DESIGN DESCRIPTION FOR SOFTWARE COMPONENTS

Vladan Jovanovic, Georgia Southern University, vladan@gasou.edu
Stevan Mrdalj, Eastern Michigan University, stevan.mrdalj@emich.edu

ABSTRACT

This paper outlines viewpoints of interest in the design of software components for software intensive information systems. Applicability of a standard for design representation is discussed indicating usage of standard languages for such representation. Additionally, a unifying meta-model for software component design is presented.

Keywords: software component, design description, design language, UML.

CONCEPTUAL MODEL FOR SOFTWARE COMPONENT DESIGN

The purpose of this paper is to explore a standardized software component design description. Our focus is on designing components to be used in component-based development (CBD) for complex information systems. In contemporary software design practices (1,2,3,4,5,6,7,8,9) it is common to use various representations during the design of complex information systems. It is because no single view can accommodate various stakeholders and their legitimate concerns, nor can it cover all aspects of design. All this is especially true for the design of the software components, even though the use of components reduces the complexity for larger systems.

Our focus is on the design description of individual components and not how components are discovered. Since there is no universally recognized definition for CBD or Component, we will work from a very general notion encompassing implementation and executable level components including executable programs, static and dynamic libraries, source code files and similar binaries. For examples we may quote J2EE and .NET as two most popular implementation technologies based on components.

We begin by providing a summary of concepts and terms used in the context of software component design descriptions as a conceptual meta-model in the form of a UML class diagram (Figure 1). The top portion represents a classification of component design elements in terms of what a component might be. It also shows a composition pattern for components. The design description is shown at the bottom portion of Figure 1 as a composition of several viewpoints each representing the involved design entity with its attributes and relationships. Next, this paper summarizes software component design in terms of design descriptions using various viewpoints (see Table 1). Design viewpoints are the means to organize the Software Design Description (SDD), to satisfy the requirements of each stakeholder, to promote a separation of concerns, and to provide a comprehensive description of a system from all relevant aspects.

This paper then describes what makes nominated viewpoints 'required' or why those elected are to be covered in a well-documented design for individual software components. For each required viewpoint, we indicate a usage of the expected standardized description techniques for describing that viewpoint. A rational requirement for the selection of representational technologies is that they should be properly defined and commonly accessible, in other words standardized.

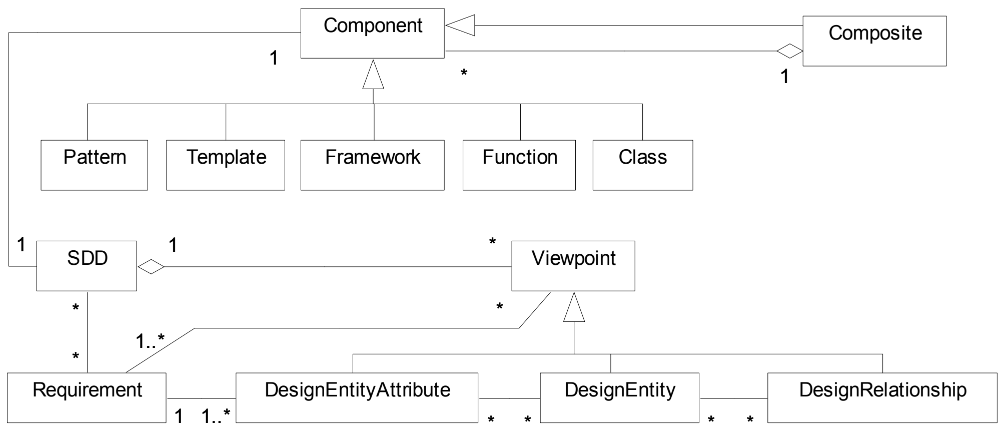

Figure 1: Conceptual Meta-Model for Software Component Design

REQUIRED VIEWPOINTS FOR COMPONENT DESCRIPTIONS

The draft of the IEEE 1016 (11) Standard for Software Design Descriptions lists ten viewpoints as required for the description of general software designs. We found that six of them apply when the scope of design is a single software component. Additional viewpoints can be defined per stakeholders needs or even imposed contractually. Contemporary development practice recognizes, see the IEEE Recommended Practice for Architectural Descriptions of Software Intensive Systems (10), that the viewpoint declaration is an explicit activity.

In order to design non-trivial software components it is indeed necessary to fully describe their design from several viewpoints. As Table 1 shows, there are six required viewpoints applicable for every system component, and while in each actual case there may be additional viewpoints of interest, they are not common for all components. For example, a human computer interface is of interest only for externally visible components with humans as direct operators. The most important viewpoint for software components is the explicit interface. Beside the interface, at minimum, the following viewpoints are expected: the decomposition of components into software objects, their dependencies, the dynamics of state changes, the interaction among collaborating objects, and the allocation of methods to various objects the component is comprised of, and methods' internal (procedural) logic.

Traditionally, for high-level designs various forms of informal box and arrow diagrams were used, for detail design structure charts were common, and for physical architecture systems diagrams were used. Instead, this paper summarizes standardized design languages of interest for components design for each viewpoint in Table 1. In addition, a unifying concept of the UML Package serves as a general visual representation (a folder) for grouping and organizing elements of various descriptions as well as a visual enclosure for a component itself (see Figures 2-5).

Documentation of components may go beyond the design description to include the performance of component realization, instructions for use etc. Our focus is on design even to the exclusion of test related information. This may change in the near future, since testing is increasingly becoming a part of the design process or it is used methodologically "instead of a design" in Extreme Programming and Test Driven Development approaches (7).

Table 1: Summary of viewpoints for software component design description.

Component Design Viewpoint	Expectation	Design Languages
Software Interface	required	UML Component Diagram UML Object Constraint Language (OCL)
Static structure	required	UML Class Diagram
Dynamic or state changes	required when applicable	UML Statechart, Petri Net
Interaction	required	UML Sequence Diagram, UML Collaboration Diagram
Dependency	required	UML Component Diagram UML Package, UML OCL
Algorithmic or detail logic	required for safety-critical, and when applicable for general purpose components	Fault Tree Diagram, Decision Table, programming language (like C++), UML Activity Diagram, UML OCL

SOFTWARE INTERFACE VIEWPOINT

The component interface description is intended to serve software architects, acquirers, designers, programmers and testers of components, as well as independent testers. It includes the details of external interfaces. This viewpoint deals with software interfaces and not with human computer interfaces, which have a separate viewpoint typically not in the scope of an individual component design. This viewpoint consists of a set of interface specifications for a component as a design entity shown in Figure 2. For each component, it provides a reference to the detailed description via the identification attribute. The component interface description should contain everything another designer or programmer needs to know to develop software that interacts with that component. The attribute descriptions for identification, type, purpose, function, and subordinates should be included in this design view. Design relationships include 'composition' and 'clientship'. For a commercial component, a design relationship may include 'ownership' and 'warranty' when those characteristics are available. It is worth nothing that the viability of the component-based technology depends on all characteristics that customers may be interested in, not just technical connectivity.

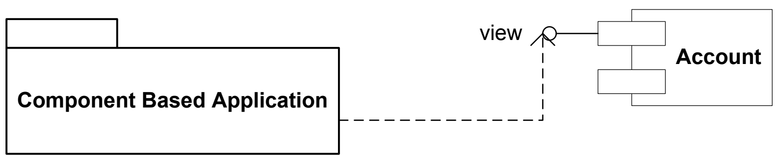

Figure 2: Interfacing in CBD

The interface description serves as a binding contract among designers, programmers, customers, and testers. It provides them with an agreement needed before proceeding with the detailed design of objects, systems or components. In addition, the interface description may be used by technical writers to produce customer documentation or it may be used directly by customers.

STATIC STRUCTURE VIEWPOINT

Every non-trivial system has its parts, subsystems, components, modules or units that it is comprised of. That is particularly obvious in CBD's practice of reusing components. Buyers, maintainers and developers of complex components all need a 'bill of materials,' see Figure 3. Those needs are described in a static structure viewpoint.

A static structure represents a component level structure in terms of classes, associations (including aggregation and composition), generalization, interfaces and objects as design entities. The purpose is to conveniently represent design abstractions including invariant-static properties as aggregates of data/object groupings together with functions operating upon them. A standardized representation language used for the static structure is the UML class diagram.

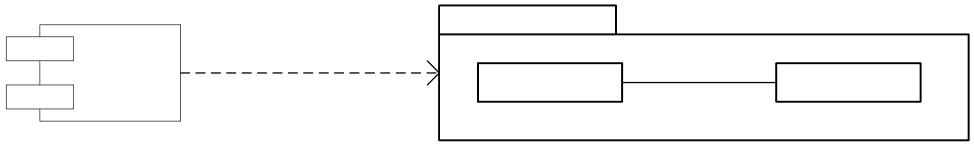

Figure 3: Static Structure Viewpoint

DYNAMIC OR STATE TRANSITION VIEWPOINT

This viewpoint is mainly of interest for reactive real-time and similar systems. Design entities are classes, components, states, events and transitions. The concurrency, timing and synchronization may be additional issues warranting extensive notation. Contemporary design languages for this viewpoint are UML Statechart Diagrams (Figure 4) and Petri-nets. Sometimes UML collaboration diagrams are used, too. Although most business components do not necessarily have very interesting state changes, there is significant number of the theoretical results available in this area.

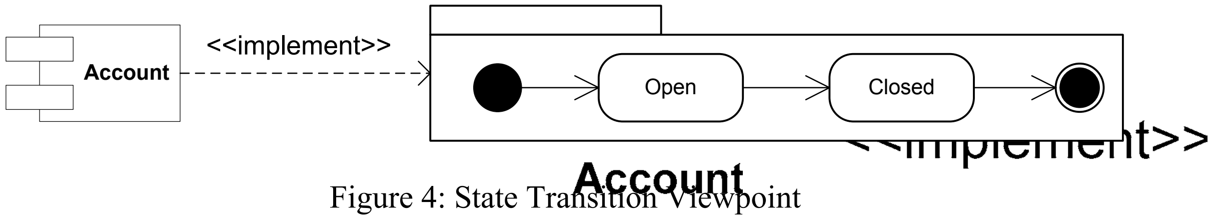

Figure 4: State Transition Viewpoint

INTERACTION VIEWPOINT

From the interaction viewpoint we will only treat the situation of developing a component using the object-oriented technology where objects collaborate in order to perform the service of a component. The assignment of responsibilities is the key issue in this viewpoint. The UML Sequence and Collaboration Diagrams are commonly used for documentation of this viewpoint. Designers frequently use design patterns and analysis of responsibilities to develop such diagrams.

This design viewpoint defines the strategies for interactions among design entities and provides the information needed to easily perceive how, why, where, and at what level actions occur. General software design and component design do not differ here in the technology used but rather in the level of rigor exercised in the evaluation of alternatives. As a component is basically a server with its internal interactions hidden, such an interaction may be documented using sequence diagrams (Figure 5). When interactions are derived from design patterns then collaboration diagrams are commonly used.

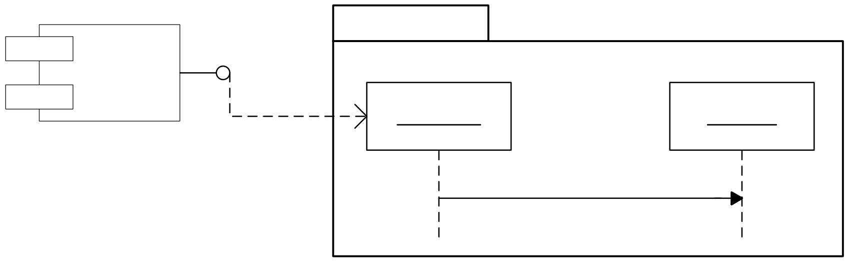

Figure 5: Interaction Viewpoint represented with a sequence diagram

Among component users this viewpoint is not prominent, since seeing into the black-box violates the principle: open for designers, closed for users. Contrary, for component designers it is one with the biggest challenge, where design skills and diligence come to the forefront. This is also an area where professionals traditionally trained on procedural software designs have the most difficulties with and where experience in the object-orientation and design patterns shows its value.

Account View

DEPENDENCY VIEWPOINT

:Ac

The dependency description specifies the relationships among the component's entities. Components may need services from the environment or from other well-known components. Such a description identifies the dependent entities, describes their coupling, and identifies the required resources. It also may specify the types of relationships that exist among the entities, as shown in Figure 3, with the implied order of implementation. The dependency relationships among packages and components might be stereotyped as 'reside', 'derive', 'implement' and 'uses' as per Figures 3 and 4. The dependency description provides an overall picture of how the component works from the outside (as opposed to, for example, an interaction diagram showing how it works inside) in order to assess the impact of requirements and design changes. It can help maintainers to isolate external entities causing component failures or resource bottlenecks. It can aid in producing the system integration plan by identifying the entities that must be developed first. This description can also be used by integration testing to aid in the production of integration test cases.

ALGORITHMIC OR DETAILED LOGIC DESIGN VIEWPOINT

This viewpoint contains the details needed prior to implementation of a software component. The detailed design description can also be used to aid in producing unit test plans. This description includes meaning and use of a design entity such as the static versus dynamic aspect, whether it is to be shared by transactions, used as a control parameter, used as a value, used as a loop iteration count, or used as a link field. In addition, data information should include a description of data validation needed for that process. This level of detail can be stored in a code-base documentation repository or a data dictionary relegated to the code. More importantly standardized representations of complex logic can be communicated using decision tables or programming language itself. Specifically, for safety critical systems, the components description requirements include fault tree diagrams. Modern Integrated Development Environments are evolving in the direction of providing increasingly sophisticated support of code refactoring and analysis so that it makes more and more sense to use the code snippets as the description of required functionality. This is becoming a common practice at the very detailed level of description for the individual methods. The development using top-down step-wise refinement with stubs is getting a new life. The languages like Java and C# are well suited to be used for detailed logic design.

SOFTWARE COMPONENT COMMERCIALIZATION

In the context of software component design representation, the question of commercialization is seldom addressed. Figure 6 provides an illustration of a capsule template for defining commercial components by exposing services beyond the basic functionality. Software components can be systematically included into reusable libraries only if the metadata and evaluation kits are standardized with designed self-tests, quality and usage profile data. This implies a need to standardize additional viewpoints for the design of commercial components. One such viewpoint would be a Meta Description that includes the component's qualities and usage profiles. Another viewpoint would be an Evaluation Kit that includes aspects of evaluations, similar to hardware boxes' self-testing capability. This can be an added value service by third parties.

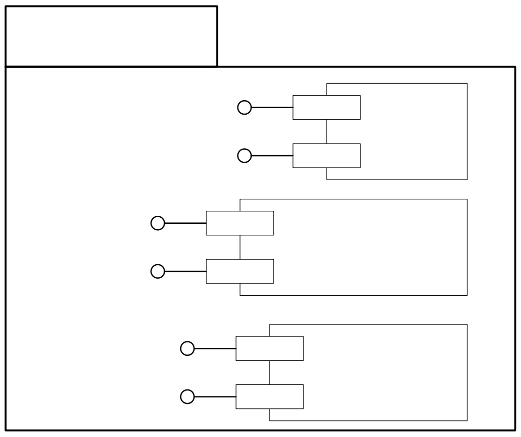

Figure 6: Component as a commercially reusable asset

As shown in Figure 6, a commercial component package "Component Capsule" will then create a façade based on both the original software component and the supplementary component descriptions (including the design description in a manner discussed in this paper). The content of the Component Capsule may depend on the willingness of a producer to disclose or on the demand of a real market for components. A potential user can explore the capsule and decide to include the capsule in its catalog of reusable assets. Entire libraries can be created using the Meta Description and Evaluation Kit viewpoints. Our hypothesis is that by separating the concerns on the software component design description and by introducing additional viewpoints, the suggested structure of commercial components can be implemented in a standardized manner providing for the automation of the trade in standardized components.

CONCLUSION

Starting from viewpoints and selecting standardized design description languages, we outlined non-exclusive but predominantly UML based notations for software components. The obtained description, while not simple, shows why it is not desirable to use a single representation in the design of software components. For example, a component diagram alone can not represent all six viewpoints. The focus on requirements for standardized SDD for components allowed us to ignore issues such as deployment, human interface, even persistent data which may occasionally surface with reuse of large grained components. Nevertheless, we expect six listed viewpoints to be of universal interest for all component designers. This paper may also serve as an example of viewpoint declarations for designers facing situations compelling them to introduce additional viewpoints.

State of the practice (1, 2, 3, 4, 5, 6, 8, 9) seems to be ahead of academic recognition of CBD in general and component SDD in particular. It is interesting to note that very little theory can be currently offered on software component design, specifically for CBD. This gap made us commit to the study of component SDD rather then CBD itself. The future direction of our work will be to focus on establishing elements for an underpinning theory and technology for evaluation and measurement of components and on further standardization of practices for component design descriptions.

REFERENCES

1. Allen, P. (2001). Realizing e-Business with Components. Addison Wesley.
2. Atkinson, C., et all (2002). Component-based Product Line Engineering with UML. Addison Wesley.
3. Cheesman, J. and Daniels, J. (2001). UML Components- A Simple Process for Specifying Component Based Software. Addison Wesley.
4. Clements, P., et all (2003). Documenting Software Architectures. Addison Wesley.
5. D'Souza, D. and Wills A. (1999). Objects, Components, and Frameworks with UML. Addison Wesley.
6. Heineman, G. and Council, W. (2001). Component-Based Software Engineering. Addison Wesley.
7. Martin, R. (2003). Agile Software Development. Prentice Hall.
8. Maurer P. (2003). Component-Level Programming. Prentice Hall.
9. Jacobson, I. , Griss, M. and Jonsson, P. (1997). Software Reuse. Addison Wesley.
10. IEEE Draft Std 4.0 1016-2003, IEEE Standard for Software Design Descriptions.
11. IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Descriptions of Software Intensive Systems.