

# Supporting Software Composition at the Programming-Language Level

Michael Franz<sup>a</sup>, Peter H. Fröhlich<sup>b</sup>, and Andreas Gal<sup>a</sup>

<sup>a</sup>*School of Information & Computer Science  
University of California, Irvine*

<sup>b</sup>*Department of Computer Science & Engineering  
University of California, Riverside*

---

## Abstract

We are in the midst of a paradigm shift toward component-oriented software development, and significant progress has been made in understanding and harnessing this new paradigm. Somewhat strangely then, the new paradigm does not currently extend all the way down to how the components themselves are constructed. While we have composition architectures and languages that describe how systems are put together out of such atomic program parts, the parts themselves are still constructed based on a previous paradigm, object-oriented programming. We argue that this represents a mismatch that is holding back compositional software design: many of the assumptions that underly object-oriented systems simply do not apply in the open and dynamic contexts of component software environments. What, then, would a programming language look like that supported component-oriented programming at the smallest granularity? Our project to develop such a language, Lagoon, tries to provide an answer to this question. This paper motivates the new key concepts behind Lagoon and briefly describes their realization (using Lagoon itself as the implementation language) in the context of Microsoft's .NET environment.

*Key words:* component-oriented software development, programming languages, distributed extensibility, language paradigms beyond object-oriented programming

---

## 1 Introduction

While the idea of a “software component” has been proposed as far back as the 1960's [1], the arrival of the Internet has propelled us into an age where component-oriented programming (COP) is becoming simultaneously viable and necessary. Viable, because an efficient component discovery and distribution mechanism is now available; necessary, because the complexity of Internet-enabled applications often exceeds the abstraction capabilities of existing programming paradigms.

The very nature of a component-oriented system is its *distributed extensibility*: by creating new components, *mutually unaware parties* can evolve the system *inde-*

*pendently of each other, and in parallel.* This decoupling of the individual components keeps complexity under control; there is no single authority that requires absolute knowledge about the inner workings of all components. Strangely, however, the implementation medium of choice for the individual components at this time remains object-oriented programming (OOP). OOP is based on a fundamentally different model, namely that of a common shared framework in which only the leaves of the object hierarchy are extended. Hence, OOP assumes the presence of a central coordinator. This collides with COP's notion of *distributed extensibility*.

The idea of distributed extensibility has profound implications for programming languages, but it was apparently not considered when object-oriented languages such as Eiffel or Java were designed. For example, Eiffel's covariant argument types require a form of global analysis that implies a "closed system" view of the world. Similarly, evolving an existing object-oriented framework (base class library) can lead to "name clashes" in already developed extensions, causing them to fail. These are just examples that show the mismatch between current OOP practice and the ultimate requirements of COP. As a result of this misalignment, today's component-oriented systems fall short of their true potential. Rather than being truly "composed" from independently developed parts, they rely on the presence of large, shared underlying frameworks that themselves cannot be evolved easily. Also, components that are based on different frameworks can often not interact.

Our research, on which we report in this paper, has focused on developing an experimental programming language ("Lagoona") that supports COP *expressly*. Lagoona retains much of the flavor and benefits of OOP languages but discards those elements that contradict the COP paradigm. We focus on two novel language mechanisms, *stand-alone messages* and *generic message forwarding* (Section 2). We then show how these allow us to eliminate or alleviate a number of (sometimes long-standing) design and implementation problems in OOP-based COP, such as issues of *interface conflicts*, *fragile base classes*, and *component reentrance* (Section 3). Section 4 describes our implementation and shows how we address the major challenges for efficient execution. The final sections describe related and future work and conclude our paper.

## 2 Lagoona

Lagoona is designed around a standard imperative language core, a choice made more for reasons of familiarity than necessity. Lagoona's object model, however, is very different from those found in established object-oriented programming languages. It separates the many roles traditionally played by classes, turning them into individual language constructs [2][3]. Figure 1 provides a concise comparison of how design concerns are mapped onto the class construct in traditional object-oriented languages and onto separate constructs in Lagoona. At the lowest level of Lagoona's object model are *messages* and *methods*. Messages are *abstract operations* that describe *what* effect they achieve, while methods are *concrete operations*

that describe *how* an effect is achieved. In other words, messages are specifications for methods, and methods are implementations of messages. At the next higher level, messages and methods are grouped into *interface types* and *implementation types*. An interface type is simply a set of messages, while an implementation type consists of a set of methods and associated storage definitions. Variables of these types are called *interface references* and *implementation references* respectively. Implementation types serve as generators for *instances*, which are first-class values that can be assigned to implementation or interface references. As with messages and methods, interface types and implementation types serve as specifications and implementations for each other. At the highest level of the object model are *modules* which encapsulate sets of messages, methods, interface types, and implementation types. Modules are *unique* in the sense that only a single copy of a certain module can exist in a given system.

So far, our description of Lagoona’s object model reads almost like the textbook definition of any object-oriented programming language. What sets Lagoona apart are the following additional relations between the concepts introduced above. Although messages are “grouped into” interface types, they are not declared in the scope of a type but rather in the scope of a module. Since modules are unique, messages are unique as well. We use the term *stand-alone messages* to express this independence of messages from types. In contrast to messages, methods *are* declared in the scope of an implementation type. This asymmetry is intentional, since we want to support multiple implementations of identical specifications on the level of messages and methods as well as on the level of interface types and implementation types [4]. To relate interface types and implementation types (including their instances), we need to define some notion of *conformance*. First, an interface type  $B$  denoting a set of messages  $M_B$  conforms to an interface type  $A$  denoting a set of messages  $M_A$  if and only if  $M_B$  is a superset of  $M_A$ :

$$A \trianglelefteq B \iff M_A \subseteq M_B \tag{1}$$

In other words, we employ *structural subtyping* between interface types. Second, an implementation type  $C$  with a set of methods implementing a set of messages  $M_C$  conforms to an interface type  $B$  denoting a set of messages  $M_B$  if and only if  $M_C$  is a superset of  $M_B$ :

$$B \trianglelefteq C \iff M_B \subseteq M_C \tag{2}$$

We thus extend structural subtyping to implementation types, and if (1) and (2) hold,  $A \trianglelefteq C$  will hold as well. This enables a form of *inclusion polymorphism* that we like to call *implementation polymorphism*. Third, an interface type never conforms to an implementation type. Of course, Lagoona allows interface types to be *cast* to implementation types, guarded by a dynamic check. Fourth, two implementation types only conform if they are the same type. In other words, we employ *occurrence equivalence* between implementation types. This completes the definition of conformance, but the fourth case raises the question how implementation

types can be reused or adapted. At runtime, Lagoona’s object model essentially reduces to a web of independent instances that communicate through messages. Assume we are sending a message  $m$  to a receiver  $r$ , which can be an interface or an implementation reference, whose type  $R$  denotes a message set  $M_R$ . We distinguish two *message send operators* with different semantics. The first operator  $\rightarrow$  is *strict* in the sense that the expression  $m \rightarrow r$  is valid if and only if  $m$  is an element of  $M_R$ :

$$m \rightarrow r \iff m \in M_R \tag{3}$$

In other words, this operator statically ensures that the message  $m$  will be “handled” by the instance bound to  $r$ . The second operator  $\Rightarrow$  is *blind* in the sense that the expression  $m \Rightarrow r$  is *always* valid. Of course, we have to guard the application of this operator by a dynamic check, similar to the one for casts mentioned above. The blind message send operator is necessary to support reuse and adaptation by intercepting and rerouting messages. Implementation types can define a *default method* which is triggered for messages that do not have an explicit method associated with them. Inside this default method, messages can be *resent* or *forwarded* to other instances. We use the term *generic message forwarding* to express that the actual message remains opaque during this process. Obviously, the strict message send operator alone would not be sufficient to support this.

Lagoona’s object model can be viewed as another step toward eliminating the dominance of the class construct in object-oriented languages. Previous steps include the separation of interfaces and implementations [5] and the separation of modules and types [6], both of which are widely accepted by now. In the remainder of this section we explain each element of Lagoona’s object model in more detail.

Lagoona’s top-level construct, the *module*, serves a variety of purposes. Modules are compilation units and result in object files which in turn are the units of deployment [7]. Modules live in a flat, global namespace and cannot be nested [8]. However, we employ a “hierarchical” naming convention based on Internet domain names, similar to the one originally proposed for Java packages. Modules are also *sealed* [9]: only explicitly exported declarations are visible to clients, and no new declarations can be added from the outside. Modules can *import* other modules and then refer to their exported declarations. These references are fully qualified, but to avoid “excessive” qualifications we allow the introduction of local *aliases* for imported modules. The module shown in Figure 2 exports all its declarations by marking them `public`. The module in Figure 3 imports the first one under the alias `S` and uses this alias to qualify further references, for example to the message `push`. However, several declarations inside the second module are not marked `public` and are therefore hidden from its clients.

As shown in Figure 2, messages are bound to (declared in) modules instead of types. Since modules are unique within a given system, and since no two messages can have the same name within a given module, our approach makes mes-

Concern	Traditional	Lagoona
Encapsulation	Class (modifiers)	Module
Specification	Class (abstract method)	Message
	Class (abstract)	Interface
Implementation	Class (concrete method)	Method
	Class (concrete)	Implementation
Modification	Class (inheritance)	Forwarding

Fig. 1. Design concerns and corresponding language constructs in traditional languages and in Lagoona.

```

module com.lagoona.stacks {
  public message void push(any obj);
  public message void pop();
  public message any top();
  public message boolean empty();
  public interface Stack {
    push, pop,
    top, empty
  }
}

```

Fig. 2. A stack abstraction in Lagoona. Messages are bound to modules, not types.

sages unique as well. If messages were bound to types, the approach taken in most conventional object-oriented languages, we could not guarantee this property in general. Surprisingly, many of the issues described in Section 3 stem from this seemingly trivial difference. We usually associate a semi-formal specification with each message. The `push` message, for example, would be characterized with the precondition “ $obj \neq null$ ” and the postcondition “ $\neg empty$ ”. Finally, we assume that a message and its specification are *immutable* once published, which is similar to the assumption made about interfaces in COM [10] and related technologies.

Messages are the basis for *interface types* (`interface` in our concrete syntax) which represent references to objects that implement a certain set of messages. In Figure 2, the interface type `Stack` is declared as supporting the messages `push`, `pop`, `top`, and `empty`. If we declare a variable `s` of type `Stack`, we can only assign objects that implement at least these four operations to `s`. As explained above, conformance to interface types is structural. The pervasive interface type `any` represents the empty message set and is the top element in the resulting type lattice. Note that the name we give to an interface type is only a convenient abbreviation; instead of using this name, we could also repeatedly declare isomorphic interface types. Conceptually, interface types in Lagoona are used to decouple independent components, similar to the use of interfaces in both COM [10] and, to a certain extent, Java. Implementation types (`class` in our concrete syntax) host methods and declarations of instance variables. Consider the implementation of the `Stack` abstraction shown in Figure 3. Each method implements exactly one message imported from the module `S`. The messages `initialize` and `finalize` have special meaning in Lagoona: They are sent by the runtime system immediately after an instance has been created and immediately before it is garbage collected. The class `Link` is essentially used as a simple record type without any methods.

Figure 4 illustrates how message forwarding between instances is used to “extend” an existing implementation type. In this example, we want to extend the stack abstraction (and its implementation) with an operation that determines the number of elements currently on the stack. First we introduce a new message `elements` which does exactly that. Next we declare a class `Stack` that has an interface reference to another stack and an instance variable for the actual counter. The method

```

module com.lagoona.simple_stacks {
  import S = com.lagoona.stacks;
  class Link {
    any object; Link next;
  }
  public class Stack {
    Link top;
    method void initialize() {
      this.top = null;
    }
    method void S.push(any obj) {
      Link x = new Link();
      x.object = obj;
      x.next = this.top;
      this.top = x;
    }
    method void S.pop() {
      this.top = this.top.next;
    }
    method any S.top() {
      return this.top.object;
    }
    method boolean S.empty() {
      return this.top == null;
    }
  }
}

```

Fig. 3. An implementation of the stack abstraction. Methods implementing messages are bound to types.

```

module com.lagoona.counting_stacks {
  import S = com.lagoona.stacks;
  public message int elements();
  public class Stack {
    S.Stack stack; int count;
    method void initialize(S.Stack stack) {
      this.stack = stack; this.count = 0;
    }
    method int elements() {
      return this.count;
    }
    method void S.push(any obj) {
      this.count++;
      S.push(obj) -> this.stack;
    }
    method void S.pop() {
      this.count--; S.pop() -> this.stack;
    }
    method any S.top() {
      return S.top() -> this.stack;
    }
    method boolean S.empty() {
      return this.count == 0;
    }
    method void default() {
      current => this.stack;
    }
  }
}

```

Fig. 4. Adding counting to the stack abstraction and its implementation.

`elements` simply returns the counter value. The methods `S.push` and `S.pop` update the counter and forward their messages to the “basic” stack instance. Although not directly related to the extension we want to produce, we also have to implement the messages `S.top` and `S.empty`. The reason is that both of these messages return a value and can therefore not be handled by the generic message forwarding mechanism implemented in the `default` method. However, implementing the `default` method as shown allows this extension to be composed with other, unrelated extensions.

### 3 Applications

In this section, we illustrate how stand-alone messages and generic message forwarding address a number of recurring design and implementation problems that are practically apparent when COP is implemented using OOPL.

COP often requires *combining* multiple interface types that were defined independently, for example if they are to be implemented by a single implementation type. Since these combined interface types can again be defined independently, the *conformance* between interface types needs to fulfill certain requirements as well. Interface combination itself is already problematic in conventional object-oriented programming languages, since it can lead to *syntactic* and *semantic* conflicts. Often, these conflicts are referred to as “name clashes,” and the problem is considered to be solved by providing language mechanisms to work around it. For example, Java

supports overloading of method names, which can be used to avoid a subset of these conflicts. More general solutions are provided in Eiffel, which supports renaming of methods in descendant classes, and in C++, which supports a form of explicit qualification of methods. However, these techniques fail to take distributed extensibility into account, because they are only applied to fix a “name clash” once it has occurred. In Lagoona, both kinds of conflicts are ruled out *by design* since messages always have a unique identity. Interface combination in Lagoona thus has the following two properties: (a) any combination of interface types results in an interface type (no syntactic conflicts), and (b) any combination of interface types preserves all constituent messages (no semantic conflicts). Solving this (long-standing) problem in fact motivated the design of stand-alone messages to a certain degree. While the problem of interface combination has been known for a long time, the related problem of interface conformance has received attention only recently. Consider two interface types A and B that were defined independently by vendors A and B. Vendors C and D define—again independently—combinations of A and B, for example  $C = A + B$  and  $D = B + A$ . While both C and D support exactly the same messages, they do not necessarily conform to each other. Most object-oriented languages rely on a declared form of conformance, i.e. types are equivalent *by name* (or *by occurrence*) instead of *by structure* (or *by extent*). The usual objection to structural conformance is that it can lead to “accidental” conformance relationships, with the archetypal example being a Cowboy and a Shape both understanding a message `draw` with different semantics. Lagoona’s stand-alone messages provide a solution to this problem as well, as we can support structural conformance between interface types without the potential for accidental conformance. Recent proposals to extend Java with a form of structural conformance [11][12] result in a more complicated and less flexible design.

Another often encountered issue in the component-oriented programming domain is the *Fragile Base Class* paradox. The concept of *inheritance* was once hailed as the “golden way” toward extensible software systems. However, the mechanism is generally not suitable for achieving distributed extensibility. 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, B cannot be implemented without knowing the implementation details of A as well: On the one hand, if the developer of A implements `MulRem` by calling `Rem` repeatedly, `Rem` (and only `Rem`!) must be overridden to maintain an accurate count. On the other hand, if `MulRem` does not call `Rem`, we have to override `MulRem` as well as `Rem`. This is known as the *fragile base class problem*, and it can be resolved by following an elaborate set of design conventions [13]. If we want to avoid it altogether, we have to restrict the use of inheritance or abolish the mechanism completely. In Lagoona, generic message forwarding takes the place traditionally occupied by inheritance. It is easy to see how to solve the example problem using this mechanism. Instead of deriving a new class B, we develop an implementation type B that has a reference to an instance

```

package com.lagoona.pubsub;
public interface Publisher {
    void attach(Subscriber me);
    void detach(Subscriber me);
    Object get();
    void set(Object data);
}
public interface Subscriber {
    void update(Publisher from);
}

```

Fig. 5. Naive publishers and subscribers in Java.

```

module com.lagoona.pubsub {
    interface Publisher {attach, detach, get, set}
    message void attach(Subscriber me);
    message void detach(Subscriber me);
    message any get();
    message void set(any data);
}
interface Subscriber {update}
message void update(interface {get} from);
}

```

Fig. 6. Smarter publishers and subscribers in Lagoona, only `get` can be sent within `update`.

of A. 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.

The language concepts introduced by Lagoona also allow to resolve the *Component Reentrance* problem in a more elegant fashion. When we use messages and interface types to specify the functionality of certain instances, we often make the assumption that each operation executes *atomically*. However, for certain design patterns that rely on “callbacks” between instances this is not the case, leading to the *component reentrance* problem [14]. Consider the *Observer* (or *Publish-Subscribe*) design pattern [15] for example, which is used to achieve loose coupling between objects by implicit invocation. A publisher encapsulates some kind of data that is of interest to subscribers. When this data changes, the publisher automatically notifies all its current subscribers. Figure 5 illustrates how this design pattern could be modeled in Java using two interfaces `Publisher` and `Subscriber`. Subscribers `attach` themselves to a publisher, and whenever `set` is invoked, the publisher in turn invokes `update` on all registered subscribers. Subscribers then use `get` to retrieve the current state of the publisher and update themselves accordingly. While this sounds great, there are in fact several problems. For example, consider subscribers that send `attach` or `detach` to the publisher within their `update` method. Since the publisher is currently traversing some kind of data structure to update all subscribers, the effect of these operations becomes highly dependent on the implementation of this traversal. Even worse, subscribers might send `set` within their `update` method, resulting in infinite recursion. The component reentrance problem can be solved by implementing publishers very defensively, e.g. by cloning the data structure before traversal and by protecting the `set` method using some kind of flag. However, the problem really boils down to what messages can be sent to the publisher from within the `update` method. If we restrict this set of messages, we can *statically* ensure that the reentrance problem does not occur. Figure 6 shows how we would model the design pattern in Lagoona. Instead of typing the `from` parameter of `update` with `Publisher`, we introduce an anonymous interface type that only supports the `get` message. While subscribers can still send other messages if they have another reference to the publisher, or if they cast the `from` parameter accordingly, our description of the design pattern is still more accurate and elegant. In Java, we would have to introduce an artificial base type, e.g.

```

module com.lagoona.iterator {
  ...
  class ArrayForwardIterator {
    any[] data;
    method void default() {
      int j = 0;
      while (j < this.data.length) {
        current => this.data[j++];
      }
    }
  }
  class Array {
    any[] data;
    ...
  }
}

method ArrayForwardIterator forward() {
  ArrayForwardIterator i =
    new ArrayForwardIterator();
  i.data = this.data;
  return i;
}
class LagoonaIterator {
  Array array;
  ...
  method void action() {
    array.forward().print();
  }
}
}

```

Fig. 7. Implementing iterators in Lagoona by leveraging generic message forwarding for broadcasting.

PublisherJustGet, that we derive Publisher from.

Finally, Lagoona’s generic message forwarding mechanism reconciles the iterator pattern with component-oriented programming. Certain programming languages, for example CLU [16] and Sather [17], offer an *iterator* construct to traverse encapsulated data structures in a modular manner. In most object-oriented programming languages, iterators are “emulated” at the library level [15][18] and the iteration loop itself must be implemented manually every time an iteration is required. Using Lagoona’s mechanism for generic message forwarding, we can implement iterators that are as powerful as library approaches, but often as convenient to use as language approaches: Since the `default` method enables us to specify a strategy for forwarding messages in the imperative core language, we are by no means limited to just a single receiver. Instead, we can implement a generic *broadcast* mechanism for messages. Figure 7 shows how we can use this idea to implement iterators in Lagoona. The container `Array` implements a message `forward`, which returns an iterator instance. The iterator contains a reference to the elements to be traversed and fully encapsulates the iteration strategy. For this example, we have limited ourselves to forward iteration, but a `backward` message could easily be added, returning an iterator instance for backward iteration. The actual iteration is performed by simply sending a message to the iterator instance. The iterator itself does not implement any message but instead broadcasts all received messages to the elements in the container. The actual action to be performed on each element is located in a method of the container elements. Message parameters can be used to pass additional context information from the current control flow to this iterator method. This approach to iterators offers a much cleaner separation between the iteration code and the application code than traditional iterator schemes. All code related to the iteration is located in the module containing the container and its iterator functionality.

## 4 Implementation

To demonstrate the viability of our language design, we decided to implement the Lagoona compiler and runtime libraries themselves in Lagoona. Moreover, instead

of emitting some machine-specific native code, we wanted the Lagoona compiler to generate portable, type-safe, and verifiable code for a virtual machine.

The first obstacle we had to overcome when implementing the Lagoona compiler was to find a way to bootstrap the compiler. For this, we first implemented a simplified Lagoona compiler in Java, using C as intermediate language. To obtain an executable, the C code generated by this bootstrap compiler is translated to executable machine code by any ANSI C compiler. The bootstrap compiler was intended to be used only during the early stages of the compiler development. As soon the compiler was complete enough to translate itself, we abandoned the bootstrap compiler and Lagoona became self-hosted.

Lagoona's object model and message dispatch mechanism differ significantly from those found in traditional object-oriented languages such as Java, Smalltalk and C++. In Lagoona, any message can be sent to any object, and if no matching method exists, a default method has to be executed. This radical reinterpretation of the terms *message* and *method* requires some extra effort for executing Lagoona on existing virtual machine architectures. Most existing virtual machines like the Java Virtual Machine (JVM) [19] are intended to execute programs written in one particular language and offer little support for mechanisms not available in that particular language. In contrast to the JVM, the Microsoft .NET framework is a virtual machine architecture, implementing the ECMA Common Language Runtime (CLR) standard, that targets a wide range of source languages including Java, C++, Visual Basic, and C#. Thus, the .NET framework seems to be the ideal target architecture for a novel language like Lagoona. Unfortunately, while offering a great deal of flexibility as far as the instruction set is concerned, the .NET framework offers far less freedom when it comes to the type system. To allow interoperability between programs written in different languages, type-safe ("managed") code has to use the rigid *Common Type System* (CTS). To be able to execute Lagoona code on the .NET virtual machine, we either had to abandon verifiability or overlay our object model onto the Common Type System. Surprisingly, the latter is possible with surprisingly little runtime overhead as we describe in the remainder of this section.

Each message  $m$  in Lagoona is represented by a pair of types at the Common Type System level. The first type,  $interface_m$  is a CTS interface type containing a  $m$  as the single abstract method. The second type,  $stub_m$  is a class that implements  $interface_m$  and contains marshaling code. Objects of this type are instantiated if a message  $m$  cannot be directly delivered to an object and has to be handled by the generic forwarding method. Lagoona types are represented as CTS classes. Implementation types correspond to a regular class and interface types to an abstract class. For every method  $method(m)$  that a Lagoona type provides, the ability to directly receive the underlying message  $m$  is indicated at the CTS level by implementing the corresponding  $interface_m$  interface. Thus, at the CTS level the generated code can use the *isinst* (is instance) instruction to check whether a message

can be delivered directly or has to be handled by the default methods. Message sent to implementation types are directly resolved to plain method invocations at the CTS level and are thus not more expensive than simple method calls in other languages. Delivering a message to an interface types requires a *isinst* check first. If no immediate delivery is possible, an object of the message stub type *stub<sub>m</sub>* will be instantiated and passed on to the default method. The Lagoona compiler performs aggressive type inference to resolve as many message send operations to interfaces to message send operations to implementation types as possible. For this, among others, the default methods of each implementation type are analyzed. Often default methods contain very simple forwarding code, i.e. just forwarding the message on to another object. In this case analyzing the procedural forwarding code allows to deduct static type information and the message send operations are optimized accordingly. However, for more complex forwarding statements, for example loop statements used for broadcasting message to all objects in a container, this analysis does not yield any useful results and the runtime check remains in place.

## 5 Related Work

Stand-alone messages can be related to the concept of *multimethods* [20]. In a language supporting multimethods, such as Cecil [21], stand-alone messages could be “emulated” by introducing an additional dispatch parameter modeling the originating module. Despite recent progress regarding type-safety and modularity of multimethods [22], the concept is not yet supported in mainstream languages. Stand-alone messages are conceptually simpler than multimethods 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.

Recent work on *units* and *mixins* [23] is related to Lagoona in a more interesting way. With Lagoona, we have argued that programming languages for component-oriented programming need to combine traits from modular languages with traits from object-oriented languages in a certain way. Namely, we have to distinguish explicitly between messages and methods and we have to separate messages from types, binding them to modules instead. Units and mixins also aim at the combination of modular and object-oriented language constructs. Units provide a module concept that is more flexible than ours: Instead of fixing the import relations of a set of modules once and for all, units allow the composition of modules through separate linking specifications. This has several important applications, e.g. for the flexible creation of extended objects. Mixins provide a variation of inheritance (in the sense of subclassing) that allows derived classes to be parameterized by different base classes. However, Lagoona’s approach to forwarding and composition already subsumes mixins: while for mixins the base class relation is determined when units are linked, in Lagoona we can actually defer this relation until objects are instantiated. In summary, the units idea is very valuable and we hope to explore the integration of a more flexible module system (with a distinct “units” flavor) into

Lagoona in the future.

Component models, such as COM [10], CORBA [24], and JavaBeans [25], are industry standards that claim to support component-oriented programming. However, the main emphasis of these models lies on defining interoperability and packaging conventions in the form of design patterns, rather than on providing comprehensive support. Many component models also address aspects that are essentially unrelated to component-oriented programming—such as distribution, concurrency, cross-platform portability, and cross-language integration—but that nevertheless increase their complexity significantly. Component models seem to be a temporary solution that will survive only until better, more comprehensive ways to practice component-oriented programming become available. We do not want to imply that component models are completely useless, but rather that they only serve a temporary purpose as far as the component-oriented paradigm is concerned.

The paradigm of generative programming (GP) [26] is based on a number of ideas: domain specific languages, aspect-oriented programming (AOP), and generic programming. In GP, software systems are described in terms of domain specific languages that are used to encode domain knowledge on a high level. These descriptions are used to drive AOP [27] tools that integrate various reusable and basically unrelated “components” and aspects to produce customized applications automatically. The functional “components” are implemented using generic programming techniques (i.e. parametric polymorphism). While GP provides an interesting approach to source-level reuse and maintenance, its “components” are not components in the sense of component-oriented programming [7]. In GP (and AOP), “components” are reusable and parameterized abstractions that only exist on the programming language level, but not in the deployed application. Thus, once an application has been produced using GP, the “components” it consists of can not be reused or updated separately from the application they were compiled into.

## 6 Conclusions

The paradigm shift towards component-oriented programming is not yet reflected in programming languages. In the absence of dedicated COP languages, current COP practice often employs OOP languages developed before the notion of distributed extensibility was recognized as being important. This paradigm mismatch results in unnecessary design complexity and increased maintenance overhead.

We have been researching programming languages that expressly support COP. In this paper, we presented Lagoona, an experimental COP language that provides several new constructs in direct support of distributed extensibility while attempting to appear “familiar” to OOP practitioners. We were able to implement Lagoona using Lagoona itself, in the context of Microsoft’s .NET framework.

## References

- [1] M. D. McIlroy, Mass produced software components, in: P. Naur, B. Randell (Eds.), *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October, 1968*, Scientific Affairs Division, NATO, Brussels, Belgium, 1969, pp. 138–155.
- [2] M. Franz, The programming language Lagoon: A fresh look at object-orientation, *Software: Concepts and Tools* 18 (1) (1997) 14–26.
- [3] P. H. Fröhlich, Component-oriented programming: Why, what, and how?, Ph.D. thesis, Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA (Mar. 2003).
- [4] P. H. Fröhlich, M. Franz, Stand-alone messages: A step towards component-oriented programming languages, in: J. Gutknecht, W. Weck (Eds.), *Proceedings of the Joint Modular Languages Conference*, Vol. 1897 of *Lecture Notes in Computer Science*, Springer-Verlag, Zürich, Switzerland, 2000, pp. 90–103.
- [5] A. Snyder, Encapsulation and inheritance in object-oriented programming languages, in: N. Meyrowitz (Ed.), *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Portland, OR, 1986, pp. 38–45.
- [6] C. Szyperski, Import is not inheritance—why we need both: Modules and classes, in: *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Vol. 615 of *Lecture Notes in Computer Science*, Springer-Verlag, Utrecht, The Netherlands, 1992, pp. 19–32.
- [7] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley / ACM Press, 1998.
- [8] P. H. Fröhlich, M. Franz, On certain basic properties of component-oriented programming languages, in: D. H. Lorenz, V. C. Sreedhar (Eds.), *Proceedings of the Workshop on Language Mechanisms for Programming Software Components (at OOPSLA)*, Technical Report NU-CCS-01-06, College of Computer Science, Northeastern University, Boston, MA 02115, Tampa Bay, FL, 2001, pp. 15–18.
- [9] L. Cardelli, Typeful programming, SRC Research Report 45, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301 (May 24, 1989).
- [10] Microsoft Corporation, *The Component Object Model (Version 0.9)* (Oct. 1995).
- [11] M. Büchi, W. Weck, Compound types for Java, in: *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Vancouver, British Columbia, 1998, pp. 362–373.
- [12] K. Läufer, G. Baumgartner, V. F. Russo, Safe structural conformance for Java, Tech. Rep. OSU-CISRC-6/98-TR20, Department of Computer and Information Science, Ohio State University, Columbus, OH 43210-1277 (Jun. 1998).

- [13] L. Mihajlov, E. Sekerinski, A study of the fragile base class problem, in: E. Jul (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 1445 of Lecture Notes in Computer Science, Springer-Verlag, Brussels, Belgium, 1998, pp. 355–382.
- [14] L. Mihajlov, E. Sekerinski, L. Laibinis, Developing components in presence of re-entrance, in: Proceedings of the World Congress on Formal Methods in the Development of Computing Systems (FM), Vol. 1709 of Lecture Notes in Computer Science, Springer-Verlag, Toulouse, France, 1999, pp. 1301–1320.
- [15] E. Gamma, J. Vlissides, R. Johnson, R. Helm, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [16] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, CLU Reference Manual, Tech. Rep. MIT/LCS/TR-225, MIT Laboratory for Computer Science (Oct. 1979).
- [17] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, Iteration abstraction in Sather, ACM Transactions on Programming Languages and Systems 18 (1) (1996) 1–15.
- [18] A. A. Stepanov, M. Lee, The Standard Template Library, Tech. Rep. X3J16/94-0095, WG21/N0482, Silicon Graphics Inc. (1994).
- [19] T. Lindholm, F. Yellin, The Java Virtual Machine Specification, 2nd Edition, Addison-Wesley, 1999.
- [20] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, F. Zdybel, CommonLoops: Merging Lisp and object-oriented programming, in: N. Meyrowitz (Ed.), Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), Portland, OR, 1986, pp. 17–29.
- [21] C. Chambers, The Cecil language: Specification and rationale, Tech. rep., Department of Computer Science and Engineering, University of Washington, Box 352350, Seattle, WA 98195-2350, USA (Mar. 1997).
- [22] T. Millstein, C. Chambers, Modular statically typed multimethods, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Vol. 1628 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 279–303.
- [23] R. B. Findler, M. Flatt, Modular object-oriented programming with units and mixins, in: Proceedings of the International Conference on Functional Programming (ICFP), Baltimore, MD, 1998, pp. 94–104.
- [24] Object Management Group, The Common Object Request Broker: Architecture and Specification (Version 2.3.1) (Oct. 1999).
- [25] Sun Microsystems, The JavaBeans Specification (Version 1.01) (Jul. 1997).
- [26] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
- [27] G. Kiczales, J. Lamping, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.