Component-Oriented Programming Languages: Messages vs. Methods, Modules vs. Types
(Extended Abstract)

Peter H. Fröhlich

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

Abstract. We are concerned with the design of programming languages that support the paradigm of component-oriented programming. Languages based on the accepted idea of combining modular and object-oriented concepts fail to provide adequate support. We argue that messages should be separated from methods to address this shortcoming. We introduce the concept of stand-alone messages, give examples for its utility, and compare it to related approaches and language constructs. Besides leading to interesting insights on the interaction of modular and object-oriented concepts, we believe that stand-alone messages also provide a useful basis for further research on component-oriented programming languages.

1 Introduction

Component-oriented programming replaces monolithic software systems with reusable *software components* and hierarchical *component frameworks* [16]. Components extend the capabilities of frameworks, while frameworks provide execution environments for components. Both are developed by independent and mutually unaware vendors for late composition by third parties. Late composition requires that component-oriented software systems support *dynamic* and *independent* extensibility. Dynamic extensibility enables the addition of new components at run-time, while independent extensibility allows components and frameworks from mutually unaware vendors to be composed.

The minimal assumptions that frameworks and components make about each other are specified using *interfaces*. An interface is an abstraction of all possible *implementations* that can fill a certain role in the composed system. We 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. We say that an implementation (or an instance) *conforms* to an interface if it provides methods for all messages in that interface. Interfaces are essential to component-oriented programming because they are the only form of coordination between component and framework vendors and the only means by which third parties can validate compositions.

A component-oriented programming language needs constructs to express interfaces and implementations and must also support dynamic and independent extensibility. 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. Dynamic extensibility requires some form of polymorphism that allows different instances of implementation types to be bound to the same interface types at run-time. Inclusion polymorphism [4] in object-oriented languages such as Java [7] is one way to achieve this, although we prefer the term *implementation polymorphism* in this context. Independent extensibility requires some form of encapsulation that isolates components from their environment except for explicitly declared dependencies. Sealed encapsulation constructs [2] in modular languages such as Oberon [12] are one way to achieve this.Therefore, combining concepts from modular and object-oriented languages should be a viable approach to the design of component-oriented programming languages [16].

However, simply *embedding* object-oriented concepts into a modular language unchanged is insufficient. If a component has to implement multiple interfaces defined by independent frameworks, syntactic and semantic interface conflicts can occur. These conflicts preclude framework combination and thus violate the principle of independent extensibility. To avoid these conflicts, messages must be given unique identities *independent* of the types in which they participate. This contradicts the object-oriented paradigm in which messages only have unique identities *within* a type. We propose the concept of *stand-alone messages* and discuss its ramifications for language design. In particular, we show that stand-alone messages simplify the integration of other desirable properties such as structural conformance. Separating the concepts of messages, methods, modules, and types opens a previously unexplored region of the design space for programming languages that seems well-suited for component-oriented programming.

In the next section we illustrate the problem of interface conflicts in a Java-like language. In Sect. 3 we develop the concept of stand-alone messages and show how it resolves interface conflicts. Section 4 discusses additional applications of stand-alone messages, and Sect. 5 surveys related work and compares it to our approach. Finally, in Sect. 6 we conclude with a summary of contributions and an outline for future work.

2 Interface Conflicts

As a simple example for the problems caused by framework combination, we attempt to develop a Stack component that is usable across four different frameworks. We assume a Java-like programming language in which (closed) packages have been replaced by (sealed) modules. Mapping interface types to interfaces and implementation types to classes is appropriate in such a language. The first framework defines the following interface:

```
module edu.uci.framework {
  public interface Stack {
    public void push (Object o);   // pre o ≠ null; post top = o
    public void pop ();            // pre ¬ empty
```

```
    public Object top ();          // pre ¬ empty; post result ≠ null
    public boolean empty ();       // "no elements?"
  }
}
```

The designer of this interface followed the textbook definition of the abstract data type Stack closely, and developing a class that implements this interface is straightforward. The second framework defines the following interface:

```
module gov.nsa.framework {
  public interface Stack {
    public void push (Object o);   // pre o ≠ null; post top = o
    public void pop ();            // pre size > 0
    public Object top ();          // pre size > 0; post result ≠ null
    public int size ();            // post result ≥ 0
  }
}
```

Instead of relying on an empty message, this vendor chose to work with the size of the Stack. To support this interface as well, we have to add a size method to our class which is again straightforward. We can even define the empty method in terms of the new size method to avoid some redundancy. Note that this relies on empty and size = 0 expressing identical semantics. The third framework defines the following interface:

```
module com.sun.framework {
  public interface Stack {
    public void push (Object o);   // pre o ≠ null; post top = o
    public Object pop ();          // pre ¬ empty; post result ≠ null
    public boolean empty ();       // "no elements?"
  }
}
```

Apart from simply removing the top element, the pop message in this interface also returns the top element. To support this interface as well, our class would have to implement two pop methods with *different* signatures.[1] However, since the signatures differ only in their return types, Java's overloading mechanism does not allow us to do this. We have just encountered an example of a *syntactic conflict* between two interfaces. In our Java-like language, it is not possible to express a class that implements this third interface in addition to the first two. The fourth and final framework defines the following interface:

```
module org.cthulhu.framework {
  public interface Stack {
    public void push (Object o);   // pre o ≠ null; post top = o
    public void pop ();            // pre ¬ empty
    public Object top ();          // pre ¬ empty; post result ≠ null
    public boolean empty ();       // "no elements?"
    public int size ();            // "how many pushes?"
  }
}
```

[1] Unlike the Java language specification [7], we distinguish the name of a message from its signature (the list of parameter types + the return type).

This interface is identical to the first interface, except for the additional size message. Unlike the size message in the second interface, this one returns the *number of remaining push operations* until some expensive internal restructuring occurs.[2] To support this interface as well, our class would have to implement two *different* size methods with *identical* signatures. However, since the signatures are identical, it is not possible to distinguish these messages. We have just encountered an example of a *semantic conflict* between two interfaces. In our Java-like language, it is not possible to express a class that implements this fourth interface in addition to the first two.

3 Stand-Alone Messages

In the Java-like language from Sect. 2, messages are declared within interfaces, while methods are declared within classes. Consequently, the identity of a message is *relative to* the interface in which it is declared, whereas the identity of a method is *relative to* the class in which it is declared. In the case of methods, this form of identity is needed to support polymorphism. Consider the following example:

```
...
edu.uci.framework.Stack stack = null;
...
stack = new edu.uci.components.ArrayedStack(16);
stack.push(new Integer(1));
...
stack = new edu.uci.components.LinkedStack();
stack.push(new Integer(1));
...
```

After we bind an instance of ArrayedStack to the reference stack, we expect the message push to invoke the *specific* push method declared for ArrayedStack. Similarly, after we rebind an instance of LinkedStack to the reference stack, we expect the *same* message push invoke a *different* push method declared for LinkedStack. Whenever the class of the instance bound to the stack reference changes, we want the identity of the methods invoked through that reference to change as well. In the case of messages, however, this form of identity is the reason for the interface conflicts described in Sect. 2. Since the identity of a message is only unique *within* an interface, combining two interfaces can result in two messages that are *not* unique within the combined interface anymore.

In order to avoid interface conflicts, we must break the symmetry between message and methods, both of which only have a unique identity within the *type* (interface, class) in which they are declared. Since methods must keep their relative identity to make polymorphism work, the only option is to decouple the identity of messages from interfaces. If messages should not have a relative identity to types anymore, the only reasonable scope in which they could be declared is that of the module. We call messages that have a unique identity relative to their declaring module *stand-alone messages*. The following example suggests a syntax for stand-alone messages in our Java-like language:

[2] This information might be necessary in a framework with real-time constraints. Implementations based on incrementally growing arrays can supply it quite naturally.

```
module edu.uci.framework {
  public message void push (Object o);   // pre o ≠ null; post top = o
  public message void pop ();            // pre ¬ empty
  public message Object top ();          // pre ¬ empty; post result ≠ null
  public message boolean empty ();       // "no elements?"
  public interface Stack { push, pop, top, empty }
}
```

This example shows how the first interface from Sect. 2 is declared using stand-alone messages. In particular, the last line of this example declares an interface *type* that consists of the four messages push, pop, top, and empty. However, note that this is very different from the original form of declaring an interface. In an external module that imports edu.uci.framework, the type edu.uci.framework.Stack would actually appear as follows:

```
interface edu.uci.framework.Stack {
  edu.uci.framework.push, edu.uci.framework.pop,
  edu.uci.framework.top, edu.uci.framework.empty
}
```

This implies that messages always have to be *fully* qualified in external modules:

```
...
edu.uci.framework.Stack stack = null;
...
stack = new edu.uci.components.ArrayedStack(16);
stack.edu.uci.framework.push(new Integer(1));
...
```

To avoid excessive qualifications, we introduce an aliasing construct for import declarations as found in Oberon [12]:

```
module com.factorial.cool.extension {
  import f1 = edu.uci.framework;
  public class CoolStack implements f1.Stack {
    ...
    public void f1.push (Object o) { ... }
    public void f1.pop () { ... }
    public Object f1.top () { ... }
    public boolean f1.empty () { ... }
  }
}
```

To adapt this class to support all interfaces described in Sect. 2 we must import the relevant modules and declare a method for each message required. Since messages are always fully qualified, no interface conflicts can result. Note that we can also add a mechanism that allows component vendors to associate a single method with a number of messages to avoid some redundancy, especially a large number of forwarding methods.

Besides being useful in a pragmatic way, stand-alone messages also lead to an interesting insight regarding language design. Consider the design space for the identity of messages and methods in programming languages. As illustrated in Table 1, both can have identities relative to either modules or types. In object-oriented programming languages such as Java [7], the identities of messages and methods are relative to types. As we have seen, this design choice does not support independent extensibility because of interface conflicts. In modular programming languages such as Oberon [12],

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

Table 1. Language design space for messages and methods.

the identities of messages and methods (procedure headers and procedure bodies) are relative to modules. While this design choice supports independent extensibility, it does not support dynamic extensibility because modules lack run-time polymorpism.[3] Using identities relative to types for messages and relative to modules for methods combines both of these drawbacks and also does not yield a practical design.

Stand-alone messages, however, lead to language designs in which the identity of messages is relative to modules, while the identity of methods is relative to types. Thus, they support both dynamic and independent extensibility and open a previously unexplored region in the design space for programming languages. We believe that this region is well-suited for component-oriented programming, and that stand-alone messages clarify the relationship between component-oriented programming and modular and object-oriented concepts.

4 Additional Applications

Interface Combination. Since the identity of stand-alone messages is relative to modules instead of types, languages that support stand-alone messages have two useful properties regarding the *combination* of interface types. First, *any combination of interface types results in an interface type*. Second, *any combination of interface types preserves all constituent messages*. As shown in Sect. 2, these properties do not hold in Java [7], leading to syntactic and semantic interface conflicts respectively. C++ [14] and Eiffel [9] require additional language mechanisms to approximate both properties (see Sect. 5).

Structural Conformance. Conformance of an implementation type A to an interface type B can either be *declared* explicitly, as in Java [7], or *inferred* based on a *structural* property, such as A providing methods for all messages of B. Structural conformance has a number of advantages, especially for software evolution [8]. More importantly, a

[3] Some modular programming languages *do* support polymorphism at compile-time or link-time. In Modula-3, for example, multiple modules can export the same interface [3]. The decision about which implementation to use is deferred until build-time.

certain degree of structural conformance is *required* for component-oriented programming [1]. However, structural conformance is often seen as being "weaker" than declared conformance, because it can result in "accidental" conformance relations that the programmer did not anticipate. A typical example of this problem is an interface type Cowboy that includes a message draw and an interface type Shape that also includes a draw message, presumably with different semantics. In a language that supports stand-alone messages, accidental conformance of this kind is not possible. The draw messages would be defined in different modules and would therefore be distinguishable. The use of structural conformance has been proposed before. In Modula-3 [3] structural conformance is used by default, but reference types can be *branded* to avoid accidental conformance. However, all brands in a composed system (a "program" in Modula-3) must be unique, which can restrict independent extensibility by mutually unaware vendors. The *compound types* proposal for Java [1] uses declared conformance for individual interfaces and structural conformance for combined interfaces. Although backward compatible with Java, compound types add additional rules to an already complex language and do not address the problem of interface conflicts at all. Another proposal for Java [8] requires that interfaces for which structural conformance should be used must extend an explicit marker interface Structural. In contrast to these approaches, structural conformance with stand-alone messages does not require any additional language constructs to avoid accidental conformance.

Minimal Signatures. An interesting application of structural conformance is that signatures of messages can be typed in a "minimal" way to express certain invariants. Consider a method that prints the top element of a Stack:

```
...
import f = edu.uci.framework;
...
// does not modify "s"
void printTop (f.Stack s) {
  if !s.f.empty() { print(s.f.top()); }
}
...
```

Instead of stating that printTop does not modify the Stack in a comment, we could add anonymous interfaces to our language and define its signature as follows:

```
void printTop (interface {f.empty,f.top} s)
```

Given this signature, only the empty and top messages could be sent to s, ensuring that printTop does not modify the stack.[4] While not providing a complete solution, this form of minimal signature specification can be used to address a subset of component re-entrance problems [11].

[4] This only holds if printTop can not cast the parameter to another type that exposes more messages.

5 Discussion

Component Models. Microsoft's COM is the component model that is most similar to our approach [10]. Instead of assigning unique identities to messages, COM assigns unique identities to interface types. Instead of relying on a transparent naming convention for modules, COM associates an automatically generated *globally unique identifier* (GUID) with each interface type. Contrary to most object-oriented programming languages, COM allows an implementation type to conform to multiple interface types *without* any conflicts. Combined interface types can also be expressed using COM's *category* mechanism. While we emphasize explicit programming language support and the associated advantages, the two approaches are equivalent as far as interface conflicts are concerned. In particular, we could map stand-alone messages to singleton COM interfaces and interface types to COM categories.

Design Patterns. Certain design patterns can be used to resolve interface conflicts [6]. In a variation of the Command pattern, "messages" are modelled as a hierarchy of classes containing "parameter slots," while "message sends" are calls to a universal dispatch method. The dispatch method performs explicit run-time type-tests and calls the actual method corresponding to the dynamic type of the "message." This approach relies on the compiler to generate unique type descriptors for each class and thus prevents any conflicts between messages. However, static type-checking is not possible to the desirable extent.Variations of the Adapter, Bridge, and Proxy patterns can be used to map multiple conflicting interface types to a single implementation type. The idea is to insert additional forwarding classes between clients of an interface type and its implementation type. Messages sent to the forwarding class are routed to the corresponding method in the implementation. While this approach preserves static type-checking, it can be tedious to write the required forwarding classes without tool support.

Renaming Messages. In Eiffel, features inherited from ancestor classes can be *renamed* in a descendant class to avoid name clashes [9]. In our terminology, an implementation type conforming to multiple interface types can explicitly choose new local names for conflicting messages. Note that clients still use the messages declared in the original interface type, but the messages are "rerouted" in a way similar to the Adapter design pattern described above. Although renaming can be used to resolve interface conflicts, the approach has two major drawbacks. First, renaming clutters up the name space of the implementation type. We may have to invent a new name for a message that is less expressive than the original one, define naming conventions to keep readability up, and repeat this "renaming excercise" whenever we want to conform to an additional interface type. Second, renaming must be extended to combined interface types in addition to implementation types. This becomes particularly clumsy in terms of syntax if we also want to support anonymous interface types.

Explicit Qualification. C++ supports the explicit qualification of member functions by classes to avoid name clashes [14]. In our terminology, message sends can be qualified by the implementation type in which a method should be invoked. As defined in C++, this mechanism does not support implementation polymorphism as required for component-oriented programming. However, we can generalize the idea of explicit qualification by allowing message sends to be qualified by interface types. Although this does not restrict polymorphism anymore, even a qualified message of the form Stack.pop is not necessarily unique, since multiple interface types with identical names could exist. Therefore, qualification must be extended to include module names as well, at which point the mechanism becomes equivalent to stand-alone messages, except for the redundant interface type.

Overloading Messages. Overloading is a form of ad-hoc polymorphism [4] supported by a number of programming languages such as Java [7] and C++ [14]. In our terminology, overloading essentially encodes parts of the signature of a message within its name and uses contextual information available when a message is sent to determine which *actual* message is being referred to. Although overloading helps to avoid some interface conflicts, it has two major limitations. First, semantic conflicts can not be avoided by overloading since the semantics of a message can not be expressed by type systems in which type checking is decidable. Second, avoiding *all* syntactic interface conflicts requires *all* combinations of parameter and return types to be distinct. This is not generally possible in the presence of subtyping and the coercions it implies.

6 Conclusions

We believe that the main contribution of this work is an improved understanding of how modular and object-oriented concepts interact and how they can be combined to support component-oriented programming. Our insight that messages should be separated from methods can be viewed as another step towards the separation of concepts subsumed by classes in traditional object-oriented languages. Previous results in this direction include the separation of interface types from implementation types [13] and the separation of modules from types [15], both of which are now widely accepted. We believe that the concept of stand-alone messages will be useful as a basis for further research on component-oriented programming languages.

We plan to continue our work on language support for component-oriented programming. Our current focus is on formally defining the experimental programming language Lag*oo*na [5] which is based on stand-alone messages and on improving its prototype compiler. Additional areas of interest include the integration of stand-alone messages with Java, the implementation of Lag*oo*na on top of COM, techniques for efficient message dispatch, formal specifications in the presence of stand-alone messages, static guarantees on abstract aliasing and representation exposure, and declarative and constraint-based approaches to the consistent integration and configuration of components and frameworks.

Acknowledgements. I would like to thank Wolfram Amme, Michael Franz, Kimberly Haas, Thomas Kistler, Ziemowit Laski, Christian Stork, and Clemens Szyperski for valuable comments. This work was partially supported by the National Science Foundation under grant EIA-9975053.

References

1. Martin Büchi and Wolfgang Weck. Compound types for Java. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 362–373, Vancouver, Canada, October 1998. Published as ACM SIGPLAN Notices 33(10).
2. Luca Cardelli. Typeful programming. SRC Research Report 45, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, May 24, 1989.
3. Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. (Modula-3) language definition. In Greg Nelson, editor, *Systems Programming in Modula-3*, chapter 2, pages 11–66. Prentice-Hall, Englewood Cliffs, NJ, 1991.
4. Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.
5. Peter H. Fröhlich and Michael Franz. The programming language Lagoona. Technical report, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, 2000. Forthcoming.
6. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
7. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Reading, MA, 2nd edition, 2000. To be published. Draft available at http://www.javasoft.com/.
8. Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. Technical Report OSU-CISRC-6/98-TR20, Department of Computer and Information Science, The Ohio State University, Columbus, OH 43210-1277, June 1998.
9. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 1997.
10. Microsoft Corporation. *The Component Object Model (Version 0.9)*, October 1995. Available at http://www.microsoft.com/COM/.
11. Leonid Mikhajlov, Emil Sekerinski, and Linas Laibinis. Developing components in presence of re-entrance. In *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM)*, pages 1301–1320, Toulouse, France, September 1999. Published as Lecture Notes in Computer Science 1709, Springer-Verlag.
12. Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps Beyond Pascal and Modula*. Addison-Wesley (ACM Press), Wokingham, England, 1992.
13. Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 38–45, Portland, OR, November 1986. Published as ACM SIGPLAN Notices 21(11).
14. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, special edition, 2000.
15. Clemens Szyperski. Import is not inheritance—why we need both: Modules and classes. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 19–32, Utrecht, The Netherlands, June 1992. Published as Lecture Notes in Computer Science 615, Springer-Verlag.
16. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley (ACM Press), Harlow, England, 1998.