International Journal of Civil Engineering and Technology (IJCIET)
Volume 8, Issue 2, February 2017, pp. 503–510 Article ID: IJCIET_08_02_052
Available online at http://iaeme.com/Home/issue/IJCIET?Volume=8&Issue=2
ISSN Print: 0976-6308 and ISSN Online: 0976-6316

MODULAR VIEWS ON SOFTWARE ARCHITECTURE SYSTEMS

P. Kiran Kumar
Research scholar, Dept of C.S.E, Bharath University, Chennai, Tamil Nadu, India

Dr. V. Khanna
Dean of Info, Bharath University, Chennai, Tamil Nadu

ABSTRACT

In general, software architecture is documented using software architecture views to address the different stakeholder concerns. The current trend recognizes that the set of viewpoints should not be fixed but multiple viewpoints. To ensure the quality of the software architecture various software architecture evaluation approaches have been introduced. The evaluation of the adopted viewpoints that are used to design and document the software architecture has not been considered explicitly. If the architectural viewpoints are not well-defined then implicitly this will have an impact on the quality of the design and the documentation of the software architecture. We present a modular view for assessing existing or newly defined software architecture viewpoint languages. The approach is based on software language engineering techniques, These user groups (stakeholders) have different concerns relevant to the models the modular views provide a means to visualize complex information and are also a way to fulfill the concerns of different user groups. In this paper the concept of modularity is to show how these multiple views addresses the stakeholders concerns. The different viewpoints are identified to construct ct the multiple views.

Index Terms: Software Architecture, Architectural Views, Structures of Software, Multiple Views, Viewpoints.

Cite This Article: P. Kiran Kumar and Dr. V. Khanna, Modular Views on Software Architecture Systems. *International Journal of Civil Engineering and Technology*, 8(2), 2017, pp. 503–510. http://iaeme.com/Home/issue/IJCIET?Volume=8&Issue=2

1. INTRODUCTION

Software systems need to evolve over time [12]. They are modified to improve their performance or change their functionality in response to new requirements, detected bugs, etc. Changes are thus part of the system maintenance; they preserve the system functionality and ensure it performs well. Other changes 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 continuous evolution increases the system complexity [12].

To successfully evolve a complex system, it is essential to understand it. The under-standing phase is time and effort consuming, due to several reasons, among which: the system size (large systems consist of millions lines of code), inappropriate documentation, lack of overall views of the system, its previous evolutions (not necessarily documented), etc. This motivates our work on sup-porting the understanding phase of software system evolution by extracting higher level system views.

We focus on extracting architectural views of existing software systems. It is widely accepted that multiple architectural views are needed to describe the software architecture [9][7][1]. Architecture relevant information can be found at different granularity levels of given systems and needs to be studied from different viewpoints. A model providing the main concepts and their relationships defines a viewpoint. The extraction rules indicate how the architectural view elements map to the elements of the architectural viewpoint. Different viewpoints are considered such as business-based, software pattern-based, cohesion-based model, etc.

The notion of multiple views has a long history in software engineering and related Þelds (such as requirements engineering, data engineering and systems engineering), where views are introduced to separate concerns and therefore to control descriptive complexity.

Despite these precursors, their role is less secure in the Þeld known as Software Architecture. This ap-pears to be the case for a couple of reasons. First, there are no coherent foundations for their use in architecture. Second, some researchers regard their introduction as problematic because multiple views introduce problems of view integration and view consistency.

Yet, practicing software systems architects routinely deploy multiple views in the description of complex systems, albeit on an informal basis. It would be useful if there were ways to conceptualize multiple views in a manner that (1) would meet the needs of practitioners and (2) make the problems of addressing them tractable to researchers in architectural description.

This paper proposes the viewpoints and multiple views for software evolution and need for the same. Section 2 discusses the related work about views and viewpoints in software evolution. Section 3 provides the concept Software Architecture systems and concerns. Section 4 provides the concept modular viewpoints and concerns .It also discusses the identified viewpoints to construct the proposed multiple views. Section 5 outlines the conclusions and future work.

2. RELATED WORK.

This section reviews related work about the views and viewpoints in the area of software evolution.

iACM Tool [2] is a prototype tool to tackle the impact analysis and change management of analysis/design documents in the context of UML based development. This taxonomy consist views such as static (class diagram) view, interaction (sequence diagram) view and the state chart diagram view. These views support the UML models.

Christain F.J.Lange, et.al. proposed a framework [3],[4] consisting of UML model elements, their properties, and software engineering tasks, that form a basis to develop new views of UML models and related information. Based on this framework they proposed eight views to support different tasks. These views are UML based views, which maintains model evolution and quality.

Multiview Software Evolution (MVSE) is a UML based framework for Object-Oriented software [11].In MVSE, evolution of complex systems is a process in which transformations are successively applied to multiple views of software (represented by models), until objective criteria are satisfied. A stakeholder view reflects the perspective of a stakeholder on a systems application and behavior. In MVSE stakeholders initiate changes to systems and describe these changes in the context of stakeholder views.

Rene Keller et.al. [10] Introduces the concepts of multiple viewpoints and multiple views in engineering change management.

Change prediction Method (CPM) tool implements the change prediction. Stephen Cook et.al. [12] proposed an approach to understand software evolution. This approach looks at software evolution from two different points of view. One is dynamic view point, which investigates software evolution trends in models and the second is static view point which studies the characteristics of software artifacts to see what makes a software system more evolvable.

The above mentioned tools and frameworks describe the multiple views and viewpoints for traditional software evolution and change management, in which only the UML models are considered. Hence, there are no such views and viewpoints exist in the literature, to address the stakeholders concerns during evolution of the different, unrelated models in MoDSE. The following section discusses the viewpoints, and construction of proposed views to satisfy the stakeholders concerns.

3. SOFTWARE ARCHITECTURE SYSTEM VIEWS

3.1. Architecture

Architecture can take two roles: one describing how the software system's architecture should be and the other describing how a software system's architecture is. Part of the usefulness of architecture analysis is to measure the discrepancy between the prescribed architecture and the architecture that describes the software produced. Where architecture is defined as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution." This is used as the starting definition in this work as it has been agreed upon through a community vetting process. As the frame-work evolved, other aspects,

Software architecture has emerged as an important sub-discipline of software engineering, particularly in the realm of large system development. Architecture gives us intellectual control over the complex by allowing us to focus on the essential elements and their interactions, rather than on extraneous details.

The prudent partitioning of a whole into parts (with specific relations among the parts) is what allows groups of people — often groups of groups of people separated by organizational, geographical, and even temporal boundaries — to work cooperatively and productively together to solve a much larger problem than any of them would be capable of individually. It's "divide and conquer" followed by "now mind your own business" followed by "so how do these things work together?"— that is, each part can be built knowing very little about the other parts except that in the end these parts must be put together to solve the larger problem. A single system is almost inevitably partitioned simultaneously in a number of different ways: Different sets of parts, different relations among the parts. Architecture is what makes the sets of parts work together as a successful whole; architecture documentation is what tells developers how to make it so. For nearly all systems, extra-functional properties (quality attributes or engineering goals) such as performance, reliability, security, or modifiability are every bit as important as making sure that the software computes the correct answer.

3.2. ARCHITECTURAL VIEWS

For most software systems it is possible to identify three classes of important concepts: application domain concepts, implementation domain concepts, and architectural concepts. Application domain concepts result from application domain analysis and jointly form domain model. Implementation domain concepts result from implementation domain analysis and jointly define infrastructure, virtual machine or platform.

Architectural concepts are not found from analysis of requirements or implementation platform. Key concepts for architecture of software need to be invented to simplify the task of bridging the product requirements and implementation platform. Thus one of the primary tasks of software architects is to establish and communicate to the rest of the team all the important concepts necessary for effective software design

and implementation. A proven way to approach this goal is by creating partial models that relate different architectural concepts and their role in addressing architectural problems and concerns.

These models reflect various aspects of software construction and execution and provide partial views on architecture of the software. Together these views make conceptual architecture of software. Architectural views are created before the system is designed to any significant degree of detail and usually exist more as a vague intuition than a precise structure. To communicate these intuitions to the development team, to define architectural partitions, and to develop detailed designs, architects must rely more on evocative concepts than formal descriptions. This is the primary reason why verbal interaction is considered so important for successful communication of conceptual architecture

4. MODULE VIEW

The modular decomposition of a system's software. Such documentation enumerates the principal implementation units -- or modules -- of a system, together with the relationships among these units. Generically we will refer to these descriptions as *module views*. As we will see, these views can be used for each of the purposes outlined in the Prologue: education, communication among stakeholders, and as the basis for analysis. Modules emerged in the 1960's and 1970's based on the notion of software units with well-defined interfaces providing some set of services (typically procedures and functions), together with implementations that hide (or partially hide) their internal data structures and algorithms. More recently, these concepts have found wide-spread use in object-oriented programming languages and modeling notations such as the UML.

Today the way in which a system's software is decomposed into manageable units remains one of the important forms of system structure. At a minimum it determines how a system's source code is partitioned into separable parts, what kinds of assumptions each part can make about services provided by other parts, and how those parts are aggregated into larger ensembles. Choice of modularization will often determine how changes to one part of a system might affect other parts, and hence the ability of a system to support modifiability, portability, and reuse. As such, it is unlikely that the documentation of any software architecture can be complete without at least one view in the module viewtype.

We will start out by considering the module viewtype in its most general form. In the next chapter we identify four common styles:

The *decomposition* style is used to focus on "containment" relationships between modules.

The *uses* style indicates functional dependency relationships between modules.

The *generalization* style is used (among other things) to indicate specialization relationships between modules.

The *layers* (or *layered*) style is used to indicate the "allowed to use" relation in a restricted fashion among modules.

4.1. What the Module View type Is For and What It's Not For

Construction: The module view can provide a blueprint for the source code. In this case there is often a close mapping between modules and physical structures, such as source code files and directories.

Analysis: Two of the more important analysis techniques are requirements traceability and impact analysis. Because modules partition the system, it should be possible to determine how the functional requirements of a system are supported by module responsibilities. Often a high-level requirement will be met by some sequence of invocations. By documenting such sequences, it is possible to demonstrate to the customers how the system is meeting its requirements and also to identify any missing requirements for the developers. Another form of analysis is impact analysis, which helps to predict what the effect of modifying the system will be. Context diagrams of modules that describe its relationships to other modules or the outside world

build a good basis for impact analysis. Modules are affected by a problem report or change request. Please note that impact analysis requires a certain degree of design completeness and integrity of the module description. Especially dependency information has to be available and correct in order to create good results.

Communication: A module view can be used to explain the system functionality to someone not familiar with the system. The different levels of granularity of the module decomposition provide a top down presentation of the systems' responsibilities and therefore can guide the learning process.

It is difficult to use the module viewtype to make inferences about runtime behavior, because it is a partition of the functions of the software. Thus, the module view is not typically used for analysis of performance, reliability, or many other runtime qualities. For those we typically rely on Component and Connector and deployment views.

4.2. Notations for the Module View type

I. Information notations

A number of notations can be used in a module view's primary presentation. One common, but informal notation uses bubbles or boxes to represent the modules with different kinds of lines between them representing the relations. Nesting is used to depict aggregation and arrows typically represent some form of *depends on* relation. Figure 4 above illustrates nesting to describe aggregation. In that figure small dots were used to indicate interfaces, similar to the UML "lollipop" notation introduced in Section 11.5 ("Notation for Documenting Interfaces").

A second common form of notation is a simple textual listing of the modules with description of the responsibilities and the interfaces for each. Various textual schemes can be used to represent the *is part of* relation, such as indentation, outline numbering, and parenthetical nesting. Other relations may be indicated by keywords. For example: the description of module A might include the line "Imports modules B, C," indicating a dependen-cy between module A and modules B and C.

II.UML

Object modeling notations like UML provide a variety of constructs that can be used to represent different kinds of modules. Figure 5 shows some examples for modules using UML notation. UML has a class construct, which is the object-oriented specialization of a module as described here. As we will see, packages can be used in cases where grouping of functionality is important, such as to represent layers and classes. The subsystem construct can be used if specification of interface and behavior is required. An example for the use of sub-systems can be found in the module decomposition style described in Chapter 2 ("Styles of the Module View-type").

4.3. Elements, Relations, and Properties of the Module Viewtype

Elements

The term "module" is used by system designers to refer a variety of software structures, including programming language units (such as Ada packages, Modula modules, Smalltalk or C++ classes), or simply general groupings of source code units. In this book we adopt a broad definition:

We characterize a module by enumerating a set of responsibilities, which are foremost among a module's properties. This broad notion of "responsibilities" is meant to encompass the kinds of features that a unit of software might provide, including services as well as internal and external state variables.

Modules can both be aggregated and decomposed. Different module views may identify a different set of modules and aggregate or decompose them based on different style criteria. For example, the layered style

identifies modules and aggregates them based on an allowed-to-use relation, whereas the generalization view identifies and aggregates modules based on what they have in common.

Relations

The relations of the module view type are:

- Defines a part-whole relation between the submodule A (the part) and the aggregate module B (the whole). In its most general form, the relation simply indicates some form of aggregation, with little implied semantics. In general, for instance, one module might be included in many aggregates. There are, however, stronger forms of this relation. An example can be found
- In the module decomposition style in Chapter 2, where this relation is refined to a decomposition relation
- Defines a dependency relation between A and B. This relation is typically used early in the design process when the precise form of the dependency has yet to be decided. Once the decision is made then *depends on* usually is replaced by a more specific form of the relation. Later we will take a look at two in particular: *uses* and *allowed to use*, in the module uses and layered styles, respectively. Other, more specific examples of the depends on relation include *shares data with* and *calls*. A call dependency may further be refined to *sends data to, transfers control to, imposes ordering on*, and so forth.
- Defines a generalization relation between a more specific module (the child A) and a more general module (the parent B). The child is able to be used in contexts where the parent is used. We will look at its use more detailed in the module generalization style. Object-oriented inheritance is special case of the is-a relation.

PROPERTIES

As we will see in Section 10.1 ("Documenting a view"), properties are documented as part of the supporting documentation for a view. The actual list of properties pertinent to a set of modules will depend on many things, but is likely to include the ones below.

Name

A module's name is, of course, the primary means to refer to it. A module's name often suggests something about its role in the system: a module called "account_mgr," for instance, probably has little to do with numeric simulations of chemical reactions. In addition, a module's name may reflect its position in some decomposition hierarchy; e.g., a name such as A.B.C.D refers to a module D that is a submodule of a module C, itself a submodule of B, etc.

Responsibilities

The responsibility for a module is a way to identify its role in the overall system, and establishes an identity for it beyond the name. Whereas a module's name may suggest its role, a statement of responsibility establishes it with much more certainty. Responsibilities should be described in sufficient detail so that it is clear to the reader what each module does.

Further, the responsibilities should not overlap. It should be possible, given a responsibility, to determine which module has that responsibility. Sometimes it might be necessary to duplicate specific responsibilities in order to support some qualities of the system such as performance or reliability. In this case describe the duplicated responsibilities by stating the condition of use. For example, module A might be responsible for controlling a device during normal operation. But there is also a module B, which has higher performance but less features, that takes over during times of high processor load.

Visibility of interface(s)

An interface document for the module establishes the module's role in the system with precision by specifying exactly what it may be called upon to do. A module may have zero, one, or several interfaces.

In a view documenting an is-part-of relation, some of the interfaces of the submodules exist for internal purposes only, that is, they are used only by the submodules within the enclosing parent module. They are never visible outside that context and therefore they do not have a direct relation to the parent interfaces.

Different strategies can be used for those interfaces that have a direct relationship to the parent interfaces. The strategy shown in Figure 4(a) is encapsulation in order to hide the interfaces of the submodules. The parent module provides its own interfaces and maps all requests using the capabilities provided by the submodules. However, the facilities of the enclosed modules are not available outside the parent.

Alternatively, the interfaces of an aggregate module can be a subset of the interfaces of the aggregate. That is, an enclosing module simply aggregates a set of modules and selectively exposes some of their responsibilities. Layers and subsystems are often defined in this way. For example, if module C is an aggregate of modules A and B then C's (implicit) interface will be some subset of the interfaces of Modules A, B. (See Figure 4(b).).

Implementation information

Since modules are units of implementation, information related to their implementation is very useful to record from the point of view of managing their development (and building the system that comprises them). Although this information is not, strictly speaking, architectural, it is highly convenient to record it in the architectural documentation where the module is defined. Implementation information might include:

- **Mapping to code units.** This identifies the files that constitute the implementation of a module. For example, a module ALPHA, if implemented in C, might have several files that constitute its implementation: ALPHA.c, ALPHA.h, ALPHA.o (if pre-compiled versions are maintained), and perhaps ALPHA_t.h to define any data types provided by ALPHA.
- **Test information.** The module's test plan, test cases, test scaffolding, test data, and test history are important to store.
- **Management information.** A manger may need the location of a module's predicted completion schedule and budget.
- **Implementation constraints.** In many cases, the architect will have a certain implementation strategy in mind for a module, or may know of constraints that the implementation must follow. This information is private to the module, and hence will not appear, for example, in the module's interface

5. CONCLUSIONS AND FUTURE WORK

The multiple views for Software Evolution are proposed. Viewpoints are identified to construct each view. Different concerns of the stakeholders are satisfied by each view, providing sufficient knowledge regarding model driven software evolution. The analytical support of all these proposed views for the laws of software evolution was discussed. it is observed that all the proposed views support majority of laws. So, the proposed views are sufficient for the stakeholder to capture the information and the laws are also important for module Driven Software Evolution. Development of framework consisting of the proposed views and the evaluation of the framework with case tools and the stakeholders concerns is the subject of future research.

REFERENCES

[1] Arie van Deursen, Eelco Visser, and Jos Warmer. Model-Driven Software Evolution: A Research Agenda, In Dalila Tamzalit (Eds.). *Proceedings 1st International Workshop on Model-Driven Software Evolution*, University of Nantes, 2007. pp. 41-49.

[2] Briand L.C, Y.Labiche, O' Sulliavan. Impact Analysis and Change Management of UML Models, *Proceedings of the International Conference on Software maintenance (ICSM'03),* IEEE, 2003.

[3] Christian F.J. Lange, Martijin A.M. Wijins, Michel R.V. Chaudron, A Visualization Framework for Task-Oriented Modeling using UML, *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07),* IEEE, 2007.

[4] Christian F.J. Lange, Martijin A.M. Wijins, Michel R.V. Chaudron. Metric View Evolution: UML-based Views for Monitoring Model Evolution and Quality, Proceedings of the 11th European Conference on Software Maintenance and reengineering (CSMR'07), IEEE, 2007.

[5] Ahmed Abd-Allah. Extending reliability block diagrams to software architectures. Technial Report USCÐCSEÐ97Ð501, Center for Soft-ware Engineering, Computer Science Department, University of Southern California, 1997.

[6] Ahmed Abd-Allah and Barry W. Boehm. Rea-soning about the composition of heterogeneous architectures. Technical Report USCÐCSEÐ95Ð 503, University of Southern California, 1995.

[7] Robert J. Allen. A Formal Approach to Soft-ware Architecture. PhD thesis, Carnegie Mellon University, May 1997. Distributed as CMU-CS-97-144.

[8] T. Ravichandran, Dr. Krishna Mohanta, Dr. C. Nalini and Dr. P. Balamurugan, Literature Survey on Search Term Extraction Technique for Facet Data Mining in Customer Facing Website. International Journal of Civil Engineering and Technology, 8(1), 2017, pp. 956–960.

[9] Dileep Kumar Padidem and Dr. C. Nalini, Process Mining Approach to Discover Shopping Behavior Process Model in Ecommerce Web Sites Using Click Stream Data. International Journal of Civil Engineering and Technology, 8(1), 2017, pp. 948–955.

[10] C. Aarthi and Dr. P.B. Ramesh Babu, Anti-Cancer Activity of Phyllanthus Reticulatus on Colon Cancer Cell Line. International Journal of Civil Engineering and Technology, 8(1), 2017, pp. 943–947.

[11] M. L. Brodie and S. L. Zilles, editors. Proceedings of the Workshop on data abstraction, databases, and conceptual modeling, Published as special is-sue of SIGPLAN Notices 16(1) 1980.

[12] Perry, D.E. and Wolf, A.L. Foundations for the Study of Software Architecture, Software Engineering Notes, ACM SIGSOFT, Vol. 17, (4), October 1992.