Objects + Views = Components?

Martin Odersky

École Polytechnique Fédérale de Lausanne

Abstract. In support of flexible composition, components need to be adaptable and they need to have first-class plugs. Abstract state machines and object-oriented programming each satisfy one of these requirements very well but satisfy the other only incompletely. This paper describes views, a language construct which provides for both requirements in a balanced way.

1 Introduction

Increasingly, software is developed from pre-existing components rather than being built from scratch. Component middleware such as Corba [Pop98], COM [Box98] or Java-Beans [Eng97] allow applications to be constructed with the help of binary-components and foster the development of a component market [Szy98]. Components can also take the form of services which are implemented as separate software systems accessed over the internet [WWC92].

Looking beyond the technical details, components are parts which exist to be composed. The most important difference with respect to modules in traditional top-down development is that future compositions of a component are unknown to the component's authors. To be useful, a component library therefore needs to provide high flexibility in component reuse and composition. It has been argued that this is a difficult task, in particular because the object-oriented languages used to implement components do not provide the right abstractions [Höl93].

This paper explores language design issues in the area of composition. We argue that a component is composable only if two requirements are fulfilled: The component needs to be *adaptable* and its plugs should be *first-class values*.

A component is *adaptable* if clients can enrich its interface after the component has been constructed and delivered. Such changes consist typically in adding some data fields or methods which the original component did not yet support. The methods are to be defined in terms of the component's existing public interface. Such additions should not affect a component's source code. After all, source code changes by clients are impossible for binary and service components and pose severe version control problems for components existing in source form.

[1] to appear in: Abstract Machines 2000, Lecture Notes in Computer Science, © Springer Verlag.

The need for adaptations arises because at the time when a component is designed and implemented one often does not know precisely in which contexts the component will be used. As a simple example, consider the task of writing a class for lists. Which methods should this class offer? A fairly small implementation could be:

```
class List[a] = {
  def head: a = ...
  def tail: List[a] = ...
  def isEmpty: Boolean = ...
}
```

Of course there are many more useful functions to be included in the List class. Functions to select elements inside the lists, to form sublists, to concatenate lists, to filter elements according to some criterion, or to map functions over all elements come to mind. Furthermore, one might want to treat lists as representations of sets, in which case membership, union, intersection and set difference also should be supported. And one can think of many other useful operations on lists. The hard question is where to stop. Meyer [Mey88] recommends a "shopping list approach" to class design where essentially all useful methods one can think of should be added to an abstraction. In the case of lists, this would probably lead to many hundreds of methods, and even then clients of the list abstraction will likely miss a method they require. It seems preferable to provide a more compact list definition and let clients customize this definition as they require. The criterion for a complete class design would then be that every useful method for the class *can* be defined in terms of the public interface, not that the method already *is* defined. The "can"-completeness criterion is much easier to fulfill; for instance our simple implementation of lists above is already complete.

```
val SymTab {
  type Scope
  type Symbol = {
    def name: String
    def tp: Type
    def location: Scope
    ...
  }
  def lookup (scope: Scope, name: String): Symbol = ...
  def enter (scope: Scope, sym: Symbol): Boolean = ...
  def newScope (outer: Scope): Scope = ...
}
```

Fig. 1. Symbol Table Module

As another example, consider the task of constructing a component for symbol tables in a compiler. Specifications and algorithms for symbol tables are potentially portable over many compiler implementations. Figure 1 shows a sketch of a generic implementation. However, it's not clear what attributes should go into a symbol, as is illustrated by the ... in the Symbol type. Clearly, symbols will have a "name" attribute. One might also settle on fields that contain a symbol's type and location. But there are also other potential attributes which depend on the situation in which a symbol table is used. For instance, if our compiler has a code generator, symbols will need a field in which to store their address. On the other hand, if the compiler is coupled with a source code browser, we might need an additional attribute in a symbol which contains the list of all usage points of that symbol. Again, the precise definition of some aspect of the symbol table component depends on the contexts the component is used.

In classical structured programming, where data structures and code are separated, symbol tables would be adapted by changing their code, filling in the ... in the Symbol type as required by the application. But source code adaptation is impossible or at least undesirable in a component setting. Furthermore, the necessary adaptations to the symbol table module also violate the encapsulation principle, since fields needed only by the code generator module are placed in the common Symbol data structure which is accessible to all. Hence, better solutions for adaptation are called for.

The other requirement for components has to do with how flexibly they can be composed. One usually regards the run-time embodiments of a component's interfaces as its "plugs". In object-oriented languages plugs are often realized as objects which have methods which are accessible to clients. A plug is *first-class* if it can be treated like a normal value. In particular, one should be able to pass plugs as parameters to functions, and it should be possible to construct data structures with plugs as elements. As a very simple example for this kind of flexibility, consider the task of displaying the information associated with a symbol. Since there are so many different devices on which this information could be displayed, it makes sense to split the task in two. Symbols only provide a method toString, which converts the information associated with the symbol to a string. In other words, symbols support the interface

```
type Printable = {
  def toString: String
}
```

Then, a display device could provide a general display service for objects that "know" how to turn their information into strings:

```
def print(obj: Printable) = ...
```

If dev was such a device and sym was a symbol it would then be possible to write dev.print(sym). Of course, this assumes that symbols are values that can

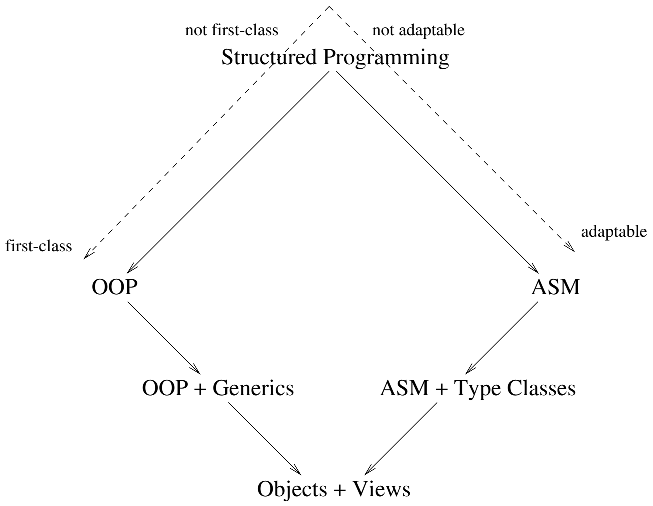

Fig. 2. Support for adaptability and first-class plugs.

be passed to the print function. In particular, the type Symbol must be compatible with the type Printable.

Since adaptability and first-class plugs are natural requirements for software components, one might expect that mainstream languages would provide support for both. Maybe surprisingly, this has not been the case so far. Figure 2 summarizes the state of the art, which will be further discussed in the next sections. Even though structured programming languages such as Ada or Modula-2 provide support for modules, which can be seen as an embodiment of components, the components thus defined are neither adaptable, nor do they have first-class plugs. Object-oriented programming leads to components with first-class plugs, but these components are not adaptable. Conversely, the approach taken by abstract state machines lends itself to the construction of components which are adaptable, but these components do not have first-class plugs. More advanced type systems and design patterns can in each case provide some of the missing functionality. Refining object-oriented programming with bounded genericity can provide adaptability to some degree, whereas refining functional programming and ASM's with a construct such as Haskell's type classes can provide some of the benefits of first-class plugs.

This paper presents *views* as a simple technique to provide both adaptability and first-class plugs. The technique uses standard object-oriented modeling techniques to model first-class plugs. A view is a syntactic construct which provides new methods for objects of an existing class, thus providing implementations for additional object interfaces. Views that extend a component can be distributed over arbitrary client components.

The discussion in this paper is in the context of statically typed languages. As notation for our examples, we use Funnel [Ode00], together with some hypothetical extensions.

The rest of this paper is structured as follows. Section 2 discusses strengths and shortcomings of the abstract state machine approach to software composition. Section 3 discusses the same for the object-oriented approach. Section 4 presents views. Section 5 discusses related work and Section 6 concludes.

2 Abstract State Machines

The abstract state machine approach reverses the usual relationship between functions and data. Rather than having mutable data structures which are operated on by immutable functions, we have immutable data, which are operated on by both mutable and immutable functions. This makes use of the following analogy between function application and selection.

$$f(x) \quad \equiv \quad x.f$$

In other words, a data selector can be seen as a single argument function over the selected data domain and *vice versa*. This equivalence makes sense on the left-hand sides of assignments as well:

$$f(x) := E \quad \equiv \quad x.f := E$$

We can thus re-interpret assignment of an expression to a variable field `f` of a record `x` as an update of a mutable function `f` at the position `x`. If we apply this re-interpretation consistently, we arrive at only immutable data, which serve as index structures for possibly mutable functions. The immutable data structures can be described as algebras, which leads to a simple formal treatment of state change. This treatment lends itself to the verification of properties of imperative programs with pointers, which are very hard to prove otherwise [Ode98].

Abstract state machines add to this the idea of specifying change by a global set of rules, which collectively specify an execution step. The question whether execution is described by a global set of rules or by more conventional statements with control flow constructs is irrelevant to the discussion in this paper and will henceforth be ignored. In any case, the abstract state machine approach leads very naturally to components which are adaptable.

For instance, consider the question which functions to provide together with a `List` type. Abstract state machine programmers would do the same functional programmers do – they would define `List` as an algebraic data type which can then be accessed by pattern matching. Using Funnel as a notation, the algebraic data type is expressed as a **class** with two **case**s and the pattern match is expressed by a call to the `match` method of a list. For instance, here is the definition of a list together with two representative functions over lists:

```
class List[a] = {
  case Nil
  case Cons (head: a, tail: List[a])
}

def append (xs: List[a], ys: List[a]) = xs.match {
  case Nil: ys
  case Cons (x, xs1): List.Cons (x, append (xs1, ys))
}
def map [a,b] (f: a → b, xs: List [a]) = xs.match {
  case Nil: xs
  case Cons (x, xs1): List.Cons (f (x), map (f, xs1))
}
```

All operations over lists can be expressed in terms of pattern matching. Hence, a list operation can be placed in an arbitrary module, and new list operations can be added in clients of the list abstraction as needed. In a word, lists designed as algebraic data types are adaptable. This might be too trivial to mention except for the fact that lists represented as objects which export all their methods are not adaptable in the same way, as clients cannot add new methods to an existing list. This point will be further discussed in the next section.

The abstract state machine approach also ensures adaptability for mutable functions. For instance, consider the situation where a symbol table module as given in Figure 1 is adapted to work with a code generator client. The code generator needs to maintain with each symbol its address. This could be achieved by defining a mutable function from symbols to their addresses in the code generator module:

```
var adr (sym: Symbol): Int := 0
```

The := 0 part in the declaration above defines the initial value of the mutable function to be 0 over all of its domain. The code generator module could then enter a symbol's address by an indexed assignment such as

```
adr (sym) := nextLocalAdr
```

and it could access a symbol's address by calling adr(sym). No change to the symbol table module is necessary. Furthermore, since addresses are dealt with only in the code generator, the definition of adr can be kept local to the code generator module, so that we have gained encapsulation as well as adaptability.

This is all good news. However, the abstract state machine approach does not satisfy our second criterion, that components have first-class plugs. For instance, we still cannot define a general print service with a method print(p: Printable) which can be applied to symbols. Indeed, neither classical structured programming nor abstract state machines provide a subtype relation which would let us treat a Symbol as a special instance of the more general Printable abstraction. This is not an oversight either. To be useful, the Printable abstraction must define a method such as toString, which turns a Printable into a String. Hence, data must be an abstraction which includes functions operating on the data. This view is at odds with the data structuring approaches of functional programming and abstract state machines, which treat data as domain structures for functions defined elsewhere. In a sense our two criteria of adaptability and first-class plugs seem to be at odds with each other. First-class plugs require grouping functions with data whereas adaptability requires grouping functions separately.

The functional programming language Haskell [HPW92] pioneered the concept of *type classes* [WB89], which can to some degree make up for the lack of first-class plugs in functional systems. It would be quite possible to extend this construct to abstract state machine languages as well.

A type class represents a shared property of a set of types – alternatively, it can be identified with the set of types which satisfy the property. A property states that a type supports a given set of functions with given signatures. For instance, here is a declaration of a type class[1].

```
type class Printable where {
   toString :: a → String
}
```

This states that a type T belongs to the type class Printable if there is a function toString, which takes a value of type T and yields a value of type String. Types become members of type classes by explicit **instance** declarations. Example:

```
instance Printable Symbol where {
   toString (sym) = ...
}
```

The **instance** declaration above declares Symbol to be an instance of type class Printable and provides the implementation of the toString method of Printable.

One can now define functions which are polymorphic over all types which are instances of a given class. Example:

```
print :: Printable a ⇒ a → ()
print x = ... toString (x) ...
```

This defines a function print which takes as parameters values of type a, where a is an arbitrary instance of type class Printable. The call to toString in print will invoke the method which is appropriate for the concrete type to which the type variable a is instantiated. For example, if sym is a Symbol then the type variable a is instantiated to Symbol and print (sym) invokes the toString method given in

[1] We use Haskell syntax except for writing **type class** where Haskell uses **class**.

Symbol's instance declaration. The type Printable a $\Rightarrow$ a $\rightarrow$ () is called a *qualified type* [Jon92] and the (Printable $\Rightarrow$) prefix is called a *context*.

Representing component interfaces as instances of type classes, one can hence write generic functions which work with any component plugs which are instances of a given type class. This yields many of the benefits of first-class plugs, but some shortcomings remain. First, generic pluggability is achieved in a somewhat roundabout fashion. Second, we still cannot define data structures over component plugs. Consider for instance a data structure of lists whose elements are printable components, not necessarily all of the same type. One might try to express this as a List[Printable], but this would be ill-formed, since Printable is a type class, not a type. The qualified type Printable a $\Rightarrow$ List[a] would not do either, since that type describes a list of objects which all have the same instance type of Printable. One could go further, expressing heterogeneous data structures by means of existential types [LÖ96], but this comes at the price of even greater conceptual complexity.

3 Object-Orientation

Instead of starting with an adaptable architecture and trying to emulate first-class plugs, mainstream programming has instead favored the object-oriented approach, which has first-class plugs built in, yet needs additional provisions to achieve some degree of adaptability.

In object-oriented design and programming, one groups data (called *instance variables*) and functions operating on the data (called *methods*) together in an object. Data and functions collectively are called the *fields* of an object. There is a subtyping relation (written <:) between object types, which lets one use an object with a certain set of fields in a context where only a subset of the fields is required. This provides all we need for first-class plugs. For instance, to make symbols printable, it suffices to make the type Symbol a subtype of Printable, which requires the definition of a toString method in Symbol:

```
class SymTab = {
  class Symbol <: Printable = {
    ... (fields as before) ...
    def toString: String = name ++ ":" ++ tp.toString
  }
}
```

Then we can write:

```
val sym = new SymTab.Symbol
...
dev.print (sym)
```

Are objects also adaptable? One might think so, because with *inheritance* it is possible to define a subclass which adds new fields to an existing class. For instance, it is possible to define in the code generator class CodeGen a new class Symbol which inherits from the original symbol class as defined in module SymTab:

```
class CodeGen = {
   class Symbol <: SymTab.Symbol = {
      var adr: Int
   }
   ...
}
```

Instance objects of class CodeGen.Symbol have all the fields of original symbols, plus the mutable adr field. So it seems we have managed to adapt symbol table modules to a code generator client. But closer inspection shows us that this is not really the case. There are two problems with adaptation through inheritance. One concerns types, the other object creation. Looking at types first, we notice that function lookup in our symbol table module still returns objects of type SymTab.Symbol, not objects of type CodeGen.Symbol. So the added adr information does not show up in the type.

```
class Symtab {
   ...
   class Symbol ...
   def lookup (scope: Scope, name: String): Symbol
   ...
}
```

To access the adr field of a code generator symbol stored in the symbol table, we need a dynamic type cast. In other words, we can gain adaptability only by subverting the type system, and running the risk of dynamic type errors. A safer alternative makes use of type parameterization. For instance, here is a re-formulation of the symbol table class using *bounded genericity*:

```
class Symbol <: Printable = {
   ... (fields as before) ...
}
class SymTab [s <: Symbol] = {
   def lookup (scope: Scope, name: String): s = ...
   def enter (scope: Scope, sym: s): Boolean = ...
   def newScope (outer: Scope): Scope = ...
}
```

The class for symbol tables is now parameterized by a type variable s which represents the actual type of symbols stored in the table. The only requirement on instantiations of s is that they are subtypes of Symbol. Some gluing is needed at the top-level of an application to instantiate the class to the right symbol type:

```
val symTab = new SymTab[CodeGen.Symbol]
```

The other problem with achieving adaptation with inheritance has to do with object creation. Taking our symbol table example, we note that we have not looked yet at code which creates symbols to be stored in the table. Let's say symbols are created during tree attribution, in class Attr:

```
class Attr = {
  ...
    new Symbol(name, tp)
  ...
}
```

This is not yet precise, because we have not yet specified which symbol should be created during tree attribution. If it is the original SymTab.Symbol that is created, we are missing an adr if we want to couple tree attribution with code generation. On the other hand, if it is a CodeGen.Symbol which is created, we have hardwired the relationship between tree attribution and code generation. Now it would be the tree attribution part which lacks adaptability, because it would not be able to deal with different kind of symbols, for instance those required by a source code browser. To overcome this problem, we can make use of the *Factory* design pattern [GHJV94]. Factory objects would have an interface of the form:

```
type Factory [t] = {
  def make (...): t
}
```

There would have to be an appropriate factory for every subtype of symbol which is being defined. E.g. in class CodeGen:

```
class CodeGen = {
  ...
  val SymFactory: Factory[Symbol] = {
    def make(name: String, tp: Type): Symbol =
      new CodeGen.Symbol(name, tp)
  }
  ...
}
```

The Attr class will be parameterized with the factory which is to be used for symbol creation. We also have to parameterize Attr with the actual type of symbols to be created. This yields:

```
class Attr [s <: Symbol] (symFactory: Factory [s]) = {
  ...
  symFactory.make (name, tp)
  ...
}
```

Even more gluing is needed at top-level, to obtain the correct instantiation of the tree attribution module:

```
val symTab = new SymTab [CodeGen.Symbol]
val attr = new Attr [CodeGen.Symbol] (CodeGen.SymFactory)
```

The preceding discussion has shown that it is possible to obtain a certain degree of adaptability in object-oriented designs, but this requires planning. If we had not foreseen the possible later extension of the symbol type, we would not have provided the necessary parameters for subtypes and factories. The hooks necessary for enabling future extensions can clutter programs considerably, in particular if we envisage multiple coexisting extensions. For example, the current compiler framework would enable a single extension of Symbol, say for a code generator or a browser, but it would not yet enable multiple coexisting extensions, say where a symbol has both an adr field for a code generator and a uses field for a browser. Multiple extensions can be supported by stacking extensions on top of each other but this requires even more complex protocols.

4 Views

In the previous two sections, we have discussed two program structuring methodologies which have complementary strengths. Each methodology supports one of our two criteria of adaptability and first-class plugs, but supports the other only incompletely and at the price of complex language features and design patterns. In this section, we propose a more symmetric combination which can address both criteria equally well. This combination arises by translating concepts first developed for type classes into an object-oriented setting.

When comparing type classes in functional programming and object-oriented type systems, we find some strong analogies:

type class	$\approx$	type
instance relation	$\approx$	subtype relation
instance declaration	$\approx$	subtyping clause

A type class corresponds to a type in the object-oriented setting. For instance, instead of having the type class Printable, we would have the type Printable, which has all "printable" types as subtypes. Consequently the instance relation between types and type classes becomes the subtype relation between types.

Finally, an instance declaration in Haskell corresponds to a subtyping clause of a class declaration. The analogy highlights an important difference between the approach of Haskell and the one of object-oriented languages: in the latter, a subtyping clause is textually part of the extending class whereas in Haskell, an instance declaration can be given anywhere. Haskell's approach is adaptable, in that instance declarations for a type can be given after the type is defined. The object-oriented approach is not, since adding new subtype edges requires changes in the source code of the subtype.

Languages with structural subtyping do not need explicit subtyping clauses in class declarations. Instead, the subtype relation is determined only by the structure of the classes: Classes with more fields become subclasses of classes with fewer fields, as long as the types of common fields are the same or stand in turn in a subtyping relation. This gives some more flexibility in the subclass relation. But it still does not solve the adaptation problem since it does not provide a way to augment a given class with new fields.

Seeing these disparities it seems promising to provide a construct on the object-oriented side which matches more closely the distributed nature of instance declarations. This construct is called a *view*. A view adds new fields to values of an existing type. For instance, the following view declaration would add the method toString to all values of type Symbol.

```
view (this: Symbol): Printable {
  def toString: String = this.name.toString ++ ":" ++ this.tp.toString
}
```

A view takes the form of an unnamed function, which takes a single parameter of some type A and yields an object of type B. The view establishes a subtype relationship between A and B by giving declarations of all fields of B which are not yet present in A. All fields of A are inherited; i.e. they form implicitly part of the resulting object. It is only possible to add new fields to a class with a **view**, not to change existing fields.

Subtype clauses

Subtype clauses can be regarded as syntactic sugar for view declarations with empty bodies. For instance, our previous class declaration

```
class Symbol <: Printable = {
  ...
  def toString: String = name ++ ":" ++ tp.toString
}
```

would be equivalent to

```
class Symbol = {
   ...
   def toString: String = name ++ ":" ++ tp.toString
}
view (this: Symbol): Printable = {}
```

Unlike a subtyping clause, a view declaration can be given anywhere in the program; it need not appear textually adjacent to the declaration of the view's parameter class.

Variables in Views

Besides methods, views can also define mutable variables. Here is an example:

```
type Adr = { var adr: Int }
view (sym: SymTab.Symbol): Adr = {
   var adr: Int
}
```

These declarations, when placed in the code generator module of a compiler, would be equivalent to the mutable function which we have used in the abstract state machine structuring:

```
var adr(sym: Symbol): Int
```

The only difference concerns selection syntax – we use the object-oriented version sym.adr instead of the previous adr(sym).

Encapsulation

More interestingly, we can also have information encapsulated in views. Consider for instance an encapsulated abstraction for address fields, which lets only even addresses be stored. This is accomplished as follows.

```
type Adr = {
   def setAdr (x: Int): ()
   def getAdr: Int
}
view (sym: SymTab.Symbol): Adr = {
   var adr: Int
   def setAdr (x: Int) =
      if (x % 2 == 0) adr := x
      else error ("bad address")
   def getAdr = adr
}
```

This view defines a variable adr along with setter and getter methods. Since the variable is not part of the view's target type Adr, it is not accessible to clients of the view.

Conditional Views

Sometimes, a parameterized type can implement a view only if some constraints on the type parameters are satisfied. Consider for instance structural equality as implemented by the interface

```
type Eq [a] = {
   def equals (other: a): Boolean
}
```

A type T supports structural equality testing if it implements the unary equals method which takes other values of type T as parameters. In other words, T supports structural equality testing if T <: Eq[T]. Now, consider the parameterized type of lists, List[a]. Should lists support structural equality testing? Clearly, we can compare lists structurally iff we can compare the list elements. This can be expressed by the following, *conditional* view declaration.

```
view [a <: Eq [a]] (this: List[a]): Eq [List [a]] = {
   def equals (that: List [a]) = {
      this.isEmpty && that.isEmpty ||
      !this.isEmpty && !that.nonEmpty &&
      this.head.equals (that.head) && this.tail.equals (that.tail)
   }
}
```

The [a <: Eq [a]] construct introduces a type parameter a for the view which is bounded by Eq[a]. The type variable can hence be instantiated to any type T which is a subtype of the bound Eq[T]. Note that this gives us a parameterized type which has or lacks method equals, depending on whether or not the type of the listed elements has the same method.

Semantics

The rest of this section discusses detailed design and implementation choices for views. The most fundamental design choice concerns name resolution of fields defined by views. Such resolution can be static, based on compile-time types or it can be dynamic, based on run-time types. To see the difference, consider a variant of our symbol table code. Assume that we want to further refine the type of symbols into variable symbols and function symbols:

```
type Symbol = {
   ... (fields as before) ...
}
type VarSymbol <: Symbol = { ... }
type FunSymbol <: Symbol = { ... }
```

In the code generator, one might need a function loadAdr which returns the instruction for loading a given symbol's address. The result of this function will depend on the kind of symbol encountered. It is therefore reasonable to implement loadAdr as a method of type Symbol, to be overridden in subtypes. But as before, we want to keep the source code in Symtab unchanged. Again as before, we resort to views for implementing the new functionality in the code generator client. Instead of a single view we now need three views, one for each kind of symbol:

```
type Loadable = {
   def load: Code
}
view (sym: Symbol): Loadable = {
   def loadAdr: Code = error ("can't load")
}
view (sym: VarSymbol): Loadable = {
   def loadAdr: Code = LLA (sym.adr)
}
view (sym: FunSymbol): Loadable = {
   def loadAdr: Code = LCA (sym.adr)
}
```

Can this work? Assume that the code generator calls the loadAdr method of a symbol it retrieved (via lookup) from the symbol table. The static return type of lookup is Symbol. So if we perform name resolution based on the static type, the loadAdr will always result in error("can't load"). In an object-oriented language such behavior would be counter-intuitive. We therefore pick dynamic name resolution, where the loadAdr method is chosen according to the run-time type of the symbol returned from lookup. If the returned symbol represents a variable, we invoke the loadAdr method of the view for VarSymbol. If the symbol represents a function, we invoke the loadAdr method of the view for FunSymbol instead. Only if the returned symbol is neither a variable nor a function, the loadAdr method of the view for the base class Symbol is invoked.

In the last example the call to loadAdr was resolved as if loadAdr was a regular method in Symbol, which was overridden in VarSymbol and FunSymbol. We can indeed often view a program with views as equivalent to a program with multiple inheritance where all views are inlined in their base classes. View methods then become regular methods and target types of views become elements of subtyping clauses. The correspondence with multiple inheritance programs serves as a guide in the definition of the semantics of views and the context-dependent syntax rules governing them.

The context-dependent syntax rules are based on the concept of the *view graph*. The view graph VG of a project has as nodes the set of types defined in all source files of the project and as edges the set of views between types. Subtyping clauses are also interpreted as view edges. Furthermore, we regard the methods defined in a class as defined in an additional view which goes from a new node representing a class implementation to the proper class type itself. A field which occurs textually in a record type T is said to be *defined* by T. A field which occurs textually in a view V is said to be *implemented* by V. The subtyping ordering $\leq$ on classes induces an ordering on views as follows: A view from S to T precedes a view from S' to T' if $S \leq S'$. We then require the following:

- The view graph must be acyclic.
- A view from S to T may only implement fields which are not already defined in S.
- When selecting a field f from an expression e of type T, there must be exactly one type reachable from T in VG which defines f.
- Let I be the node representing a class implementation and let T be a type reachable from I in VG. Then every field f in T must have a best implementation for I. This means: First, there must be a view on a path between I and T which implements f. Second, among all the views on paths between I and T which implement f, there must be one view which precedes all others according to the induced subtyping ordering on views. This requirement corresponds to the usual requirement for multiple inheritance that inheritance of fields must be unambiguous.

It is worth noting that some of these rules are global in that they require knowledge of the complete view graph. To check such global requirements in a system with separate compilation, one could delay some type checking until link time. As an alternative one could also maintain a project-wide repository in which all current view declarations are maintained. Consistency of the view graph can then be checked incrementally, as views are added and deleted.

5 Related Work

The shortcomings of the object-oriented approach in the area of adaptability have been known for some time. They have given rise to a varied body of research. The work on subject-oriented programming [HO93] identified the need for multiple, independent roles played by a single object. Aspect-oriented programming [KLM+97], hyperspaces [TOHJ99], and adaptive programming [MSL00] all address this requirement with systems which act as source to source transformations. The idea is in each case that different aspects of a program are defined in separate source files which are then merged using a "weaver" tool. Binary component adaptation [KH98] provides mechanisms to perform similar changes on Java class files. Both weavers and binary component adaptation are possible approaches for implementing views. Compared to these approaches, views are more restrictive in that they support only strict extensions of existing classes. One advantage of this restriction is that strict extensions commute, so that the semantics of the weaver tool becomes a non-issue.

Multi-methods [BKK86, Cha92] are another approach to extend the limitations of pure object-oriented programming. Here, a method is no longer attached to a single class, and method selection takes the dynamic type of all method arguments into account. Like views, multi-methods can be defined independently of their argument classes. But multi-methods are more general than views in that they abandon the concept of a distinguished receiver argument and in that they require multiple dispatch. Open objects [MC99] are an idiom of multi-methods which is comparable to views.

Like views, Haskell's type classes allow a distributed definition of functionality over data types. Qualification of type variables in a qualified type corresponds to F-bounded qualification of views [CCH+89]. A number of different designs for type classes have been developed [HHPW94, NP93, Jon92, Jon93, Jon00]. Our view proposal is most closely related to *parametric type classes* [CHO92], in that the bound of a type variable a may have additional parameters other than a itself. Type classes are usually studied in systems without subtyping. Where subtyping is added [Kae92], name resolution is based on static types, whereas views are based on dynamic types.

A language construct called "view" has also been proposed for functional programming [Wad87]. These views serve as alternative decompositions of sum types such as Haskell's algebraic data types. By contrast, the views presented here provide alternative accesses to product types such as records or objects. Both designs have in common that a view can be defined in program parts other than the one which defined its base type.

6 Conclusion

Do objects plus views equal components? This paper has argued that truly reusable component libraries require the ability to adapt components by extending them with new functionality, and to compose components via first-class plugs. Object-oriented programming and abstract state machines each provide one of these two keys to successful composition. The view construct aims at providing both of these keys.

Work is currently underway on a complete design and implementation of views in the context of Funnel, an implementation language for functional nets. Only future experience will tell whether the two keys are sufficient for the construction of truly reusable component libraries. That's why the title of this paper still ends in a question mark.

References

[BKK86] D.G. Bobrow, K. Kahn, and G. et al. Kiczales. CommonLoops — merging Lisp and object-oriented programming. In *Proc. OOPSLA'86*, pages 17–29, 1986.

[Box98] Don Box. *Essential COM.* Addison Wesley, 1998.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP*, 1992.

[CHO92] Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181, June 1992.

[Eng97] Robert Englander. *Developing Java Beans.* O'Reilly & Associates, 1997.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[HHPW94] Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *Proc. 5th European Symposium on Programming*, pages 241–256, 1994. Springer LNCS 788.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proc. OOPSLA*, pages 411–428, September 1993.

[Höl93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proc. ECOOP*, volume 707 of *Springer Verlag, Lecture Notes in Computer Science*, pages 36–56, 1993.

[HPW92] Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992. more recent versions at `www.haskell.org/definition`.

[Jon92] Mark P. Jones. A theory of qualified types. In *Proc. 4th European Symposium on Programming*, pages 287–306, February 1992. Springer LNCS 582.

[Jon93] Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 52–61, June 1993.

[Jon00] Mark P. Jones. Type classes with functional dependencies. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 230–244. Springer Verlag, March 2000.

[Kae92] Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 193–204, June 1992.

[KH98] Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proc. European Conference on Object Oriented Programming*, Lecture Notes in Computer Science. Springer Verlag, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming*, Finland, June 1997.

[L9̈6] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, May 1996.

[MC99] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proc. ECOOP*, number 1628 in LNCS, pages 279–303. Springer Verlag, June 1999.

[Mey88] Betrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MSL00] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology*. Kluwer, 2000.

[NP93] Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 409–418, 1993.

[Ode98] Martion Odersky. Programming with variable functions. In *Proc. International Conference on Functional Programming*, September 1998.

[Ode00] Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, March 2000.

[Pop98] Alan Pope. *The Corba Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison Wesley, 1998.

[Szy98] Clemes Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TOHJ99] P. Tarr, H. Ossher, W. Harrison, and S. "Sutton Jr.". *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. ICSE*, pages 107–119, 1999.

[Wad87] Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.

[WB89] Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.

[WWC92] Gio Wiederholt, Peter Wegner, and Stefano Ceri. Towards megaprogramming. *Communications of the ACM*, 1992. November.

This article was processed using the LaTeX macro package with LLNCS style