Components have no Interfaces

Richard Rhinelander[*,†]
Department of Computing Science
University of Kitara, Australia
rr@kit.edu

Abstract

While the discussion ”on what is a component” seems to finally come to a (wrong) conclusion, this paper argues heavily against one of the most dramatic misconceptions of this still immature field, i.e., that software components are specified by their interfaces. We explain what problems result from this misconception and we discuss requirements to a component model which avoids the problems discussed. Most interestingly to the rest of the community, such a component model is not an interface model, but much more a model where components are *not* defined by their interfaces.

keyword (selected) predictable assembly of components, performance/efficiency and reliability of component-based systems, systems for the description and prediction of non-functional component properties, deployment attribution / constraints, COP and Model-driven Development (MDA), role of composition frameworks, interoperation among component frameworks, dynamic composition of component-based systems, component-oriented development processes, relating architectural principles/approaches to component software, architecture description languages suitable to guide COP, addressing variability requirements in component-based solutions, system design for independent extensibility, system design for the use of third-party components, component versus application evolution, components in distributed embedded systems, incl. mobile phones and PDAs, domain-specific (vertical) standards, organizational aspects, business aspects, what worked / what didn't work in practice and lessons learned, plus some other keywords, not mentioned here explicitly, but including SOA, metamodelling, UML and various other politically incorrect terms.

[*]Author name in alphabetical order.

[†]The German Research Council would not be happy to hear that this work is supported by it. Therefore, various grant references are suppressed.

1 Introduction

One of the major driving forces and motivation of the whole field of component based software engineering[1] is the fundamental question of ”what is a component”. The discussion was certainly made ”interesting” by views ranging from ”anything but reusable” [HC91] to a principally undefinable entity [CE00]. Particularly damaging is the list of component properties of Clemens Szyperski (most often cited as a definition, hence we do not cite it here) that a component ”is a unit of third party deployment, has contractually specified interfaces and that is has no state”. For teachers, in particular the latter property is still a pain in the neck. However, for the author (of this position statement, not the property list) the second property shows the irresponsibility of the author (of the list, not this paper) much more clearly. Not alone with letting us alone what ”contractually specified” means (a term what was most often used inconsistently and wrongly in the community, until a hallmarking paper [RS02] (by one of the most overrated researchers in the community)); much worse (back again with the other irresponsible author (the one of the property list, not the hallmarking paper), the whole idea that a component has interfaces (what so ever specified) is totally wrong and results in various misconceptions. One of the more severe problems caused was a review comment on a paper of the author (of this position paper, but a review comment was on another paper), that rejected the idea of a novel component model because the reviewer insisted that component model is in fact the same as an interface model. Beyond this dramatic mislead of at least one reviewer, there are various lighter problems of less concern: (in fact, all relating to topics listed in the Call for Papers).

However, as (of course totally by accident) the organisers of this year’s WCOP explicitly asked for position statements on how dark is the component black-box, the author clearly sees a feeling of guilt in the workshop organisers attitude and is willing to forgive and to share his more fundamental insights in the true nature of components. (Attention to reviewers:) The contribution of this paper is the clarification of the relation between components and interfaces, in particular interfaces mistakenly pretending to specify component-QoS-properties plus a sketch of a solution to save the world (”world” in the broader meaning of the word: ”universe”). The remainder of the paper is organised as follows: in section 2, interfaces and components, we (unexpectedly) discuss the terms interfaces and component.[2] In section 3 we conclude that components have no interfaces and conclude. Related

[1]In fact formerly, component phrased software engineering, as the majority of self-declared component experts simply declared objects or classes as components and re-submitted their old concepts. However, we now beat back in declaring components as services.

[2]We intentionally use interface in its plural form while using the term component in its singular, as one component can have several interfaces. (oh shit. In fact, they don’t.)

work (of course of a much lesser depth of thought as the author's work (the author of the this paper, not the ones of the related work)) is presented in section 4. Section 5 than again states the main message of this position paper, but now in a form that you can cut of of the paper and wear it as a sticker (which less humble characters may do to demonstrate their intellectual superiority).

2 Interface(s) and Component(s)

Message number one is: An interface is an abstraction of an encapsulated software unit. In case of a procedure (as software unit) the signature would be the interface, but that is a little pathologic, as interfaces are mostly known as an abstraction of a module (as a software unit), as introduced by Parnas 1972 [Par72]. Argued in an exemplary way, modules defined to encapsulate design decisions. Interfaces are used as the sole way to access modules. The idea is, that the module implementation can change (if the design decision is changed to use an alternative implementation), but the interface is stable. Therefore, it is clear that the interface must be an abstraction of the module. If it were not, than any change of the module would also affect the interface. Although this sounds simple, Parnas also stated that an interface should contain sufficient information to use the module and to implement the module. Interestingly, this is in direct contradiction to the "interface is an abstraction"-idea, as an abstraction intentionally omits information and, hence, may not for all cases provide sufficient information for usage or implementation. However, in any case, the interface is used to describe a module. As a module is always supposed to be in a fixed environment (at least if it is seen as a way to encapsulate design decisions and not as a means for software reuse), one has not to worry whether the module really provides the functionality promised in the interface. However, things are different for components (which are delivered without context and can be deployed in different contexts). Therefore, we continue with:
Message number two is: Interfaces are first class entities. They can exist without any component. (For example, this is the case in domain standards, where interfaces are defined and the semantics of the service [3] signatures listed in the interfaces is described.) As a consequence, between two first class entities (namely components and interfaces) there can be several relationships:

implements: that is what we know from Java. The meaning of "A implements B" is that a class (err, component) A contains the code of the service signatures of the *implemented* interface B.

[3] The user of the term "service" instead of "method" or even "procedure / function" is a concession to the zeitgeist.

depends: that is also easy. The meaning of A requires B means that component A contains code, that depends external services asp specified in interface B.

provided: totally meaningless for isolated components: what functionality a component actually provides depends not only from the component itself, but also from several factors, in particular whether external component which the component depends on are connected and which properties they have. hence, the provided relationship between a component and an interface can only occur in architectural diagrams where a component is put into its context (which includes the wiring to other components).

required: totally meaningless for isolated components: what functionality a component actually requires depends on the functionality one component is used for in a specific context. Without knowing what functionality of a component is actually called, one cannot specify which functionality is really required. Like above, the required-relationship between a component and an interface only occurs in architectural diagrams where a component is put into its context.

The message number three is: What information is actually specified in an interface, is given by the interface model (actually interface meta model). The even mor important massage is: an interface model is not a component model, as, trivially, an interface is not a component and both are independent first-class entities.[4] Various interface model classifications have been published, one of the most often cited is the one from Beugnard et al [BJPW99]. In this classification the authors interestingly omit the dimension of classification and just list classes (of probably increasing complexity): "Syntactic level, behavirouràl level, synchronisation level, QoS level". In fact, it is clear why the classification dimension is omitted, because in fact, it were two dimensions intermixed. Therefore, in a brilliant work[5], Becker et al presented at CBSE 04 a landmarking paper about an interface model classification with *two* dimensions, namely, the level of granularity you are specifying properties for (single services of an interface or the interface itself) and whether you talk on functional or extra-functional properties.[6] While moving from functional signatures to functional protocols (as a behaviour specification you stay constant in

[4]Listen! Listen carefully, thou reviewer of CBSE who rejected one of my papers because this difference was not clear to you. Dopey!

[5]Probably not as brilliant as the paper which got rejected, as the reviewers where still able to grasp the concepts of the paper.

[6]In case you are always confused with a clear borderline between functional and extra-functional properties: functional properties are all those you can specify with a Turing machine, those you cannot, are extra-functional. However, this is an academic statement: I got to know that in consulting and industrial practice Turing machines are sometimes omitted in functional specifications.

the functional-extra-functional axis, but move on in granularity. Differently, when going from behaviour to QoS specifications. Here you move back in granularity but more to extra-functional properties. The basic message however, is, what is specified in an interface is not fixed. Although mist time it are service signatures, it was firstly argued by Nierstrasz in 1993 that a list of service descriptions is insufficient and one needs in addition a specification of valid call sequences [Nie93].[7] In a different dimension you can extent interfaces by specifying the quality of services. In practice, one is mainly interested in performance metrics (such as response time or throughput) or reliability metrics (mean-time-to-failure or availability). To summarise: although not used in Java, there are interfaces which describe not only the signature of services but also some of their quality properties, and this is perfectly ok with the interface concept. (In particular, there is no need to introduce new terms, like ”gates”.)

3 Components and Interfaces

Unlike modules, components are to be used in different contexts. Therefore, they have to make dependencies to the context explicit. Naively, one would think that a component uses one or several interfaces to specify what functionality they expect from the environment (to be implemented). While this is naive even for purely functional interfaces, the whole tragedy comes most clear when considering QoS-specifying interfaces. What QoS of a required service our component should ask for? For example, assume that our QoS-specifying interfaces allow to specify response time. What response time a component C should ask for a specific external service e for? The answer is simple: this depends on what the component's C services (who are using e) promise as a response time to callers. However, this is not the right answer, because what the response time of C's services is also not clear. In fact, it depends from the C callers, hence it is context dependent. Altogether: as long as we do not know, which QoS is expected from C, we cannot specify the ”required” interfaces of C. With the same line of argumentation, we also cannot specify what QoS C will provide, as long as we do not know, which QoS will be provided by the required external service e to our component C. Although not such seductively clear, the argumentation holds for functional properties: should our component C always ask for all external services which are referenced in its code? Of course, it could (and has to be (type-) safe), but that would restrict reusability in all cases, where actually not all of C's implemented functionality is used by callers. In practice, this is often the case and it exactly should be the case. As it should be

[7]Yes, it was Nierstrasz, not Henzinger in 2000. In between even the author argued for specifying component protocols, at least one year before some (suitably chosen) others.

cheaper to reuse a component which implemented actually more functionality than I want to use in a specific context is perfectly ok. And in fact, if such a reuse is made unnecessarily expensive by formally required external services which in fact are never used, the whole component reuse idea gets questionable.

From the above argumentation, we can derive the following two requirements for component models:

A distinction between unbound components and components which are wired into a context. As discussed in the introduction of this paper, the central question of the whole research field of component orientation centers around the question of what is a component.Much confusion exist, as it is usually not differentiated for which level of abstraction the component is defined for. In fact, a clear distinction of component abstraction levels is lacking. From the above discussion on the relation between interfaces and components, we learned that at least two different levels of components exist. The first level is concerned with unbound components which can implement on interfaces or depend on interfaces. The second level are architecturally bound components, which provide and require interfaces. You can refine these two layers of abstraction even further, by distinguishing between the following layers:

provided component type: a component in a coarse grained architecture: you just describe that a component exists which will implement some functionality. As in this level of coarse grained architectural design you are mainly interested in decomposing a system (i.e., identifying components) you should not forced to specify interfaces the component depends on. Hence, implementations of this type can depend on interfaces which are not specified in the provided type.

complete component type: this component already includes a specification of the interfaces it implements and those it depends on. An implementation of this type must not depend on more interfaces as specified in the complete type.

implementation component type: while the words ”implementation” and ”type” sound contradicting, the point is that this type contains additional information which may relate very strongly to its implementation. However, it is still a specification of an unbound component which abstracts from the actual implementation. Which information is specified in this in particular is discussed in the next subsection.

deployed component: This component is deployed. This means it is linked to its resources and the interfaces it references to are itself connected to external components.

run-time component: a lump of code in the memory, we do not discuss any further.

A different way of specification of unbound components than interfaces. As the discussion shows, one has to specify the relation between what a components gets from the environment (as specified by the required interfaces the component is bound to at deployment) and what the component provides (as specified by by the provided interfaces the component is bound to at deployment). Now three different scenarios exist:

1. A component is bound to required interfaces, hence it has to be computed what the component in this context can actually provide (given the required interfaces, what would be ”maximal” provided interfaces, i.e., the interfaces which describe all functionality at the best QoS the component can provide with the required interfaces given.)
2. A component is bound to provided interfaces, hence it has to be computed what the component in this context hast to actually ask for (given the provided interfaces, what would be ”minimal” required interfaces, i.e., the interfaces which describe the least functionality at the worst QoS the component has to ask for to provide the functionality specified by the provided interfaces given.)
3. A mixed case: some provided and some required interfaces are bound. Now the question is what is a suitable combination of provided and required interfaces for the unbound ports, if such a combination exists at all.

Note that in any case, one can ask all questions also for protocol specifying interfaces. (The answer for questions one and two for protocol specifying interfaces was even so academic (not o say: confusing) that it was rewarded with a dissertation in a until than well-regarded south-western German university. [Reu01])

4 Related Work of lesser depth of thought

Omitted to to space restrictions of the paper and time restrictions of the author. After the paper is accepted and some additional pages are granted according to its outstanding importance, the author may consider writing this section.

5 Conclusions

Cut of here and place it in your name badge of the conference. Wearing it during the conference dinner is supported by the SUSIMSSRUC (Society to use superior innovative marketing to support scientific results of unclear character), pronounced (soozie-hmmm-ruk.)

Lessons learned:

1. Components have no interfaces. This is true for any unbound component, and particularly obvious for QoS-specifying interfaces.
2. It is ok to say, that a component implements an interface, if the interface is not specifying any QoS-properties.
3. It is ok to say, that a bound component provides and interface, even if the interface is specifying QoS-properties.
4. It is not ok to say that a component requires an interface, unless the component is bound to its calling component(s).
5. It is not ok to forget to cite parametric contracts.
6. A component model is not an interface model.

Acknowledgements

As this paper did not went through the regular review process of the research group, the reputation of the other members is not affected by the publication. However, they also lack the credit.

References

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[HC91] J. W. Hooper and R. O. Chester. *Software Reuse – Guidelines and Methods*. Plenum Press, New York, NY, USA, 1991.

[Nie93] Oscar Nierstrasz. Regular types for active objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-93)*, volume 28, 10 of *ACM SIGPLAN Notices*, pages 1–15, 1993.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Reu01] Ralf H. Reussner. *Parametrisierte Verträge zur Protokolladaption bei Software-Komponenten*. Logos Verlag, Berlin, 2001.

[RS02] Ralf H. Reussner and Heinz W. Schmidt. Using Parameterised Contracts to Predict Properties of Component Based Software Architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, 2002.