International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

COMPOSABILITY OF COMPONENTS IN COMPONENT BASED SOFTWARE DEVELOPMENT (CBD)

H. P. S. Dhami

AP(CSE)-Dean Acedemics, RBIENT, Hoshiarpur

Abstract

Component-based Software Engineering (CBSE) (also known as Component-based Development (CBD)) is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. Software components vary from normal software parts in the sense that they own composition potentialities, named composability. Composability is the capability to select and assemble simulation components in various combinations into simulation systems to satisfy specific user requirements. Lack of proper composition of software components is a main concern between components users & developers. *The defining characteristic of composability is the ability to combine and recombine components into different simulation systems for different purposes.* Present component technologies are not prowling much support for the non functional properties of components that generally become a cause of poor composability. If a component is enable to compose in various environments, and then there is a need to add some programmability with the components. A proposal is to use light weight components such that the overheads (that are not required in a particular application) do not get transported with the body of component. Based on this suggestion, an attempt is made to propose the model of “Template Component” with “Component Generator” that will generate components according to the requirements of the

specific application. This idea calls for concern about the composability since beginning that is at the time of generation of the component.

Keyword: CBSE, CBD, HLA

1. Problem Decomposition

In order to optimize the design, construction and maintenance process of software, we need to apply the divide-and-conquer principle by decomposing our systems and problems into smaller parts, which can be decomposed again, recursively. This decomposition process must continue until a level is reached where each building block (a) can be understood and constructed effectively, and (b) deals only with a single concern (we will discuss the motivation for this later). The word 'problem' in 'problem decomposition' is not restricted to end-user requirements, but applies to anything from given requirements to the implementation of a simple task or algorithm.

The decomposition process is a way to analyze and manage complexity –in other words, it is a problem solving technique– but at the same time, it may provide a basis for system construction and maintenance. This is because it has same result as that of the decomposition process –as it applies to the design phase– determines the structure and the building blocks for constructing the system[1].

We make the following important assumptions about software development:

- The method of decomposition determines what the building blocks are and how they are related.
- We can always identify useful and appropriate abstractions and structures for a particular application by analyzing the related problem domain.
- A software development method *should* be *structure-preserving*: this means essentially

[1] This assumes that the same modeling paradigm is used in all development phases.

that traceability between the 'input'-artifacts (e.g. requirements) and 'output'-artifacts (esp. code) is such that an iterative rather than a waterfall-style development process is supported.
Concluding, the decomposition process should take domain knowledge as an input, and result in a structured set of building blocks that offer a clear mapping to the structure and abstractions of the problem domain.

2. Introduction

Composability is an increasingly important issue in system development. The main objective of Component-based Development (CBD) or CBSE is to reduce time to market & cost, and on the other hand increase quality of software system. CBSE is able to achieve it by developing software components once & use it many times. In CBSE, components are developed autonomously from software development. So, a process of component evaluation, adaptation & composition must be performed before using components into component-based software systems. At the time of development of the component we don't have a clear idea about the design structure of the software system, where the component has to be deployed. A component is responsible to provide some functionality to a system where it is going to be incorporated (plugged). Sometimes it might happen that a component is incorporated easily into the system but fails to perform its desired operation & system's performance may get affected. This problem generally arises because components are not properly composed into the system. Components [1], whose interfaces are syntactically compatible, exhibit undesirable behavior when used together. The problem of proper composition of software components is an important issue between component developers & components users. Component composition goes one step further than integration in that the result of component composition is a software assembly that can be used as a part of a larger composition. The problem of reasoning about how well components will work together is the most vital problem faced by component based system developers today. Components alone are not responsible for composition failure; it also depend on other factors like nature of the

ISSN (Online) : 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

connectors, architecture of the system, run time behavior, composition rules etc. A standard architecture[6] will require, on which application can be built & deployed. The architecture should provide appropriate level of functionalities and should also help to automate the creation of standard 'plumbing' to plug the application into itself. A component may be obtained from runtime environment & integrated in the application. On the other hand a component may be obtained from a repository serving as a supplier from some other organization.

The capability of component may improve if we are able to predict the behavior of a component's behavior in a specific application under specific conditions. The components interface, many times, doesn't have sufficient information for good composition. So, one idea may be to give all required functional & non-functional information with the interface. But this will make components interface heavy & inadequate. A set of bond can also be associated with a component that will give information about input & output parameters, pre & post conditions etc. But this may increase documentation overhead. A proposal is to use light weight components such that the overheads (that are not required in a particular application) do not get transported with the body of component. Based on this suggestion, an attempt is made to propose the model of "Template Component" design that will help to generate the component according to the requirements of the specific application & this approach is discussed in this paper.

Some approaches to the composition of software have been proposed in literature J. A. Stafford and Kurt Wallanu [1] have described problem of composition due to inadequate interfaces. As per Barbier[2], the Composability of a software component is defined as "Whole-Part Theory Approach". The foundation of this approach is encapsulation of sub-components by component, emergent and resultant properties for component with regards to their sub component & finally state & life time dependencies. Gordon S Novak Jr [3] explained the method of company reusable software components through views.

In Gordon S Novak Jr.[4] research, the correct research topics of Composability have been discussed & it also points out some problems of composability. Composition anomalies have been discussed in Lodewijk Bergmans' [5]. Orcas Neierstrase & T. D. Meijler [7] addressed some issues of software composition as lack of suitable framework, Composition model and Compositional language. Kikel D Pretty, Eric W Weisel [8] and Jeffrey Voes [9] focus on composition problem for embedded systems.

3. Composability

Software components [2] vary from normal software parts in the sense that they own composition potentialities, commonly named Composability or Compositionality. A highly computable system provides recombinant components that can be selected & assembled in various combinations to satisfy specific user requirements. Many definition of Composability are stated here. While, Carine Lucas, Patric Steyaert and Kim Mens [4] composability is a much desired quality for software artifacts, there is no consensus whatsoever on what composability really is, not how it can be achieved.

Composability means "The ease with which a component can be integrated & perform the functionalities as desired by the specific application".

It is the ability to rapidly configure, initialize, and test an exercise by logically assembling a simulation from a pool of reusable components [10].

Composability of a component deals with its plug-ability with other components & its dynamic run time behavior in the application. The essential attributes that make a component composable are self containment and statelessness.

The composition between software components depends on [1]:

- The nature of components.
- The nature of connector (Protocols & data models),
- The architecture of the assemblies (Constraints on interaction), and

- Runtime construction process.

Two types of composability can be defined: Syntactic & Semantic [8]. Semantic composability is the actual implementation of composability; it required that the composable components be constructed so that their implementation details, such as parameter passing mechanism, external data access, and trimming assumptions are compatible for all of the different configurations that might be composed. The question in syntactic composability is a question of whether components can be connected. In contrast, semantic composability is a question of whether the models that make up the composed simulation system can be meaningfully composed.

3.1 Levels of Composability

As the term "composability" in the literature is compared it is apparent there is one way in which the meanings often differ. It differs on the question of what is being composed and what is formed by the composition. Various different answers can be found in the literature; they will be referred to as *levels* of composability. Nine levels of composability are defined here. These levels have been drawn from various sources, some of which explicitly or implicitly include several of the levels defined here in composability (e.g., [13], [21]). Composability levels from different sources have been combined. Those listed here have different meanings and implications, but there may be some overlap in component and scale between them.

1. *Application (also called event-level).* Applications such as real systems, simulations, networks, communications equipment and auxiliary software components are composed into simulation events, exercises or experiments. For this to be a level of composability, rather than simply integration, the composition must be done in way that allows combining and recombining the applications into different systems and events. This level of composability is also called "event-level" [11].

2. *Federate(also called* federation-level*)*. Federates are composed into persistent federations. A federation is persistent if is reused for a number of different purposes (such as events, exercises, or experiments), though possibly with some changes to the set of federates that have been composed. The composition may be supported by an interoperability protocol, such as DIS (Distributed Interactive Simulation), ALSP (Aggregrate Level Simulation Protocol), and HLA (High Level Architecture). Examples of this level of composability include the Joint Training Confederation and the Combat Trauma Patient Simulation [12]. This level of composability has also been called "federation-level" [11]. The terms "federate" and "federation" have specific HLA meanings; here they are being used with more generic meanings analogous to their HLA meanings to denote simulations linked together, but not necessarily with HLA.
3. *Package*. The Pre-assembled packages comprising sets of models that form a consistent subset of the battle space are composed [10].
4. *Parameter*. Parameters are used to configure pre-existing simulations [10].
5. *Module.* Software modules are composed into software executables. The executables may be federates in a federation or standalone simulation systems. The OneSAF family of software products is expected to have this level of composability [14] [15] [16].
6. *Model (also called* object-level, component). Various models of smaller-scale processes or objects (means simulated real-world objects) are composed into composite models of larger-scale processes or objects. Models of physical processes, such as rainfall and wind, may be composed into composite models of larger-scale physical phenomena, such as weather. The composite models may be implemented as modules or federates. This level of composability has also been called "object-level" [11], "component" [10], and "reconfigurable models" [17].

ISSN (Online) : 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

7. *Data*. Data sets are composed into databases and these data sets may be initially distinct because they describe different entities, they are from different sources, or they represent different aspects of some phenomena. Different data sets were composed to represent electronic warfare in DIS [18]. SEDRIS is intended to support such composability for natural environment databases.
8. *Entity (also called* federate-level). Entities are composed into groupings. This level of composition may be hierarchical, with several layers of groupings composed into higher level groupings. This level of composition is typically done with data, rather than with software, as in ModSAF and WARSIM. This level of composition has also been called "federate-level" [11].
9. *Behavior*. Low-level atomic behaviors are composed into high-level composite behaviors, which are to be executed by autonomous simulation entities in a computer generated forces system or constructive simulation. The behaviors may be expressed in a variety of forms. Examples include hierarchically organized finite state machines as used in ModSAF and its variants [19] and process flow diagrams [20].

3. Reason of Poor Composability

The following reason may be the cause of poor composability:

1. Defective software components [9]
2. Lack of suitable architecture keeping composition in mind.
3. Problem with assumptions (contractual requirements) between components [9].
4. Inputs received that are outside the range of any profile that the original designer anticipated [9].
5. Dependency between components is not actually foreseen & precisely specified [2].

Table 1 summarizes these composability levels.

Components	**Composition**	**Example(s)**
Application	Event	Unified Endeavor
Federate	Federation	Joint Training Confederation, Combat Trauma Patient
Package	Simulation	JSIMS
Parameter	Simulation	JSIMS
Module	Executable	OneSAF
Model	Composite model	ModSAF, OneSAF
Data	Database	Electronic warfare in DIS, SEDRIS
Entities	Military unit	ModSAF, WARSIM
Behavior	Composite behavior	Finite state machines, Process flow diagrams

Table 1 Levels of Composability.

4. Template Component

When a component is plugged into a subsystem, then its subsystem is expecting some functionality from the component and the component also expects some support from the subsystem. If any one of the above two fails to fulfill the responsibility, the component would not be able to provide proper functionality to the subsystem. [Figure. *1*]

Fig. 1 Responsibility dependency in a Component Based System.

A component can only be accessed through its interface. Such an interface should contain all necessary information about component operations and about the context in which a component will be deployed. In general purpose component technologies, the interface are usually implemented as object interface supporting polymorphism by late binding, while late binding allow connecting of components that are completely unaware of each other beside the connecting interface, this flexibility along with a performance penalty and increases risk for system failure. Also the predictability of the system's performance or other properties decrease since the composition of the components occurs at runtime. To make a component properly composable into a subsystem, some extra information will be needed like input and output parameter, possible error codes, memory requirements etc. Lack of these information sometimes create problems in composition of components. One way is to add more information (function & non-function requirements) with the interface of a component. But this will make components interface heavy and inadequate. Another alternative is to add all required conditions within the internal logic of a component. These requirements will be different for different environments, and only few of the

International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

conditions will be used at a time. Such components will take more memory at runtime because the whole components will add to the program whether we may need some functionality. If a component is enable to compose in various environments, and then there is a need to add some programmability with the components. Components should be developed to be delivered in such a manner so that problem of integration and composition do not arise and overheads (those are not required in a specific application) do not get transported with the body of the component. The composability issue must not be an after thought as it is normally not possible to modify a component once it has been designed and implemented.

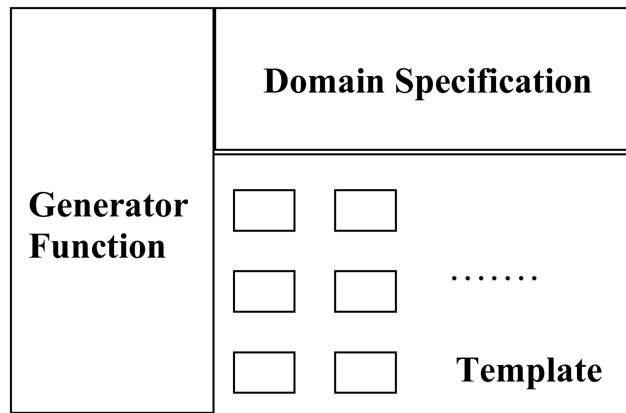

Fig 2. Proposed

One way is to use light weight components that express the internal logic in a base class and use a common generator function that will generate the component according to the variations or modifications required by the specific application.

The basic idea is – i) To design the component with attributes and functionalities that will always be essential in its any deployment.

ii) It should have scope for addition of, or modification in, other functionalities as per requirements of specific applications.

Thus we have proposed here the idea of “Template Component” (fig 2) that can be developed in two steps.

International Journal of Enterprise Computing and Business Systems

ISSN (Online) : 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

The first one, we names it as "Generic Component", contain the basic functionality of a component, with all general features.

In the second step, there will be a "Component Generator" function that will generate the component according to the need of the specific application. Once the template component is defined, the component generator will automatically call the template component and generate that component with required functionalities. The main motivation of "Template Component" is to reduce the size of components and produce light weight composable components that will take less memory and execute time. Template Components will provide all necessary information for composition, by separating actual implementation details at run-time. A client of a template component needs to get the work done without having to worry about which algorithm will be required to do it under varying circumstances. It will provide a greater degree of flexibility, generality and efficiency. Components generated with their method would be easily composable into a subsystem.

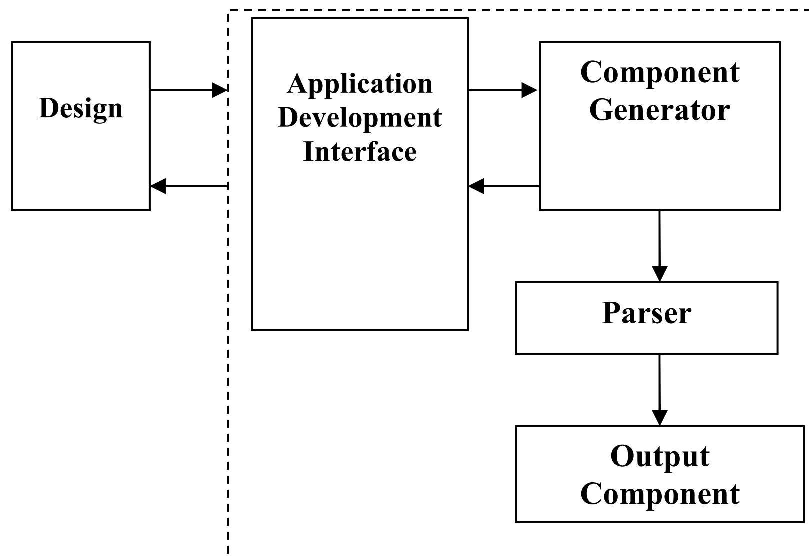

Fig 3. A template component development environment

In our proposal "Template Component" model (fig 3.), there will be on ADI (Application Development Interface), a set of template components along with a common "Component Generator" function in a local repository. Here Components will be Template Components analogous to templates in the .NET Framework, which will contain all essential features. If a designer needs a component (with some Constraints) then he will request to ADI, ADI then send

this request to component generator, The component generator will accept the required specification and sent it to Component Descriptor. A Component Descriptor keeps record about all Template Components along with its specifications; a routine will then match the specification and pick up required Component.

Component Generator then generates the Component according to the specific user requirements. Here Component Generator will act like a template processor that will generate the Components according to the application need. Such a component would be pluggable and Composable in applications and will perform desired functionality.

For example, if a Component has to be deployed in two different Component based software system, one for stand alone environment and other for client-server environment (or for Mobile Computing Environment), then it night be possible that both environments have different requirement at run-time but the basic logic would be same. Our proposed Component Generator function will generate the Component according to the need of the specific Component based application environment.

5. Advantages

- Template Components takes less memory of run-time.
- Template Components would promote modularity & flexibility.
- Such Components would be more suitable for embedded system.

6. How to Enhance Composability

Composability of Component should increase in such a manner that increase (at least not decrease) the quality of a software in which component is going to be deployed. Following points may be useful to enhance the Composability of a Component.

i. Components should be developed as Black Box.
ii. Communication between the two components should be limited.
iii. Components interface should be smaller.
iv. Components should be properly combined into assembly according to its functionality neither unnecessary dependency may reduce comparability.
v. A Component needed some contextual requirement, in which it is going to be composed. This information should be kept as minimum as possible.

7. Conclusion

The aim of this research work is to point at the possibility of applying this approach for developing of light components that would be suitable for composition. Component based Software Engineering will be as successful as comparable the components would be. When CBSE would be mature enough to provide components "On Demand" then only the proper culture of software development with Component would come into being.

We propose here a model that addresses the question, though in limited sense. The Component generation model would be able to generate some Component only if the corresponding template is available and that two of the automatic modification of the template to generate the Components is possible. Another possibility is to make the development in customization of a canonical component rather than automatic generation.

8. References

1. J A Stafford and Kurt Wallnau, "Component Composition and Integration", pp 187-192, in "Building Reliable Component Based Systems" by Ivica Crnkovic, Magnus Larson, Artech House publishers ISBN i-58053-327-2.

2. Barbier F., “Composability for Software Components –An Approach Based on the Whole Part Theory”, Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 02) page 101-106, 2002.
3. Gordon S Novak Jr., “Composing Reusable Software Components Through Views”, Proc. 9th Knowledge Based Software Engineering Conference (KBSE-94) pp 39-47, Montercy CA, Sept 1994, IEEE Computer Society Press.
4. Carine Lucas, Patric Steyaert, Kim Mens, “Research Topics in Composability”, Proc. Of the CTOO-96 Workshop at ECOOP presented at the ECOOP 96 Workshop on Composability issue in Object Oriented, published in Special Issues in Object Oriented Programming: Workshop Reader of the 10th Europiean Conference on Object Oriented Programming, pp 81-86, 1996.
5. Lodewijk Bergmans, Bedir Tekinerdogan, M. Glandrup and Mehmet Aksit, “On Composing Separate Concerns, Composability and Composition Anomolies”, ACM OOPSLA Workshop on Advanced Separation Concerns, USA 2000.
6. Subrahmanyam Allamaraju et al, “Professional Java Server Programming” pp-10 61, J2EE 1.3 Edition, Apress Publisher.
7. Oscar Neierstrasz, Theo Dirk Meijler, “Research Direction in Software Composition”, ACM Computing Surveys (CSUR) Volume 27, Issue 2 (June 1995) page 262-264, ISSN: 0360-0300.
8. Kikel D Pretty, Eric W Weisel, “A Formal Basis for A Theory of Semantic Composability”, In Proc. Of the Spring 2003 Simulation Interoperability Workshop, 2003, 035-SIW-054.
9. Jeffrey Voes, “Predicting System Trustworthiness”, pp 201-204, in “Building Reliable Component Based Systems” by Ivica Crnkovic, Magnus.
10. JSIMS Composability Task Force, “JSIMS Composability Task Force Final Report”, September 30 1997.
11. G. M. Post, “J9 Composability Summary Comments”, Electronic mail, June 12 2002.

12. M. D. Petty and P. S. Windyga, “A High Level Architecture-based Medical Simulation”, *SIMULATION*, Vol. 73, No. 5, November 1999, pp. 279-285.
13. M. Biddle and C. Perry, “An Architecture for Composable Interoperability”, Proceedings of the Fall 2000 Simulation Interoperability Workshop, *Proceedings of the Fall 2000 Simulation Interoperability Workshop*, Orlando FL, September 17-22 2000, 03S-SIW-073.
14. United States Army, *One Semi-Automated Forces Operational Requirements Document*, Version 1.1, Online document at URL http://www-leav.army.mil/nsc/stow/saf/onesaf/onesaf.htm/, August 21 1998.
15. A. J. Courtemanche and R. B. Burch, “Using and Developing Object Frameworks to Achieve a Composable CGF Architecture”, *Proceedings of the Ninth Conference on Computer Generated Forces and Behavioral Representation*, Orlando FL, May 16-18 2000, pp. 49-62.
16. A. J. Courtemanche and R. L. Wittman, “OneSAF: A Product Line Approach for a Next-Generation CGF”, *Proceedings of the Eleventh Conference on Computer-Generated Forces and Behavior Representation*, Orlando FL, May 7-9 2002, pp. 349-361.
17. A. Diaz-Calderon, C. J. J. Paredis, and P. K. Khosla, “Organization and Selection of Reconfigurable Models”, *Proceedings of the 2000 Winter Simulation Conference*, Orlando FL, December 10-13 2000, pp. 386-393.
18. D. D. Wood and M. D. Petty, “Electronic warfare and Distributed Interactive Simulation”, in T. L. Clarke (Editor), *Distributed Interactive Simulation Systems for Simulation and Training in the Aerospace Environment*, SPIE Critical Reviews of Optical Science and Technology, Vol. CR58, SPIE Press, Bellingham WA, 1995, pp. 179-194.
19. R. B. Calder, J. E. Smith, A. J. Courtemanche, J. M. F. Mar, and A. Z. Ceranowicz, “ModSAF Behavior Simulation and Control”, Procedings of the Third Conference on Computer Generated Forces and Behavioral Representation, Orlando FL, March 17-19 1993, pp. 347-356.

ISSN (Online) : 2230-8849

http://www.ijecbs.com

Vol. 2 Issue 1 January 2012

20. S. D. Peters, N. D. LaVine, L. Napravnik, and D. M. Lyons, “Composable Behaviors in an Entity Based Simulation”, Proceedings *of the Spring 2002 Simulation Interoperability Workshop*, Orlando FL, March 10-15 2002.
21. M. D. Petty and *Eric W. Weisel,* Virginia Modeling, Analysis and Simulation Center Old Dominion University, Norfolk VA 23529, “A Composability Lexicon”, Unpublished manuscript, July 4 2002.