www.ijecs.in
International Journal Of Engineering And Computer Science ISSN: 2319-7242
Volume 4 Issue 8 Aug 2015, Page No. 13956-13963

Component-Based: The Right Candidate for Restructuring the Nature of Software Development in Organizations

Badamasi Imam Ya'u

International Islamic University Malaysia,

P. O. Box 10, 50728 Gombak, Kuala Lampur, Malaysia

Email: badayau@gmail.com

Abstract

Component-based software development is an emerging field in software engineering aims toward the cost effective development of composite components or a complete system by reusing pre-built components or subsystems that are perhaps stored in a repository. In this paper the techniques of current software model such as objects and classes in object-oriented programming language and architecture models have been studied and hence coming up with the rationale of using a component-based software model with a unique property of compositionality and encapsulation as contrarily to what happens in the current component models.

1.0. Introduction

It is ubiquitous that many people interchange the meaning of software and computer programs. As we know, a computer program is a step by step sequence of instructions that perform or carryout a particular task, and therefore it is virtually concerned as line by line written code. In the other hand, software is a broader term that not only confines about instruction coding but also entails all necessary details such as other programs incorporated in the software, software installation requirements, configuration files, system documentation, license, user documentation, website information etc. [1]. The process of developing, organizing, maintaining, managing, specifying, evolving the software system is known as software engineering. Software engineering is a field of engineering applied to software system with the aim of rapid and cost effective production of software to satisfy the end users. The notion of software engineering began in 1968 during the software crisis as a result of the then emergence of new powerful third generation computer hardware [2]. Right from that time, the needs for software always increased as the computer based systems are produced. More and more electronic products incorporate computer software in different forms, and the cost of these software products plays significant role in the total system cost. So because of the dependability on those complex computer-based systems, the production of cost effective software systems is essential for the national and international economy which is the primary goal of software engineering.

This paper focuses on Component-based development which is a branch of software engineering aims towards a systematic way of reusing pre-built software components or sub-systems in a cost effective approach to develop larger and larger systems.

Prior to any software development, are a lot of activities that involve software requirement specification, analysis, design etc. These requirements are mostly enquired from the intended customers, stakeholders, clients and other end users. Different techniques are available and used to gather the necessary requirements; these include interview, discussion, administration of questionnaire, documentation, etc. Components are to be built on the basis of systematic compositionality and encapsulation model where each component encapsulates computation (data) and its composition connector or operator encapsulates controls for such interaction. Each component has an interface through which it receives and sends information.

2.0. Motivation

Component-based development (CBD) is an important area of software engineering that is emerging and focuses more on the reusability of components. For long, industries have been looking a way or technique that will be used to reduce the cost of software production and maintenance as well as decreasing the time of this software to markets. Component-based models promise all these [3]. The backbone of any CBD technique is its underline software component design which describes its functionality, reusability, adaptability and defines how it can be constructed; updated, deployed as well as explaining what operations can be used on those components and what are the constraints of these operations.

Basically, two types of component models [4] do exist: Object oriented models where objects behave as components and communicate with each other. Typical example is Enterprise javaBeans (EJB) in object oriented programming. The other type is where architectural units behave as components in software architecture; typical example of this model is architecture description languages (ADLs). The aforementioned models are the two major categories of component models that cover all existing current software component models such as JavaBeans, COM, web services, UML 2.0, Koala, KobrA etc.

Ideally, components are developed according to acceptable standard life cycle. That is, each component should undergo three stages of development: design phase, deployment phase and runtime phase. By undergoing these stages, we assume that a component conforms to the acceptable CBD desiderata. The criteria for this desiderata is that, from the design phase, components should be preexisting software fragments developed and stored in repositories by different or independent software developers and further allow the reusability of these software units by other independent parties to compose larger composite system that will satisfy their need.

3.0. Software Requirement Specification

SRS is the technique of defining, describing, and showing the total characteristics, behavior, patterns etc. of the system or new system to be developed. To develop or produce anything, we need some raw ingredients as an input that will be processed and yield an output. Requirements are the building blocks and key ingredients of any software system, and should therefore be properly elicited. Variety of requirements elicitation methods are adopted [5]. These include verbal interviews, survey, questionnaires, scenarios etc. Requirement elicitation is carried out by system developer mostly through interaction with clients and users. In the phase, two types of requirements are obtained: Functional and non-functional requirements. Functional requirements are those requirements that describe the overall behavior expected of the target system, i.e. the functionalities or services the system should provide. They are sometimes referred to as use cases which completely describe the interactions between a user and the software. They include inputs, outputs, exceptions etc.

In the other hand, non-functional requirements are requirements which do not directly define the services or functions of a system but rather control the functionality of the system. They are system properties or constraints that determine the behavior of the target system. They entail quality standards, reliability, response time, storage capacity performance requirement and design constrains that are imposed during the software design or implementation. Unlike functional requirements, non-functional requirements relate to the entire system rather than to individual fragments of the system. In this case, their failure significantly prevents the functionality of the whole system.

4.0. Component-based Approach

To develop a component-based software system that will support incremental composition a special component-based model of Kung Kiu Lau with properties of encapsulation and compositionality which is coupled with exogenous connectors that help improves the lapses therein current component models [6]. The encapsulation property wraps the computations that occur in the components and control mechanism that passes to and pro from the exogenous connector whereby making data and computation private from outsiders of the component; while making total freedom for the exogenous connector to initiate and pass control. This feature makes the model so efficient in alleviating tight coupling during component's communication. A good description of this model follows in the subsequent sections.

4.1. Basic Entities of Components

In this approach, a component based model is defined to have the properties of compositionality and encapsulation. To express these properties, two basic entities by which each component possesses need to be understood. These are computation unit and connector [7].

- ***Computation unit:*** is more or less a private section where a component stores a set of methods, data (services); and provides these services when they are needed outside of the component (for other computation units). The methods are invoked but the invocation in this respect is not direct as in object oriented components. Everything is private and encapsulated within the computation unit; i.e. there no inter method invocation between computation units, rather any communication should be passed and controlled via an intermediary.
- ***Connector:*** This is the intermediary that passes control between computation units or components. Since we are talking about composition, the intermediary is not meant to be a single connector merely for a single component. To deal with composition, two types of connectors do exist: invocation and composition connector.
    - ➢ *Invocation connector:* is a unary level i.e. the lowest level connector that is encapsulated in a single component. It receives control from outside of the component, passes this control to the computation unit to invoke the chosen methods or services and sends or passes back the control after the execution of the request to where it was initially came outside of the component. The invocation connector therefore encapsulates control in this respect.
    - ➢ *Composition connector*: Like invocation connector, a composition connector also coordinates and encapsulates control but in this case for a set of components. Because of the hierarchical nature of the composition. Variety of n-ary composition connectors are used to accomplish the compositional task. These include: sequencing e.g. sequencer to pass control serially; a pipe to pass result of one component that is required in other unit. We use selector connector for branching among the components, loop for iteration and guard connector for verification.

4.2. Kinds of Component

To exploit the usefulness of invocation and composition connectors, components are categorized into two classes; atomic and composite components [8].

- Atomic component consists of computation unit and invocation connector and therefore encapsulates computation.
- Composite component entails the combination of atomic components connected by composition connector such as sequencing, selector, pipe etc. it

encapsulates computation and control by connecting the subcomponents to the interface of the composite component.

The figure below depicts the two kinds of components i.e. atomic and composite components as shown in [9].

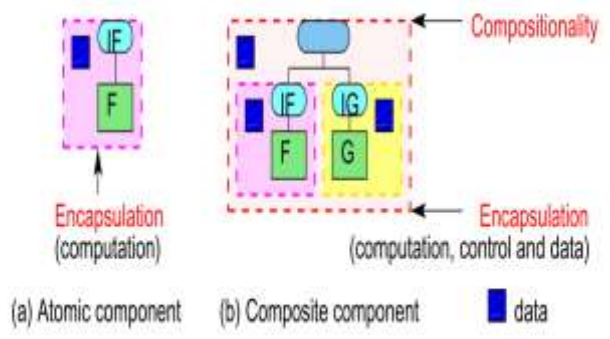

Fig 1: Atomic and Composite component

4.3. Properties of Component Model

Virtually, software component models have some distinct special properties that distinguish them from others. The distinguishing characteristics that make this component model so special and work in a superb manner are the following two main techniques:

4.3.1. Encapsulation

Encapsulation [10] is a term that specifies a region of control and ownership which a component maintains. Components usually have data and processes by which those data are computed. To differentiate a component with class or object in object oriented programs, both data and computation should be encapsulated in the component model making them both private from the external environment. While in other hand, an object can only encapsulate data. By making the data and the computation private, I mean there is no direct access to that region. Any access should be provided by interfaces. This makes the computation to be done within the component unit itself without having invoked to other units. If we look at the picture depicted in the fig 4 above, we see that in all cases data in a blue rectangular box is encapsulated. In 1(a) the invocation connector and the computation unit are surrounded by dotted line, showing that computation is encapsulated in the atomic component. So no any outside method can directly have access to the computation without the intervention of the invocation connector. While in 1(b), both computation and control are encapsulated by the dotted line to make the composition connector and the individual computations in F and G private to other components.

4.3.2. Compositionality

The general idea is how to utilize our understanding of software component with the characteristics of encapsulation to continue building more and more composite components. Component should then be compositional, i.e. having two or more components in a repository, say F and G as depicted in figure 3, with each atomic component maintaining the property of encapsulation as in 1(a) above, then a composite component say H can be built by assembling F and G as shown in 1(b) above coupled with inheriting both encapsulation and compositionality properties. Since computation and control are encapsulated for any composition, depending upon the number of preexisting components we have in the repository and the new ones constructed on the ground that the system needs, we continue to compose larger complex composite components or system with similar property of compositionality [11].

4.3.3. Exogenous Connectors

As we have seen so far, current component models use connectors as a mechanism for message passing; the connectors being just a medium such as bus in C2 [12] for establishing communication between components. The pitfall here is that, everything is initiated and coordinated in the components. Computation as well as origination of control are done in the components; thus enables the component to initiate method call (or remote procedure call) and manipulate their returns. This technique only allows the connector to coordinate the flow of the messages between the components and hence maximizes coupling among the components and connectors.

The rationale in exogenous connector is to alleviate external dependency and coupling so that computation and control are separated and done in the component and connectors respectively. In exogenous connector as shown in fig 2 below, there is no direct interaction between component A, B, C, D and E, rather all components respond by sending their reactions to the intermediate connectors Con1, Con2, Con3 and Con4. In this, the exogenous connectors encapsulate the entire control of the interaction; they initiate the method calls in the components and coordinate any flow between the connected components.

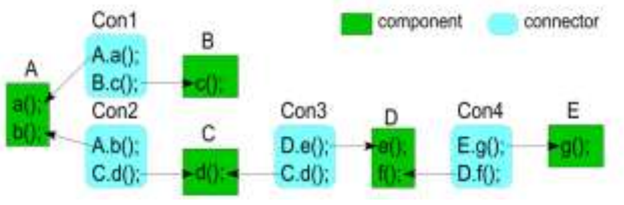

Fig 2: components connected by exogenous connector

In current component models where messaging passing of either direct or indirect methods are adopted, the connectors are merely channels that support interaction; since computation and control are handled in the components, the connectors are not meant to be stored in the repository, for they are not reusable; they are specifically meant for particular application. For better wellbeing of component-based method, the adaptation of exogenous connectors to separate control from computation is really expedient.

4.4. Example Using Partial Architectures

Partial architectures with open interfaces are used during component development process to enable incremental composition to the completion of the development where they become components with closed interfaces [13]. To describe how the composition will be done, let us use a simple example of the following requirements.

R1. Customers scan items from their shopping basket in supermarkets.

R2. A machine reads the bar code of the item and displays the amount.

R3. Customers then pay the amount of their items by cash or using credit card

R4. The machine then verifies the payment and prints receipt for the customer

In this simple example we have four requirements that merely require a component scanner that will read the car code, a component display, cash, card reader, authentication, and print components.

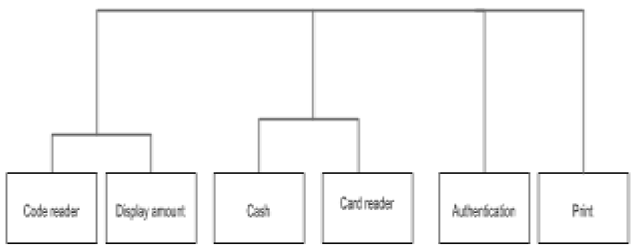

Fig 3: Partial architectures of self-service machine

The above example is incomplete in the sense it only shows the partial architectures rather than depicting a complete system with all level composition connectors. Like in the first composition, we need a pipe connector to push the data from code reader to display amount; in the second composition a selector is needed for branching and finally pipe connector to compose the entire system. However, during the progression of the project, a tool will be developed to implement the task.

5.0. Conclusion

Component-based software development brings lots of benefits to organizations and software vendors for the provision of reusability of components. To serve the same purpose, this project was started from the scratch and aims for mapping requirements directly to components as traditional structured models fall short to achieve. It involves automatic extraction and incremental mapping of parts of speech from natural language requirements specification to partial architecture under construction. In realizing this, the project adopts a special component model with the properties of encapsulation and compositionality which uses exogenous connectors that ensure loose coupling in the system.

6.0. Reference

1. Stephen R. Schach (2002). Object-Oriented and Classical Software Engineering. McGraw-Hill Higher Education, 1221 Avenue of the Americas, New York, NY 10020. Fifth edition.
2. Sommerville (2001). Software Engineering. Sixth edition, Pearson Education limited.
3. A. W. Brown. Large-Scale, Component-based development. Object and component technology series, 2000.
4. K-K. Lau and Z. Wang: Software Component Models. IEEE Transactions on Software Engineering Vol. 33, No. 10 October 2007.
5. M. Saeki, H. Horai and H. Enomoto: Software Development Process from natural Language Specification. In proc. 11th ICE, pages 64-73. ACM, 1989, ACM.
6. K.-K. Lau, A. Nordin and T. Rana. Constructing Component-Based Systems Directly from Requirements Using Incremental Composition.
7. K.-K, Lau M. Orgaghi, Z. Wang: A Software Component Model and its

preliminary formalization. In: de Boer. F.S., Bonsangue, MM., Graf, S., de Roever, W.-P (eds) FMCO 2005. LNCS, vol. 4111, pp. 1-21. Springer, Heidelberg (2006).

8. K.-K. Lau and I. Ntalamagkas. Component-based Construction of Component Systems with Active Components. School of Computer Science, University of Manchester.
9. K.-K, Lau (2010). A Software Component Model with Encapsulation and Compositionality.
10. K.-K. Lau and F. Taweel. Data encapsulation in software components. In H. Schmidt et al., editor, proc. CBSE 2005.
11. K.-K. Lau and I. Ntalamagkas. A compositional approach to active and passive components. In proc. EUROMICRO- SEAA 2008, pages 76-83. IEEE, 2008.
12. K.-K. Lau, P. V. Elizondo and Z. Wang. Exogenous Connectors for Software Components.
13. CBSE 2005, LNCS 3489, pp. 90-106 2005.
14. J. A. Wang (2000). Towards Component-Based Software Engineering. Consortium for computing in Small Colleges: Rocky Mountain Conference, JCSC 16, 1(November 2000).