Components Are Classes

Martin Odersky
École Polytechnique Fédérale de Lausanne (EPFL)

Component Software – State of the Art

As software grows more complex and mature, components become more important.

But programming languages lag behind.

Current programming languages are better at expressing small components than at expressing larger ones.

In the small:

$$\begin{array}{lcl} \textit{Component} & \hat{=} & \textit{Function} \\ \textit{Composition} & \hat{=} & \textit{Application} + \textit{fixed points (i.e. recursion).} \end{array}$$

Functions are first class.

In the large:

?

What's the Difference Between Large and Small?

- Large parts have more connections than small parts.

  $\Rightarrow$ *naming* becomes important.

- Large parts have internal structure

  $\Rightarrow$ *information hiding* becomes an issue.

- In a statically typed language, large parts may contain *types*.

What is a Component?

A *component* is a *reusable* program part, to be combined with other parts in larger applications.

To be reusable in new contexts, a component needs *interfaces* describing its *provided* as well as its *required* services.

Most current components are not very reusable.

Most current languages can specify only provided services, not required services.

Note: Component $\neq$ API !

No Hard < Links >!

A component should refer to other components not by hard links, but only through its required interfaces.

Another way of expressing this is:

> *All references of a component to others should be via its members or parameters.*

In particular, there should be no global static data or methods that are directly accessed by other components.

Components as Functors

One established language abstraction for components are SML functors.

Here,

Component	$\hat{=}$	*Functor* or *Structure*
Interface	$\hat{=}$	*Signature*
Required Component	$\hat{=}$	*Functor Parameter*
Composition	$\hat{=}$	*Functor Application*

Sub-components are identified via sharing constraints.

Shortcomings:

- No recursive references between components
- Structures are not first class.

Components as Classes

In Scala:

Component	$\hat{=}$	*Class*
Interface	$\hat{=}$	*Abstract Class*
Required Component	$\hat{=}$	*Abstract Member* or *"Self"*
Composition	$\hat{=}$	*Symmetric Mixin Composition*

Advantages:

- Components instantiate to objects, which are first-class values.
- Recursive references between components are supported.
- Sub-components are identified by name

  $\Rightarrow$ no explicit "wiring" is needed.

Language Constructs for Components

To express components as classes, we need:

- A way to *nest* classes inside other classes (already present in Java).
- A way to *compose* classes forming larger classes, e.g. by multiple inheritance or mixin composition.
- A way to *abstract* over required services of a class. There are two complementary ways of doing this:
  - Abstract over members (either types or values)
  - Abstract over the type of **this**.

A theoretical foundation for these constructs is the νObj calculus [ECOOP03].

These constructs subsume generative SML modules.

Example: Symbol Tables

Here's an example, which reflects a learning curve I had when writing extensible compiler components.

- Compilers need to model symbols and types.
- Each aspect depends on the other.
- Both aspects require substantial pieces of code.

The first attempt of writing a Scala compiler in Scala defined two global objects (*aka* modules), one for each aspect:

First Attempt: Global Data

```
object Symbols {
  class Symbol {
    def tpe: Types.Type;
    ...
  }
  // static data for symbols
}
```

```
object Types {
  class Type {
    def sym: Symbols.Symbol
    ...
  }
  // static data for types
}
```

Problems:

1. Symbols and Types contain hard references to each other.

   Hence, impossible to adapt one while keeping the other.

2. Symbols and Types contain static data.

   Hence the compiler is not *reentrant*, multiple copies of it cannot run in the same OS process.

   (This is a problem for the Scala Eclipse plugin, for instance).

Second Attempt: Nesting

Static data can be avoided by nesting the Symbols and Types objects in a common enclosing class:

```
class SymbolTable {
    object Symbols {
        class Symbol { def tpe: Types.Type; ... }
    }
    object Types {
        class Type {def sym: Symbols.Symbol; ... }
    }
}
```

This solves the re-entrancy problem.

But it does not solve the component reuse problem.

- Symbols and Types still contain hard references to each other.
- Worse, since they are nested in an enclosing object they can no longer be written and compiled separately.

Third Attempt: Type Abstraction

Question: How can one express the required services of a component?

Answer: By abstracting over them!

Two forms of abstraction: *parameterization* and *abstract members*.

Only abstract members can express recursive dependencies, so we will use them.

```
abstract class Symbols {
  type Type;
  class Symbol { def tpe: Type }
}
```

```
abstract class Types {
  type Symbol;
  class Type { def sym: Symbol }
}
```

Symbols and Types are now classes that each abstract over the identity of the "other type". How can they be combined?

Symmetric Mixin Composition

Here's how:

```
class SymbolTable extends Symbols with Types;
```

Instances of the SymbolTable class contain all members of Symbols as well as all members of Types.

Concrete definitions in either base class override abstract definitions in the other.

Fourth Attempt: Mixins + Self Types

The last solution modeled required types by abstract types.

This is sometimes verbose, when we have to give bounding interfaces for abstract types.

It is also limiting, because in Scala one cannot instantiate or inherit an abstract type.

Another approach makes use of *self-types*:

```
class Symbols
    : Symbols with Types {
    class Symbol { def tpe: Type }
}
```

```
class Types
    : Types with Symbols {
    class Type { def sym: Symbol }
}
```

```
class SymbolTable extends Symbols with Types;
```

Self-Types

- If a class comes with an explicit type annotation, as in:

  class C : T { ...

  then T is called a *self-type* of class C.
- If a self-type is given, it is taken as the type of **this** inside the class.

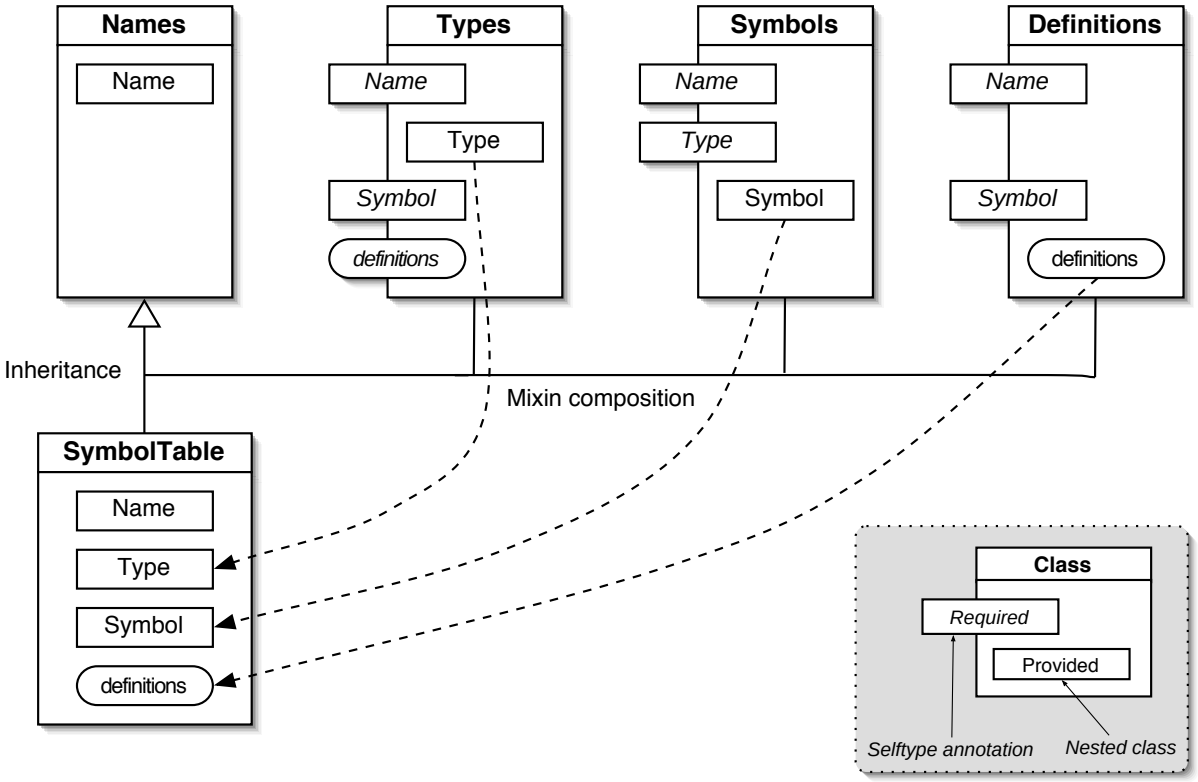
  (Without an explicit type annotation, the self-type is taken to be the type of the class itself.)
- Self-types need not have a relation with the class being defined.
- Only when a class is instantiated, it is checked that it conforms to its self-type.

Key insight:

The required interface of a class is its self-type.

Symbol Table Schema

Here's a schematic drawing of *scalac*'s symbol table:



We see that besides Symbols and Types there are several other classes that also depend recursively on each other.

Benefits

1. The presented scheme is very *general* – any combination of static modules can be lifted to a assembly of components.
2. Components have *documented interfaces* for required as well as provided services.
3. Components can be *multiply instantiated*

   $\Rightarrow$ *Reentrancy* is no problem.
4. Components can be flexibly *extended* and *adapted*.

Example: Logging

As an example of component adaptation, consider adding some logging facility to the compiler.

Say, we want a log of every symbol and type creation.

To print logging information, we use the following abstract class, which can be instantiated with arbitrary implementations.

```
abstract class Log {
  def println(s: String): unit
}
```

The problem is how insert calls to the println method into an existing compiler

- without changing source code,
- with clean separation of concerns,
- without using AOP.

Logging Classes

The idea is that the tester of the compiler would create subclasses of components which contain the logging code. E.g.

```
abstract class LogSymbols extends Symbols {
  val log: Log;
  override def newTermSymbol(name: Name): TermSymbol = {
    val x = super.newTermSymbol(name);
    log.println("creating term symbol " + name);
    x
  }
  ...
}
```

... and similarly for LogTypes.

How can these classes be integrated in the compiler?

Inserting Behavior by Mixin Composition

Here's an outline of the Scala compiler root class:

```
class ScalaCompiler extends SymbolTable with ... { ... }
```

To create a logging compiler, we extend this class as follows:

```
class TestCompiler extends ScalaCompiler with LogSymbols with LogTypes {
  val log = new ConsoleLog;
}
```

Now, every call to a factory method like newTermSymbol is re-interpreted as a call to the corresponding method in LogSymbols.

Note that the mixin-override is non-local – methods are overridden even if they are defined by indirectly inherited classes.

Sub-Systems

One possible objection to the presented scheme is that all classes making up a system exist as operands of single mixin composition, and are hence all on the same level.

Sometimes, we would like to keep a hierarchy of nested sub-systems, as in

```
class Outer: ... extends ... {
    object Inner extends ... { ... }
    ...
}
```

but with Inner compiled in a separate source file.

In traditional languages this is difficult once Inner refers to type members of Outer.

Nested, separately compiled systems can be expressed using abstract types:

```
class Outer {
    object inner extends Inner {
        type outer : Outer.this.type = Outer.this
    }
}
...
class Inner {
    type outer <: Outer;
}
```

Conclusion

Components can be modelled well with classes, **IF**:

- We can nest classes
- We can compose classes
- We can abstract over types and the identity of self.

The result is a simple and very powerful composition technique.

Still missing (and orthogonal) is

Encapsulation:

- How can we encapsulate classes (as opposed to objects) with interfaces?

Addendum I: The Expression Problem

With similar abstraction techniques, we can solve the *expression problem*:

> *How can a system be extended at the same time with new data variants and with new operations over data?*

Requirements:

1. Separate compilation,
2. strong static type safety,
3. no code modification.

See:

[1] Matthias Zenger and Martin Odersky. Independently Extensible Solutions to the Expression Problem. EPFL Technical Report IC/2004/33

Addendum II: Generalized Algebraic Data Types

Here's how GADT's would be expressed in Scala.

```
abstract class Term[T];
case class Lit(x: int);
case class Succ(t: Term[int]) extends Term[int];
case class IsZero(t: Term[int]) extends Term[boolean];
case class If[T](c: Term[boolan], t1: Term[T], t2: Term[T]) extends Term[T];

def eval[a](t: Term[a]): a = t match {
    case Lit(n)         ⇒ n
    case Succ(u)        ⇒ eval(u) + 1
    case IsZero(u)      ⇒ eval(u) == 0
    case If(c, u1, u2) ⇒ if (eval(c)) eval(u1) else eval(u2)
}
```

Caveat: In current Scala, this would not type check.

Reason: Type variable bindings in a pattern are "forgotten" on the

right of the "⇒".

Hence, eval's result type would be Any, not a.

But maybe we should change that?

Example: A function exists which tests whether a given array has an element which satisfies a given predicate:

```
def exists[T](xs: Array[T], p: T ⇒ boolean) = {
    var i: int = 0;
    while (i < xs.length && !p(xs(i))) i = i + 1;
    i < xs.length
}
```

... and a function forall, which uses exists:

```
def forall[T](xs: Array[T], p: T ⇒ boolean) = {
    def not_p(x: T) = !p(x);
    !exists(xs, not_p)
}
```

Note that forall uses a named nested function not_p.

Example: A function exists which tests whether a given array has an element which satisfies a given predicate:

```
def exists[T](xs: Array[T], p: T ⇒ boolean) = {
  var i: int = 0;
  while (i < xs.length && !p(xs(i))) i = i + 1;
  i < xs.length
}
```

... and a function forall, which uses exists:

```
def forall[T](xs: Array[T], p: T ⇒ boolean) = !exists(xs, x: T ⇒ !p(x));
```

The alternative formulation of forall uses an anonymous function.

Example: A function exists which tests whether a given array has an element which satisfies a given predicate:

```
def exists[T](xs: Array[T], p: T ⇒ boolean) = {
  var i: int = 0;
  while (i < xs.length && !p(xs(i))) i = i + 1;
  i < xs.length
}
```

... and a function forall, which uses exists:

```
def forall[T](xs: Array[T], p: T ⇒ boolean) = !exists(xs, x: T ⇒ !p(x));
```

Finally, here is a function to test whether a 2-dimensional matrix has a row consisting of zeroes:

matrix exists (row =¿ row forall (0 ==))

```
def hasZeroRow(matrix: Array[Array[int]]) =
  exists(matrix, row: Array[int] ⇒ forall(row, 0 ==));
```

How To Do Better?

Hypothesis 1: Languages for components need to be *scalable*; the same concepts should describe small as well as large parts.

Hypothesis 2: Scalability can be provided by unifying and generalizing functional and object-oriented programming concepts.

To validate these hypotheses we have designed and implemented a concrete programming language, **Scala**.

Part I: A Quick Introduction to Scala

Some key aspects of Scala are:

(1) interoperability with Java and .NET,
(2) a uniform object model,
(3) higher-order functions,
(4) uniform abstraction concepts for both types and values,
(5) symmetric mixins for composing classes.
(6) object decomposition with pattern matching,
(7) XML support.

(1) – (5) are quickly explained in the following.

1. A Java Like Language

Here is a sample program in Java:

```
class PrintOptions {
    public static void main(String[] args) {
        System.out.println("Options selected:");
        for (int i = 0; i < args.length; i++)
            if (args[i].startsWith("-"))
                System.out.println("  "+args[i].substring(1));
    }}
```

And here is the same program in Scala:

```
object PrintOptions {
    def main(args: Array[String]): unit = {
        System.out.println("Options selected:");
        for (val arg ← args)
            if (arg.startsWith("-"))
                System.out.println("  "+arg.substring(1));
    }}
```

Interoperability

Scala is completely interoperable with Java (and more recently also to C#).

A Scala component can:

- access all methods and fields of a Java component,
- create instances of Java classes,
- inherit from Java classes and implement Java interfaces,
- be itself instantiated and called from a Java component.

None of this requires glue code or special tools.

This makes it very easy to mix Scala and Java components in one application.

2. A Unified Object Model

In Scala, every value is an object and every operation is a method invocation.

Example: A class for natural numbers

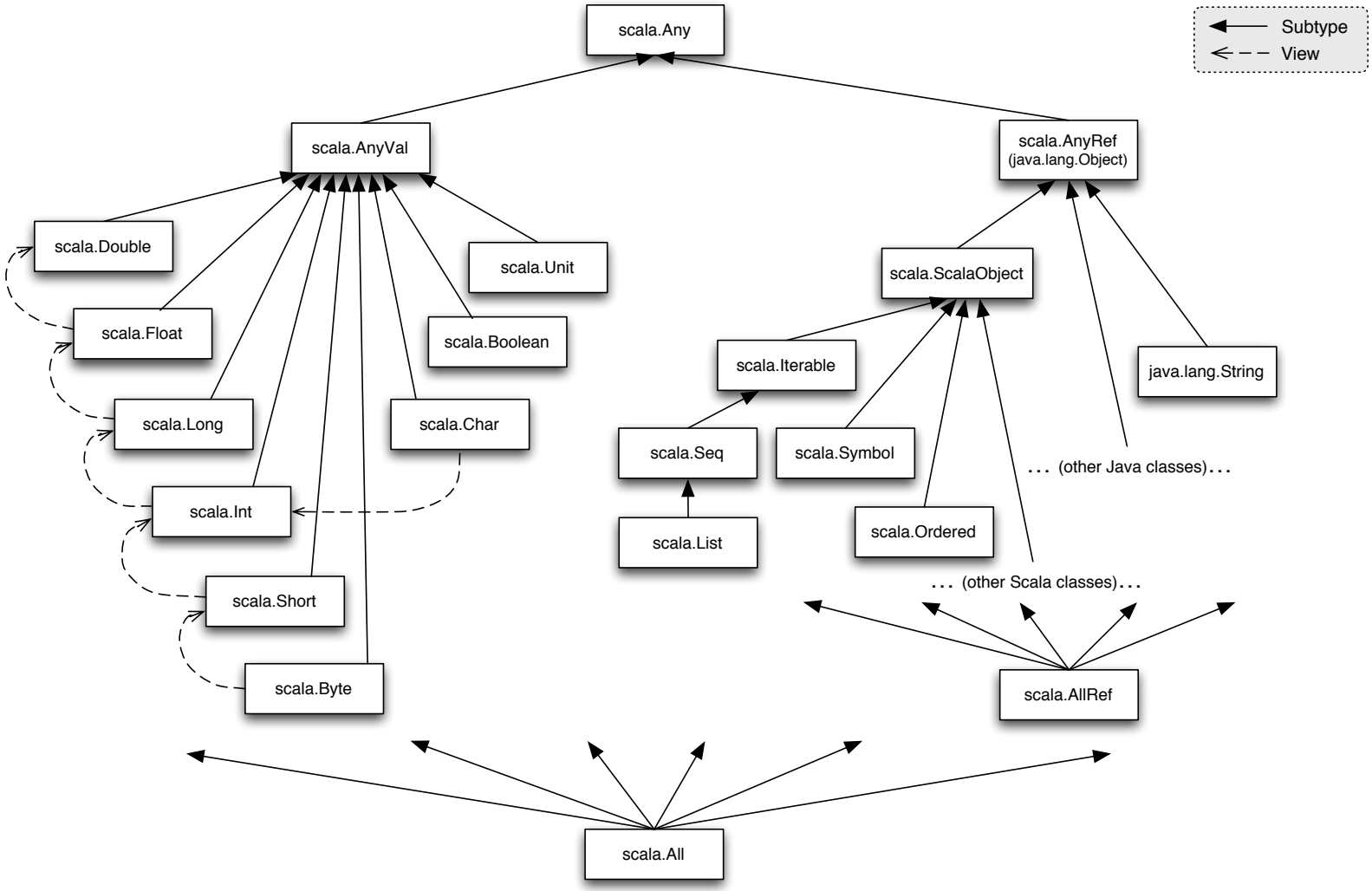
```
abstract class Nat {
  def isZero : boolean;
  def pred : Nat;
  def succ : Nat = new Succ(this);
  def + (x : Nat): Nat = if (x.isZero) this else succ + x.pred;
  def − (x : Nat): Nat = if (x.isZero) this else pred − x.pred;
}
```

Here are the two canonical implementations of Nat:

```
class Succ(n : Nat) extends Nat {
  def isZero : boolean = false;
  def pred : Nat = n
}
```

```
object Zero extends Nat {
  def isZero : boolean = true;
  def pred : Nat =
    throw new Error("Zero.pred");
}
```

Scala's Class Hierarchy



3. Operations Are Objects

- Scala is a functional language, in the sense that every function is a value.
- Functions can be anonymous, curried, or nested inside each other.
- Familiar higher-order functions are implemented as methods of Scala classes. E.g.:

```
matrix exists (row ⇒ row forall (0 ==)))
```

- Here, matrix could be of type of List[List[int]], using Scala's List class:

```
class List[+T] {
  def isEmpty : boolean;
  def head : T
  def tail : List[T];
  def exists(p : T ⇒ boolean): boolean =
    !isEmpty && (p(head) || (tail exists p));
  ... }
```

Functions are Objects

- If functions are values, and values are objects, it follows that functions themselves are objects.
- In fact, the function type $S \Rightarrow T$ is equivalent to

  scala.Function1[S, T]

  where Function1 is defined as follows in the standard Scala library:

  abstract class Function1[−S, +T] { **def** apply(x : S): T }

  (Analogous conventions exist for functions with more than one argument.)
- Hence, functions are interpreted as objects with apply methods.
- For example, the anonymous "incrementer" function x : int ⇒ x + 1 is expanded as follows.

  new Function1[int, int] { **def** apply(x : int): int = x + 1 }

Abstract Types

Here is a type of "cells" using object-oriented abstraction.

```
abstract class AbsCell {
  type T;
  val init: T;
  private var value: T = init;
  def get: T = value;
  def set(x: T): unit = { value = x }
}
```

The AbsCell class has an abstract type member T and an abstract value member init.

Instances of that class can be created by implementing these abstract members with concrete definitions.

```
val cell = new AbsCell { type T = int; val init = 1 }
cell.set(cell.get * 2)
```

The type of cell is AbsCell { **type** T = int }.

Path-dependent Types

It is also possible to access AbsCell without knowing the binding of its type member.

For instance: **def** reset(c : AbsCell): unit = c.set(c.init);

Why does this work?

- c.init has type c.T
- The method c.set has type c.T ⇒ unit.
- So the formal parameter type and the argument type coincide.

c.T is an instance of a *path-dependent* type.

In general, such a type has the form $x_0. \ldots .x_n.t$, where $n \geq 0$,

- x_0 is an immutable value
- $x_1, \ldots, x_n$ are immutable fields, and
- t is a type member of x_n.

Safety Requirement

Path-dependent types rely on the immutability of the prefix path.

Here is an example where immutability is violated.

```
var flip = false;
def f(): AbsCell = {
    flip = !flip;
    if (flip) new AbsCell { type T = int; val init = 1 }
    else new AbsCell { type T = String; val init = "" }
}
f().set(f().get)  // illegal!
```

Scala's type system does not admit the last statement, because the computed type of f().get would be f().T.

This type is not well-formed, since the method call f() is not a path.

Family Polymorphism

Scala's abstract type concept is particularly well suited for *family polymorphism*, where several types vary together covariantly.

Example: The subject/observer pattern (also known as *publish/subscribe*):

```
abstract class SubjectObserver {
  type S <: Subject;
  type O <: Observer;
  abstract class Subject : S {
    private var observers : List[O] = List();
    def subscribe(obs : O) = observers = obs :: observers;
    def publish = for (val obs ← observers) obs.notify(this);
  }
  abstract class Observer { def notify(sub : S): unit; }
}
```

The top-level class SubjectObserver has two member classes:

- The Subject class defines methods subscribe and publish.
- The Observer class only declares an abstract method notify.

Note that the Subject and Observer classes do not directly refer to each other.

Instead, they refer to two abstract types S and O which are bounded by Subject and Observer.

Family Polymorphism ctd

The mechanism defined in the publish/subscribe pattern can be used by inheriting from SubjectObserver

Example:

```
object SensorReader extends SubjectObserver {
    type S = Sensor;
    type O = Display;
    abstract class Sensor extends Subject {
        val label : String;
        var value : double = 0.0;
        def changeValue(v : double) = { value = v; publish; }
    }
    abstract class Display extends Observer {
        def println(s : String) = ...
        def notify(sub : Sensor) = println(sub.label + " has value " + sub.value);
    }
}
```

In object SensorReader, type S is bound to Sensor, and type O is bound to Display.

The two formerly abstract types are now defined by overriding definitions.

This "tying the knot" is always necessary when creating a concrete class instance.

On the other hand, it would also have been possible to define an abstract SensorReader class which could be refined further by client code.

In this case, the two abstract types would have been overridden again by abstract type definitions.

```
abstract class AbsSensorReader extends SubjectObserver {
    type S <: Sensor;
    type O <: Display;
    ...
}
```

SelfTypes + Mixins *vs.* AOP

Similar strategies work for many adaptations for which aspect-oriented programming is usually proposed. E.g.

- security checks
- synchronization
- choices of data representation (e.g. sparse vs dense arrays)

Generally, one can handle all before/after advice on method join-points in this way.

Relationship between Scala and Other Languages

Main influences on the Scala design:

1. Java, C# for their syntax, basic types, and class libraries,
2. Smalltalk for its uniform object model,
3. Beta for systematic nesting,
4. ML, Haskell for many of the functional aspects.

(Too many influences in details to list them all)

Component Composition in Other Languages

1. Java, C#: Mostly static data. No true reusable components.
2. Smalltalk: Object-level instead of class-level composition. Less problems (and less security) for lack of static types.
3. Beta: Nesting + an extra-language layer of "fragments".
4. ML: Parameterization only; No inheritance or recursive dependencies are possible.

Related Language Research

Mixin composition : Bracha (linear), Duggan, Hirschkowitz (mixin-modules), Schaerli et al. (traits), Flatt et al. (units, Jiazzi), Zenger (Keris).

Abstract type members : Ernst (gbeta), Jolly et al. (Concord).

Explicit self types : Vuillon and Rémy (OCaml)

Conclusion

- Scala enables a new method for software construction "in the large".
- Static data and references are replaced by member and selftype abstraction and symmetric mixin composition.
- Programs are classes which can be instantiated multiple times.
- This enables better separation of concerns and more flexible component adaptation.

Try it out: scala.epfl.ch

Thanks to the (past and present) members of the Scala team:

Philippe Altherr, Vincent Cremet, Julian Dragos, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger.

Component Abstraction

There are two principal forms of abstraction in programming languages:

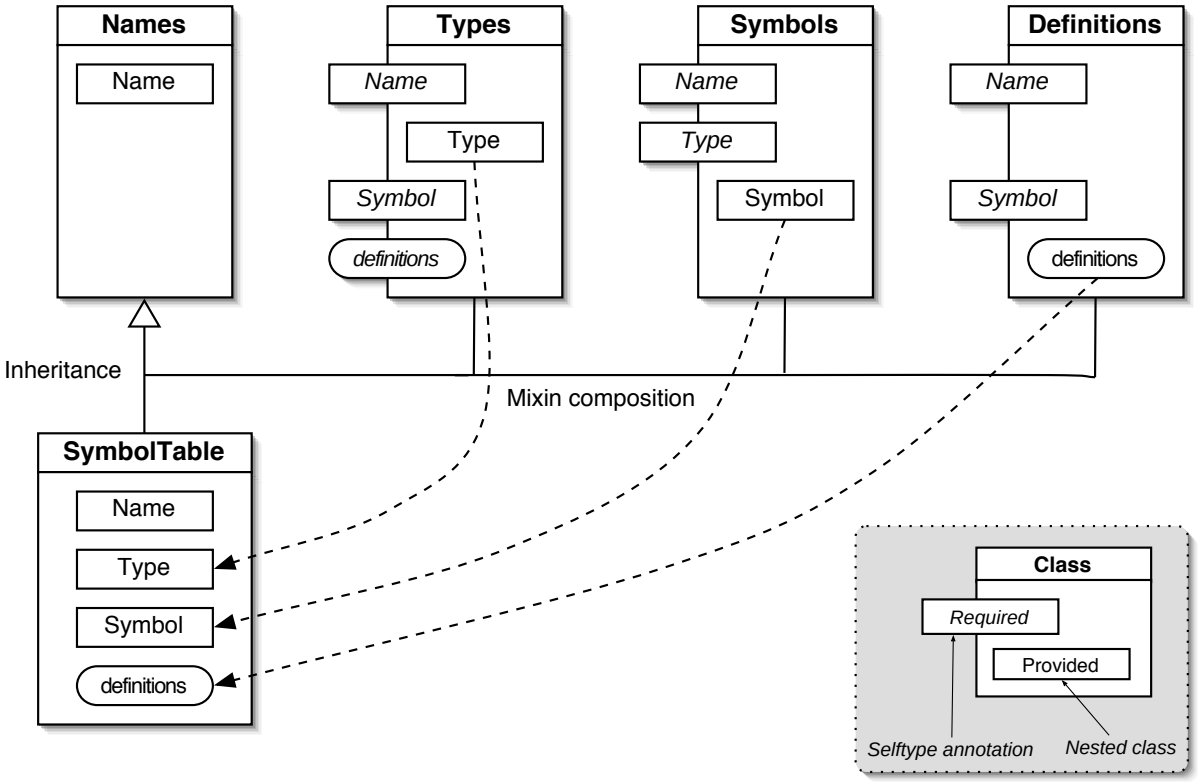
parameterization (functional)
abstract members (object-oriented)

Scala supports both styles of abstraction for types as well as values.

Both types and values can be parameters, and both can be abstract members.

Symbol Table Schema

Here's a schematic drawing of *scalac*'s symbol table:



We see that besides Symbols and Types there are several other classes that also depend recursively on each other.

Benefits

1. The presented scheme is very *general* – any combination of static modules can be lifted to a assembly of components.
2. Components have *documented interfaces* for required as well as provided services.
3. Components can be *multiply instantiated*

   $\Rightarrow$ *Reentrancy* is no problem.
4. Components can be flexibly *extended* and *adapted*.

In *principle*, software should be constructed from re-usable parts ("components").

In *practice*, software is still most often written "from scratch", more like a craft than an industry.

Programming languages share part of the blame for this.

Most existing languages offer only limited support for components.

This holds in particular for statically typed languages such as Java, C#, and Haskell.