Software System Understanding via Architectural Views Extraction According to Multiple Viewpoints

Azadeh Razavizadeh[1], Sorana Cîmpan[1], Hervé Verjus[1], and Stéphane Ducasse[2]

[1] University of Savoie, LISTIC Lab, France
[2] INRIA Lille-Nord Europe, RMoD Team, France
{razavizadeh;verjus;cimpan}@univ-savoie.fr

Abstract. Changes and evolution of software systems constantly generate new challenges for the recovery of software systems architectures. A system's architecture, together with its elements and the way they interact, constitute valuable assets for understanding the system. We believe that offering multiple architectural views of a given system, using domain and pattern knowledge enhance understanding of the software system as a whole. To correlate different sources of information and existing software system, different viewpoints are considered. Viewpoints enable one to model such information and guide the extraction algorithms to extract multiple architectural views. We propose a recursive framework, an approach that expresses different kinds of information as viewpoints to guide the extraction process. These multiple viewpoints models improve the consideration of architectural, conceptual, and structural aspects of the system.

1 Introduction

Software systems need to *evolve* over time [15]. They get modified to improve their performance or change their functionality in response to new requirements, detected bugs, *etc.* Some changes are part of the system maintenance; others evolve the system, generally by adding new functionalities, modifying its architecture, *etc.* Thus, there are several evolution phases for which different processes may be employed. The evolution process ideally begins with the system comprehension and continues with finding a suitable set of system modifications. It has been measured that in maintenance and evolution phases, at least half of the engineers' time is spent on the system comprehension [7]. Thus, to successfully evolve a complex system, it is essential to *understand* it. The understanding phase is time and effort consuming, due to several reasons, among which: the system size (large systems consist of millions lines of code), lack of overall views of the system, its previous evolutions (not necessarily documented), *etc.* This motivates us on supporting the software system understanding phases.

Architectures serve thus as education means [1] and guide software evolution, by providing a high-level abstract model of existing software systems. Software

architecture reconstruction is a reverse engineering approach that aims at reconstructing viable architectural views of software applications [7].

We focus on *extracting architectural views* of existing software systems. It is widely accepted that multiple architectural views are useful when describing the software architecture [13][5][1]. Many approaches focused on providing high abstraction level views in order to facilitate program understanding [11][8] [19]. Architecture relevant information can be found at different granularity levels of given systems and needs to be studied from different viewpoints. A viewpoint is a collection of patterns and conventions for constructing one type of view. It reflects stakeholders concerns and guides the construction of views [10]. We consider different viewpoints according to such concerns: business-based, pattern-based, cohesion-based, activity-based, *etc.*

The main contributions of the proposal presented in this paper are a recursive framework for extracting architectural views and an enhanced definition or architectural viewpoints allowing their modularity and reuse.
Structure of the paper: In the next section, we propose a recursive framework and its extraction process. Section 3 presents architectural viewpoints composed of viewpoint model and viewpoint extraction algorithm. Section 4 presents the architectural view. In Section 5, we use an example to illustrate our approach. Section 6 presents related work and the paper closes in section 7 with a conclusion.

2 A Recursive Framework

We adopt a generic and recursive framework (Figure 1) for extracting architectural views: at each recursion of the framework, a specific view is extracted (also results) from another view, using a guiding viewpoint. Our framework is reproductible by chaining horizontal and/or vertical recursions of the framework.

- *horizontal* recursion: starting from a given architectural view (often the implementation view but other views can be considered) other architectural views can be elaborated, each based on a specific viewpoint;
- *vertical* recursion: architectural views are organized in a pipe-line style, such that the output (view) of an extraction recursion is used as an input (view) for another one.

Each horizontal or vertical recursion corresponds to an instance of the generic framework and thus extracts a new architectural view using a specific viewpoint.

An extraction process corresponds to the recursive framework application: an architectural view that is extracted from a view given as input of an extraction process is considered as the view generated by the framework. Several views may be extracted from a given view, using different viewpoints. The recursive framework application always begins with the generation of an *implementation view*. This recursion re-engineers (using the Moose [6] re-engineering environment) the software system from its source code and produces a Famix model as an *implementation view*: it consists in generating an (architectural) *implementation view* of the source code (*i.e.*, flat files containing the source code expressed

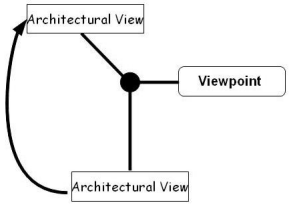


Fig. 1. The recursive framework principle

in a OO Programming Language). This *implementation view* is composed of architectural elements such as classes, methods, and packages. Using this view as input, architectural views can be generated by horizontal recursions. Further on, generated architectural views can be used as input for vertical cascading of the framework (Figure 3). Thus, for obtaining an architectural view, at least two recursions of the framework are needed. The following section will present architectural viewpoints.

3 Architectural Viewpoints

The IEEE standard definition of viewpoints (previously presented) is the fundamental concerns of stakeholders and the convention for constructing the related views [10]. Our framework formalizes this by separating the concerns reflected in the viewpoint in a viewpoint model (VptM) and, the convention to extract these concerns in viewpoint extraction algorithms (VptEA). By this separation, the definition of a viewpoint model makes abstraction of the extraction mechanisms (see 3.1). Moreover, this separation enables us to generalize as much as possible the convention for constructing the views and to reuse the knowledge on view construction in the form of reusable generic extraction algorithms (see 3.2). Starting from an implementation view of the software system, we provide several architectural views according to different viewpoints. We propose two viewpoint classes: *matching* and *discovering*. *Matching viewpoints* are used when stakeholders can represent their expectations as models representing the main concepts present in the system. The extraction algorithm constructs then the architectural view by "matching" the system's elements to the expected representation. This allows us to have generic extraction algorithms that are (re)used in the definition of several viewpoints. It is clear that *matching* viewpoints put their force on the modeling side. Examples of such viewpoints are *business domain*-based viewpoints which consider the principal business domain concepts and their relationships, and *software pattern*-based viewpoints [9][23] which identify architectural elements conform to a given pattern. *Discovering viewpoints* focus on the algorithm side of the viewpoint (more than the modeling side). The specificity of such viewpoints is entailed in the extraction algorithm, which specify how the elements of the input view are to be grouped leading to architectural

elements discovery. In this case, the model part of the viewpoint definition is less important. Examples of such viewpoints are the *activity*-based viewpoint which identifies the architectural elements according to their level of interaction with their environment, and the *cohesion*-based viewpoint [17][14] which identifies a set of related architectural elements according the strength of their dependencies. The framework can easily integrate other viewpoints.

3.1 Viewpoint models

For defining a viewpoint model, several sources of information can be used: rough understanding of the system's functionality, documentation skimming, demo interviewing, or even the available experts's opinions [4]. Viewpoint models are mainly represented using concepts and relationships among them. As such models represent an expected architecture representation, they are represented similarly to architectural views. Thus, a subset of the Architectural Meta Model presented in section 4 is used, concepts being represented as architectural elements. Each viewpoint reveals certain aspects of the software system. For example, the extracted architectural view of the business domain-based viewpoint presents the system architectural elements organization in accordance with the business domain concepts. Such a view mainly provides an overall view of the system in terms of the business concepts and helps different stakeholders in their system understanding. Let us consider a Banking software application example. The business domain concepts of such an application can be *Bank, Client, Account, Credit card*. The corresponding VptM of the business domain-based viewpoint comprises these concepts as architectural elements and their relationships (*i.e.*, *a Bank may have more than one Client; a Client may have more than one account; a Credit card concerns a specific Account, etc.*). This VptM supports and guides the architectural view extraction process.

3.2 Extraction Algorithms

Let us recall that we focus on extracting architectural views; and we start from an implementation view of the system. In the case of *matching* viewpoints, the extraction algorithm attempts to identify elements from the input view related to each viewpoint model concept. Then, it makes a group of these elements and links them to the architectural element of the viewpoint. The input view's architectural elements that do not correspond to any concept of the VptM are grouped in another group labeled *Outside domain*. This first extraction result allows one to identify concepts which are not reflected in the code. Moreover, further analysis of the *Outside domain*'s elements may lead to the discovery of new concepts. In the case of *discovery* viewpoints, the algorithms mainly focus on the relationships among architectural elements of the input view but differ in the metrics they consider (*i.e.* number of intra/inter invocations, number of methods, *etc*). The process can also be considered at a finer grain level if needed: the extraction algorithm can target different architectural elements (*i.e.*,

classes, methods, packages) of the implementation view. Thus, different architectural views can be generated accordingly. Moreover, at the implementation view level, the extraction algorithm can identify subsets of classes according to the VptM concepts. A subset of a class is a group of related attributes and methods and can be considered as *traits* [16]. As a consequence, an architectural element (*i.e.*, a class) can be seen as a set of (potentially and partially) overlapping traits. The same trait can be found in different architectural elements. In the Banking software application example, we use a business domain-based viewpoint and thus, the matching algorithm presented before. This algorithm searches all classes (considered as architectural elements) of the implementation view which contain the VptM's concepts in their name (*i.e., Bank, Account, Credit card, etc.*); when found, the classes are put in a group labeled with the corresponding concept and linked to the related architectural element. The classes that do not correspond to any concept of the VptM are grouped as *Outside domain*.
The *matching* extraction algorithm is also used in software pattern-based viewpoints; in this latter, the viewpoint model is a Software Design Pattern Model. For instance the MVC pattern model contains *Model, View, Controller* as concepts of the MVC's domain with their relationships. In this case, the extraction algorithm groups architectural elements according to MVC's concepts and their relationships.

4 Architectural Views

Our definition of a view is based on the IEEE standard. An architectural view is a way to present the system using the elements that are relevant to stakeholders concerns [10]. The multiple views of a system allow one to give support to understanding the system to different classes of stakeholders.

We propose a simple *Architectural Meta Model* for representing architectural views: a system architectural view is represented as a set of interconnected *architectural elements* (Figure 2). An architectural element is mapped to a group of system's elements (or a group of architectural elements) of the architectural view (of the framework) given as input.

The *Architectural Meta Model* deals also with relationships among architectural elements. When extracting an architectural view, using the framework, the relationships among architectural elements of an architectural view are deduced from relationships among architectural elements of the architectural view given as input to the framework. Thus, the extraction process takes into account both the identification of the architectural elements and their relationships.
Cascading Architectural Views: Our recursive approach enables one to use a previously extracted architectural view as an input for a new extraction process with a new (or even the same) architectural viewpoint. Therefore, each extracted architectural element (that is a group of elements of the architectural view given as input) may be also considered as an input of the recursive framework. Applying the framework recursively consists in defining and organizing architectural elements of the generated view according to the viewpoint model

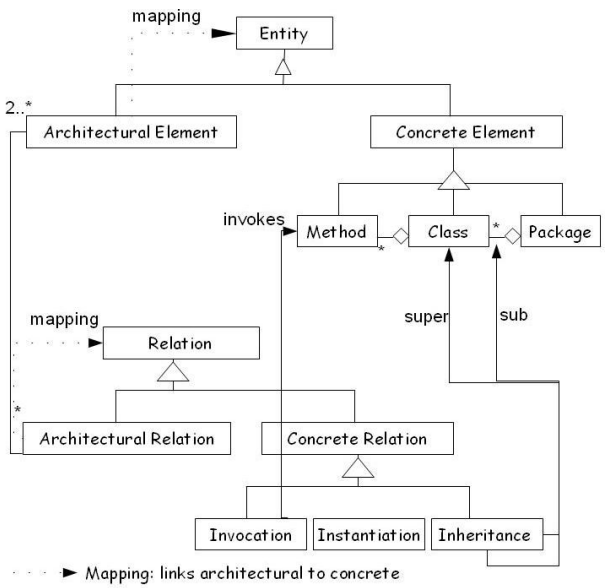
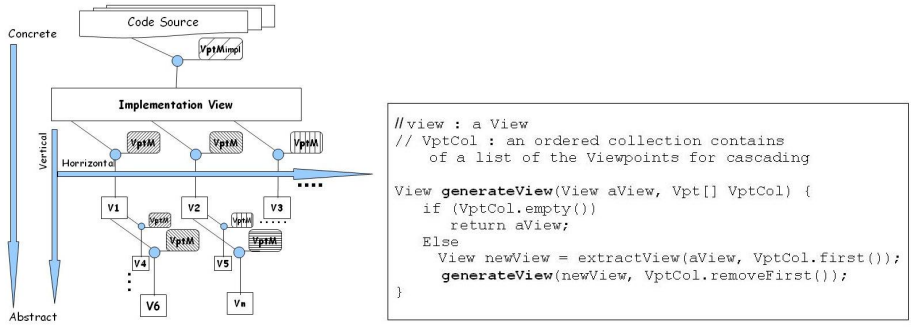


Fig. 2. Architectural Meta Model

and extraction algorithms. As consequence, each architectural element of this generated view may be refined according to a new viewpoint model, and so on. Considering again the Banking software application example, we are cascading extraction processes (*i.e.*, several framework recursions): starting from the source code of the software application, the first extraction process produces the *implementation view* which serves as the input view for another extraction process; this latter considers a business domain-based viewpoint and generates a business domain-based architectural view. This business domain-based architectural view is itself employed as input of a third extraction process using a MVC pattern viewpoint. This third extraction process identifies MVC viewpoint architectural elements (*i.e.*, *Model, View, Controller*) of each architectural element of the business domain-based architectural view (given as input) and generates new architectural views accordingly. For cascading views, one places the different viewpoints in an ordered collection and recursively applies the generate views algorithms (See Figure 3b). As a result of such architectural viewpoints cascade, we obtain an abstract architectural view for which the architectural elements reveal business domain concepts (second viewpoint used) and are composed of those of *Model*, *View* and *Controller* elements (third viewpoint used).

5 Illustration Example

Let us go back to the Banking software application and illustrate the application of the *matching* viewpoint extraction algorithm with the business VptM. The Banking software system source code entails 88 classes. The stakeholder begins first by defining a business viewpoint model that contains three concepts: *Account, Client, Card*. This proposed model is a very abstract and basic model.



a) b)

Fig. 3. a) Cascading multiple-viewpoint in a multiple-view perspective b)Extraction (recursive) algorithm

It reflects a non-expert user's intuitive system understanding. Starting from the *implementation view*, using this VptM (architectural concepts and relationships) and the *matching* extraction algorithm, an architectural view is generated. This view contains thus 3 architectural elements (3 concepts) plus the *Outside domain* architectural element (Figure 4b). Further investigations can be done focusing on the *Outside domain* group/architectural element: the classes associated to the group *Outside domain* can be deeply analyzed for similarities identification. Several classes may share a concept that might correspond to a concept in the domain, which was not formalized (or that has been forgotten) in the viewpoint model. This later can thus be updated. The extracted architectural view is equally updated in order to include the new architectural element. The remaining elements of the *Outside domain* group generally can be now subject to another extraction process using a given VptM. The extraction process entails, for example, "software design know-how" that is formalized in another viewpoint model. The new detected concepts presented in Figure 4c are thus extracted from the *Outside domain* architectural element. The results obtained include the detection of 7 new architectural concepts that are part of the business domain. Considering the number of classes (*implementation view*'s elements), detecting 10 concepts is a good sign. We should remark that the detection of concepts which are not parts of domain is a false positive. It entails for example "software design know-how" that is formalized in another viewpoint model. Figure 4a shows also the links established between the *implementation view* and the *business view*. These links facilitate maintaining the consistency between the abstract and the concrete representations of the system.

6 Related Work

This section addresses those works that deal with software architecture reconstruction. Various works are proposed in order to extract architectures of an

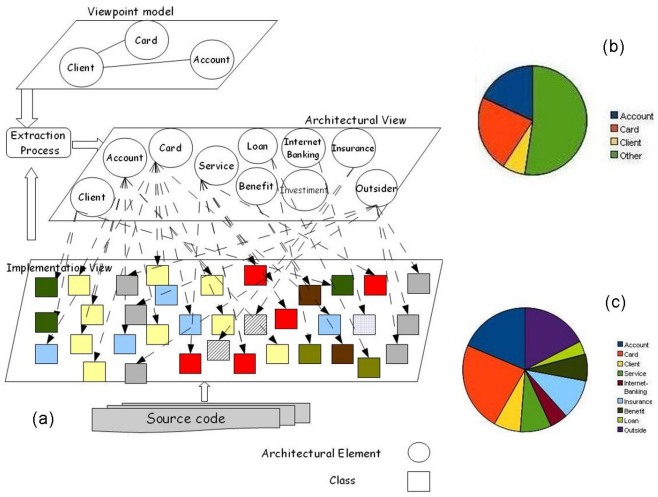


Fig. 4. a)Bank application view extraction; b)Extraction result with the initial VptM; c)Extraction result after exploration of the *Outside domain* architectural element

object-oriented systems. We distinguish these works according to two criteria: the extraction process input and the technique of this process.

The inputs used by extraction approaches are various. Most often the source code is used, but some researchers have highlighted the importance of considering alternative sources of information during the architecture extraction such as: developer knowledge [20][12]; bug reports and external documentation [2]; or considering an ontology of the software system's domain [3]. In our approach we propose to infer a viewpoint as input in order to guide the extraction from the source code of a system. The use of viewpoints to generate views is a key aspect of our approach. The viewpoint is not limited to be developer knowledge, bugs or documentation, thus it can be one or all of them. This viewpoint may be: a software pattern, a business model or even an interesting concept requested by user; therefore it is generic compared with other approaches. The main difference is the separation of two existing concepts of viewpoint definition according to IEEE standard: concerns (as a viewpoint models) and conventions (as an extraction algorithms). This separation increases the generic aspect of our approach.

The techniques used to reconstruct architecture of an existing system are various. Approaches like [18] and [21] consider external constraints (represented as queries) to be checked against the reality of source code or recovered architectural elements. [20],[12] and [22] propose an automatic reconstruction technique based on reflexion models, starting with a structural high-level model. In Murphy et al. proposition, users iteratively refine a structural high level view model to gain information about the source code. The technique is based on the definition of a set of mappings between this high level model and the source code. Our

technique is a reflexion model; the main difference is that we propose a recursive framework to apply this reflexivity. This recursion leads in define multiple views from any generated (or existing) view. At each recursion a view extracts from another view using a viewpoint.

7 Conclusion

The presented approach for architecture reconstruction allows us to extract information for the stakeholders who are interested in high level architectural views of a software system. We propose a generic and recursive framework that considers various viewpoints (from different sources of information) and generates multiple views of an existing system. The first instantiation of the framework uses a view of the source code as input. This process of instantiation considers horizontal and vertical recursions to extract cascading architectural views. The approach stresses on separating the defined notion of architectural viewpoint into: *viewpoint model* and *viewpoint extraction algorithm*. This enables us to propose two classes of viewpoints: *matching* viewpoints emphasizing the model side (*e.g.*, business-domain based and pattern-based viewpoints) and the *discovering* viewpoints emphasizing the extraction algorithm side (*e.g.*, activity-based viewpoint, cohesion viewpoint). Each architectural element in an extracted view entails a link towards the group of elements it represents (from the input view). This link is a valuable asset for system maintenance and allows maintaining the consistency among different views of the system during its evolution. The approach presented in this paper can be used at any object-oriented source code granularity level. The straightforward approach uses classes as the first architectural element kind, but any other slicing like method-based and/or package-based can be used.

Acknowledgements: This work has been partially funded by the french ANR JC05 42872 COOK Project.

References

1. P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002.
2. D. Cubranic and G. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings 25th International Conference on Software Engineering (ICSE 2003)*, pages 408–418, New York NY, 2003. ACM Press.
3. F. Deissenboeck and D. Ratiu. A unified meta-model for concept-based reverse engineering. In *Proceedings of the 3rd International Workshop on Metamodels, Schemas, Grammars and Ontologies (ATEM'06)*, 2006.
4. S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2008.
5. A. Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 122–134, 2004.

6. S. Ducasse, M. Lanza, and S. Tichelaar. Moose: an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems. In *Proceedings of CoSET '00 (2nd International Symposium on Constructing Software Engineering Tools)*, June 2000.
7. S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 2009.
8. P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Nov. 1997.
9. Y. Guo, Atlee, and Kazman. A software architecture reconstruction method. In *Working Conference on Software Architecture (WICSA)*, pages 15–34, 1999.
10. IEEE Architecture Working Group. *IEEE P1471/D5.0 Information Technology — Draft Recommended Practice for Architecural Description*, Aug. 1999.
11. M. Jazayeri. On architectural stability and evolution. In *Reliable Software Technologies-Ada-Europe 2002*, pages 13–23, Berlin, 2002. Springer Verlag.
12. R. Koschke and D. Simon. Hierarchical reflexion models. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE 2003)*, page 36. IEEE Computer Society, 2003.
13. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995.
14. A. Lakhotia. Rule-based approach to computing module cohesion. In *Proceedings 15th ICSE*, pages 35–44, 1993.
15. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.
16. A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, Nov. 2005.
17. 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.
18. K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views — a case study. *Journal of Computer Languages, Systems and Structures*, 32(2):140–156, 2006.
19. H. A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.
20. G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.
21. M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 170–178, 2002.
22. M. P. Robillard and G. C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE'02: Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, New York, NY, USA, 2002. ACM Press.
23. P. Tonella and G. Antoniol. Object oriented design pattern inference. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, pages 230–238. IEEE Computer Society Press, Oct. 1999.