagrams. Here again are the main steps of our approach:

- An execution trace is generated when executing some scenario of business value (use-case). We only keep the events between application classes.
- The clusters that are identified are strongly linked to business functions. In a sense they represent chunks of elementary functionality;
- The working of the system is abstracted by displaying only the method called between the clusters.
- The resulting event set is further compressed using event patterns repetition compression.

Our cluster identification technique is to be contrasted with the ones that try to identify the components based on some software engineering metrics such as high cohesion and low coupling. As explained in paragraph 2, we are not targeting the recovery of any structural view of the software. Rather, we wish to generate a view that would support the understanding of software behavior. Therefore the conceptual grouping of classes as well as the analysis technique cannot rest on static analysis techniques. With the latter, there is no hope that the component will represent steps of some complex behavior related to the business function. The comparison of the dynamic clusters with the static structure of the program (for example its packages) is useful to show where the responsibilities pertaining to some behavior are actually implemented. But, as we already explained in paragraph 2, the criteria that lead to the static structure of the code are usually not based on the working of scenarios. Therefore, there little hope to end up with a set of functional components that would match the packages of the source code.

Our technique has been integrated in the IBM's RSA environment as an Eclipse plugin. In summary, the key contribution of this paper is to show that the automatic generation of an abstract view of a legacy system is indeed possible and we presented the way we achieved this. To refer back to the definition of program understanding by Biggerstaff at al. [8] we generated a view that would let an engineer understand the program behavior as interacting abstract functional components that are qualitatively different from the tokens of the source code i.e. the classes. Our approach therefore represents a step toward the true support of program understanding at higher level of abstraction.

So far, we only worked on single scenarios at the time (i.e. the main flow of a use-case). As future work we will develop multi-scenarios clustering and analysis techniques with alternatives. This may allow us to identify reusable functional components. Moreover we will integrate some zooming feature to be able to perform the same clustering analysis on the functional components themselves. Then we will develop a new view with the links between the different abstraction levels.

8. RELATED WORK

In the literature, many techniques have been proposed to recover the structure of a system by splitting it into components. They range from document indexing techniques [22], slicing [34] to the more recent "concept analysis" technique [27][30] or even mixed techniques [17]. All these techniques are static i.e. they try to partition the set of source code statements and program elements into subsets that will hopefully help to rebuild the architecture of the system. But the key problem is to choose the relevant set of criteria (or similarity metrics) [35] with which the "natural" boundaries of components can be found. In the reverse-engineering literature, the similarity metrics range from the interconnection strength of Rigi [24] to the sophisticated information-theory based measurement of Andritsos and Tzerpos [1][2], the information retrieval technique such as Latent Semantic Indexing [22] or the kind of variables accessed in formal concept analysis [31]. Then, based on such a similarity metric, an algorithm decides what element should be part of the same cluster [21]. In their work, Xiao and Tzerpos compared several clustering algorithms based on dynamic dependencies. In particular they focused on the clustering based on the global frequency of calls between classes [37]. This approach does not discriminate between situations where the calls happen in different locations in the trace. This is to be contrasted with our approach that analyzes where the calls happen in the trace. Very few authors have worked on sampling or segmentation techniques for trace analysis. One pioneering work is the one of Chan et al. [9] to visualize long sequence of low-level Java execution traces in the AVID system (including memory event and call stack events). But their approach is quite different from ours. It selectively picks information from the source (the call stack for example) to limit the quantity of information to process. The problem to process very large execution traces is now beginning to be dealt with in the literature. For example, Zaidman and Demeyer proposed to manage the volume of the trace by searching for common global frequency patterns [38]. In fact, they analyzed consecutive samples of the trace to identify recurring patterns of events having the same global frequencies. In other words they search locally for events with similar global frequency. This is then quite different from our approach that analyzes class distribution throughout the trace. Another technique is to restrict the set of classes to "trace" like in the work of Meyer and Wendehals [23]. In fact, their trace generator takes as input a list of classes, interfaces and methods that have to be monitored during the execution of the program under analysis. Similarly, the tool developed by Vasconcelos, Cepêda and Werner [33] allows the selection of the packages and classes to be monitored for trace collection. In this work, the trace is sliced by use-case scenarios and message depth level and it is then possible to study the trace per slice and depth level. Sartipi and Safyallah [26] use a patterns search and discovery tool to separate, in the trace, the patterns that correspond to common features from the ones that correspond to specific features.

8. REFERENCES

[1] Andritsos P., Tzerpos V. 2003. Software Clustering based on Information Loss Minimization. Proc. IEEE Working Conference on Reverse engineering. 2003

[2] Andritsos P., Tzerpos V. 2005. Information Theoretic Software Clustering. IEEE Trans. on Software Engineering 31(2). 2005

[3] Anquetil N., Lethbridge T.C. – Experiments with Clustering as a Software Remodularization Method. Proc IEEE WCRE, 1999.

[4] Bass L., Clements P., Kazman R. 2003. Software Architecture in Practice, 2nd edition. Adison-Wesley Inc.

[5] Belady L., Lehman M. 1976. A Model of Large Program Development. IBM Syst. Journal 15(3), pp. 225-252.

[6] Bennett C., Myers D., Storey M.-A., German D. 2007. Working with 'Monster' Traces: Building a Scalable, Usable Sequence Viewer. Proc. of the 3rd International Workshop on Program Comprehension through Dynamic Analysis.

[7] Bergey J., Smith D., Weiderman N., Woods S. 1999. Options Analysis for Reengineering (OAR): Issues and Conceptual Approach. Software Engineering Institute, Tech. Note CMU/SEI-99-TN-014, 1999.

[8] Biggerstaff T. J., Mitbander B.G., Webster D.E. 1994. Program Understanding and the Concept Assignment Problem. Communicaitons of the ACM, CACM 37(5), 1994.

[9] Chan A., Holmes R., Murphy G.C., Ying A.T.T. 2003. Scaling an Object-oriented System Execution Visualizer through Sampling. Proc. of the 11th IEEE International Workshop on Program Comprehension (ICPC'03).

[10] Clements P.et al. 2002. Documenting Software Architectures: Views and Beyond, Addison Wesley.

[11] Dugerdil Ph. 2007. Using trace sampling techniques to identify dynamic clusters of classes. Proc. of the IBM CAS Software and Systems Engineering Symposium (CASCON), 2007

[12] Dugerdil Ph., Jossi S. 2008. Empirical Assessment of Execution Trace Segmentation in Reverse-Engineering. Proc. ICSOFT 2008, Porto Portugal.

[13] Dugerdil Ph., Jossi S. 2009. Computing Dynamic Clusters. 2nd Indian / ACM Conference on Software Engineering (ISEC) 2009, Pune, India, February 23-26.

[14] Hamou-Lhadj A. Lethbridge T. 2004. A Metamodel for Dynamic Information Generated From Object Oriented Systems. Electronic Notes in Theoretical Computer Science 94, pp.59-69.

[15] Hamou-Lhadj A. Lethbridge T. 2006. Summarizing the Content of Large Traces to Facilitate the Understanding of the Behavior of a Software System. Proc. of the IEEE Int. Conference on Program Comprehension (ICPC'06).

[16] Hamou-Lhadj A., Lethbridge T.C 2002. Compression Techniques to Simplify the Analysis of Large Execution Traces. Proc. of the IEEE Workshop on Program Comprehension (IWPC).

[17] Harrman M., Gold N., Hierons R., Binkeley D. 2002. Code Extraction Algorithms which Unify Slicing and Concept Assignment. Proc IEEE Working Conference on Reverse Engineering (WCRE'02).

[18] www.hibernate.org

[19] www.springsource.org

[20] Koskinen J. – Software Maintenance Costs. University of Jyväskylä, Finland, http://users.jyu.fi/~koskinen/smcosts.htm

[21] Maqbool O., Babri H.A. 2007. Hierarchical Clustering for Software Architecture Recovery. IEEE Trans. On Software Engineering, Vol. 33, No. 11.

[22] Marcus A. 2004. Semantic Driven Program Analysis. Proc IEEE Int. Conference on Software Maintenance (ICSM'04).

[23] Meyer M., Wendehals L. 2005. Selective Tracing for Dynamic Analyses. Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05).

[24] Müller H.A., Orgun M.A., Tilley S., Uhl J.S. 1993. A Reverse Engineering Approach To Subsystem Structure Identification. Software Maintenance: Research and Practice 5(4), John Wiley & Sons. 1993

[25] Repond J. 2009. Génération de diagrammes UML à partir d'une analyse dynamique. Bachelor Thesis, HEG, Univ of Applied Sciences of Western Switzerland, Geneva.

[26] Sartipi K., Safyallah H. 2006. An Environment for Pattern based Dynamic Analysis of Software Systems. Proceedings of the 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA'06).

[27] Siff M., Reps T. 1999. Identifying Modules via Concept Analysis. IEEE Trans. On Software Engineering 25(6). 1999.

[28] Simon H.A. 1969. The architecture of complexity. In: The Sciences of the Artificial, MIT Press. (Reprinted in 1981)

[29] Tilley S.R., Santanu P., Smith D.B. 1996. Toward a Framework for Program Understanding. Proc. IEEE Int. Workshop on Program Comprehension.

[30] Tonella P. 2001. Concept Analysis for Module Restructuring. IEEE Trans. On Software Engineering, 27(4).

[31] Tonella P. 2003. Using a Concept Lattice of Decomposition Slices for Program Understanding and Impact Analysis. IEEE Trans. On Software Engineering. 29(6).

[32] Tzerpos V., Holt R.C. 2000. ACDC : An Algorithm for Comprehension-Driven Clustering. Proc. IEEE Working conference on reverse Engineering (WCRE).

[33] Vasconcelos A., Cepêda R., Werner C. 2005. An Approach to Program Comprehension through Reverse Engineering of Complementary Software Views. Proceedings of the 1st International Workshop on Program Comprehension through Dynamic Analysis (PCODA'05).

[34] Verbaere M. 2003. Program Slicing for Refactoring. MS Thesis, Oxford University.

[35] Wiggerts T.A. 1997. Using Clustering Algorithms in Legacy Systems Remodularization. Proc IEEE Working Conference on Reverse Engineering (WCRE '97).

[36] Wirfs-Brock R., McKean A. 2003. Object Design, Roles, Responsibilities and Collaborations. Addison-Wesley.

[37] Xiao C., Tzerpos, V. 2005. Software Clustering basd on Dynamic Dependencies. Proc. of the IEEE European Conference on Software Maintenance and Reengineering (CSMR'2005).

[38] Zaidman A., Demeyer S. 2004. Managing trace data volume through a heuristical clustering process based on event execution frequency. Proc. of the IEEE European Conference on Software Maintenance and Reengineering (CSMR'2004).