On Certain Basic Properties of Component-Oriented Programming Languages

Position Paper

Peter H. Fröhlich
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
phf@acm.org

Michael Franz
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
franz@uci.edu

ABSTRACT

The essence of component-oriented programming is a new understanding of extensibility in which the development and integration of extensions is a distributed activity, not a centralized one as in previous software development paradigms. Component-oriented programming languages must therefore be designed to have certain basic properties that support rather than impede the distributed extensibility of software systems. We discuss a number of existing language mechanisms in this regard and provide examples from Lag*oo*na, an experimental component-oriented programming language we are developing.

1. INTRODUCTION AND BACKGROUND

In 1986, Nancy Leveson observed the following regarding software safety [12]: "A fair conclusion might be that 'why' is well understood, 'what' is still subject to debate, and 'how' is completely up in the air." Today, the same is true for the paradigm of component-oriented programming: There is broad agreement that "component-oriented programming is good," but there is much less agreement on "what component-oriented programming is" and certainly none on "how to do component-oriented programming." The number of approaches and technologies that have been proposed for component-oriented programming in the past provide ample evidence of this.

With the publication of Douglas McIlroy's classic paper [13] in 1968, software components became a "silver bullet" for software engineering [2] and reappeared regularly through the decades. Components in this "classic" sense are primarily concerned with reuse, either in the form of buying needed components cheaper than their development cost, or in the form of assembling multiple products out of existing components. However, software reuse has been possible (if not practiced) for a long time: In the paradigms of structured, modular, and object-oriented programming, components in this sense took the form of procedures, modules, and classes. If "component-oriented programming" is to mean anything as a software development paradigm, its essence must be something other than reuse. Also, if this essence exists, it should be possible to design component-oriented programming languages to support it.

During our work on Lag*oo*na [6, 8] it became clear that the essence of component-oriented programming is its understanding of *extensibility*. A software system is usually considered extensible only if certain well-defined means for adding functionality have been designed into it.[1] Procedure variables can serve as such well-defined means in the structured and modular paradigms, while subtype polymorphism can be used in the object-oriented paradigm. Although these mechanisms enable the construction of extensible systems, the *process* of developing and integrating extensions is *not* defined by the respective paradigms. Programming languages for these paradigms usually require the developer of an application and an extension to be the same party (or at least to have access to the source code), and restrict the integration of an extension to compile time. Component-oriented programming breaks with this centralized process by mandating that both development and integration of extensions is possible in a *distributed* fashion:

- Any interested party can develop an extension.
- New extensions can be integrated at any time.

These requirements are usually known as *independent extensibility* and *dynamic extensibility* [18], but we prefer the more encompassing term *distributed extensibility*.[2]

This insight is not new, but it has a tendency to get lost among terms such as *architecture*, *component*, *connector*,

[1] Obviously every software system is "extensible" in a very basic sense since we can rewrite arbitrary amounts of it. However, this is neither practical for making extensions, nor useful for classifying software systems.

[2] Note that we do not claim that distributed extensibility is impossible to achieve using the technology of previous paradigms. It is, however, harder to achieve.

configuration, *framework*, *market*, or *middleware* that are used pervasively in the literature [16, 18, 11]. Although we refrain from doing so here, all of these terms can indeed be explained solely with the requirement of distributed extensibility. For example, components and frameworks arise from the need to delineate the extensions present in a system from the core system itself. Focussing on distributed extensibility also proved valuable in making several key design decisions for Lagoona, decisions we believe to be true for other component-oriented programming languages as well. However, while this perspective helps in pointing out "what not to do" and "what still needs to be addressed" when designing a component-oriented programming language, it is not helpful in finding out "how to do it." This is where the language designer's creativity comes into play.

In the following, we first discuss several fundamental design decisions for component-oriented programming languages in section 2 and then focus on a number of specific language mechanisms and their impact on distributed extensibility in section 3. Section 4 concludes the paper with a summary.

2. FUNDAMENTAL DESIGN

The design of a component-oriented programming language starts with three fundamental insights: the need for *interfaces*, the need for *modules*, and the need for *polymorphism*.

An interface is an abstraction of all possible *implementations* that can fill a certain role in the composed system.It thus describes minimal assumptions that frameworks and components can make about each other. Interfaces are essential to component-oriented programming because they are the only form of coordination between frameworks and components and the only means by which compositions can be validated. We can view interfaces as sets of *messages* (abstract operations) and implementations as sets of *methods* (concrete operations). Messages describe *what* effect is achieved by an operation, while methods describe *how* that effect is achieved. Multiple *instances* of an implementation can exist concurrently, and multiple implementations can be part of a component. We say that an implementation (or an instance) *conforms* to an interface if it provides methods for all messages in that interface. In programming languages, interfaces and implementations should be modeled as *interface types* and *implementation types* respectively. In this manner, we can define the conformance of an implementation to an interface by the conformance of the corresponding types.

Modules define the static structure of a system by providing rigid boundaries which can not be crossed arbitrarily. They thus help to isolate frameworks and components from each other, to limit their interaction, and to make dependencies explicit. Modules must be *sealed* [4] to be suitable for this purpose: From outside the module we can neither access members that were not exported explicitly, nor can we add new members retroactively. Polymorphism supports the dynamic structure of a system by allowing different instances of different implementation types to be bound to the same interface type at runtime. Inclusion polymorphism [5] as known from object-oriented languages is one way to achieve this, although we prefer the term *implementation polymorphism* in this context.

Note that these considerations do not restrict us to a particular model of computation. Component-oriented programming languages could be based on an imperative, a functional, or possibly even a logical model. For Lagoona we chose an imperative model in the tradition of Oberon [20], mainly because of our previous experience with it.

3. LANGUAGE MECHANISMS

We now consider a number of language mechanisms and their impact on the distributed extensibility of software systems in detail. Both known facts and open issues will be described to varying extents.

3.1 Modules Revisited

We already discussed the central role of modules in section 2. However, a number of further issues often arise in regard to this basic construct, not the least of which is the confusion of modules and classes. It has been shown that although classes *can* play the role of modules, the two *should* be conceptually different because they serve different purposes [17], and many recent language designs have indeed separated modules from classes.[3] One major reason for this is that modules can package a number of related classes into a single deployable unit.

This in turn raises another question: Since certain components might exceed the complexity that can conceivably be packaged into a single module, should it not be possible to nest modules? Aside from a number of semantic difficulties with hierarchical module systems (or nested classes for that matter), we have to consider what constitutes a deployable unit again. If nested modules are still deployed individually, nesting becomes irrelevant for distributed extensibility. On the other hand, if nested modules are deployed in one "super module," we might have to distribute the same (source-level) modules a number of times because they are part of different components. A flat module space is simpler and has a number of other valuable properties regarding component-oriented programming [19].

Another concern is the identity of components, and therefore that of modules. Distributed extensibility requires that the presence of a particular extension in a system can not preclude the presence of any other extension.[4] Two otherwise unrelated modules must therefore never have the same name, they must have unique identities. Since no form of "unique identity" can be achieved without some convention, our goal should be to make the conventions as unintrusive and transparent as possible. Microsoft's COM [14] uses randomly generated identifiers for this purpose, but these are hardly transparent. For Lagoona, we have adopted a convention similar to that originally proposed for Java: module names are prefixed with "inverted" Internet domain names, such as `net.lagoona.base.Stack`. Although not enforcable, we believe that this convention is a good tradeoff, especially when coupled with an import declaration that can introduce abbreviations.

[3] We will encounter this idea of separating concerns in language design a number of times in the following.

[4] The exceptions to this rule are of dynamic nature and concern invariants the system needs to maintain in order to function properly, for example when the extensions are device drivers of an operating system.

3.2 Interface Conflicts

Components often need to conform to *multiple* interfaces. Consider a component that presents the results of a database query within a compound document. Instances of this component have to react to notifications from the database *and* the compound document framework to keep their presentation current. Since these framework could have been developed by different organizations, conforming to both of their interfaces must not lead to conflicts.

However, if we use existing object-oriented technology to achieve polymorphism, such conflicts will sooner or later occur. The reason for this is that messages are bound to interface types and therefore only have a unique identity there. When two interface types are *combined* as in the scenario sketched above, syntactic and semantics conflicts can arise between messages with the same name.

In Lagoona, we have introduced the concept of *stand-alone messages* that are bound to modules instead of types [9]. Since modules have a unique identity already, messages in Lagoona are also always unique. Therefore, *any* combination of interface types results in a legal interface type and also preserves all constituent messages. Other programming languages provide mechanisms for *resolving* interface conflicts once they occur, however this is not useful if we want to achieve distributed extensibility. Lagoona avoids interface conflicts by design instead.

Stand-alone messages lead to an interesting insight regarding language design if we consider the design space for the identity of messages and methods (as defined in section 1 above) relative to modules and types: Binding both messages and methods to types leads to object-oriented languages, while binding both to modules leads to modular languages. Component-oriented languages require that messages are bound to modules to avoid conflicts, while methods remain bound to types to support polymorphism.

3.3 Structural Conformance

For the design of a component-oriented programming language, the need of components to support multiple interfaces also has implications for the *conformance* of two types. In the scenario outlined above, assume that database notifications are described by an interface A, while document notifications are described by an interface B. Consider the consequences of someone introducing a new interface type AB that combines both interfaces. If we use *declared conformance* between types, a component that supports interfaces A and B is *not* compatible with the new interface AB, even though it describes identical requirements. For this reason, a component-oriented programming language must offer *structural conformance* between types [3], and this is indeed what we alluded to in section 1.

Structural conformance is often seen as "weaker" than declared conformance, because it can result in "accidental" conformance relations that the programmer did not anticipate. However, in a language that supports stand-alone messages, accidental conformance is not possible: Even messages with identical names and signatures are distinct in such a language, because they were defined in different modules. In Lagoona, we can thus safely support structural conformance as required by the component-oriented paradigm, without any of the drawbacks usually associated with this.

3.4 Inheritance and Forwarding

The concept of *inheritance* in object-oriented programming was once hailed as the "golden way" towards extensible software systems. However, the mechanism is generally not suitable for achieving distributed extensibility.[5] Assume a container class A that supports operations *Add* for adding an element, *Rem* for removing an element, as well as $MulRem$ for removing *several* elements at once. We want to define a derived class B that also supports queries about the number of elements currently in the container. However, we can not implement B without knowing the implementation details of A as well: If the developer of A implements $MulRem$ by calling *Rem* repeatedly, we have to override *Rem* to maintain an accurate count; if $MulRem$ does not call *Rem*, we have to override $MulRem$ instead. This is known as the *fragile baseclass problem* in the literature, and it can be avoided by following an elaborate set of design conventions [15]. If we want to avoid it altogether, we have to restrict the use of inheritance or abolish the construct completely.

In Lagoona, we have chosen the latter option and replaced inheritance with a *generic forwarding mechanism*. When an object (instance of an implementation type) receives a message, and a corresponding method implementing this message exists, that method is executed. If no matching method is found, but a special *default method* is implemented, that method is executed instead. Inside a default method we can generically *resend* the message to other instances. If neither a matching method nor a default method exists, execution is aborted with an exception. It is easy to see how to resolve the problem in our example using this mechanism. Instead of deriving a new class B, we develop an implementation type B that has a reference to an A instance. We implement the methods corresponding to the messages *Add*, *Rem*, and $MulRem$ by first maintaining our count and then sending the message to the A instance. We also implement a default method to forward all other messages to A.

There are, however, certain problems with this approach. First, it is hard to make guarantees about a particular message send since an empty default method can be used to ignore all messages an object does not explicitly implement. We currently resolve this by offering two versions of message sends to the developer, one that is strict and requires a corresponding method in the receiver, and one that is non-strict and makes no guarantees that a message is ever handled. While this at least makes the decision explicit, it is not entirely satisfactory. Second, forwarding all received messages to another object in the default method can result in a "sudden acquisition of functionality" that was not originally intended. This issue is currently not resolved in Lagoona. Finally, there are certain useful applications of inheritance that are not easily expressed through forwarding, in particular when we define hierarchical data structures like abstract syntax trees. While a final decision for Lagoona is still pending, we are considering to allow inheritance *within*

[5] In fact, the example of inheritance caught our attention on the website announcing this workshop. We believed the issue to be settled...

a module, but not *across* module boundaries where it becomes a problem for distributed extensibility. We hope to resolve these problems in the near future.

3.5 Implementation Aspects

We would like to briefly point out three important implementation aspects of component-oriented programming languages: portability, efficiency, and safety. Components (and therefore modules) should be deployed in a "binary" form which allows their integration with a minimum of intervention. However, if the developer of a component has to provide numerous "binary" versions for several different platforms, the resulting management overhead can become a serious problem. Therefore, the "binary" form should be a *portable* intermediate representation. Instead of virtual instruction sets as used by the Java Virtual Machine or the .NET architecture, we propose the use of *abstract syntax trees* in the form of *slim binaries* [7]. Such a format has a number of properties that make it more suitable to optimization in the context of dynamic compilation, and thus leads to portable *and efficient* execution. It is also suitable for *dynamic optimization*, which is particularly relevant to component-oriented programming [10]: While we have to deploy frameworks and components as well-encapsulated entities in this setting, nothing prevents us from "tearing down" these barries at runtime in order to perform optimizations such as program specialization or inlining that would otherwise not be possible. Finally, abstract syntax trees have proven to be useful for establishing certain safety properties that are as important for component-oriented programming as they are for mobile-code systems where we first studied them [1].

4. CONCLUSIONS

We have argued that the essence of component-oriented programming lies in a new understanding of the notion of extensibility as it applies to software systems. Previous paradigms were only concerned with making systems extensible *in principle* and implied the existence of a *central* authority managing all extensions. Component-oriented programming *requires* that extensions can be developed and integrated in a *distributed* fashion, without any central authority. We have shown that several basic properties of component-oriented programming languages can be derived solely from this understanding of extensibility. The fundamental properties implied by distributed extensibility are the need for interfaces, modules, and polymorphism. However, there is still a large number of individual choices to be made within this framework. We discussed modules, stand-alone messages, structural conformance, and inheritance in more detail. Space limitations forced us to ignore the issues of representation exposure, configuration management, and advanced type systems, all of which are of importance for component-oriented programming.

Acknowledgments

We would like to thank Kimberly Haas and the anonymous referees for valuable comments on earlier versions of this paper. We are also indebted to Clemens Szyperski for a number of interesting discussions and to the workshop organizers for being extraordinarily patient. This work was partially supported by the National Science Foundation under grants EIA-9975053 and CCR-0105710.

5. REFERENCES

[1] W. Amme, N. Dalton, M. Franz, P. H. Fröhlich, V. Haldar, P. S. Housel, J. v. Ronne, C. H. Stork, and S. Zhenonchin. Project transPROse: Reconciling Mobile-Code Security With Execution Efficiency. In *Proceedings of the DARPA Information Survivability Conference and Exhibition*, pages II.196–II.210, Anaheim, CA, June 2001.

[2] F. P. Brooks, Jr. No Silver Bullet: Essence and Accidents in Software Engineering. *IEEE Computer*, 20(4):10–19, Apr. 1987.

[3] M. Büchi and W. Weck. Compound Types for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 362–373, Vancouver, British Columbia, Oct. 1998.

[4] L. Cardelli. Typeful Programming. SRC Research Report 45, Digital Systems Research Center, May 1989.

[5] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, Dec. 1985.

[6] M. Franz. The Programming Language Lagoona: A Fresh Look at Object-Orientation. *Software: Concepts and Tools*, 18(1):14–26, Mar. 1997.

[7] M. Franz and T. Kistler. Slim Binaries. *Communications of the ACM*, 40(12):87–94, Dec. 1997.

[8] P. H. Fröhlich and M. Franz. The Programming Language Lagoona. Technical report, Department of Information and Computer Science, University of California, Irvine. Forthcoming.

[9] P. H. Fröhlich and M. Franz. Stand-Alone Messages: A Step Towards Component-Oriented Programming Languages. In *Proceedings of the Joint Modular Languages Conference*, pages 90–103, Zürich, Switzerland, Sept. 2000.

[10] T. Kistler and M. Franz. Continuous Program Optimization: Design and Evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001.

[11] G. T. Leavens and M. Sitaraman, editors. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[12] N. G. Leveson. Software Safety: Why, What, and How. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[13] M. D. McIlroy. Mass-Produced Software Components. In *Proceedings of the NATO Software Engineering Conference*, pages 138–155, Brussels, Belgium, Oct. 1968.

[14] Microsoft Corporation. *The Component Object Model (Version 0.9)*, Oct. 1995.

[15] L. Mikhajlov and E. Sekerinski. The Fragile Base Class Problem and Its Solution. Technical Report 117, Turku Centre for Computer Science, Turku, Finland, June 1997.

[16] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[17] C. Szyperski. Import is not Inheritance—Why we need both: Modules and Classes. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 19–32, Utrecht, The Netherlands, June 1992.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[19] C. Szyperski. Modules and Components: Rivals or Partners. In L. Böszörmeny, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth*. Morgan-Kaufmann, 2000.

[20] N. Wirth and J. Gutknecht. *Project Oberon: The Design and Implementation of an Operating System and Compiler*. Addison-Wesley, 1992.