Component–Oriented Approach for Designing Enterprise Architecture

Zoran Stojanovic
Delft University of Technology, The Netherlands

Ajantha Dahanayake
Delft University of Technology, The Netherlands

INTRODUCTION

One of the main challenges enterprises face today is how to manage complexity of systems being developed, effectively utilize the power of the Internet, and be able to rapidly adapt to changes in both technology and business. The new paradigm of component-based development (CBD) has been introduced as an excellent solution for building complex Internet-enabled enterprise information systems (Brown, 2000; Szyperski, 2002). The basic idea of CBD originates from the strategy successfully applied in other engineering disciplines that a system developed from components is more flexible and easier to develop. CBD provides higher productivity in system development through reusability, more effective system maintenance, higher quality of solutions and the possibility for parallel work. Moreover, it provides better system adaptability through replaceability of parts, localization and better control of changes, system scalability, and the possibility of using legacy assets.

CBD has often been presented as a new silver bullet for complex, enterprise-scale system development in the Internet age (Udell, 1994). However CBD inherits many concepts and ideas from the earlier encapsulation and modularization, "divide-and-conquer" initiatives in information technology (IT). The NATO Conference in 1968 recognized that producing software systems should be treated as an engineering discipline providing system assembling from software components (McIlroy, 1968). Parnas (1972) defines concepts and requirements for decomposing system into modules. These principles of separation of concerns, encapsulation, and plug-and-play building blocks have been applied in different ways through the concepts of functions, subroutines, modules, units, packages, subsystems, objects, and now components.

The CBD paradigm has been first introduced at the level of implementation and deployment. CBD middleware technologies, such as CORBA Components (Siegel, 2000), Sun's Enterprise Java Beans and Microsoft's COM+/.NET, are now used as standards for the development of complex enterprise distributed systems. While the technology solutions are necessary in building the system, one cannot simply program and deploy components using a component middleware without any prior plan to follow from business requirements towards implementation. For the effective use of the CBD paradigm and in order to gain real benefits of it, the component way of thinking must be applied in earlier phases of the development lifecycle, such as system analysis and design. CBD methods and approaches proposed so far do not provide a complete and consistent support for various component concepts. Components are often treated as implementation concepts—packages of binary or source code that can be deployed over the network nodes. During the system analysis and design, components, if used, are often represented as larger-grained business objects. This suggests using components mainly at the system implementation and deployment as software code packages, while still following the principles of object-oriented modeling, analysis and design. At the same time, the role and usefulness of the component concept as a bridge between business and technology issues have not been truly recognized yet. Components can be identified very early in the system lifecycle, namely derived from business requirements, and then used as central artifacts of the whole development process. In this way, the whole process would be structured and organized around the same set of component concepts. That can provide a necessary alignment and traceability from business services to implementation assets. The benefit from the separation of concerns using components will be gained at all levels of system development.

BACKGROUND

Component technologies are now widely used in the development of complex Internet-enabled systems. First, VBX controls, DCOM/COM, CORBA and Java Beans, and now COM+/.NET, CORBA Components and Enterprise Java Beans (EJB) represent the standard component-

based implementation solutions. The physical perspective on components as binary packages of software is still predominant. The standard unified modeling language (UML) treats components as packages of binary code and uses them in describing system implementation through component and deployment diagrams (Booch, Rumbaugh, & Jacobson, 1999). Components in UML represent physical things that can be deployed over network nodes. The Catalysis approach defines a component as a package of software code as well as other software artifacts (D'Souza & Wills, 1999). According to Szyperski, a software component is a unit of composition with contractually specified interfaces and explicit context dependencies (Szyperski, 2002). A software component can be deployed independently and is subject to composition by third parties. Gartner Group (1997) defines a runtime software component as a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at runtime. Definition of a business-oriented component concept can be found in Herzum and Sims (2000), where a business component is the software implementation of an autonomous business concept or business process, and in Andersen Consulting (1998), where a business component is a means for modeling real-world concepts in the business domain. When introducing components, the question about similarities and differences between objects and components are naturally arising. According to Udell (1994), components represent a new silver bullet for system development in the Internet age, while objects have failed to provide higher level of reusability. In the UML, components are nothing else than larger-grained objects deployed on the network nodes. In Szyperski (2002), a component comes to life through objects, and therefore, it would normally contain one or more classes, as well as traditional procedures and even global variables. In a debate over this topic (Henderson-Sellers, Szyperski, Taivalsaari, & Wills, 1999) granularity has been seen as the main issue in distinguishing components and objects. According to Catalysis, components are often larger-grained than traditional objects, and can be implemented as multiple objects of different classes.

The academia and industry have just started to recognize the importance of new CBD methods, processes, techniques and guidelines. The methods and approaches are often greatly influenced by the object-oriented concepts, constructs and principles, dictated by the use of the standard UML. The Rational Unified Process (RUP) (Jacobson, Booch, & Rumbaugh, 1999), Catalysis, and the Select Perspective (Allen & Frost, 1998) can be considered as the first generation of the CBD methods. The business component factory (BCF) (Herzum & Sims, 2000), the UML components approach (Cheesman & Daniels, 2000), and the KobrA approach (Atkinson et al., 2002) represent the second generation of the CBD methods. These methods are more focused on components concepts than previous ones. They provide a comprehensive support to CBD throughout the system lifecycle, and represent remarkable achievements in the field. On the other hand, there are certain shortcomings. The analysis and evaluation of CBD methods can be found in Dahanayake, Sol, and Stojanovic (2003).

SERVICE-BASED COMPONENT CONCEPT

Components have been so far used mainly as implementation artifacts. However, the components are equally useful and important if used as modeling and design artifacts for building the logical architecture of the system (Stojanovic & Dahanayake, 2002). The essence of the component approach is the explicit separation between the outside and the inside of the component. This means that only the question WHAT is considered (what useful services are provided by the particular building block to the context of its existence?) not the HOW (how are these services actually implemented?). A component fulfills a particular role in the context, by providing and requiring services to/from it. A component has a hidden interior and exposed interface. It participates in a composition with other components to form a higher-level behavior. At the same time, every component can be represented as a composition of lower-level components. Well-defined behavioral dependencies and coordination of activities between components are of a great importance in achieving the common goal. The metamodel of the core component concepts is shown in Figure 1.

According to the role(s) a component plays in the given context, it exposes corresponding behavior by providing and requiring services to/from its context, or by emitting and receiving events. The services a component provides and requires form the basic part of its contract. Services can be of different types such as performing computation, providing information, communication with the user, and so forth. They are fully specified in a contract-based manner using pre-conditions, post-conditions and other types of constraints. A component must handle, use, create or simply be aware of certain information to provide its services properly. In order to be used in a different context or to be adaptable to the changes in its context, a component can possess configuration parameters that can adapt the component according to new requirements coming from the outside. A component can possess a set of non-functional parameters that characterize the "quality" of its behavior. Figure 2 shows the component specification concepts.