Component-Oriented Languages: Messages vs. Methods, Modules vs. Types

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

Abstract Programming languages should support the paradigm of component-oriented software development. Component-oriented programming languages must explicitly distinguish messages vs. methods and modules vs. types. Most object-oriented languages unify these concepts and are therefore unsuitable for component-oriented programming.

Position Statement Software development paradigms influence programming language design and vice versa [1]. For example, structured, modular, and object-oriented programming are often associated with Pascal, Ada, and Smalltalk respectively. Although these associations are not essential, languages that directly support the basic concepts of a paradigm usually lead to more efficient and more maintainable solutions. However, current approaches to component-oriented programming (COP) [2, 3, 4] are largely based on object-oriented design patterns with very little language support. This situation is comparable to practicing object-oriented programming in Pascal. We believe that history will repeat itself: Drawbacks of existing languages will enourage language designers to support concepts essential to COP, leading to *component-oriented programming languages* (COPLs).

Three concepts essential to COP are polymorphism, modularity, and independent extensibility [5]. Polymorphism allows components to be dynamically substituted for each other. Modularity isolates components from their environment except for explicit dependencies. Independent extensibility means that independently developed extensions can be combined without leading to errors or conflicts in the component model or language.[1]

Let us clarify some additional terminology: A *message* is an abstract operation, while a *method* is a concrete operation. Messages specify *what* effect is achieved, while methods specify *how* an effect is achieved. In COP, a set of messages is a component *interface* while a set of methods is a component *implementation*. A *module* is a syntactic grouping of declarations describing the *static* structure of a software system [5], while a *type* serves a similar purpose regarding a system's *dynamic* structure [6]. In contrast to modules, types are usually constructed out of more primitive types.

A language supporting polymorphism and modularity is easily defined. For example, we could add a module construct to Java.[2] However, this language does not support independent extensibility. If we model interfaces using the `interface` construct and implementations using the `class` construct, subtle conflicts can occur when a single implementation requires multiple interfaces (see Figure 1). Since messages have a unique identity only *within* a type, messages with identical identifiers (id) loose their identity in the combined type. If the signatures (sig) of two such messages can not be disambiguated, a syntactic conflict occurs. If their signatures are identical, it may not be possible to write a method conformant to both messages, causing a semantic conflict.

	sig(a) $\neq$ sig(b)	**sig(a) $=$ sig(b)**
id(a) $\neq$ id(b)	No conflict	No conflict
id(a) $=$ id(b)	Syntactic conflict	Semantic conflict

Figure 1: Conflicts Between Messages

To avoid these conflicts, COPLs must *explicitly* distinguish messages vs. methods and modules vs. types [7]. In particular, messages must have unique identities within modules to retain their identities across all interfaces in which they participate. Methods, however, must remain subordinate to types for polymorphism to work as required. Only two quadrants of the design space illustrated in Figure 2 have been explored, but both are unsuitable for COP.

	Message $\in$ Type	**Message $\in$ Module**
Method $\in$ Type	Object-Oriented	*Component-Oriented*
Method $\in$ Module	Useless?	Modular

Figure 2: Language Design Space

Using *separate* language constructs for messages, methods, modules, and types seems to be the simplest approach to design a COPL. We are currently developing an experimental programming language—code-named Lag*oo*na—based on the insights described here.

References

1. C. Ghezzi and M. Jazayeri: *Programming Language Concepts*. 2nd edition, Wiley & Sons, New York, NY, 1987.
2. Microsoft Corporation: *The Component Object Model 0.9*. July 1995.
3. Sun Microsystems: *The JavaBeans Specification 1.01*. July 1997.
4. Object Management Group: *The Common Object Request Broker: Architecture and Specification 2.3.1*. October 1999.
5. C. Szyperski: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Harlow, England, 1998.
6. L. Cardelli: *Type Systems*. The Computer Science and Engineering Handbook, CRC Press, Chapter 103, 1997.
7. P. H. Fröhlich and M. Franz: *Component-Oriented Programming in Object-Oriented Languages*. Technical Report 99-49, Department of Information and Computer Science, University of California, Irvine, CA, October 1999. Revised December 1999.

[1] Application-level errors or conflicts can *not* be ruled out completely.

[2] Java's `package` construct is not closed and thus unsuitable [5].