

# QCCS: A methodology for the development of contract-aware components based on Aspect Oriented Design

Anne-Marie Sassen<sup>1</sup>, Gabriel Amorós<sup>1</sup>, Petr Donth<sup>2</sup>, Kurt Geihs<sup>3</sup>, Jean-Marc Jézéquel<sup>4</sup>, Karine Odent<sup>4</sup>, Noël Plouzeau<sup>4</sup>, Torben Weis<sup>3</sup>

## 1. Introduction

---

QCCS (Quality Controlled Component-based Software development) [1] is an IST project sponsored by the European Commission that will develop a methodology and supporting tools for the creation of components based on contracts and Aspect Oriented Programming (AOP).

Components that have been designed according to the QCCS methodology will have proven properties, which are formally specified in contracts and can therefore be safely applied to build complex systems and services. Also they may be re-used in other situations. Because each component knows its pre- and post-conditions for usage, it is easy to discover situations in which the component is used erroneously. Hence our work results in a methodology and supporting technology which copes with two key problems:

- Enabling the smooth and sound integration of components from multiple independent sources into complex systems and services.
- Supporting the design and creation of new components.

The QCCS project started in November 2000 and will end in March 2003. In this paper we will describe the results achieved in the first half of the project.

## 2. Technical approach

---

The QCCS' methodology will complement other existing methodologies and enhance them. In particular, the methodology will focus on non-functional issues for the specification part; and it will be focused on aspect weaving and transformation for the design side.

QCCS uses software architecture models extensively to support its methodology of model weaving and transformation. These models are based on extensions of the UML metamodel [2]. UML has been chosen as the QCCS standard modeling language because of its widespread use in the industry, its extensibility properties and the strong growth of transformation tools in academic research as well as in industrial tool companies.

While UML is a powerful and widespread modeling language, it is nothing more than a notation and therefore must be used within a software development process. Building such a process for quality controlled component construction is precisely one of the main objectives of the QCCS project and therefore QCCS participants have been carefully examining methodological issues.

---

<sup>1</sup> SchlumbergerSema, Madrid, Spain

<sup>2</sup> KD Software, Prostějov, Czech Republic

<sup>3</sup> Technical University Berlin, Germany

<sup>4</sup> IRISA, Rennes, France

Ideally the process must support both the application developer which uses components and the component developer which builds them. The Catalysis method [3] has been chosen as an initial methodological framework for the project. This choice is based on the following factors.

1. Components put emphasis on interactions between them and their environment. These interactions are structured as protocols (which can sometimes be complex). These protocols must be subject to modelling with the same expressiveness as for instance for types. In other words, one should spend much more time specifying interactions sequences (for instance as acceptable patterns of operation calls). The Catalysis method provides the protocol models with the same possibilities found usually for types only: protocol abstraction and refinement.
2. Components need strong means to define their properties from the user point of view. One of the most important conceptual tools is the *contract* notion [4], which enables components to describe their abilities in extremely precise terms. A well-known example of this technique is a pre/post condition definition on object operations. Catalysis supports this and adds important notions such as guarantees; they specify properties that must be held true during some operation execution (this is different from object invariants, which must be true only at entry and exit of operations).

To be able to extend Catalysis, one needs to go to the metalevel and extend the standard models.

Since we address two different activities (programming *with* quality-controlled components and programming these components), we need two different powerful metamodels:

- one which allows application developers to use and extend components in any application; this metamodel will be an extension of the UML metamodel;
- one which allows component developers to create new components.

In the next section we will describe the first metamodel.

### **3. A metamodel for the use and extension of components**

---

#### **3.1. Input and output contracts**

The interface description of components provides information about the component's functional properties. However, non-functional properties need to be described as well, in order to be able to reuse the component in a predictable manner. These can be described by contracts [5], and therefore the metamodel will be extended with contracts. All requirements of a component need to be made explicit, also the requirements which a component demands of other components in order to be able to carry out its job. For instance, an e-shop component could have an interface featuring the following method:

```
setDataBase( in db : IDataBase ) : boolean
```

Some written documentation that comes along with the component may explain that the application has to invoke this method first before doing anything else. That means one of the component's interfaces expressed the requirement implicitly. To make it explicit, these requirements may also be modelled by contracts, with the little difference that the component does not offer this kind of contracts: it demands them. Hence, we distinguish between output contracts, which are offered by the component to the outer world, and input contracts on which the component relies.

#### **3.2. Contract Relationships**

Components and objects bear different kinds of flexibility. It is generally not possible to derive from a component and to overload some of its methods. However, a component offers means for interface discovery and configuration management through introspection mechanisms. Therefore, a tool can find out at runtime which properties are available and which type they expect; this facility helps the user in configuring the component. Most of these properties can be

altered at runtime, but some are static and have to be set during or before deployment. However, this mechanism does not offer more than a tool supported parameterisation of a component which does neither affect the *offered* nor the *required* interfaces or non-functional properties. To illustrate the inherent problem of this approach we come back to our e-shop component example and we assume that it offers an e-payment interface that is valid only in some situations, for instance only if data exchanges can be safely encrypted over the communication link. Such requirements on a component's environment can be modelled by an input contract. If this required contract cannot be fulfilled then the component should not offer the e-payment interface. This mechanism prevents the developer from making a fundamental design error. Therefore we need an appropriate UML notation to assist designers and design tools in manipulating contracts and contract relationships.

When dealing with quality of service contracts [5] we will discover that a set of contracts may have exclusive-or semantics. That means only one contract can be active at a certain time. These contracts often share common subsets. To ease the modelling we introduced the concept of compound contracts. A compound contract is a composition of other contracts. A composed contract can play two different roles. It is either a required or an offered contract. Another case where compound contracts are useful is the combination of functional and non-functional contracts. For example, some component demands a certain throughput for an SQL interface it is using. By grouping the interface and the non-functional contract in a compound contract, we can express this relationship.

### 3.3. Contract Selection and Negotiation

Sometimes the contracts can be selected at design time. However, in other cases (especially in QoS-enabled applications) contracts depend on dynamic factors and have to be chosen at runtime. For example, a webcast application that transmits a rock concert over the internet may decide to switch to a contract that does not offer any video but acceptable sound when the bandwidth is no longer sufficient. On the other hand, the e-payment contract may be selected at design time because some communication component ensures that there will be an encrypted link between client and server. Since this document does not aim at presenting mechanisms to select and parameterise the contracts at runtime we refer the reader to our *Management Architecture for Quality of Service* (MAQS) in [6] and [7] as an example for dynamic contract negotiation.

### 3.3 UML Metamodel Extensions

We propose an extension to the metamodel of the UML 1.4 beta1 specification that allows to model components and contracts as discussed above [8]. Figure 1 shows our new metaclasses and some excerpts from the UML 1.4 beta1 metamodel.

The biggest change is the addition of *Contract* and its subclasses. A *Contract* has the attribute *isOptional*, which is set to *true* if the developer can decide not to accept the contract. Some contracts however are mandatory for the use of the component, so they are not optional. The *isStatic* attribute determines whether the contract has to be selected before deployment or whether selection is possible at runtime. The *Contract* is a superclass of *Interface* and has a *NonFunctionalContract* subclass. The standard *Interface* element is thus redefined in our model.

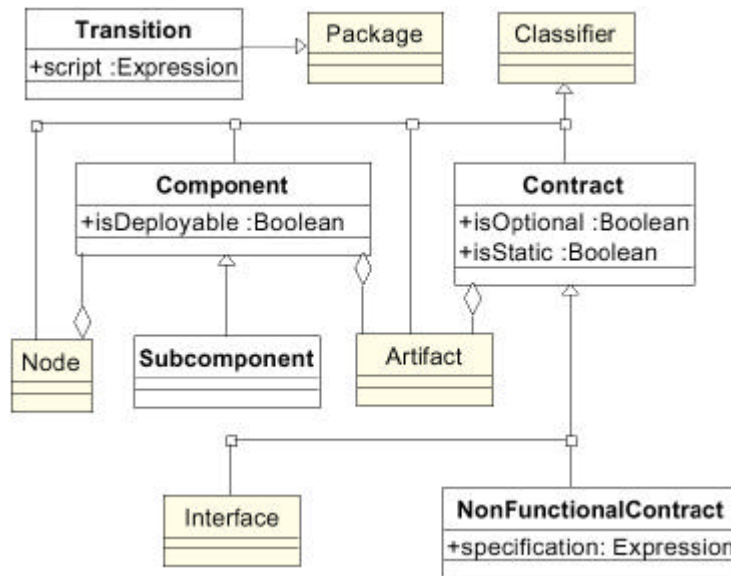


Figure 1. The contract metamodel

- *NonFunctionalContract* instances can be associated with *Interfaces*, which represent functional contracts by grouping them in compound contract as shown in Figure 1. For example, imagine a *NonFunctionalContract* that will deny access to a certain API when the process is running in super-user mode. To model this, one would just draw a *Contract* that has a *Dependency* relationship with the *Interface* and the *NonFunctionalContract*. Connecting the non-functional contract directly with the interface does not work, because that would not allow to model two components requiring the same interface but with different non-functional contracts attached to it.
- *Artifact* is a metaclass that was introduced by UML 1.4 beta1; it is linkable to a contract. In this case, the artifact that represents some physical data will only be installed if its contract has been selected. If the developer decides not to use an optional contract then there is no need to install the data that is used to implement the functionality offered by this contract.
- *Subcomponent* is derived from *Component*, because a subcomponent behaves like a component, but has further restrictions: it cannot be deployed independently. The word *independently* means here independent of its sibling subcomponents. A subcomponent may be connected to its parent component with a *Dependency* relation that has the *trace* stereotype. We now need a way to describe which contract is offered and which contract is required by a component. If a component offers a contract then they are connected by a *Realization* relationship (dashed line with closed arrow). A normal *Dependency* relationship (dashed line with open arrow) between contract and component indicates that the contract is required by this component. The same applies to compound contracts. They can offer and require subcontracts.

### 3.4. Notation

Our metamodel extension introduces some new metaclasses. Consequently, we have to specify their graphical notation. The UML 1.4 already defines the notation for a component but we have altered it slightly. We added the *isDeployable* attribute to components. If this attribute's value is false then the components' names should appear in italic font. This is still compatible with the current notation since a UML 1.4 component is by definition always deployable. *Subcomponents* are displayed like normal components. The only difference is that the top right corner of its rectangle is cut off. The notation for *Contract* is structurally the same as for classes. That means contracts may have compartments, including but not limited to compartments for attributes and operations. The shape of a contract is not a rectangle. Instead, it resembles the

shape of a convoluted sheet of paper like shown in Figure 2. *Interfaces*, which are in our metamodel a specialization of the contract metaclass, are an exception to this rule. For the sake of compatibility, their notation does not change.

### 3.5. Example

Let us look at a complete example using the metamodel. We model a simple *Database* component. Figure 2 shows the specification of the *Database* component as given by the components vendor. The database component called “*Component*” realizes two compound contracts (realization is indicated by the closed arrow). One contract combines a *SQL* interface with the non-functional contracts *Availability* and *Thruput*. The other compound contract combines the *Admin* interface with the *Availability* contract. This small example illustrates that different interfaces can be combined with different sets of non-functional contracts.

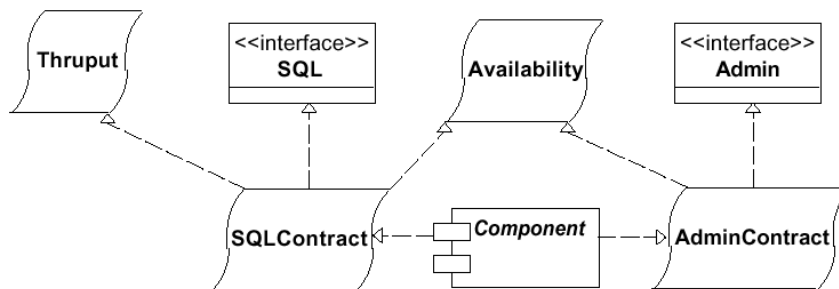


Figure 2. Example of a *database component*

### 3.6 QML

Our model stores quantitative properties of contracts as QML expressions in the specification attribute of the *NonFunctionalContract* metaclass. The full QML definition is given in the QML reference document [9].

### 3.7. Using Aspect Orientation for the realisation of contracts

In Figure 2, we do not see how contracts are realised. We just see which contracts are required by a component, and which are offered, and their dependencies. Contracts are modelling non-functional properties of the components. Normally the implementation of these properties are scattered all over the code of the component. Therefore, a useful way to describe the design of the contracts is by using Aspect Orientation. When the designer wants to link a contract (which is an aspect) to a certain component, it means that a new UML model should be generated in which the aspect is woven into the original design. In order to do this automatically with a transformer, the transformer needs to:

1. know where the aspect-specification touches the design (the so-called join points have to be specified)
2. have a weaving algorithm.

At this moment, we believe that an aspect and its join-points may be specified in almost all cases by standard UML, and only in some cases a small extension to UML is necessary. We are also experimenting with a weaving algorithm based on an algorithm of Ullman [10], which is efficient in most cases. More details about this may be found in [11].

The output of this model transformation is another UML model in which the aspect is woven into the design. The designer can continue with the design in the normal way with his standard UML tools, but, he has been able to benefit from reusing an existing design solution. All he needed to do was select a certain standard contract, adapt it to the specific situation, and specify the join points of the existing design with the specific contract.

Of course, since contracts depend on other contracts, several aspects will be woven into the design, resulting in a complex model. However, this doesn't matter, because if for some reason the design of the system needs to be changed, other contracts can be linked to the components, join points may be specified, and a new (woven) design will be generated.

#### **4. Conclusions and further work**

---

QCCS is aiming at a development methodology for contract-aware components. In order to define such a methodology we have defined a meta-model for the contract notion in UML, and we are currently defining how to model contracts in UML using aspect orientation. A tool will be able to interpret contracts and the join points to an existing UML design, and to generate a result design which possesses the required non-functional property. During the project we want to test our new methodology on a workflow system. We have identified mobility to be an important cross cutting concern in the design of our workflow system. The workflow system is a client-server architecture, along the three tier architectural form. A client application is interacting with a business server that manages all workflow related data. Persistency is managed by a relational database that interacts with the business domain server. Mobility has many possible meanings. In the ideal case, a user can work completely disconnected from the network with full read/write access to his/her data. In a more modest and more realistic case, the user can read all his/her data and write part of it. If there is a mobile network involved, the application may provide more of a "normal" service, perhaps with a degraded throughput and/or response time.

The methodology will also be validated on an eCommerce application. There, the crosscutting concern identified was persistency. For both mobility in the workflow system and persistency in the eCommerce system we have specified a standard contract in QML and the metamodel described above. The next step is to model this contract in UML, and to add join points to the current design of the workflow and e-commerce application.

#### **References:**

1. QCCS website: [www.qccs.org](http://www.qccs.org)
2. UML Revision Task Force, *UML 14. beta*. 2000.
3. The Catalysis Method. [www.trireme.co.uk](http://www.trireme.co.uk)
4. Meyer, B., *Applying Design by Contract*. IEEE Computer Special Issue on Inheritance and Classification, 1992. **25**(10): p. 40-52.
5. Beugnard, A., et al., *Making Components Contract Aware*. IEEE Computer Special Issue on Components, 1999. **13**(7).
6. Becker, C. and K. Geihs. *Generic QoS Support for Corba*. in *ISCC'00*. 2000. Antibes, France
7. Lorcy, S., N. Plouzeau, and J.-M. Jézéquel. *Reifying Quality of Service Contracts for Distributed Software*. in *TOOLS USA*. 1998.
8. Weis, T., C. Becker, K. Geihs and N. Plouzeau. *A UML Meta-model for Contract Aware Components*, Springer LNCS 2185, 2001
9. Frolund, S. and J. Koistinen, *Quality of service Specification in Distributed Object Systems*. Distributed Systems Engineering Journal, 1998. **5**(4).
10. Ullman, J.R. An algorithm for subgraph isomorphism. Journal of the ACM, Vol 23, No.1, Jan 1976, pp 31-42.
11. Jézéquel, J.-M., N. Plouzeau, T. Weis and K. Geihs, *From Contracts to Aspects in UML Designs*. In *Aspect-Oriented Modelling with UML Workshop*, AOSD 2002.