Component Composition with Scala

Bogdan Dumitriu
(SN: 0402044)

June 1, 2006

Abstract

Software components that are developed by distinct teams with no coordination have been traditionally hard to integrate. The difficulty lies with the fact that the services a component requires in order to function are usually hard coded in its source code. This entails that in order to replace one or more of them with different ones, one needs to change the source code of the original component, which fails to qualify as "clean" reuse. The Scala programming language addresses this issue by providing three language concepts, *abstract member types*, *modular mixin composition* and *selftype annotations* that, combined in a smart way, allow the programmer to lift the specification of the required services out of the component code. In this paper we present the three mechanisms (together with some additional presentation of the Scala language) and show how they can be used for component composition by working through two examples.

1 Introduction

Seamless integration of components is something that the software industry has yet to achieve. And this despite the fact that putting software components together in the way components are put together throughout virtually all other branches of engineering has been a well-established goal for decades now. Failure to fully achieve this goal is perhaps even more surprising if we realize that large software companies themselves have had a constant, vested interest in bringing this about, which suggests that massive funding must have gone into its research. Reality, however, confirms that we are still some distance away from turning software writing (and particularly software composition) from craftsmanship into an industrial process.

Perhaps one of the reasons why this task is so difficult is that there does not seem to be an agreement regarding what software components really are. Each author tends to have his or her own definition and understanding of this concept, either striving for strictness – reducing software components to very specific entities only – or, on the contrary, for generality – trying to cover as many types of software as possible. Since this article is based on [7], we choose to adopt the very loose definition of the concept given therein, namely that components are "program parts which are used in some way by larger parts or whole applications." Such a definition allows for unlimited variability both in terms of a component's form and size. The goal, then, is to ensure that such pieces of software can be combined with others in a fashion which requires little or no adaptation work, regardless of whether or not the various parts were designed to work together in the first place.

Theoretical approaches on component composition are generally well articulated and inspiring, but essentially remain at a level of statements of intent, providing no immediate solutions. This does not make them less interesting to explore, however. For instance, in [9] Szyperski classifies the motivation for using software components in four tiers, with the most refined – and least achieved in practice – of which being dynamic administration of deployed systems. This entails having such confidence in the reusability of components (correctness, robustness, and efficiency-wise) that we can simply replace them with different ones on site. He also claims that in order to produce more mature components we first need to master *compositional reasoning* which would presumably allow us to predict (in a mathematical sense) what the out-come of composing two or more components will be, regardless of context. In a discussion about how to reuse components in a generative programming setting [2], Czarnecki proposes to replace the single system-centric development with families of system-centric development in order to produce truly reusable components. This implies a process consisting of domain analysis, design and implementation which would yield components that, through iterative refinement, would eventually become reusable throughout the family of systems. It is unlikely, however, that such components could also be composed automatically across such families.

If we turn to more practical approaches, we notice that even though we have come a long way in terms of modularity and ease of reuse over the years, especially with the introduction of the object-oriented paradigm, we still have some ground left to cover. If we look at the two most important representatives of the current state of the art, the Java and the .NET platforms, we notice that all modules routinely describe the services that they *offer* in an abstract manner, through interfaces. This makes it easy for us to reuse them without much knowledge of how those services are implemented. Why, then, can we not simply promote such modules to the rank of full-fledged reusable components? It is because they lack a proper description of the services they *require* in order to function. The common practice is to hard code references to such services into the source code of the modules, thus impeding the replaceability of the former. In essence, if module A requires module B in order to function, and we want to replace module B with another module, C, which provides similar functionality, we need to extensively modify the source code of A. A multitude of design patterns have been proposed and are in use (with [3] being the reference in the field) for alleviating these hard coded dependencies, but they can only partially overcome the limitations of today's systems. What we need instead is to come up with new systems which allow a component to specify both its required and provided interfaces, such that matching components can be "plugged in" these interfaces in a transparent manner.

Knit [8], "a module language and toolset for managing systems components," for instance, is an attempt to such a system. It builds on C as a base language and it provides additional abstractions called units (much like modules in functional languages) which define *imports* and *exports* or, in the terminology used thus far, required and provided services. These imports and exports are actually C functions which are used and, respectively, defined in the body of the unit. Units can be composed freely into systems by only ensuring that in the plugging together process, the exports of a unit match (part of) the imports of another and that overall each unit's imports are matched. In addition, Knit allows the specification of architectural invariants which are checked automatically at the composition phase.

The solution we will focus on in this paper, however, is *Scala*, a complex and powerful language that, among many other purposes, can also be used for elegantly composing components. Most notably, the mechanism promoted by Scala allows changing the services a component uses as well as use those that the component provides without making any modifications to the source code of the original component. Moreover, the composition process takes classes as input and produces classes as output, so the process is indefinitely scalable. Finally, an advanced type system ensures that all compositions are statically type safe, thus raising the degree of confidence in the result.

The first part of this paper, section 2, deals with presenting the features that enable Scala to allow the abstraction of the services a component requires: abstract type members, modular mixin composition and selftype annotations. In the second part we explain how these features can be put to use by means of two examples. Section 3 shows how a rudimentary broker system can be created by component composition, while section 4 describes a solution to the so-called expression problem, which is also an instance of a composition problem. We end in section 5 with some concluding remarks.

2 Scala support for composition

In this section we give an overview of the Scala language that is focused especially on those parts of Scala that facilitate software composition. Since Scala is a very featureful system, this is by no means a comprehensive presentation. However, we will try to provide the reader with enough background that the contents of this article can be understood without further reading. For a more detailed description of the Scala language, we refer you to [4] as well as to the various documentation that can be found on Scala's home page [1].

2.1 What is Scala?

Scala is a programming language designed by Martin Odersky and developed by a group of people at Ecole Polytechnique Fédérale de Lausanne. According to the definition on its web site [1], "Scala is a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented and functional languages." What this definition first and foremost entails is that Scala is a combination of an object-oriented language and a functional language. Unlike OCaml, that adds object-orientation to a base functional language, Scala takes the opposite approach by adding functional features to a strong object-oriented language core that supports (nested) classes and objects, inheritance and mixin-based composition. Functional aspects include the fact that all functions (including methods) are values, support for higher order functions, currying and sequence comprehensions (similar to for comprehensions in Haskell, but applicable to any iteratable sequence). In addition, Scala allows lightweight representation of and pattern matching on algebraic data types in a fashion similar to functional languages using the concept of case classes.

Scala is a statically typed language, with its type system being founded in νObj calculus (described in [5]). Aside from all features that are present in modern object-oriented programming languages, Scala also includes support for variance annotations, abstract types as class members, inner classes that are bound to the outer object (building on the notion of *singleton type*: each object implicitly defines its own singleton type), modular mixin composition and selftype annotations (both of which will be discussed in more detail later).

Another characteristic of Scala is that it interoperates cleanly with Java and C#, by featuring a similar compilation model. All it takes in order to use Java libraries in Scala, for instance, is an import statement identical to the one used in Java itself:

```
import java.util.ArrayList;

object Test {
  def main(args: Array[String]): unit = {
    val list = new ArrayList;
    for (val elem <- args)
      list.add(elem);
  }
}
```

In the code above we simply import the Java `ArrayList` class and then use it without further complication in our Scala program. This code can be compiled directly by running `scala Test.scala` (assuming it is saved in a file called `Test.scala`). A quick guide to this code snippet: type annotations are specified in Scala using brackets (e.g., `Array[String]` defines an array which contains strings); the notion of static members does not exist in Scala, we use class definitions for non-static members and object definitions (as in the code above) for static members; much syntax is optional, including empty lists of parameters (as in the case of `new ArrayList;` above), semicolons and some block curly braces; `for (val elem <- args) expr` is a simplified for comprehension that only contains a generator, `val elem <- args`, and does not produce a sequence as its result. For comprehensions can contain multiple generators and multiple filters. They can also produce results by using the keyword `yield` after the list of generators and filters.

From the multitude of features that Scala has to offer, its authors have identified three as being essential to software composition. Theoretically speaking, any language providing these features would enable smooth composition of components. The features, as listed in [7] are *abstract type members*, *selftype annotations* and *modular mixin composition*. We explain below what each of them are and how they are realized in Scala.

2.2 Abstract type members

This is perhaps the most common of the three, and it refers at the possibility of having abstract type variables defined in classes just as we can have abstract methods in "traditional" object-oriented programming languages. In fact, a lot of languages already support this, albeit in a different form, as generics (Java) or templates (C++). Both generics and templates allow such specifications of abstract type members immediately following a class declaration (e.g., `class MyClass<T>` defines the abstract type member `T` as belonging to class `MyClass` both in Java and C++). In the case of Scala, the mechanism is a bit more versatile, since Scala allows the definition of such types alongside class variables and class methods, as show in the following code snippet:

```
abstract class Buffer {
  type T;
  val size: int;
  protected var buff = new Array[T](size);

  def prepend(elem: T): unit = ...
  def append(elem: T): unit = ...
  def remove(pos: int): T = ...
  def clear: unit = ...
}
```

Here, the declaration `type T` introduces `T` as an abstract type member, largely in the same way that a declaration `class Buffer<T>` would have done the same in Java. In fact, the latter is available in Scala just as well, with the sole syntactic distinction that we have to use right brackets instead of angle brackets to specify the type parameter. Scala supports generics as syntactic sugar over its base mechanism consisting of abstract type member definitions. We actually used Scala generics in the code above when creating the array that will function as support for our buffer by specifying that the elements that will be stored in the array will have the (abstract) type `T`.

Abstract type members have to be instantiated in order to be able to create instances of the class that declares them. This can be done either in a subclass of this class or directly in the code using the abstract class, as shown below:

```
val mybuffer = new Buffer
  { type T = char; val size = 10 };
mybuffer.append('a');
```

Why are abstract type members important for component composition? It is because they are part of the mechanism for defining a class' required interfaces. Abstract type members can be considered as "hooks" left in a component so that other components can be plugged therein. Except for highly general-purpose components (such as those implementing data structures, for instance), it is usually the case that one wants to limit the scope of values that a component's abstract type can be instantiated to. In composition terms, we use the abstract type to specify the kind of service that the component requires, with the limitation being the one that defines the service. This limitation can be enforced in Scala by using *upper parameter bounds*, which restrict the values that can be assigned to an abstract type to subclasses of a certain class (or trait, as we will discuss later).

Imagine we want to define a new type of buffer which also contains a method for pretty printing its elements. However, this can only be achieved if its elements themselves can be pretty printed. We could then define a *trait* (for now, you can regard it simply as an abstract class) called `PrettyPrintable`:

```
trait PrettyPrintable {
  def pp: String;
}
```

and specify in our new buffer class that `T` has to be a subclass of `PrettyPrintable`. This enables us to use the `pp` method with the buffer's elements, since we know that whatever class will be plugged in as `T`, it will provide the pretty printing service. Our buffer class then can be defined as follows:

```
abstract class PPBuffer
  extends Buffer {
  type T <: PrettyPrintable;

  def pp: String = {
    var result = "";
    for (val elem <- buff) {
      result = result + elem.pp + "\n";
    }
    result;
  }
}
```

The <: operator imposes `PrettyPrintable` as the upper type bound of type `T`. The keen reader might have also noticed that by implementing the pp method, the new buffer is itself a subclass of PrettyPrintable as well, but as Scala only allows multiple inheritance in the form of mixin composition (which we have not discussed yet), we refrained from declaring it as such. Also as a side note, the last expression in an sequence of expressions is the one returned by the sequence itself. We do not thus have to prefix the last expression (`result`) in the code above with `return`, since this is implicit in Scala.

2.2.1 Views

Views represent an interesting Scala feature, especially from the perspective of component adaptation. It is often the case that we need to integrate a component into an existing system, but the component's interface and/or behavior is not exactly the one that the system expects. A layer of adaptation generally has to be introduced then to interface the two, which can be cumbersome in many settings because it usually implies changing a lot of existing code in order to introduce calls to adapter methods. Such difficulties have been a constant nag in the industry and keep hindering the composition process.

Scala offers an interesting solution to this problem in the form of views. With this mechanism we can essentially *view* a component as a different one, the properties of which we can ourselves define according to our needs. If our existing system needs to use a set and the component that we want to integrate is merely a list, we can define a view that will allow us to view the list as a set. Views in Scala perform implicit conversions from one type to another. They are defined as regular methods prefixed with the `implicit` modifier. The compiler then will consider all the views in scope as potential implicit conversions. Whenever in an expression the expected type is `T` and the actual argument has type `U`, if there is a view converting `U` into `T`, it will be automatically inserted around that argument. The most specific view is always selected if more are available and it is a compilation error if more than one view is most specific.

To illustrate, assume that our system expects instances of sets as defined by the following the trait:

```
trait Set[T] {
  def add(x: T): Set[T];
  def contains(x: T): boolean;
}
```

and that our list is defined as follows:

```
class List[T] {
  def prepend(x: T): List[T] = { ... }
  def head: T = { ... }
  def tail: List[T] = { ... }
  def isEmpty: boolean = { ... }
}
```

Since our system expects instances of `Set[T]`, we would not expect to be able to pass instances of `List[T]` to it since this would not be type correct. However, with the aid of views this is actually possible in Scala. An interface (but not behavior) conversion from lists to sets could be encoded like this:

```
implicit def list2set[T](xs: List[T]):
  Set[T] = new Set[T] {
    def add(x: T): Set[T] =
      xs prepend x
    def contains(x: T): boolean =
      !xs.isEmpty && ((xs.head == x) ||
                      (xs.tail contains x))
  }
```

A syntax-related remark: methods with a single argument can be used as infix operators in Scala, i.e., `x op y` is equivalent to `x.op(y)`. Returning to views, it is interesting to see that the view that we are defining is actually already used in its own definition. The result of the expression that defines the body of the add method, `xs prepend x`, has the type `List[T]`, but we can see that the method signature specifies that a `Set[T]` has to be returned. The compiler will detect this and implicitly add `list2set` around the expression `xs prepend x` to accommodate this. The same happens in the expression `xs.tail contains x`, where `list2set` is wrapped around `xs.tail` in order to allow the use of the `contains` method.

Scala also features so-called *view bounds*, which allow the programmer to restrict the abstract type used to to parameterize a method[1] to only those types for which a view exists that transforms them

[1] View bounds are not available for abstract type members.

into a specified type. `def m[T <% U](x: T)` indicates that the method `m` will only operate on those types `T` for which a view is defined that transforms them into `U`, where `U` has to be a concrete type.

2.3 Modular mixin composition

Specifying required and provided services is, as argumented, vital to promoting component reusability in random contexts. However, what we primarily need in order to be able to compose components in the first place is the composition mechanism itself. There are many approaches to how composition is done in programming environments. The most common techniques for composition are through class members (an instance of a class cooperates with an instance of another class by holding a pointer to it in one of its data members) and through multiple inheritance (a class that extends two or more classes effectively composes them into a single new class). We will discuss the latter technique in this section.

C++ allows multiple inheritance, but unfortunately the mechanism is limited by the so-called diamond problem: if classes B and C both inherit from class A, then a class D will not be able to inherit from both B and C because this would introduce the state of class A twice. To alleviate this, Java has disallowed multiple inheritance and offers a different mechanism based on interfaces in order to allow a class to conform to multiple specifications. Since interfaces can only contain method declarations there is no state duplication (given that there is no state). Unfortunately, Java interfaces prevent code reuse, because they cannot also include the definitions of the methods they declare. Scala tries to solve both problems by combining single inheritance with mixin class composition to effectively obtain the code reuse benefit of multiple inheritance while smartly avoiding the diamond problem.

We begin our discussion about mixin composition in Scala with explaining *traits*. Traits are most succinctly described as stateless classes. According to the Scala language specification, traits cannot have either constructor parameters or variable definitions, since either of these would qualify as state. In this sense traits are much like interfaces in Java. Where they differ, however, is in that they can provide method definitions along method declarations. Traits can be used in all places where abstract classes can be used and they also constitute the only type of classes that can be *mixed in* other classes. Why multiple inheritance with traits (through the mixin mechanism) avoids the diamond problem will become clear later once we discuss class linearization and the actual supertype of traits. The second problem, code reuse, is solved through the fact that traits allow method definitions.

To illustrate traits, as well as single inheritance in Scala, consider the following example:

```
trait Logger {
  def log(message: String): unit;
}

trait Debugger extends Logger {
  def debug(message: String,
            active: boolean) = {
    if (active) { log(message); }
  }
}

class FileLogger(path: String)
  extends Logger {
  import java.io._;
  val f = new FileWriter(new File(path));

  def log(message: String) = {
    f.write(message + "\n");
  }
  def close = { f.close; }
}
```

First we define the `Logger` trait, which specifies the method that needs to be implemented by a component in order to provide logging services. The `Debugger` trait extends `Logger` with a `debug` method, which introduces an extra parameter that allows turning logging on or off. Notice in the `Debugger` trait that the `debug` method is also defined, not just declared. Finally, class `FileLogger` provides a concrete implementation of the `Logger` trait by using a file for logging. There is nothing uncommon so far, except perhaps some more Scala syntax. You will notice that some methods lack the result type. This is because it can be inferred automatically by the compiler. Another peculiarity is that the `import` clause can be used anywhere and that it uses "_" as a wildcard. Finally, the default constructor of Scala classes is specified directly after the class declaration, as you can see in the case of `FileLogger`. Moreover, all the parameters in the list automatically become immutable members of the class. All the code in the class that is defined outside of a method is considered to be part of the constructor (such as the initialization of `f` in our case).

Mixin class composition intervenes if we want to define a `FileDebugger` class that supports logging as well as debugging. In order to do this, it would be best to inherit the basic logging functionality from `FileLogger` and add the definition of the `debug` method from the `Debugger` trait to it. In other words, we would like to mix in the `Debugger` trait with the FileLogger class. The following code shows how we can do this:

```
object Test {
  def main(args: Array[String]): unit = {
    class FileDebugger
      extends FileLogger(args(0))
      with Debugger;
    var dbg = new FileDebugger;
    dbg.debug("some message", true);
    dbg.close;
  }
}
```

Scala allows local class definitions, as you can see above. Specifying `args(0)` as a parameter to `FileLogger` achieves the equivalent of a super call in Java, i.e., `super(args[0])`. Going back to the mixin mechanism, what it permits is the reuse of "the delta of a class definition, i.e., all new definitions that are not inherited" [7]. In our case, the `Debugger` mixin contains the `log` and the `debug` methods, but only the latter is inherited in `FileDebugger`, since the former is inherited by `Debugger` from `Logger`. In a mixin composition, the class following the `extends` keyword is called the *superclass*, while the classes introduced by the `with` keyword are called *mixins*.

For a mixin composition to be correct, a condition has to be met: the superclass in the composition must be a subclass of all the superclasses of the mixins. This condition is necessary because, as we explained, only the methods defined in a trait itself are inherited if the trait is used as a mixin, but the code of these methods might refer to members of the trait's superclass and without this condition we would have no guarantees that those members would be present in the class that is the result of the mixin composition.

The inheritance relationships introduced by the mixin mechanism can be represented as a directed, acyclic graph. Since a class inherits members from all its base classes (direct or indirect), it is important to be able to precisely specify which member is inherited, if multiple matching members are defined in the base classes. For this to be possible, we need to first define a linearization of the directed acyclic graph. According to the Scala language specification (see [1]), the linearization of a class C for the declaration C `extends` C_1 `with ... with` C_n is:

$$\mathcal{L}(C) = \{C\}, \mathcal{L}(C_n) \vec{+} \cdots \vec{+} \mathcal{L}(C_1)$$

where $\vec{+}$ indicates concatenation where elements of the right operand replace identical elements of the left operand:

$$\{a, A\} \vec{+} B = \begin{cases} a, (A \vec{+} B) & \text{if } a \notin B, \\ A \vec{+} B & \text{if } a \in B. \end{cases}$$

Based on this linearization, we can now define how concrete and abstract members are inherited in class C. The concrete member that appears in the leftmost class (of those defining that member) of $\mathcal{L}(C)$ is the one propagated to C, *regardless* of abstract definitions of matching members that might appear in a class left to the one defining the concrete member. This is a very important property, since it makes mixin composition symmetrical in a sense. More precisely, if trait C_i contains an abstract definition of m and trait C_j contains an concrete definition of the same m, the concrete one will have precedence regardless of the order in which C_i and C_j are specified in the composition. This is essential since it can be the case that another member m' is defined in a reverse manner (concrete definition in C_i and abstract one in C_j) and then, without this symmetry, there would be no way to mix in C_i and C_j in which for both m and m' the concrete definition to be the one that is inherited. If only abstract definitions are available for a member, then the one defined in the class (of those defining that member) that is leftmost in $\mathcal{L}(C)$.

2.4 Selftype annotations

There are certain systems consisting of two or more types that (1) mutually reference one another and (2) tend to vary together (as one component is specialized, the others follow). In languages like Java or C++ the problem with such systems is that the types that extend the original types will end up not referring each other (as would be preferable), but referring each other's supertype. This particular problem can be solved in Scala by attaching an abstract type to a class which changes the type of the `this` variable within that class (which would otherwise have the type of the class itself). How does such a *selftype annotation* help us solve the problem we stated? It is perhaps easiest to explain with an example.

Consider that we want to define a finite state machine framework that works with the concepts of `State` and `FSM`. A `State` has to be able to perform an action and to specify transitions, while a `FSM` has to be able to run an input-driven loop that takes it from an initial state to an end state and perform the actions associated with the states in between. A `FSM` needs a reference to a `State` in order to be able to use its provided services, while a `State` might need a reference to the `FSM` in order to perform its action. We can attempt to encode such a system in Scala as follows:

```
trait FiniteStateMachine {
  type T;
  type S <: State;
  type F <: FSM;

  trait State {
    def runState(fsm: F): unit;
    def getNext(x: T): S;
  }

  abstract class FSM(ss: S, endState: S) {
    protected var curState = ss;

    def readInput: T;

    def run = {
      while (curState != endState) {
        val x = readInput;
        curState = curState.getNext(x);
        curState.runState(this);
      }
    }
  }
}
```

The first thing to notice is that neither `State` nor `FSM` refers the other one directly. Having such hard links would lead us to the original problem we set out to solve in the first place, since any extension of `State` would keep on referring the base `FSM` class and vice-versa. What we do instead is introduce the `S` and `F` abstract types which we restrict with an upper type bound to subclasses of `State` and `FSM`, respectively. This mechanism allows refinements of the two classes to bind the types `S` and `F` to whatever concrete types they define and thus use those concrete types in the signatures of the methods (as we will see soon). However, before we get there, we have to point out that the code above is not entirely type correct. The culprit is the statement `curState.runState(this)`, because it passes `this`, which has the type `FSM` to the method `runState` which requires the type `F`. But `F` is an abstract type that will eventually be bound to a subtype of `FSM`, and passing an instance of a superclass where one of its subclasses is expected is clearly a type error.

In order to solve this problem, Scala allows us to change the type of the self reference `this` to an arbitrary type by using the keyword `requires`. In our case, we want to change the type of `this` in class `FSM` from `FSM` to `F`, so we encode this as follows:

```
abstract class FSM(ss: S, endState: S)
  requires F
{ ... }
```

With this modification, the code above becomes type correct. According to [7], the use of selftypes is guaranteed to be correct by requiring that:

> (1) the selftype of a class must be a subtype of the selftypes of all its base classes,
> (2) when instantiating a class in a `new` expression, it is checked that the selftype of the class is a supertype of the type of the object being created.

We can now proceed to show how the two classes (`State` and `FSM`) can be specialized together and how their specialized versions reference each other instead of each other's superclass. This is made possible by the fact that both the curState variable in the FSM class and the fsm parameter to the runState method in the State class are typed as abstract types in the code that we have already presented. Then, in the specialized class, we can bind the abstract types to the concrete ones we are defining and thus use those as our references. The code below shows how this is done.

```
object MyFSM extends FiniteStateMachine {
  type T = char;
  type S = PrintState;
  type F = CharFSM;

  abstract class PrintState extends State {
    val msg: String;

    def runState(fsm: CharFSM) = {
      Console.println(msg +
        " after reading character " +
        fsm.input.charAt(fsm.ipos-1));
    }
  }

  class CharFSM(ss: PrintState,
                fs: PrintState, i: String)
    extends FSM(ss, fs) {
    val input: String = i;
    private var pos: int = 0;

    def readInput: char = {
      if (pos == input.length)
        pos = 0;
      pos = pos+1;
      input.charAt(pos-1);
    }

    def ipos = pos;
  }
}
```

The object definition of above extends the original FiniteStateMachine class and creates a singleton instance called MyFSM of the extended class. Concrete states of this extension (PrintStates) simply print a predefined message to the screen, while the FSM itself (CharFSM) reads the input from a predefined string, character by character, circularly. T, the abstract type that defines the input, is accordingly bound to char.

In what the subject at hand is concerned, covariant extension of types, you can notice that S is bound to PrintState, while F is bound to CharFSM. This renders the use of the CharFSM reference in State, fsm, type correct. We could, in a different extension of the two base classes, also use the protected variable curState defined in the class FSM to access the members defined in the class extending State. This is possible because the type of curState is the abstract type S, which gets bound in the extension to the concrete state class being defined. The advantage of the combined abstract type - selftype annotation mechanism is that we do not have to (like we would in Java of C++) perform an unsafe downcast of, e.g., the fsm argument from FSM to CharFMS in order to use the members defined in CharFSM only (such as input or ipos).

This last piece of code shown below illustrates how MyFSM can be used in order to finalize the definition of a concrete finite state machine that takes characters as its input and goes through three states. This part does not introduce any new language features or concepts, so we present it for completeness, but refrain from any further discussion about it.

```
object Test {
  import MyFSM._;

  val s1: PrintState = new PrintState {
    val msg = "In state 1";
    def getNext(x: char): PrintState =
      x match {
      case 'a' => s2
      case 'b' => s3;
    }
  }

  val s2 = new PrintState {
    val msg = "In state 2";
    def getNext(x: char): PrintState =
      x match {
      case 'a' => s1;
      case 'b' => this
      case 'c' => s3;
    }
  }

  val s3 = new PrintState {
    val msg = "In state 3";
    def getNext(x: char): PrintState =
      this;
  }

  val fsm =
    new CharFSM(s1, s3, "aaabbaabcd");

  def main(args: Array[String]): unit = {
    fsm.run;
  }
}
```

3 The Scala composition paradigm

We use the rest of this paper to explain, by means of two examples, how the three language concepts we have presented so far can be put to use in order to achieve the purpose stated in the introduction of this paper. If you recall, the main problem we set out to solve was how to allow a component to be reused in a context different than the original one in which it was designed without having to intervene in the source code of the component itself. Our first example which shows how we can do that with Scala is a simple broker system.

This system, as we have imagined it, consists of two distinct servers, one for buyers, or *bidders*, (the bid server) and one for sellers (the ask server). The intent is that when assuming a certain role in a transaction, a client will connect to the server fulfilling that role and make his/her bid for buying stock or offer of selling stock. A third, backend server continuously tries to match bids with offers, on success allowing transactions to complete. A fourth module will be used to represent the clients' accounts, which constitute the data of our system. Closing a transaction will in effect take shares from the seller's account and put them in the buyer's one. This is, in a nutshell, how the system is intended to function.

There are thus four components that form the system: `Account`, `MatchServer`, `AskServer` and `BidServer`. While the first of them, `Account`, only provides services to the others, without requiring any services itself, the other three are highly interdependent. The `MatchServer` needs the services of both the `AskServer` and the `BidServer`, because it needs to be able to notify each when transactions are completed. Each of these two, in turn, need the services of the `MatchServer` and those of `Account`, since they need to first register bids and offers and later, upon transaction completion, to modify the accounts of the two parties. This system of interdependencies is illustrated in figure 1.

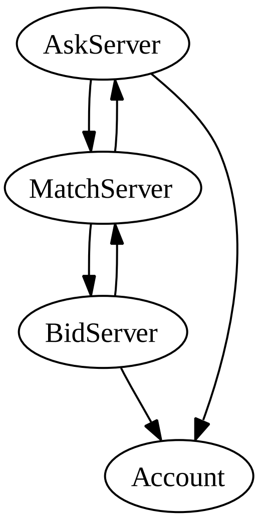

Figure 1: Component dependencies

There are a number of problems with common approaches to implementing such a system which are discussed at length in [7]. The first problem is that by using top-level classes to represent such components, we automatically defeat our goal since such a model creates hard links among components. The second problem that we have with top-level classes is that as soon as they need to use static data members, the overall system stops being reentrant. In other words, we can not run two (or more) instances of the system side by side, because their data would get mixed. Finally, if we try to solve the first problem by various means (such as introducing contexts that map symbolic names to components, for example, and passing these contexts around the system) we usually either lose static type safety or we have to write a system based on complex design patterns which (1) affect performance and (2) allow a myriad of opportunities for misuse which cannot be detected by the system.

This brings us to the conclusion that it is advisable to use nesting of classes in order to systematically solve the problem of reentrancy. The outer class will represent the entire system and each component will be implemented as an inner class. This allows us, at least in Scala, to isolate the static data of each system as one or more nested objects that are also part of the outer class in a structure similar to this one:

```
class System {
  class Component1 = ...
  class Component2 = ...
  object staticData = ...
}
```

If you recall, object definitions are Scala's way of expressing static members. If `s1` and `s2` are two instances of the class `System`, each of them will have its own copy of the `staticData` object, so we can safely use them side by side. Moreover, Scala's singleton types ensure that one cannot use members of `s1.staticData` interchangeably with matching members of `s2.staticData`, since such members have different types. This provides for increased type safety when using multiple instances of `System` in parallel.

However, having all our components bundled in a single class is by no means a modular construction, so the next challenge is to figure out how to split them in multiple source files, while still keeping them in a nested setting. The difficulty in doing this lies with the fact that nested classes or objects need to be able to reference other nested classes or objects that are defined in different source files, as part of different outer classes. The Scala solution to this involves defining the selftype for each outer class in such a way that it is a mixin of all the outer components that it needs in order to function. In other words, we use the selftype annotation for the class in order to specify its required services. We show below how this works for the case of the broker system:

```
// in file Accounts.scala
trait Accounts {
  class Account { ... }
}

// in file MatchServers.scala
trait MatchServers
  requires (MatchServers
            with AskServers
            with BidServers) {
  abstract class Input {
    val client: String;
    val symbol: String;
    val amount: int;
    val price: int;
  }

  object MatchServer { ... }
}

// in file AskServers.scala
trait AskServers
  requires (AskServers
            with Accounts
            with MatchServers) {
  object AskServer { ... }
}

// in file BidServers.scala
trait BidServers
  requires (BidServers
            with Accounts
            with MatchServers) {
  object BidServer { ... }
}
```

Except for the `Accounts` component, which has no required services, all the other components declare a selftype annotation which ensures that when they will actually be instantiated, the type of the created object will conform to the mixin type specified in the selftype annotation. By being able to enforce this with the aid of the system, it is possible for all these classes to use the nested members defined by their required components as if they had defined them themselves. For instance, the `AskServer` object can use the `Input` object defined by `MatchServers` simply by referring to it as `Input`. Similarly, the `MatchServer` object can use any member defined by the `AskServers` and `BidServers` without any prefix, simply because the system knows that at runtime, the type of the self reference `this` in any instance of the class `MatchServers` will be a subtype of both `AskServers` and `BidServers`.

Creating an instance of the broker system is now a matter of mixing in all the components together in a concrete object (or class). By mixing everything together we effectively ensure the right selftype required by each component and thus the overall composition is type correct.

```
object BrokerSystem extends Accounts
                   with MatchServers
                   with BidServers
                   with AskServers {
  def main(args: Array[String]) = { ... }
}
```

This example (the complete source code of which you can find in the appendix) has shown how selftype annotations can be used together with mixin composition in order to abstractly specify the required services for a component. This mechanism is elegant, lightweight, flexible and ensures type correctness at compile time. While appropriate for many systems, it can be the case that for some scenarios the approach we presented above is either too coarse or, on the contrary, too fine grained. In the latter case, we can simply replace all selftype annotations with the type of a class that mixes in all the components that are required. In our case, this class would be defined in a similar way as the `BrokerSystem` object above, but leaving out the definition of the main method. Assuming we have defined such a `BrokerSystem` class, we can, for instance, replace the selftype of `MatchServers` with this:

```
trait MatchServers
  requires BrokerSystem { ... }
```

If we need to specify the required services more accurately, we can drop the selftype annotations altogether and use abstract type members and abstract method definitions in the component classes. Since in order to instantiate a class all its members have to be defined, we can be sure that after we mix in all of the components, the required abstract members will be overridden by a concrete implementation from some other component. Consider the following specification of `MatchServers`, for instance:

```
trait MatchServers {
  type AskServer <: AskServerInterface;
  type BidServer <: BidServerInterface;

  def completeBid(bid: Input): unit;
  def completeAsk(ask: Input): unit;

  object MatchServer { ... }
}
```

Now, instead of requiring simply that `AskServers` and `BidServers` become superclasses of the concrete instance of `MatchServers`, we drop this generic requirement and replace it with the `AskServer` and `BidServer` abstract member types and specify a predefined interface for them in the form of upper type bounds. Similarly, method definitions that we expect to be present in the composed component are specified explicitly.

4 Using Scala to solve the expression problem

In this last example we outline how Scala can be used for solving the so-called *expression problem.* The entire contents of this section is based on a technical report by Zenger and Odersky [6], parts of which we summarize in the context of this paper. The report offers two solutions to the same problem, one in an object-oriented approach and one in a functional approach. We only discuss the former in this section, while we leave it to the interested reader to consult [6] for the latter.

Before going to the solution, however, let us first see what the expression problem is. Assume the common setting of a data type that is defined by a number of cases and operations that perform various functions over this data type (computing information, changing the data type, etc.). Such a data type - operations pair can be extended by adding more elements to either of its members. We can either extend the data type to include new cases or we can extend the set of operations that we perform over the data type. Or, naturally, we can extend both.

Our goal is to allow both these types of extensions without changing *or* duplicating the existing code. Furthermore, we want the composed system (including the extensions) to be statically typed so that we know it is consistent and, lastly, it should be possible to combine independent extensions without using any “glue code”. This problem is clearly an instance of the general software composition one analyzed by this paper, albeit one quite different than our previous broker example. You will notice, however, that the same three concepts we have discussed are used in this case as well in order to ensure an elegant and flexible solution.

It is in fact the requirements outlined in the previous paragraph that make the common solutions to this problem incomplete. Existing solutions fail to comply with one or more of these requirements. Some do not ensure type safety either because they rely on reflection or type cases. Some that offer type safety require modifications of the existing code base in order to integrate the extensions, thus implicitly also failing to meet the independent extensibility requirement. More complex solutions that ensure type safety and require no need to modify the base code still do not support independent extensibility.

Our toy language throughout this example will be a boolean expression language that will initially consist of only one constructor and one operation. We will then build on this language by adding extra data type constructors as well as extra operations. The core language, that only supports boolean values and the `eval` operations, can be expressed like this:

```
trait BoolLang {
  type E <: BoolExp;
  trait BoolExp {
    def eval: boolean;
  }
  class Val(v: boolean) extends BoolExp {
    val value = v;
    def eval = { value; }
  }
}
```

Notice the use of the abstract type `E` which allows for further extensibility of the language. Had we not introduced this abstract type, the only operation that the `BoolExp` type would have ever supported is the `eval` operation. The reason for this is that, as we will show immediately, additional data constructors need to refer variables of the expression type. Without `E`, our only option would have been to use `BoolExp` as that type, which would have limited the operations that can be used with those variables to just `eval`. Of course, when we want to create concrete instances of our language, we will have to bind `E` to a concrete type, as shown in our next code snippet.

```
object Test extends BoolLang
  with Application {
  type E = BoolExp;
  val e = new Val(true);
  Console.println(e.eval);
}
```

Mixing in the `Application` system-defined trait is an alternative Scala offers to explicitly defining a `main` method. The entire code of the singleton object `Test` will be executed as the body of the `main` method.

Let us now show how `BoolLang` can be extended by two different components in order to support the `And` and the `Or` data constructors (first component) and the `Not` constructor (second component). The two extensions are independent of one another and can each be used separately to provide a language with `Val`, `And` and `Or` and one with `Val` and `Not`, respectively.

```
trait BLAndOr extends BoolLang {
  class And(l: E, r: E) extends BoolExp {
    val lexp = l;
    val rexp = r;
    def eval = { lexp.eval && rexp.eval; }
  }
  class Or(l: E, r: E) extends BoolExp {
    val lexp = l;
    val rexp = r;
    def eval = { lexp.eval || rexp.eval; }
  }
}

trait BLNot extends BoolLang {
  class Not(e: E) extends BoolExp {
    val exp = e;
    def eval = { ! exp.eval; }
  }
}
```

As you can see, extending the data constructors of the language is fairly straight-forward and does not imply any modifications of the original source code, since the implementation of the `eval` method can be supplied in the extensions. All three constructors refer boolean expressions using the `E` abstract type in order to provide extensibility, as explained before.

Aside from using them independently, we can also combine the two extensions in order to obtain a language that supports all four constructors (Val, And, Or and Not). In order to do this, we will naturally use mixin composition to gather all the definitions in a single component. Notice the simplicity of the mechanism.

```
trait BLAndOrNot
  extends BLAndOr with BLNot;
```

Once we have combined the extensions, we can use the resulting boolean expression language much in the same way as we used the base one before. Only now, we have access to the additional constructors, which lets us define more interesting expressions:

```
object Test extends BLAndOrNot
  with Application {
  type E = BoolExp;
  val e = new And(
            new Or(new Val(true)
                 , new Val(false))
          , new Not(new Val(true))
          );
  Console.println(e.eval);
}
```

The second type of extension that we need to illustrate is the extension of the operations that can be applied to the data type. Given the object-oriented approach adopted in this example, extension of operations is a bit more verbose than that of constructors. Nevertheless, it still complies with our specifications in that it does not require any changes to or duplication of the base code. First, we provide the extension of the base language with a new operation for showing the data type.

```
trait BLShow extends BoolLang {
  type E <: BoolExp;
  trait BoolExp extends super.BoolExp {
    def show: String;
  }
  class Val(v: boolean)
    extends super.Val(v) with BoolExp {
    def show = { value.toString; }
  }
}
```

Comparing this to the previous extensions, we notice here that we had to create another version of BoolExp in which to add the definition of the show method. Using the same name for the trait has the effect of hiding the old one, so any client using the BLShow version of the language would only have access to the BoolExp type that supports both the eval and the show operations. Furthermore, the abstract type E is restricted by this extension to subclasses of the new BoolExp in order to ensure that a data member that takes a reference to E will be able to use the show operation as well. Naturally, we need to also subclass the (single) constructor, Val, in order to provide a definition for the newly introduced operation. We again use the same name, Val, as in the original language in order to shadow the old definition that only supported the eval operation.

Our next step is to extend the BLAndOrNot language (that we previously obtained by combining the BLAndOr and the BLNot languages) so that it supports the new show operation. We do this again by using mixin composition at two levels. We mix the BLShow extension in the BLAndOrNot one and, nested in the new trait, we mix the new BoolExp in all the data constructors defined by BLAndOrNot. Finally, we supply implementations for the show operation in all the constructors.

```
trait BLShowAndOrNot
  extends BLAndOrNot with BLShow {
  class And(l: E, r: E)
    extends super.And(l, r) with BoolExp {
    def show = { "(" + lexp.show +
                 " && " +
                 rexp.show + ")"; }
  }
  class Or(l: E, r: E)
    extends super.Or(l, r) with BoolExp {
    def show = { "(" + lexp.show +
                 " || " +
                 rexp.show + ")"; }
  }
  class Not(e: E)
    extends super.Not(e) with BoolExp {
    def show = { "~(" + exp.eval  + ")"; }
  }
}
```

Notice that the E abstract type that we use as the type of the l, r and e parameters is the one defined in BLShow. Since that E is upper bounded by the BoolExp that supports eval and show, we can correctly use the show operation in the code above.

Following the same mechanism as before, we can create an instance of the new BLShowAndOrNot language and use the new operation. The binding type E = BoolExp binds E to the extended BoolExp, since this is the only one visible at this point (given that the Test singleton object inherits from BLShowAndOrNot).

```
object Test extends BLShowAndOrNot
  with Application {
  type E = BoolExp;
  val e = new And(
            new Or(new Val(true)
                 , new Val(false))
          , new Not(new Val(true))
          );
  Console.println(
    e.show + " = " + e.eval);
}
```

Things get even more verbose if we want to extend the language with an operation that transforms the tree over which it operates (as opposed to before, when we only computed a value over the tree). The added complexity comes from the fact that we need to construct in the code of our operation new instances of the data type. However, we cannot simply create instances of the respective constructor classes because this would bind the resulting type of our operation to only that version of BoolExp that is defined thus far. This would effectively prevent the extension from being combined with others that extend the data type. In order to avoid this, we need to create abstract factory methods for each data constructor. We can then use these methods in the code of the extension and postpone their definition up to the point where a concrete language is instantiated.

The extension introduces a new operation called switchArgs, which exchanges the two arguments of binary constructors (And and Or). Except for the factory methods, the rest of the mechanism used to define the BLSwitchAndOrNot extension is identical to that used in the definition of BLShow.

```
trait BLSwitchAndOrNot
  extends BLAndOrNot {
  type E <: BoolExp;
  trait BoolExp extends super.BoolExp {
    def switchArgs: E;
  }
  def Val(v: boolean): E;
  def And(l: E, r: E): E;
  def Or(l:E, r: E): E;
  def Not(e: E): E;
  class Val(v: boolean)
    extends super.Val(v) with BoolExp {
    def switchArgs = { Val(v); }
  }
  class And(l: E, r: E)
    extends super.And(l, r) with BoolExp {
    def switchArgs = {
      And(r.switchArgs, l.switchArgs);
    }
  }
  class Or(l: E, r: E)
    extends super.Or(l, r) with BoolExp {
    def switchArgs = {
      Or(r.switchArgs, l.switchArgs);
    }
  }
  class Not(e: E) extends
    super.Not(e) with BoolExp {
    def switchArgs = { Not(e.switchArgs); }
  }
}
```

If we need to use the BLSwitchAndOrNot boolean expression language we have just defined, we need to provide implementations for all the factory methods. This is not a problem at this point, since we are binding the abstract type E to BoolExp, so no further extension can be assumed anyway. The following code illustrates how this can be done.

```
object Test extends BLSwitchAndOrNot
  with Application {
  type E = BoolExp;
  def Val(v: boolean) = { new Val(v); }
  def And(l: E, r: E) = { new And(l, r); }
  def Or(l: E, r: E) = { new Or(l, r); }
  def Not(e: E) = { new Not(e); }
  val e = And(
            Or(Val(true), Val(false))
          , Not(Val(true))
          );
  Console.println(e.switchArgs.eval);
}
```

The final point on our list would have been to show how independent extensions that each introduce new operations can be combined together with mixin composition. Unfortunately, the mechanism needed for this is no longer supported in the last versions of Scala. The problem is that in order to combine such extensions, we need to mix in classes that are not traits, which is no longer allowed. The 2.x versions of Scala only allow traits to be mixed in, for reasons explained in section 2.3.

Nevertheless, we end this example by presenting the code that would combine the independent extensions that add the `show` and the `switch` operations, respectively, had Scala not dropped mixin composition of regular classes. The difference from the previous time when we combined the `BLAndOr` and the `BLNot` extensions into the `BLAndOrNot` one is that now, in the absence of a language mechanism that would do this automatically, we have to perform the mixin of all the nested types that define data constructors manually. The code that realizes this is shown below.

```
trait BLShowSwitchAndOrNot
  extends BLShowAndOrNot
  with BLSwitchAndOrNot {
  type E <: BoolExp;
  trait BoolExp
    extends super[BLShowAndOrNot].BoolExp
       with super[BLSwitchAndOrNot].BoolExp;
  class Val(v: boolean)
    extends super[BLShowAndOrNot].Val(v)
       with super[BLSwitchAndOrNot].Val(v)
       with BoolExp;
  class And(l: E, r: E)
    extends super[BLShowAndOrNot].And(l, r)
       with super[BLSwitchAndOrNot].And(l, r)
       with BoolExp;
  class Or(l: E, r: E)
    extends super[BLShowAndOrNot].Or(l, r)
       with super[BLSwitchAndOrNot].Or(l, r)
       with BoolExp;
  class Not(e: E)
    extends super[BLShowAndOrNot].Not(e)
       with super[BLSwitchAndOrNot].Not(e)
       with BoolExp;
}
```

5 Conclusion

In this paper we have shown that the Scala programming language is equipped with features that enable an intuitive “design pattern” for ensuring component composition with minimal adaptation effort. We have discussed the three essential features, abstract type members, modular mixin composition and selftype annotations that make this possible and presented small examples to clarify each. In addition, we believe we have also achieved a composition-centric introduction to the Scala language itself, especially through the source code of our examples.

The second part of the paper explained the design pattern by working out two examples of composable systems. These showed how hard coded references can be abstracted out of components and specified explicitly at different possible levels of granularity, while in the same time maintaining full static type safety. The latter example also constitutes an interesting solution to the expression problem in itself, albeit a partial one since, as explained, with the new version of the Scala language it can no longer support independent extensions.

References

[1] Scala home page. `http://scala.epfl.ch/`.

[2] Czarnecki, K., and Eisenecker, U. W. Components and generative programming (invited paper). In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering* (London, UK, 1999), Springer-Verlag, pp. 2–19.

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[4] Odersky, M., and al. An overview of the scala programming language. Tech. Rep. IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[5] Odersky, M., Cremet, V., Röckl, C., and Zenger, M. A nominal theory of objects with dependent types. In *Proc. ECOOP'03* (July 2003), Springer LNCS.

[6] Odersky, M., and Zenger, M. Independently extensible solutions to the expression problem. In *Proc. FOOL 12* (Jan. 2005). `http://homepages.inf.ed.ac.uk/wadler/fool`.

[7] Odersky, M., and Zenger, M. Scalable component abstractions. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), ACM Press, pp. 41–57.

[8] Reid, A., Flatt, M., Stoller, L., Lepreau, J., and Eide, E. Knit: Component composition for systems software. In *Proc. of the 4th Operating Systems Design and Implementation (OSDI)* (Oct. 2000), pp. 347–360.

[9] Szyperski, C. Component technology: what, where, and how? In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering* (Washington, DC, USA, 2003), IEEE Computer Society, pp. 684–693.

Appendix A

```
// file Accounts.scala
trait Accounts {

  class Account(client: String) {
    def buy(symbol: String, amount: int, price: int) = {
      Console.println("Client " + client + " bought " + amount + " " +
        symbol + " shares at " + price + " euro a share.")
    }

    def sell(symbol: String, amount: int, price: int) = {
      Console.println("Client " + client + " sold " + amount + " " +
        symbol + " shares at " + price + " euro a share.")
    }
  }

  def getAccountFor(client: String) = {
    new Account(client);
  }

}

// file MatchServers.scala
import scala.collection.mutable._;

trait MatchServers requires (MatchServers with AskServers
                                          with BidServers) {
  abstract class Input {
    val client: String;
    val symbol: String;
    val amount: int;
    val price: int;
  }

  object MatchServer {

    val bids = new ArrayBuffer[Input];
    val asks = new ArrayBuffer[Input];

    def registerBid(bid: Input): unit = {
      val omatch = asks.elements find (x => (x.symbol == bid.symbol) &&
                                            (x.amount == bid.amount) &&
                                            (x.price <= bid.price))
      if (!omatch.isEmpty) {
        val ask = omatch.get;
        completeBid(bid);
        completeAsk(ask);
        // delete match from asks
      } else {
        bids += bid;
      }
    }

    def registerAsk(ask: Input): unit = {
      val omatch = bids.elements find (x => (x.symbol == ask.symbol) &&
                                            (x.amount == ask.amount) &&
                                            (x.price >= ask.price))
      if (!omatch.isEmpty) {
        val bid = omatch.get;
        completeBid(bid);
        completeAsk(ask);
      } else {
        asks += ask;
      }
    }
  }
}

// file BidServers.scala
import concurrent.ops._;

trait BidServers requires (BidServers with Accounts
                                      with MatchServers) {

  def completeBid(bid: Input): unit = {
    getAccountFor(bid.client).buy(bid.symbol, bid.amount, bid.price);
  }

  object BidServer {

    private def readInput: Input = {
      // read c, s, a & p from socket
      new Input {
        val client = c;
        val symbol = s;
        val amount = a;
        val price = p;
      }
    }

    private def worker: unit = {
      while(true) {
        val in = readInput;
        MatchServer.registerBid(in);
      }
    }

    def run: unit = {
      spawn(worker);
    }
  }
}

// file AskServers.scala
import concurrent.ops._;

trait AskServers requires (AskServers with Accounts
                                      with MatchServers) {

  def completeAsk(ask: Input): unit = {
    getAccountFor(ask.client).sell(ask.symbol, ask.amount, ask.price);
  }

  object AskServer {

    private def readInput: Input = {
      // read c, s, a & p from socket
      new Input {
        val client = c;
        val symbol = s;
        val amount = a;
        val price = p;
      }
    }

    private def worker: unit = {
      while(true) {
        val in = readInput;
        MatchServer.registerAsk(in);
      }
    }

    def run: unit = {
      spawn(worker);
    }
  }
}

// file BrokerSystem.scala
object BrokerSystem extends Accounts
                    with MatchServers
                    with BidServers
                    with AskServers {
  def main(args: Array[String]) = {
    AskServer.run;
    BidServer.run;
  }
}
```