Formal Specification of Business Components – a Design by Contract Perspective

Anca Ioana Andronescu, Oana Muntean

Abstract: *In software developers' world two aspects are common knowledge: software fails are expensive, so we need reliable software; software itself is expensive, so we want reusable software. Design by Contract, advocated by Bertrand Meyer, is known as one of the most comprehensive methods for ensuring reliability, while delivering reusable software components.*
A business component is a component that offers a certain set of services of a given business domain. In order to assemble business components with little effort to customer-individual application systems, it is necessary to establish functional, content-related and methodical standards. This paper discuss the problem of applying Design by Contract principles in the specification of business components in order to simplify their reusability between companies and software developers.
Key words: *Business Components, Design by Contract, Formal Specification*

INTRODUCTION

When they consider using a new software development method or tool, many people view *productivity* as the major expected benefit. In object-oriented technologies, productivity benefits follow not just from the immediate benefits of the approach, but from its emphasis on quality. A major component of quality in software is reliability: a system's ability to perform its job according to the specification (*correctness*) and to handle abnormal situations (*robustness*).

Although reliability is desirable in software construction regardless of the approach, it became particularly important in the object-oriented methods because of the special role given by these methods to reusability. We can say that by itself, reusability helps in building reliable software: this is the case when we are able to use component libraries produced and validated by a reputable outside source, rather than developing our own solution for every single problem we encounter. In order to be sure that our object-oriented software will perform properly, we need a systematic approach to specifying and implementing object-oriented software elements and their relations in a software system. This article discusses the problem of applying Design by Contract principles for the formal specification of business components, at design level. The effort is made in order to simplify business components' reusability between companies and software developers.

1. Background

1.1 Components. Business Components

The construction of business applications from components promises solutions that are flexible and well suited to user requirements. McIlroy first introduced the idea of software systems made up from pre-fabricated software components somewhere in 1968. The main point is to combine components from different vendors to an application, which is individual to each customer and where the components are seen as plug-and-play black-boxes. Thus, the advantages of both standard and individual software production can be combined. The principle of modular design that is underlying component based software systems brings both technological and economical advantages.

According to Fellner and Turowski, the term *component* is defined as follows:
A component consists of different (software) artefacts. It is reusable, self-contained and marketable, provides services through well-defined interfaces, hides its implementation and can be deployed in configurations unknown at the time of development.

A (software) artefact can be anything from simple executable code to graphics, text, data that describes the initial state of the component, as well as specification or user documentation.

The main idea behind components is reusability. A component is reusable if it can be integrated in other software systems without modification (excepting it's parameters of course). The property of being self-contained means that parts of the component (software artefacts) can be assigned unequivocally in order to distinguish it as a unit from other parts of the system. The fact that a component is marketable can be translated into the possibility of identifying a component as a separate good that can be sold on a software market. A *business component* is a component that implements a certain set of services out of a given business domain. They are software components that are designed to support specific business applications like Finance or CRM.

1.2 Basic Principles of Design by Contract

One way to prove the correctness of a program with respect to a formal specification is the Hoare calculus. Built upon this formal method, Bertrand Meyer developed a method of software engineering called Design by Contract [2]. The central idea of DBC is that a software system can be seen as a set of communicating components (or entities) which have obligations to other entities based upon formalized rules between them. A functional specification, or "contract", is created for each module in the system before it is coded. Program execution is then viewed as the interaction between the various modules as bound by these contracts.
Human contracts are written between two parties: the *supplier,* which performs some task for the other (the *client*). Each party expects some benefits from the contract, and accepts some obligations in return.

A contract document protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope. The same ideas apply to software. In this case, a contract governs the relations between the method and any potential caller.

1.3 . Formal Specifications and OCL

Problem specifications are essential for designing, validating, documenting, communicating, engineering, and reusing solutions for software systems. The process of writing a "correct" specification is very difficult – probably as difficult as writing a correct program.

Formal specifications, which refer to a mathematical description of the system's requirements, can greatly benefit requirements specification. Generally speaking, a formal specification is the expression, in some formal language and at some level of abstraction, of a collection of properties that a system must satisfy. Formality helps in obtaining high-quality specifications within software development processes; it also provides the bases for their automated support. Usually, "formal" is confused with "precise" (the former term entails the latter, but the reverse is not true). To be formal a specification must be expressed in a language made of three components: rules for determining the grammatical well-forming of sentences (the syntax); rules for interpreting sentences in a precise, meaningful way within the domain considered (the semantics); and rules for inferring useful information from the specification (the proof theory).

The Object Constraint Language (OCL) is a modeling language with which you can build software models. It is defined as a standard "add-on" to the Unified Modeling Language (UML), the Object Management Group (OMG) standard for object-oriented analysis and design [3]. Some of the most important characteristics of OCL are: it is a declarative language; it is not a programming language, it is a modeling language; it is a formal language; it is a typed language.

In OCL we can identify the same three kinds of expressions that DBC delivers: preconditions, postconditions and invariants [5]. Preconditions specify conditions that must hold before a method can execute. As such, they are evaluated just before a method

executes. Preconditions specify obligations that a client of a software component must meet before it may invoke a particular method of the component. In contrast, postconditions specify conditions that must hold after a method completes. Postconditions specify guarantees that a software component makes to its clients. If a postcondition is violated, the software component has a bug. An invariant specifies a condition that must hold anytime a client could invoke an object's method. Invariants are defined as part of a class definition. A violation of an invariant may indicate a bug in either the client or the software component.

2. Need for Specification of Business Components

The idea of specifying business components became more and more important in order to offer support for a composer who combines business components from different vendors to an application system.

According to the definition in the first paragraph, using business components implies their need for standardization. Therefore, the interface and behaviour of a component have to be described in a consistent and unequivocal way, through a methodical standard that will enable a common understanding of the component specifications.

Such a standard can be achieved by identifying the objects to be specified and by defining standardized notations. This ensures the reusability of components and their exchange between companies and software developers can be simplified. According to Turowski and Conrad, the specification of a business component is defined as a complete, unequivocal and precise description of its external view, that also shows which services a business component provides and under which conditions.

One of the recommendations for standardized specifications for business components was made by the working group of the German Informatics Society (GI), who proposed a methodical standard constituted by identifying the objects to be specified and by defining a standardized notation accepted all participating parties [4]. The term *specification* of a business component is defined as a complete, unequivocal and precise description of its *external* view. In other words it describes which services a business component provides and under which conditions. The resulting model defines seven different contract levels for the specification of components, as presented in Figure 1.

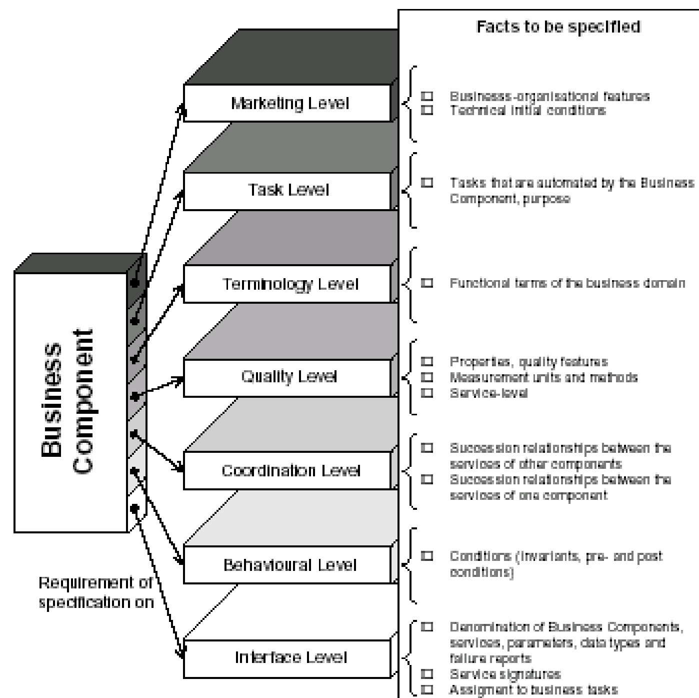

Figure 1. Levels and facts to be specified [4]

The *interface* (or syntactic) *level* contains basic agreements as names of services and data types, signatures of services, as well as the declaration of error messages. The resulting contract guarantees that service client and service donator can technically communicate with each other. Mainly, this layer focuses on the technical and not on the semantic aspects of communication. Besides, this level describes which functional terms that have been introduced in the Terminology Level are related to which data types and which tasks on the service level correspond to which services of the business component.

The specifications at *behavioral level* provide a closer description of the behavior and describe how a given component acts in general or in special cases. As an example, we could define an invariant condition for a business component *Stock*, which says that the reordering quantity has to be higher than the minimum inventory level. On this level, both the provided and required services have to be specified. The Object Constraint Language (OCL), which complements the UML, is an example for a notation to specify behavioral facts. Agreements at *coordination* (or synchronization) *level* regulate the sequence in which services of a business component may be invoked, and synchronization demand between its services. The coordination level is also used to regulate the sequence in which services of *different* business components may be invoked. On this level it is recommended to use the OCL, enhanced by temporal operators. These temporal operators are used to specify the order in which different services can be performed.

On the *quality level* are described non-functional characteristics of business components. Examples for such characteristics are the response time of a service, its availability, its performance or maintenance needs.

The *terminology level* serves as central registry and keeps all used terms and their definitions in a dictionary.

At the *marketing level* specifies features of the business component that are important from a business-organizational point of view, for instance contract terms, version, business domain, or vendor contact persons. As notation predefined tables are used.

Therefore, we can see the OCL specification of a business component, especially at the behaviour level, as an application of the DBC principles regarding the construction of reusable software.

4. A Practical Example

As it has been mentioned before, in the methodical standard proposed by GI, there are two levels (behavioral and coordination) that have to be formally specified using OCL. We will consider the example of the business component "Flights" in order to emphasize the importance of using OCL in the specification of business components. At the interface level, the specification of business component "Flights" is depicted in Figure 2. As a notation for this level, the OMG Interface Definition Language (IDL) was proposed. IDL is suitable for describing the services, data types and error messages mentioned before.

```
interface Flights
	{
		typedef integer FlightNumber;

		struct Airport
		{
			FlightNumber flightNumber;
			String plane;
			String airportName;
			String destination;
```

```
                Integer departureTime;
                Integer arrivalTime;
        }
        sequence <Airport> ListofAirports;
        exception UndefinedAirport();
        Boolean isValidFlight(in FlightNumber flightNumber);
        Boolean isValidAirportName(in string airportName);
        Boolean isValidAirport(in FlightNumber flightNumber, string airportName);
        FlightNumber SearchFlightNumber(in String aiportName, String destination, Integer
        departureTime) raises (UndefinedAirport);
        String SearchOrigin(in  FlightNumber flightNumber) raises
        (UndefinedAirport);
        String SearchDestination(in  FlightNumber flightNumber) raises (UndefinedAirport);
        ListOfAirports SearchAirportWithWildcard(in string airportName);
};
```

Figure 2. "Flights" business component - Specification of the interface level

This business component offers seven services. There are also declared some data types and an exception. Because the extern interface is empty, the business component does not require and use services of another business component. Figure 3 shows the specification written in OCL of the business component on the behavior level. First, the context of the specification must be defined. For instance, the context of the first expression is the whole business component "Flights" and the second expression refers to the service "searchFlightNumber"of the business component. The first expression is an invariant (a constraint that should be true for an object during its complete lifetime) and specifies that each airport which is managed by the business component must have a flight with a flight number greater than zero. Furthermore, the name of each airport must not be empty. The second expression (which is a precondition) specifies that the service "searchFlightNumber" may only be called if an airport and a destination with the given names exist. We can define similar expressions for the other services of the business component.

```
(1)
context Flights
inv  self.ListOfAirports → forAll(a:Airport | a.flightNumber > 0)
self.ListOfAirports → forAll(a:Airport | a.aiportName <> ' ')
(2)
context Flights:: SearchFlightNumber(an:String aiportName, dest: String , dt: Integer): FlightNumber
pre: self.ListOfAirports → exists (a:Airport | a. airportName = an and a.destination = dest)
```

Figure 3. "Flights" business component - Specification of the behaviour level

For the specification of the coordination level we will use OCL, enhanced by temporal operators. In [1] the authors propose an extension of the OCL with temporal operators. Because the proposed notation is just an extension of the OCL, its advantage is a smooth integration of the behaviour and coordination levels. The following temporal operators can be used (A, B are Boolean terms): *sometime_past* A: A was true at one point in the past; *always_past* A: A was always true in the past; A *sometime_since_last* B: A was true sometime in the past since the last time B was true; A *always_since_last* B: A was always true since the last time B was true; *sometime* A: A will be true sometime in the future; *lways* A: A will be true always in the future; A *until* B: A is true until B will be true in the future; A *before* B: A will be true sometime in the future before B will be true in the future; *initially* A: At the initial state A is true.

The example business component "Flights" is simple and its services have no dependencies on other. But its services may be used by other business components, for instance a business component "BookingProcessing" offers a service for booking a seat on a flight. This service may only be called after it is verified that there are seats available

on the plane associated with the flight and that the airport's details are valid. These constraints are specified in Figure 4.

BookingProcessing::DoBooking(b:Booking; f: flightNumber; d:Date;passenger:String)
pre: **sometime_past**
(isValidAirportName (Booking.Flight) and seatsAvailable(f:flightNumber, d:Date))

Fig.4. "Flights" business component - Specification of the coordination level

CONCLUSIONS AND FUTURE WORK

Because one main goal of every program is reliability, that is, correctness and robustness this paper addressed the problem of business components specification in order to assure software reliability. Based on Hoare calculus formal method, Bertrand Meyer developed a method of software engineering called Design by Contract, which applies especially in object-oriented environments. At the same time, we have referenced Turowski and Conrad proposal, which defined seven different contract levels for the specification of business components. In order to emphasize that DBC principles can be applied to the formal specification of the facts proposed by the authors mentioned above, we have presented an example of how to describe, using OCL, the interface, behaviour and coordination level of a business component named "Flights".

One problem regarding OCL, is that, despite the fact that the language was created especially to be easily read and write, practice revealed that it is suitable only for the design phase of software development because people who do not have a special mathematical background find it hard to understand OCL specification.

Starting from this paper's observations, future work will include the design of a object oriented CASE tool capable to support business rules explicit manipulation and to include DBC principles for the specification of the contracts that exist between business components.

REFERENCES

[1] S.Conrad, K.Turowski - Temporal OCL: Meeting Specification Demands for Business Components, Unified Modeling Language: Systems Analysis, Design, and Development Issues. IDEA Group Publishing, 2001

[2] B. Meyer: *Applying Design by Contract*, in *Computer (IEEE)*, vol. 25, no. 10, October 1992

[3] Object Management Group, Object Constraint Language Specification in OMG Unified Modeling Language Specification, June 1999, Chapter 7

[4] Turowski, K., et al. - Standardized Specification of Business Components, Memorandum of the working group Component Oriented Business Application System, February 2002

[5] J. Warmer, A. Kleppe, The Object Constraint Language: Getting Your Models Ready for MDA, Addison Wesley, 2003

[6] J.M.Zaha, A. Albani - Tool based support for teaching formal specification of business components, *Oxford University (ed.)*: Teaching Formal Methods: Practice and Experience, Oxford, 2003

ABOUT THE AUTHORS

Asist. Prof. Anca Ioana Andronescu, PhD Student, Department of Economic Informatics, Academy of Economic Studies, Bucharest, Phone:+40 21 3191900, E-mail: andronescua@ase.ro

Oana Muntean, PhD Student, Education Sales Representative, ORACLE ROMANIA Phone: +40 21 3072791, E-mail: oana.muntean@oracle.com