# Objects + Views = Components?

Martin Odersky
EPFL

Abstract State Machines - ASM 2000



## Components

- Components have become all the rage in software construction.
- Everybody talks about them, but hardly anybody uses them.
- Why is that?

## What is a Component?

- A component is a part of a greater assembly - either another component or a whole program.
- The purpose of a component is to be *composed* with other components.
- Typically, the other components and the composition is not known at the time a component is constructed.
- "*Pluggable parts*"; a component's plugs are its interfaces.



## What Makes a Component Composable?

- To support composition in flexible ways, components should be adaptable and their plugs should be first class values.
- Adaptable: The ability to change an interface of a component after the component has been constructed and delivered.
  - Changes are typically additions of new methods.
  - Changes to a component may not affect the original source code.
- First-class: The ability to treat plugs of components as normal values. In particular,
  - Plugs can be parameters to functions.
  - It should be possible to construct data structures with plugs as elements.

## Example for Adaptation: Symbol Tables

- Consider the task of writing a symbol table component for a compiler.
- What attributes should a symbol have?

  Name
  Type
  Location
  If there is a code generator: Address ?
  If there is a browser: Usage info ?
  Anything else?

- There is no good a-priori answer to these questions!
- What's needed is a minimal implementation of symbols which can be *customized* by the client.



## Example for First-Class Plugs: Printing

- Say we want to provide ways to display the information associated with a symbol.
- But we don't know a priori on what device the contents should be printed.
- This is easy to solve: Simply provide in the symbol an implementation of the interface

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

- Then we can define for each device a general print service

```
def print (t: Printable) = …
```

## Printing a Symbol Table

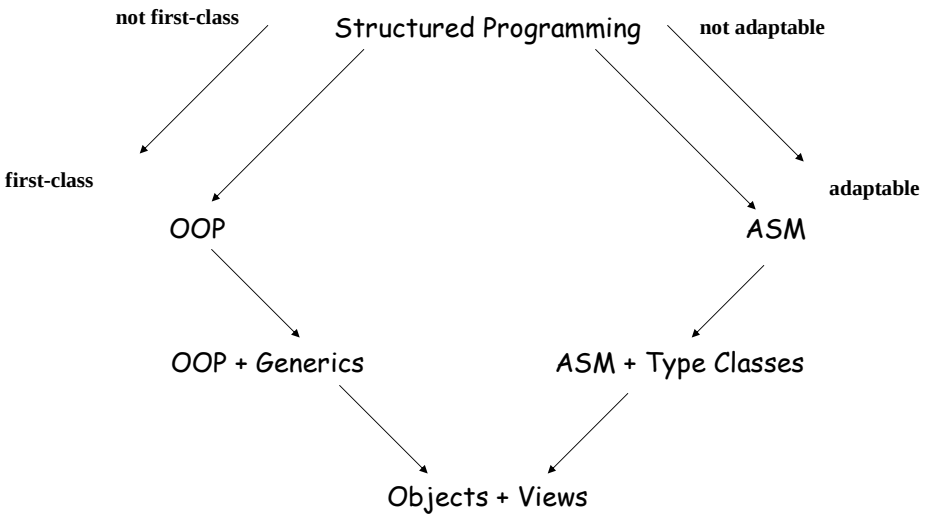
- The printing service can be invoked as follows:

```
sym: Symbol
dev.print (sym)
```

- Of course, this assumes that symbols are values that can be passed to the print function.
- In particular the type Symbol must be compatible with the type Printable.
- (The notation we use here is *Funnel*, the functional net language which is currently being developed in our group).



## State of the Art



## Structured Modular Programming

- Programs are partitioned into modules.
- Modules define data types and functions or procedures which access data.
- Modules can hide data by means of abstract types (e.g. in Modula-2: opaque types).
- Is this structure adaptable?



## Symbol Table Module

- Here's a definition of a module for symbol tables. We use Funnel as notation, but restrict ourselves to concepts found in Modula-2.

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

Question: What changes are necessary to add address fields to symbols, which are to be maintained by a code generator module?

## Classical Modules Do Not Produce Adaptable Components

- Customization of symbol tables requires changing their code.
- We need to add new fields to the definition of Symbol.

$\Rightarrow$ Classical modules are not adaptable.

- Classical modules do not have first-class plugs either.
- It is not possible to pass a Symbol as parameter to a function which takes a Printable.
- This is not an accident, as subtyping would require dynamic binding.



## Gaining Adaptability - The ASM Approach

- ASM's reverse the usual relationship of state and data. Rather than having mutable state as part of a data structure, we have immutable data as domains of mutable functions.
- This makes use of the equality

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

- In other words, field selectors can be seen as functions over the data they select.
- The analogy makes sense for mutation as well:

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

(the idea goes back to Algol W (1965), has been largely ignored since).

## The ASM View Helps in Proofs

- It's very hard to prove properties of programs which contain assignments to fields accessed via references such as

$$x.f := E.$$

- Hoare-logic does not apply, since references violate the substition principle for assignment:

$$\{ [E/x] P \} \quad x := E \quad \{P\}$$

- The above equation holds as long as x is a simple variable, but breaks down if x is a field accessed via a reference.



- Of course, nothing is gained per se by renaming x.f to f(x).
- But there is one important difference between the two forms:
- The ASM form f (x) := E allows x to have *structure* (for instance x could be a value of an inductively defined type).
- We can make use of that structure in program proofs, using structural induction over indices, analogously to the use of range induction in programs that use linear arrays.
- See: "Programming with Variable Functions", ICFP 1998.

## The ASM View Helps in Program Structuring

- The fields of a record all have to be defined in the same place. (we simplify for the moment by disregarding inheritance).
- On the other hand, mutable functions over a common domain can be placed anywhere, not necessarily where the domain is defined.
- In particular, new mutable functions can be defined after an index structure is defined and shipped as part of component

$\Rightarrow$ Components are adaptable!



## Example: Address Fields for Symbols

- To add address information to symbols, we simply define:

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

- This definition can be placed in the code generator module; no change to the symbol table module is necessary.
- Address attributes can be be encapsulated in the code generator module, they need not be visible outside of it.
- So we have gained both adaptability and better encapsulation.
- But: components are still not first class.
- For instance, it's still not possible to pass a symbol to a generic print function.

## First-Class Components - The OOP Approach

- A plug which is packaged as an object is a first class value.
- Example: Symbols

```
class SymTab = {
  class Symbol extends Printable = {
    ... (fields as before) ...
    def toString: String = ...
  }
}
```

- Then we can write

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



## Are Objects Adaptable?

- One might think they are, because of *inheritance:*

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

- Symbols in CodeGen inherit the fields and methods of symbols in SymTab, and add the CodeGen-specific field adr.
- Can this work?

## Problem: Types

- SymTab.lookup still retuns Symtab.Symbols not CodeGen.Symbols:

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

- Hence, a dynamic type cast is needed to extract the extra address information from a symbol table.



## Problem: Object Creation

- Furthemore, symbols are typically created in another component (say class Attr).

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

- Symbols thus created do not have adr fields.
- If we want to add them for supporting a code generator we have to change the Attr component.
- So adaptability is lost.

## Objects + Generic Types

- We can solve the typing problem by making all participants generic over the actual types of symbols used.
- Example:

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

- Some gluing is needed at top-level:

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

- The payoff is that no type casts are needed.



## Factories

- We can solve the creation problem by using the *Factory* design pattern.
- The idea is that all components which create symbols will be parameterized with a factory object which does the actual creation. Example:

```
type Factory [ T ] = { def make: T }
class Attr [ST <: Symbol] (symFactory: Factory [ ST ]) = {
   ...
      symFactory.make (name, type)
   ...
}
```

- Even more gluing is needed at top-level:

```
attr = new Attr [CodeGen.Symbol]  (CodeGen.symFactory)
```

## Evaluation of OOP

- Some degree of adaptability can be achieved by using generic types and design patterns with OOP.
- However: This requires a lot of planning.
- Need to parameterize by both types and factory objects.
- Multiple coexisting extensions can be supported by *stacking*, but this requires even more planning.



## ASM Structure + Type Classes

- Rather than trying to make OOP more adaptable, we can also try to emulate first-class plugs in the ASM structure.
- This approach has been pioneered by Haskell's *type classes*.
- A type class represents a property of a type.
- The property states that a type supports a given set of methods.

## Type Classes

- Here's a declaration of a type class (Haskell uses just class instead of type class):

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

- This says that a type T belongs to Printable if there is a function toString, which takes a T and yields a String.
- Types have to be declared explicitly as members of a type class:

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



## Qualified Types

- Functions can be generic over all types which belong to a given type class. Example:

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

- This says that function print can take any parameter which has an instance of Printable as type.
- The call to toString in print will pick the method appropriate for the run-time type of print's parameter.
- The qualification Printable a ⇒ is called a *context*, and the type of print is called a *qualified type*.

## Do Type Classes Yield First-Class Plugs?

- Not quite, since a type class is not a type.
- Plugs can indirectly be members of type classes, but they still cannot be values of (general) types.
- Hence it is not possible to create a list of printable objects, say. The "type" of such a list would be List[Printable], which is not well-formed.
- We can push this further (for instance by adding existential types) but the concepts become rather heavy.
- Is there a simpler way?



## Type Classes vs OOP

- Can we translate type classes to an OOP setting?
- Observe the analogies:

| | | |
|---|---|---|
| Type class | ≈ | Type |
| Type/type class instance relation | ≈ | Type/type subtyping relation |
| Instance declaration | ≈ | Extends clause |

- Important difference:
  - Extends clauses are given with the subclass.
  - Instance declarations can appear anywhere.
- Hence, instance declarations are adaptable but extends clauses are not.

## Views vs Type Classes

- Idea: Introduce a way to add new fields and functions to an existing class. Example:

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

- This declaration makes Symbol a subtype of Printable, by giving implementations of all methods in the supertype.
- Extends clauses can be regarded as syntactic sugar for view declarations that come with a class.
- Like type classes, views can be declared anywhere, not just in the component that defines their subtype.



## Views vs Mutable Functions

- Views can also define fields. Example:

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

- This is equivalent to the mutable function

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

- Selection syntax is still in OO style. We use sym.adr instead of adr(sym).

## Views and Encapsulation

- Fields defined by a view may be encapsulated by functions.
- Example:

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

- Then sym.setAdr (x) is legal but sym.adr := x is not.



## Views are Stackable

- Let's say, we want addresses to be printed with symbols that have them. This can be achieved as follows.

```
view (sym: Symbol): Printable = {
  def toString = sym.name ++ ":" + sym.type ++ " at " ++ sym.adr
}
```

- Note that the implementation of the Printable view refers to sym.adr, which is defined in the Adr view.

## Views can be Conditional

- Parameterized types sometimes implement views only if their element types satisfy certain conditions.
- Example: Define a type Comparable as follows:

```
type Comparable [ T ] = {
   def equals (other: T)
   def less (other: T)
}
```

- Then objects of a type U can be compared iff U <: Comparable [U].
- Question: Are lists comparale?
- Answer: Only if their elements are. That is,

```
view [ T <: Comparable [T]] (xs: List [T]): Comparable [List [T]] = ...
```



## The Small Print

1. For types A and B, let V(A,B) be the set of subtype paths
   $A = A_0,...,A_n = B$ such that there exist view declarations from $A_i$ to $A_{i+1}$, for all i.
   We require: V(A,B) is either empty, or it has a minimum path relative to the subsequence ordering.

- This is a global restriction, which can be checked only at link time.
- The restriction is necessary for ensuring coherence. (It also disallows cyclic views.)

2. View fields can appear in a selection only in those regions of the program text where the view is in scope.

- Visibility of views is analogous to visibility of other declarations.

## The Small Print

1. For types A and B, let V(A,B) be the set of subtype paths $A = A_0,...,A_n = B$ such that there exist view declarations from $A_i$ to $A_{i+1}$, for all i.
   We require: V(A,B) is either empty, or it has a minimum path relative to the subsequence ordering.

- This is a global restriction, which can be checked only at link time.
- The restriction is necessary for ensuring coherence. (It also disallows cyclic views.)

2. View fields can appear in a selection only in those regions of the program text where the view is in scope.

- Visibility of views is analogous to visibility of other declarations.



## Related Work

- The lack of adaptability of the object approach has been realized by many others. It has sparked a number of proposals, among them:
  - Subject-oriented programming (Harrison & Osher)
  - Adaptive programming (Lieberherr)
  - Aspect-oriented programming (Kiczales et al.)
- The presented work can be regarded as an instance of aspect-oriented programming.
- But aspect-oriented programming is much more general - everything that does not fit into the notion of components as generalized procedures can be called an aspect.
- Often, general aspects are realized by program transformations.

## Conclusion

- The combination of objects and views leads to adaptable components with first-class plugs.
- We are currently implementing these ideas in Funnel.
- A paper in ESOP 2000 gives an overview of Funnel and its underlying foundation of functional nets.
- The goal of the current implementation work is to provide flexible concepts and tools for program composition in a Java environment.