Component-Oriented Programming in Object-Oriented Languages

Peter H. Fröhlich
phf@acm.org

Michael Franz
franz@uci.edu

Technical Report No. 99-49
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

October 28, 1999
Revised: December 11, 1999

Abstract

Current approaches to component-oriented programming are based on traditional object-oriented languages and concepts. However, most existing object-oriented languages fail to address subtle interface compatibility issues that become paramount in a component-based setting. We explore both syntactic and semantic interface incompatibilities and discuss why they are difficult to handle. We argue that resolving these incompatibilities requires breaking with a fundamental idiom of object-oriented languages: the subordination of messages to interfaces and classes. We propose a solution based on the concept of stand-alone messages as found in the experimental programming language Lagoona and discuss its ramifications.

1 Introduction

Component-oriented programming aims to replace traditional monolithic software systems with reusable *software components* and layered *component frameworks* [34]. Components extend the capabilities of frameworks, while frameworks provide an execution environment for components. Both are developed by independent and mutually unaware vendors, and their composition into a running system is performed by a third party, possibly the end-user.

A component-based approach to software development promises many advantages, almost all of which result from the possibility of vendors specializing in a single domain of expertise. For example, instead of developing a complete text processing application, independent vendors can concentrate on providing document versioning, spell-checking, hyphenation, or intelligent assistants within a

common text processing framework. Development resources can thus be concentrated on a single component (or a single framework), hopefully increasing its reliability and efficiency beyond what would be possible otherwise.

One of the most important factors for making the vision of pervasive software components a reality are interfaces. An *interface* is an abstraction of all possible implementations that can fill a certain role in the composed system [22]. This abstraction allows us to concentrate on what is required of an implementation to fulfill its task and to disregard irrelevant details. In the component-based setting, interfaces are used to describe both the assumptions that frameworks make about components and the assumptions that components make about frameworks.

In programming languages, interfaces are usually only syntactic in nature. Behavioral specifications that implementations are expected to conform to are given as informal comments, and type-systems are used to guard against a subset of possible errors [8, 34]. More elaborate techniques that merge formal specifications and programming languages also exist, but are much less widely accepted [39]. However, even with those techniques, behavioral conformance of an implementation to an interface cannot in general be proven automatically as doing so would entail solving the halting problem [30].

In this paper, we take the point of view of component vendors as opposed to framework vendors. The major task of a component vendor is to develop a software component that conforms to the interface specified by a framework vendor. However, certain components might only be viable as products if they can be reused across multiple frameworks. For example, a specialized hyphenation component for an "exotic" language will only be interesting to a limited market segment. To broaden its potential market such a component should be usable across multiple text-processing frameworks. Since the respective interfaces have been specified independently, conforming to multiple interfaces can complicate the task of the component vendor considerably. These complications are in fact the main motivation for this paper as will become obvious in Sect. 2. Note that similar problems also occur when independently developed frameworks are combined.

Object-oriented programming has been described as a foundation technology for software components [34, 37]. Indeed, conventional object-oriented programming languages like Java [1] and C++ [32] are often used to implement components. Component interfaces, on the other hand, are usually described using an interface definition language (IDL). These IDLs are specific to a certain component model such as COM [5, 11] or CORBA [17], and also have an object-oriented character. The resulting complexity of this approach can be daunting and prompted us to investigate simpler alternatives [13, 14, 19].[1] In particular, we are interested in designing a programming language—codenamed Lag*oo*na—that provides only what is essential in a component-based setting, but not more.

[1] Note that at least in the case of COM, the underlying *core model* is not complex at all. We discuss COM and its relation to our work in more detail in Sect. 6.

A first insight gained from the Lag*oo*na project is that most current object-oriented languages are by themselves unsuitable as a basis for component-oriented programming. They fail to properly address certain interface incompatibilities that arise when a component must implement several interfaces, each defined by an independent framework vendor. Furthermore, neither the component vendor nor the composing end-user can resolve these incompatibilities in a straightforward way. Surprisingly, we can trace the incompatibility problems back to a fundamental idiom found in object-oriented languages: the subordination of messages to interfaces and classes. Our conclusion is that breaking with this idiom is the only clean way to solve the problem.

The remainder of this paper is organized as follows. Section 2 illustrates the interface compatibility problems through a series of examples in a Java-like language. Section 3 analyzes syntactic and semantic incompatibilities in detail and explains their origin in the object-oriented paradigm. Section 4 introduces the concept of stand-alone messages and shows how it solves the compatibility problems through as series of examples in the programming language Lag*oo*na. Section 5 gives a brief overview of additional Lag*oo*na language features. Section 6 reviews related work and compares it to our approach. Section 7 concludes the paper with a summary of contributions and an outline for future work.

2 Interface Compatibility Problems

We illustrate the interface compatibility problems in object-oriented languages through a series of examples based on the infamous abstract data type `Stack`. Independent framework vendors specify `Stack` interfaces that implementations have to conform to. Playing the role of a component vendor targeting the tiny market for `Stack` components, we want to ensure that our implementation can be reused across all frameworks.

To make the examples more concrete, we assume an imperative, class-based, object-oriented programming language similar to Java [1]. We ignore features that would only complicate the discussion without solving the problems we want to illustrate. In particular, we disregard visibility specifiers, ad-hoc polymorphism (overloading), and parametric polymorphism (templates). In spirit our language is also similar to BOPL [29], an idealized object-oriented language that models common features of Simula, Smalltalk, C++ and Eiffel. In contrast to BOPL we support separate type and class hierarchies. Mapping component instances to objects, component implementations to classes, and component interfaces to (Java-like) interfaces seems natural in such a language.

The interface of the first framework, shown in Fig. 1, specifies the four canonical operations on (unbounded) stacks rather informally, although some relevant pre- and postconditions are also stated. Our implementation of this interface in terms of the Java utility class `java.util.Vector` is given in Fig. 2.

Now consider how this implementation can be made conformant to another interface, shown in Fig. 3. Apart from `pop` and `top` being specified in terms of `size`, their signatures and semantics are unchanged from the previous interface

```
package framework1;
interface Stack {
    // Add object to stack as topmost object.
    // require o != null; ensure this.top () == o;
    void push (Object o);
    // If stack not empty, remove topmost object.
    // require !this.empty();
    void pop ();
    // If stack not empty, return topmost object.
    // require !this.empty(); ensure result != null;
    Object top ();
    // No object on stack?
    boolean empty ();
}
```

Figure 1: An interface for the abstract data type `Stack`.

```
package component;
class Stack implements framework1.Stack {
    java.util.Vector rep = new java.util.Vector ();
    void push (Object o) { rep.add(0, o); }
    void pop () { rep.remove(0); }
    Object top () { return rep.elementAt(0); };
    boolean empty () { return rep.size() == 0; };
}
```

Figure 2: Our implementation conforming to Fig. 1.

```
package framework2;
interface Stack {
    // Add object to stack as topmost object.
    // require o != null; ensure this.top () == o;
    void push (Object o);
    // If stack not empty, remove topmost object.
    // require this.size > 0;
    void pop ();
    // If stack not empty, return topmost object.
    // require this.size > 0; ensure result != null;
    Object top ();
    // Number of objects on stack?
    // ensure result >= 0;
    int size ();
}
```

Figure 3: An interface compatible with Fig. 1.

```
package component;
class Stack implements framework1.Stack, framework2.Stack {
    java.util.Vector rep = new java.util.Vector ();
    void push (Object o) { rep.add(0, o); }
    void pop () { rep.remove(0); }
    Object top () { return rep.elementAt(0); };
    boolean empty () { return this.size() == 0; };
    int size () { return rep.size(); }
}
```

Figure 4: Our implementation conforming to Fig. 1 and Fig. 3.

```
package framework3;
interface Stack {
    // Add object to stack as topmost object.
    // require o != null;
    void push (Object o);
    // If stack not empty, remove and return topmost object.
    // require !this.empty(); ensure result != null;
    Object pop ();
    // No object on stack?
    boolean empty ();
}
```

Figure 5: An interface syntactically incompatible with Fig. 1 and Fig. 3.

(Fig. 1). Furthermore, we can easily map the value returned by `size` to the meaning of `empty`. Our modified implementation given in Fig. 4 conforms to both interfaces, thus doubling our potential market share.

We say that two interfaces are *compatible* if we can define a single implementation that conforms to both. As Fig. 4 illustrates, the interfaces from Fig. 1 and Fig. 3 are compatible in this sense. If no implementation can sensibly conform to both interfaces, we say that the interfaces are *incompatible*. Note that this notion of compatibility naturally depends on the programming language at hand. However, we were careful to include only common object-oriented features in our discussion. We are therefore confident that the definition makes sense for a large number of object-oriented languages.

The interface published by the third framework vendor is shown in Fig. 5 and seems to be inspired by the Java utility class `java.util.Stack`. The lack of a `top` message in this interface poses no obvious problem for interface compatibility. However, the new signature of `pop` makes this interface *syntactically incompatible* with those shown in Fig. 1 and Fig. 3. In the purely object-oriented paradigm that we are considering here, two messages with the same name but different signatures cannot be part of the same interface or class, because signatures are not considered during message sends and method selection. Neither extending our model in the direction of Smalltalk [16], where the number of parameters *is* relevant for method selection, nor using the non-object-oriented overloading mechanisms of Java or C++ helps to disambiguate the two messages in this example.[2]

Finally, consider the interface from the fourth framework, shown in Fig. 6. Since `push`, `pop`, `top`, and `empty` are unchanged from the interfaces in Fig. 1 and Fig. 3 we can concentrate on the remaining `size` message. It has a signature that is identical to the signature of `size` in Fig. 3. However, its semantics are

[2]Since the messages only differ in their *return types*, the resolution mechanisms would have to be context-dependent [32].

```
package framework4;
interface Stack {
    // Add object to stack as topmost object.
    // require o != null; ensure this.top () == o;
    void push (Object o);
    // If stack not empty, remove topmost object.
    // require !this.empty();
    void pop ();
    // If stack not empty, return topmost object.
    // require !this.empty(); ensure result != null;
    Object top ();
    // No object on stack?
    boolean empty ();
    // Pushes until internal resize?
    // ensure result >= 0;
    int size ();
}
```

Figure 6: An interface semantically incompatible with Fig. 3.

changed, from returning the number of actual objects on the stack, to returning the number of "free spaces" still available until some internal resize operation occurs in the implementation. The new `size` message makes this interface *semantically incompatible* with the one in Fig. 3, since there is no way to distinguish two semantically different messages with identical names and signatures in the same interface or class.

3 Messages in Object-Oriented Languages

The examples in the preceding section illustrate the problem of interface incompatibility in object-oriented programming languages. We can summarize it as follows:

- Two interfaces are *compatible* if both can be implemented by a single component; otherwise they are *incompatible*.
- Two interfaces are *syntactically* incompatible if they assign *conflicting signatures* to the same message.
- Two interfaces are *semantically* incompatible if they assign *conflicting meanings* to the same message.

In a component-based setting as described in Sect. 1, both syntactic and semantic incompatibilities will occur eventually. The main reason for this is the lack

	$\mathbf{a_{sig} \neq b_{sig}}$	$\mathbf{a_{sig} = b_{sig}}$
$\mathbf{a_{id} \neq b_{id}}$	No conflict	No conflict
$\mathbf{a_{id} = b_{id}}$	Syntactic conflict	Semantic conflict

Figure 7: Possible outcomes of combining messages $a \in A$ and $b \in B$.

of coordination among framework vendors. However, this lack of coordination is also one of the main requirements for the component-based approach to work. Controlling free markets in a centralized fashion has not proven successful in the past. Hence, we would like to prevent interface incompatibilities without enforcing a stricter coordination policy.

The crux of the problem can be found by examining the mechanism used to combine existing interfaces into new ones. In object-oriented languages, messages are pairs of *identifiers* and *signatures*. The identifier describes the name of a message, while the signature describes the types of its arguments and results (possibly including declaration order). Interfaces are *sets* of messages, i.e., each message has a *unique identity* within the interface, namely its identifier. Note that polymorphic method invocation is based on exactly this identity.

If two interfaces A and B are to be combined into a new interface C, the resulting interface must again be a set to be consistent with the object-oriented model. As long as no two messages $a \in A$ and $b \in B$ share the same identifier $a_{id} = b_{id}$ (i.e., $A \cap B = \emptyset$), the combination of interfaces can be described as the union $C = A \cup B$. However, if two messages $a \in A$ and $b \in B$ actually *do* share the same identifier, syntactic or semantic incompatibilities can result. As shown in Fig. 7, the actual incompatibility depends on the signatures a_{sig} and b_{sig}. Figure 8 illustrates how messages "fall out" of their originating interfaces into a new one (for brevity we use single parameter signatures).

In case of identical names and identical signatures, two messages from different interfaces are folded into a single message in the resulting interface (message "c" in Fig. 8). If the messages had different (informal) meanings in their respective interfaces, we can not distinguish them anymore, and the interfaces are semantically incompatible. In case of identical names and different signatures, the two messages cannot both be included in the resulting interface, as method invocation is only driven by the message identifier (message "b" in Fig. 8).[3] The object-oriented "tradition" seems to be that semantic conflicts are silently accepted by the compiler, while syntactic conflicts result in error messages. We speak of a "tradition" since resolving both kinds of conflicts in exactly *this* way is obviously an ad-hoc decision.

The only way to address these incompatibilities is to give messages a *unique identity* that is *independent* of the interfaces or classes they participate in. At the heart of the interface compatibility problem, we thus find a fundamental limitation of many current object-oriented languages. Messages are considered

[3]As pointed out in Sect. 2, the Smalltalk approach of including the number of parameters into the dispatch process is not a general solution either.

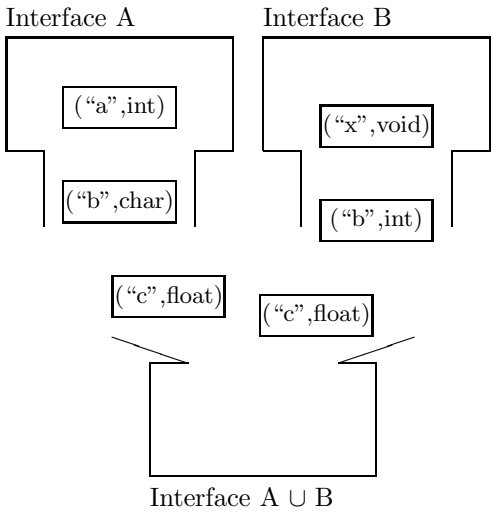

Figure 8: Messages are "falling out" of interfaces A and B into a new one.

subordinate to interfaces (classes) rather than having a unique identity of their own. This is somewhat surprising if we consider the classic definition of the object-oriented paradigm [20]:

- Everything is an *object*.
- Objects communicate by sending ... *messages* (in terms of objects).
- Objects have their *own memory* (in terms of objects).
- Every object is an *instance* of a *class* (which must be an object).

While we do not agree with this definition completely, it is easy to see that it does not relegate messages to an inferior status. In particular, it could be argued that a message should itself be an object, similar to a class. However, even in Smalltalk [16], where this definition was first applied, messages are subordinate to classes.

An alternate explanation of this fundamental problem in the object-oriented paradigm can be found in its attempt to unify two distinct concepts: modules and types. It has often been argued that classes (and thus interfaces) are "better modules," because they support encapsulation *and* extension [24]. Modules support only encapsulation, while types (in the sense of extensible record types) support only extension. However, in light of the above discussion it should be clear that messages cannot safely be subordinate to entities that keep changing through extensions. To retain a unique identity, messages have to be subordinate to some "stable anchor" that ensures that the same message always represents the same meaning. Modules can fill exactly this role.

It could be pointed out that an object-oriented language like Java already supports "messages" with unique identities. For example, we could model "messages" as a hierarchy of classes and "message sends" as a general dispatch message similar to the "Command" design-pattern [15]. However, such an approach is both clumsy and unsafe. It is clumsy because parameter passing and type-tests must be performed explicitly, and it is unsafe because the compiler can not statically enforce type-safety. In a component-based setting this approach is especially useless for the latter reason.

Some object-oriented languages also support additional mechanisms not considered so far. In Eiffel [23] for example, conflicting messages can be *renamed* in a derived class. However, this is either not type-safe or leads to semantic incompatibilities. If all messages are given new and unique names, message sends using the conflicting name will lead to run-time errors or invoke the method of a base class. For abstract base classes (the equivalent to interfaces) the latter again results in a run-time error. On the other hand, if one message retains the conflicting name, message sends that expect different behavior will invoke a semantically incompatible method.

Another mechanism is used in C++ [32], where message sends can be explicitly qualified by the class in which a method should be invoked. However, this results in static binding, not in dynamic method selection. The idea of explicit qualification points towards another solution though. We could require that messages are always explicitly qualified by the interface that first introduced them, thus giving them a unique identity across other interfaces and classes. However, interfaces themselves are not necessarily unique. In Java for example, two interfaces with the same name and identical messages could be declared in two different packages. If both interfaces are to be combined, we would have to extend explicit qualification to package names. However, since packages in Java are also not "closed" in the sense of modules in languages such as Ada or Modula-2, we would also have to replace the package concept with a "real" module concept. Starting out with a "real" module concept seems preferable to us, and is in fact the solution we adopt in Lag*oo*na.

4 Stand-Alone Messages in Lagoona

The analysis in Sect. 3 shows that support for component-oriented programming is fundamentally limited in object-oriented languages. To avoid syntactic and semantic incompatibilities between component interfaces, messages must have *unique identities* that are *independent* of the interfaces or classes they participate in. In this section, we introduce the concept of *stand-alone messages*. We will again use a series of examples based on the abstract data type `Stack`, this time expressed in the experimental programming language Lag*oo*na [13].

Lag*oo*na is an imperative, modular, and object-oriented[4] programming language loosely based on Oberon [31]. The basic compilation unit is the *module*

[4]For obvious reasons, we use the term "object-oriented" somewhat reluctantly, but the better term "component-oriented" does not have an established meaning yet.

```
MODULE Framework1;
  MESSAGE
    (* Add object to stack as topmost object. *)
    (* REQUIRE o # NIL; ENSURE Top() = o; *)
    Push (IN o: ANY);
    (* If stack not empty, remove topmost object. *)
    (* REQUIRE ~Empty(); *)
    Pop ();
    (* If stack not empty, return topmost object. *)
    (* REQUIRE ~Empty(); ENSURE result # NIL; *)
    Top (): ANY;
    (* No object on stack? *)
    Empty (): BOOLEAN;
  TYPE
    Stack = {Push, Pop, Top, Empty};
END Framework1.
```

Figure 9: A Lag*oo*na interface for the abstract data type `Stack`.

as opposed to the class or interface in conventional object-oriented languages. Modules contain declarations of constants, variables, messages, types, procedures, and methods.

Figure 9 shows the Lag*oo*na equivalent to the stack interface originally given in Fig. 1 above. As before, we disregard visibility considerations in these examples. However, we do not make other simplifications to the language for presentation purposes. Note that stand-alone messages are introduced in the module scope, on the same level as all other declarations in Lag*oo*na (except for local variables, of course). In particular, messages are not subordinate to a type, although so-called *message set types* can be formed out of a number of messages. Message set types can be used to reference objects that implement all messages mandated by the type. A variable of type `Framework1.Stack` could either reference an object implementing at least `Framework1.Push`, `Framework1.Pop`, `Framework1.Top`, and `Framework1.Empty`, or no object at all (denoted by `NIL`).

As before, we associate an informal specification with each message. However, since messages in Lag*oo*na have a unique identity, a message always refers to *exactly that* specification. In a manner similar to conventional type-checking, this allows guarding against accidentally using one message where another is required. In particular, neither syntactic nor semantic incompatibilities are possible at the interface level in Lag*oo*na.

An implementation conforming to the interface given in Fig. 9 is shown in Fig. 10. For simplicity, we assume the presence of an utility class, just like in Sect. 2. Note how the implementation explicitly *imports* and *qualifies* each message it implements or sends. Module identifiers are formed out of "inverted" domain names in Lag*oo*na—similar to the approach taken for package names

```
MODULE Component;
  IMPORT
    F1 := Framework1, V := Com.Lagoona.Util.Vectors;
  TYPE
    Stack = RECORD rep: V.Vector; END;
  METHOD (OUT self: Stack) INITIALIZE ();
    BEGIN NEW (self.rep);
  END INITIALIZE;
  METHOD (INOUT self: Stack) F1.Push (IN o: ANY);
    BEGIN V.Add (0, o) -> self.rep;
  END F1.Push;
  METHOD (INOUT self: Stack) F1.Pop ();
    BEGIN V.Remove (0) -> self.rep;
  END F1.Pop;
  METHOD (IN self: Stack) F1.Top (): ANY;
    RETURN V.ElementAt (0) -> self.rep;
  END F1.Top;
  METHOD (IN self: Stack) F1.Empty (): BOOLEAN;
    RETURN V.Size (0) -> self.rep = 0;
  END F1.Empty;
END Component.
```

Figure 10: Our Lag*oo*na implementation conforming to Fig. 9.

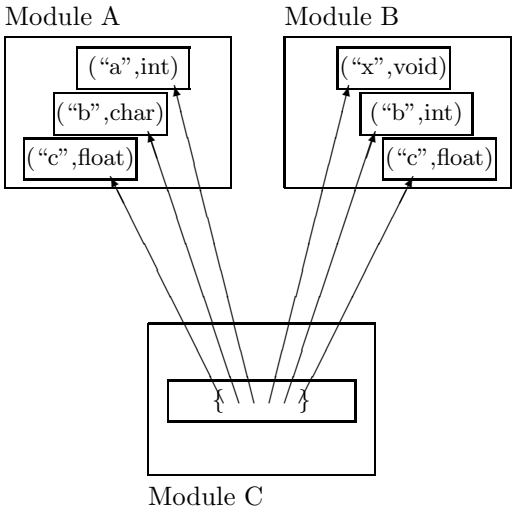

Figure 11: Messages retain their identities even in new interfaces.

in Java—and can thus reasonably be expected to be unique. To distinguish message sends from conventional "method calls" we use the operator "`->`" instead of the more common dot notation. The `INITIALIZE` method is invoked implicitly as an initializer whenever an object of this type is created.

The concept of stand-alone messages in Lagoona is illustrated in Fig. 11. Compared to the conventional object-oriented mechanism from Fig. 8 above, messages retain a unique identity (and thus the meaning they are informally given in the module they are defined in), even if they are composed into new interfaces in other modules.

A somewhat extreme example of the power of stand-alone messages is an implementation conforming to *all four* stack interfaces we introduced in Sect. 2. Such an implementation is shown in Fig. 12, assuming that the appropriate interfaces—which we omit for brevity—have been defined in Lagoona. Note how Lagoona enables the component vendor to use his knowledge about the specifications of individual messages to provide a single method implementation for multiple message specifications. Thus duplication of method bodies can be avoided and no additional forwarding methods have to be introduced. Compared to the interface shown in Fig. 9, the other interfaces that this component implements differ as follows:

- Framework 2 uses `Size` instead of `Empty`.
- Framework 3 omits `Top` and returns objects through `Pop`.
- Framework 4 attaches different semantics to `Size`.

The resulting implementation can be reused unchanged across all four frameworks.

5 Additional Features of Lagoona

In the previous section we have illustrated how Lagoona's stand-alone messages can be used to solve the interface compatibility problem. In this section we discuss some additional features of Lagoona that were designed to better support a component-oriented programming style.

We already briefly introduced *message set types* above. The main purpose of declaring a named message set type such as `Framework1.Stack` is to give a convenient name to a set of messages, nothing more. Message set types support "set-like" *union* and *difference* operations. For example, given messages `A`, `B`, `C`, and `D`, the type `S = {A, B, C}` could also be expressed as `S = {A} + {B, C} - {D}`. Compatibility between two message set types is defined using a subset relation between the sets of messages they denote. Given the type `T = {A, B, C, D}` and variables `s: S` and `t: T`, the assignment `s := t` is legal, while the reverse assignment `t := s` is not. The obvious consequence of this definition is that type compatibility between message set types is *structural*. However, this structural compatibility is nevertheless *safe* in Lagoona because individual messages have and retain unique identities.

```
MODULE Component;
  IMPORT
    F1 := Framework1, F2 := Framework2, F3 := Framework3,
    F4 := Framework4, V := Com.Lagoona.Util.Vectors;
  TYPE
    Stack = RECORD rep: V.Vector; END;
  METHOD (OUT self: Stack) INITIALIZE ();
    BEGIN NEW (self.rep);
  END INITIALIZE;
  METHOD (INOUT self: Stack) F1.Push, F2.Push, F3.Push, F4.Push (IN o: ANY);
    BEGIN V.Add (0, o) -> self.rep;
  END F1.Push, F2.Push, F3.Push, F4.Push;
  METHOD (INOUT self: Stack) F1.Pop, F2.Pop, F4.Pop ();
    BEGIN V.Remove (0) -> self.rep;
  END F1.Pop, F2.Pop, F4.Pop;
  METHOD (INOUT self: Stack) F3.Pop (): ANY;
    RETURN V.Remove (0) -> self.rep;
  END F3.Pop;
  METHOD (IN self: Stack) F1.Top, F2.Top, F4.Top (): ANY;
    RETURN V.ElementAt (0) -> self.rep;
  END F1.Top, F2.Top, F4.Top;
  METHOD (IN self: Stack) F1.Empty, F3.Empty, F4.Empty (): BOOLEAN;
    RETURN F2.Size () -> self = 0;
  END F1.Empty, F3.Empty, F4.Empty;
  METHOD (IN self: Stack) F2.Size (): INTEGER;
    RETURN V.Size () -> self.rep;
  END F2.Size;
  METHOD (IN self: Stack) F4.Size (): INTEGER;
    RETURN MAX(INTEGER); (* fake value *)
  END F4.Size;
END Component.
```

Figure 12: Our Lag*oo*na implementation conforming to all four interfaces.

Furthermore, the combination of set-like operations and structural compatibility enables the expression of *subtyping* as well as *supertyping* in a straightforward way. Thus variables and formal parameters can be typed "as little as possible" to exactly specify the relevant requirements. For example, a method `X` that only applies the `Framework1.Push` message to some object can be declared as `X(stack:{Framework1.Push});` instead of having to mandate an implementation of the complete `Framework1.Stack` message set.

Implementations (classes) are modelled as *object types* in Lagoona, which basically are record types with attached *methods* as shown in Fig. 10 and Fig. 12. This idea is fairly standard, except for the fact that object types do *not* declare which message sets they implement explicitly. Furthermore, the structural type compatibility mechanism introduced between message set types is reused. A variable of object type A can be assigned to a variable of message set type B if and only if A implements all messages denoted by B. However, between two object types, name-based compatibility is used. Type compatibility between message set types and object types is checked when they are actually used in assignments or as actual parameters. This supports the evolution of software components better than static declarations do, at the price of detecting some type-errors "slightly" later than possible with explicit declarations.

Message sends and method dispatch is more interesting, since Lagoona does not support any kind of inheritance (in the sense of reuse of field or method declarations). Instead, a *generic message forwarding* mechanism is provided. Avoiding inheritance (and delegation) helps to guard against certain problems caused by the self-referential nature of these mechanisms. It has been shown that restricting inheritance (and delegation) in order to make them safe in a component-based setting essentially results in making them equivalent to forwarding [25, 26, 37].

Our generic forwarding mechanism works as follows. When an object receives a message, and a corresponding method implementing this message exists in the object type, that method is executed. If no matching method is found, but the special method `DEFAULT` is implemented, it is executed instead. Inside a *default method* we can *resend* the message to other objects. However, resending is a generic operation in which the actual message remains opaque. Note that an empty default method can be used to ignore all messages an object does not explicitly implement. Finally, if no matching method and no default method exists, execution is aborted with an exception.

For this flexible message forwarding approach to be safe, some restrictions on message sends have to be imposed. In particular, for messages that return a result, we must be able to statically determine whether they are handled or not. Otherwise using the result of the message send would be unsound. In Lagoona, we decided to offer two clear alternatives to the programmer. If a message is sent to a message set variable, it must statically exist in the set of messages denoted by the corresponding message set type. Note that this rule applies to functional and non-functional messages. If a message is sent to an object type variable, and if the message returns a result, an implementation for it must statically exist in the object type. If the message does not return

a result, the message might be handled (possibly after multiple forwards), it might be ignored, or an exception might be raised. Thus developers can decide on the exact message-handling semantics they need at a fine-grained level.

6 Related Work

Previous work related to the problem of interface incompatibility has been surprisingly hard to find. The only reference we are aware of is the work of Ossher and Harrison on combining separate inheritance hierarchies [28]. Ossher and Harrison also identified the distinction between syntactic and semantic conflicts, however they did not propose a concrete approach how these conflicts could be addressed at the language level.

The use of structural conformance to increase the expressiveness of type-systems has also been advocated by Baumgartner, Läufer, and Russo, for C++ [2] and more recently for Java [21]. They give a number of good motivating examples, although not from the perspective of component-oriented programming. However, their proposals neither handle the problem of "accidental" conformance in an elegant way (messages are still subordinate to signatures) nor address the interface compatibility issues we have been concerned with. On the other hand, their conformance relation is more flexible than ours, since it allows contravariance of parameter types, covariance of result types, and several other mechanisms to be taken into account.

Büchi and Weck also argue for some degree of structural conformance in their *compound types* proposal for Java [6]. They provide a good motivation from a component-oriented perspective and show that explicitly declared compatibility has severe drawbacks in this setting. Compound types combine declared and structural type compatibility in order to maintain the relation between named types and specifications on the one hand, while allowing for flexible compositions of multiple named types on the other hand. Thus, compound types can be used to model composition of interfaces more flexibly while at the same time supporting type-safety and type-based checking of specifications. However, compound types are actually a special case of Lag*oo*na's stand-alone messages, and our approach shows that their claim that "*... structural equivalence ... does not support behavioral typing...*" is not necessarily true if messages have unique identities. Furthermore, their work also does not address issues of interface compatibility.

Finally, the idea of stand-alone messages is also related to research on multi-methods in object-oriented languages. In a language with multi-methods such as Cecil [9], stand-alone messages could be "emulated" by introducing an additional dispatch parameter modelling the originating module. However, multi-methods are not yet widely accepted and are not supported in mainstream languages. They also lead to a more functional style of programming that adherents of the object-oriented style sometimes dislike. Furthermore, supporting type-safe multi-methods in the presence of separate compilation is still an active area of research [27]. Efficiency considerations also have to be taken into account and

are not clearly understood. We argue that stand-alone messages are conceptually simpler because they only rely on the established notion of modules and add no additional concerns for separate compilation. They also maintain the established, object-oriented programming style.

As far as component-models are concerned, the approach taken in COM [5, 11] seems to be most similar to Lag*oo*na. Instead of giving unique identities to messages, COM assigns unique identities to interfaces using automatically generated *globally unique identifiers* (GUIDs). Interfaces are "frozen" once they are published, ideally together with the associated semantics. Contrary to the object-oriented programming languages we examined, COM allows a class to implement multiple interfaces *without* merging them, i.e. interfaces retain their unique identity across all classes that implement them. It therefore seems straightforward to model some of Lag*oo*na's features in COM. Stand-alone messages could be expressed using "singleton" COM interfaces consisting of only one message, while message-set types could be expressed using COM's *category* mechanism.[5]

In contrast to the *core* model of COM sketched above, the *actual* COM model is much more complex. Since COM is a language-neutral standard, many additional features must be exposed in order to allow low-level languages such as C and C++ to be integrated. For example, programmers have to be aware of the complex details of memory management and in-processes vs. out-of-process COM servers and their specific restrictions. Aspects of persistence and distribution are also part of the COM specification, even though they do not seem to be of central importance to a component-oriented programming style. It is possible to hide some of COM's complexity via so-called Direct-To-COM compilers, for example by providing automatic garbage collection on top of COM's reference counting [18]. However, it seems that even the designers of COM are sometimes overwhelmed by its complexity, as evidenced by the recent discovery of an inconsistency within COM itself [33].

7 Conclusions

Component-oriented programming requires that component and framework interfaces are explicitly specified. A single component must also be able to implement multiple interfaces. In traditional object-oriented languages, we have shown through a series of examples how the second requirement can—and eventually will—lead to interface compatibility problems, either of a syntactic or a semantic nature. We have argued that the root of this problem can be traced back to the fact that most current object-oriented languages reduce messages to inferior constructs subordinate to interfaces and classes.

[5] It might indeed be interesting to investigate this similarity in more detail by developing an implementation of Lag*oo*na on top of COM. Such an implementation could leverage the wide support COM has gained in industry while offering a considerably simpler programming model to the software developer.

	Message ∈ Type	**Message ∈ Module**
Method ∈ Type	Object-Oriented: C++, Eiffel, Smalltalk	Component-Oriented: Lag*oo*na
Method ∈ Module	Useless?	Modular: Ada, Modula-2

Figure 13: Language design space for messages and methods.

We have proposed stand-alone messages as a solution to this problem and have illustrated their application through another series of examples, showing that they can solve both syntactic and semantic interface incompatibilities. Furthermore, we have briefly described the experimental programming language Lag*oo*na that supports stand-alone messages and other constructs facilitating a component-oriented programming style.

Figure 13 illustrates the language design space we have explored in developing Lag*oo*na. Messages (the building blocks for interfaces) and methods (the building blocks for implementations) can be subordinate to either types or modules. In the object-oriented paradigm, both are subordinate to types, leading to the interface compatibility problems described above. In the modular paradigm, both are subordinate to modules, and no interface incompatibilities can result. However, the modular paradigm does not allow us to express the idea of *implementation polymorphism*, i.e. the ability to dynamically exchange conforming implementations "behind" an interface.

The component-oriented paradigm seems to require a combination in which messages are subordinate to modules in order to retain their unique identity, while methods are subordinate to types in order to model implementation polymorphism. Note that we still need a way of typing variables with interfaces, which is the purpose of message set types in Lag*oo*na. Finally, messages could be made subordinate to types and methods subordinate to modules. However, this does not seem to yield an interesting programming paradigm, and we are not aware of any languages in this category.

Future work will focus on improving Lag*oo*na in various ways so as to still better support component-oriented programming. Numerous interesting problems in this direction have not been addressed so far. For example, issues of *abstract aliasing* and *representation exposure* become important in a language that achieves code-reuse purely through forwarding between black-box components. In particular, we plan to investigate extending Lag*oo*na to provide static guarantees about the possible aliases that can be created at runtime [10, 12, 36]. Another important area is the formalization of Lag*oo*na's type-system and a proof of its soundness, especially in the presence of separate compilation [8]. We are also interested in more sophisticated approaches to interface specification, especially in the presence of inter-component call-backs [7]. Finally, we want to show that the Lag*oo*na approach to component-oriented programming is scalable, which will require implementing a suitably large, non-trivial soft-

ware system from scratch, and maintaining it through a series of extensions and adaptions.

Acknowledgements

We would like to thank Clemens Szyperski, Ziemowit Laski, Thomas Kistler, and Christian Stork for valuable comments on an earlier version of this paper. We are also grateful to Michael Guntersdorfer, Peter Housel, Juei Chang, and Marcellus Wallace for fruitful discussions.

References

[1] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, second edition, 1998.

[2] Gerald Baumgartner and Vincent F. Russo. Signatures: A language extension for improving type abstraction and subtype polymorphism in C++. *Software—Practice and Experience*, 25(8):863–889, August 1995.

[3] Thomas J. Bergin and Richard G. Gibson, editors. *History of Programming Languages*. ACM Press / Addison-Wesley, 1996.

[4] A. Michael Berman, editor. *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, October 1998. Published as SIGPLAN Notices 33(10), October 1998.

[5] Don Box. *Essential COM*. Addison-Wesley, 1998.

[6] Martin Büchi and Wolfgang Weck. Compound types for Java. In Berman [4], pages 362–373. Published as SIGPLAN Notices 33(10), October 1998.

[7] Martin Büchi and Wolfgang Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, August 1999.

[8] Luca Cardelli. Type systems. In Tucker [35], chapter 103, pages 2208–2236.

[9] Craig Chambers. The Cecil language: Specification and rationale. Technical report, Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA, March 1997. Available at http://www.cs.washington.edu/research/projects/cecil/.

[10] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In Berman [4], pages 48–64. Published as SIGPLAN Notices 33(10), October 1998.

[11] Microsoft Corporation. *The Component Object Model (Version 0.9)*, October 1995. Available at http://www.microsoft.com/COM/resources/COM1598C.ZIP.

[12] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. SRC Research Report 156, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, July 29, 1998.

[13] Michael Franz. The programming language Lagoona: A fresh look at object-orientation. *Software — Concepts and Tools*, 18(1):14–26, March 1997.

[14] Michael Franz. On the architecture of software component systems. In R. Nigel Horspool, editor, *Systems Implementation 2000*, pages 207–220. Chapman & Hall, February 1998.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[17] Object Management Group. *The Common Object Request Broker: Architecture and Specification (Version 2.3.1)*, October 1999. Available at http://www.omg.org/cgi-bin/doc?formal/99-10-07.

[18] Dominik Gruntz and Beat Heeb. Direct-To-COM compiler provides garbage collection for COM objects. In *Proceedings of the Second Component User's Conference (CUC'97)*. Available at http://www.oberon.ch/resources/com/dtc_cuc_paper/index.html.

[19] Jurg Gutknecht and Michael Franz. Oberon with Gadgets: A simple component framework. In Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors, *Implementing Application Frameworks: Object-Oriented Frameworks at Work*. John Wiley & Sons, September 1999.

[20] Alan C. Kay. The early history of Smalltalk. In Bergin and Gibson [3], pages 511–597.

[21] 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, June 1998. Available at http://www.cis.ohio-state.edu/~gb/Papers/Java-conformance.pdf.

[22] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press / McGraw-Hill, 1986.

[23] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[24] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[25] Leonid Mikhajlov and Emil Sekerinski. The fragile base class problem and its solution. Technical Report 117, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, June 1997.

[26] Leonid Mikhajlov, Emil Sekerinski, and Laibinis Linas. Developing components in the presence of re-entrance. Technical Report 239, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, February 1999.

[27] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer, June 1999.

[28] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In Andreas Paepcke, editor, *Proceedings of the 1992 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, pages 25–40, October 1992. Published as SIGPLAN Notices 27(10), October 1992.

[29] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[30] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.

[31] Martin Reiser and Niklaus Wirth. *Programming in Oberon: Steps beyond Pascal and Modula*. Addison-Wesley, 1992.

[32] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[33] Kevin J. Sullivan, Mark Marchukov, and John Socha. Analysis of a conflict between aggregation and interface negotiation in Microsoft's component object model. *IEEE Transactions on Software Engineering*, 25(4):584–599, July/August 1999.

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

[35] Allen B. Tucker, editor. *The Computer Science and Engineering Handbook*. CRC Press, 1997.

[36] Jan Vitek and Boris Bokowski. Confined types. In A. Michael Berman, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 82–96, October 1999. Published as SIGPLAN Notices 34(10), October 1999.

[37] Wolfgang Weck. Inheritance using contracts and object composition. In Weck et al. [38], pages 105–112.

[38] Wolfgang Weck, Jan Bosch, and Clemens Szyperski, editors. *Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97)*, number 5 in TUCS General Publications, Turku Center for Computer Science, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, September 1997.

[39] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.