Component Composition with Scala

Bogdan Dumitriu

Department of Computer Science
University of Utrecht

June 1st, 2006

Outline

Software Engineering

The reality of software engineering:

- tons of software components exist
- … and more are created daily
- specifying services offered by a component is commonplace
- changing services used by a component
  - needs complex design patterns
  - is usually not possible
- *software writing is currently an art*
- can we turn it into an industry?

Software Engineering

The reality of software engineering:

- tons of software components exist
- … and more are created daily
- specifying services offered by a component is commonplace
- changing services used by a component
  - needs complex design patterns
  - is usually not possible
- *software writing is currently an art*
- can we turn it into an industry?

Composition

Composition in other braches of engineering:

- components are written to conform to standards
    - e.g., integrated circuits, car parts
- systems are assembled from parts
- no adaptation is necessary

Composition in software:

- components comply to no standards
- systems are not assembled from parts
- at most, they reuse certain independent parts
- to put components together, “glue” code is usually needed

Composition

Composition in other braches of engineering:

- components are written to conform to standards
  - e.g., integrated circuits, car parts
- systems are assembled from parts
- no adaptation is necessary

Composition in software:

- components comply to no standards
- systems are not assembled from parts
- at most, they reuse certain independent parts
- to put components together, “glue” code is usually needed

The Software Composition Problem
Reasons

Why can't we build software from components?

- components are only made to work in one environment
- to reuse them in a different context we need to abstract over:
    - provided services
    - required services
- required services are generally hard coded
- design patterns can alleviate this, but
    - some are difficult to understand
    - can be circumvented since they are not enforced by the system
    - complicate the design significantly
    - often involve unsafe type casts

The Software Composition Problem

Requirements

To advance towards seamless composition of components:

- lift all hard coded links into abstract specifications
- required services should be as easy to specify as provided ones
- mechanism of composition with static type checking
- composition should be possible without any changes to components

The Scala programming language allows all this and

- infinite scalability (components are classes)

The Software Composition Problem
Requirements

To advance towards seamless composition of components:

- lift all hard coded links into abstract specifications
- required services should be as easy to specify as provided ones
- mechanism of composition with static type checking
- composition should be possible without any changes to components

The Scala programming language allows all this and

- infinite scalability (components are classes)

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
  - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
  - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
  - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
  - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
    - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
    - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

What is Scala?

Scala

- is a programming language
- is an object oriented programming language
- is a functional programming language
- has a featureful static type system
  - generics, variance annotations, abstract types, selftypes, views
- has a large variety of syntactic constructs
- interoperates smoothly with Java and C#

Using Scala with Java

Collecting arguments

```
import java.util.ArrayList;

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

Scala Mechanisms for Composition

Scala features that enable composition:

- nested classes
- abstract type members
- modular mixin composition
- selftype annotations
- views

Abstract Type Members

Abstract type members

- type variables that appear at the class level
- generalize the idea of abstract members to types
- the type can be used throughout the class
- similar to generics

Abstract Type Member T

```
abstract class SomeClass {
  type T;
  [...]
}
```

Abstract Type Members

Buffer with abstract type member

```
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 = ...
}
```

Abstract Type Members

To create instances of `Buffer`, we have to provide `T`.

Instantiation of abstract type

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

Abstract Type Members

Composition-wise,

- type members are “hooks” for plugging in components
- allow class methods to not use concrete types
  - thus eliminating hard coding of components

Unrestricted type members are rarely useful ⇒ upper type bounds

Abstract Type Members

Composition-wise,

- type members are “hooks” for plugging in components
- allow class methods to not use concrete types
  - thus eliminating hard coding of components

Unrestricted type members are rarely useful $\Rightarrow$ upper type bounds

Abstract Type Members

Upper Type Bounds

Upper type bounds:

- have similar uses as type classes in Haskell
- restrict the range of an abstract type to subclasses of the bound
- compositionally, can be used to specify required services

Sample Bound

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

Abstract Type Members

Upper Type Bounds

Upper type bounds:

- have similar uses as type classes in Haskell
- restrict the range of an abstract type to subclasses of the bound
- compositionally, can be used to specify required services

Sample Bound

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

Abstract Type Members
Upper Type Bounds

Upper type bound

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

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

Abstract Type Members

Views

Views:

- different mechanisms to restrict type variables
- useful for component adaptation
- allows to "view" a component as a different one
- implicit methods applied automatically by the compiler
  - when an expression does not match expected type
  - when a member of an expression doesn't exist in its type
- application is controlled by scoping and specificity

Abstract Type Members

Views

Set & List

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

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

Abstract Type Members

Views

View from List to Set

```
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))
  }
```

Abstract Type Members

Views

Using the view

```
object Test {
  def addToSet[T](elem: T, set: Set[T]) = {
    set.add(elem);
  }
  def test = {
    val l: List[char] = new List;
    addToSet('a', l);
  }
}
```

Restricting a type

```
def m[T <% U](x: T) = { ... }
```

Abstract Type Members
Views

Using the view

```
object Test {
  def addToSet[T](elem: T, set: Set[T]) = {
    set.add(elem);
  }
  def test = {
    val l: List[char] = new List;
    addToSet('a', l);
  }
}
```

Restricting a type

```
def m[T <% U](x: T) = { ... }
```

Modular Mixin Composition

Mixin Composition:

- Scala mechanism for multiple inheritance
- avoids the "diamond problem"
- but still allows code reuse
- based on stateless classes called traits
- compositionally, the method for building systems

Modular Mixin Composition
Traits

Traits:

- stateless classes
  - no constructor parameters
  - no variable definitions
- allow method definitions $\Rightarrow$ code reuse
- can be used anywhere abstract classes can
- are the only classes that can be *mixed in*

Modular Mixin Composition
Traits

Defining traits

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

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

Modular Mixin Composition

Traits

Using traits

```
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; }
}
```

Modular Mixin Composition

Mixing in traits

```
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;
  }
}
```

Modular Mixin Composition

Mixins, formally:

- first class: *superclass*, the rest: *mixins*
- superclass must be a subclass of all the superclasses of the mixins
- inheritance relationships $\Rightarrow$ DAG

Linearization of C extends C_1 with ... with C_n

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

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

Modular Mixin Composition

Linearization of `FileDebugger extends FileLogger with Debugger`:

$$
\begin{aligned}
\mathcal{L}(\text{FileDebugger}) &= \{\text{FileDebugger}\} \vec{+} \mathcal{L}(\text{Debugger}) \vec{+} \mathcal{L}(\text{FileLogger}) \\
&= \{\text{FileDebugger}\} \vec{+} \{\text{Debugger}\} \vec{+} \mathcal{L}(\text{Logger}) \vec{+} \\
&\{\text{FileLogger}\} \vec{+} \mathcal{L}(\text{Logger}) \\
&= \{\text{FileDebugger}\} \vec{+} \{\text{Debugger}\} \vec{+} \{\text{Logger}\} \vec{+} \\
&\{\text{FileLogger}\} \vec{+} \{\text{Logger}\} \\
&= \{\text{FileDebugger}, \text{Debugger}, \text{FileLogger}, \text{Logger}\}
\end{aligned}
$$

Modular Mixin Composition

Concrete members

The concrete member that appears in the leftmost class in $\mathcal{L}(C)$ will be inherited in the composition, *even if there are matching abstract members more to the left*.

Abstract members

The abstract member that appears in the leftmost class in $\mathcal{L}(C)$ will be inherited in the composition, *unless there is at least a matching concrete member anywhere in $\mathcal{L}(C)$*.

Modular Mixin Composition

Concrete members

The concrete member that appears in the leftmost class in $\mathcal{L}(C)$ will be inherited in the composition, *even if there are matching abstract members more to the left*.

Abstract members

The abstract member that appears in the leftmost class in $\mathcal{L}(C)$ will be inherited in the composition, *unless there is at least a matching concrete member anywhere in* $\mathcal{L}(C)$.

Selftype Annotations

Selftype annotations:

- all mixins of a class have to be concrete types
- what if we need a class to extend an abstract type?
- use selftype annotations
- changes the type of the self reference *this*
- compositionally, the same purpose as abstract types

Selftype Annotations

Family polymorphism

Family polymorphism:

- systems with two or more types that
  - mutually reference one another
  - tend to vary together
- in Java/C++, extended types will refer the supertypes
- in Scala, this can be solved using selftype annotations

Selftype Annotations

FSM Example

Finite State Machine (1)

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

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

Selftype Annotations

FSM Example

Finite State Machine (2)

```
  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);
      }
} } }
```

Selftype Annotations
FSM Example

Selftype Annotation for FSM

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

Requirements (according to [1]):

- selftype of a class must be a subtype of the selftypes of all its base classes
- 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

Selftype Annotations

FSM Example

Selftype Annotation for FSM

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

Requirements (according to [1]):

- selftype of a class must be a subtype of the selftypes of all its base classes
- 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

Selftype Annotations

FSM Example

State Definition

```
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));
    }
  }
```

Selftype Annotations

FSM Example

FSM Definition

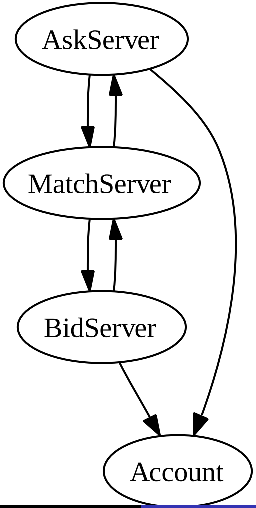
```
  class CharFSM(ss: PrintState, fs: PrintState, i: String)
    extends FSM(ss, fs) {
    val input: String = i;
    private var pos: int = 0;

    def readInput: char = {
      [...]
      input.charAt(pos++);
    }

    def ipos = pos;
} }
```

Broker System
Description

Example: A Broker System

- original problem: reuse component without changing source code
- the system's components:
    - two interface servers, one for bidders, one for sellers
    - one backend server, to match transactions
    - component representing clients' accounts
- components are highly interconnected

Broker System
Description



Broker System
Issues

So, why isn't this easy?

- basic solution with components as top level classes
  - introduces hard links
  - prevents re-entrancy in the presence of static data
- pass components around in a context object
  - loses type safety
- use various design patterns
  - reduces performance
  - it is up to the user to enforce their use

Broker System
Issues

So, why isn't this easy?

- basic solution with components as top level classes
  - introduces hard links
  - prevents re-entrancy in the presence of static data
- pass components around in a context object
  - loses type safety
- use various design patterns
  - reduces performance
  - it is up to the user to enforce their use

Broker System

Issues

So, why isn't this easy?

- basic solution with components as top level classes
    - introduces hard links
    - prevents re-entrancy in the presence of static data
- pass components around in a context object
    - loses type safety
- use various design patterns
    - reduces performance
    - it is up to the user to enforce their use

Broker System
Solution

Guideline to solution (in Scala):

- use nested classes
- outer class: entire system
- inner classes: components

System with nested classes (and objects)

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

Broker System
Solution

file Accounts.scala

```
trait Accounts {
  class Account { ... }
}
```

file MatchServers.scala

```
trait MatchServers
  requires (MatchServers
            with AskServers
            with BidServers) {
  abstract class Input {
    val client: String;
    [...]
  }
```

Broker System

Solution

file Accounts.scala

```
trait Accounts {
  class Account { ... }
}
```

file MatchServers.scala

```
trait MatchServers
  requires (MatchServers
            with AskServers
            with BidServers) {
  abstract class Input {
    val client: String;
    [...]
  }
```

Broker System
Solution

file AskServers.scala

```
trait AskServers
  requires (AskServers
            with Accounts
            with MatchServers) {
  object AskServer { ... }
}
```

file BidServers.scala

```
trait BidServers
  requires (BidServers
            with Accounts
            with MatchServers) {
  object BidServer { ... }
}
```

Broker System
Solution

file AskServers.scala

```
trait AskServers
  requires (AskServers
            with Accounts
            with MatchServers) {
  object AskServer { ... }
}
```

file BidServers.scala

```
trait BidServers
  requires (BidServers
            with Accounts
            with MatchServers) {
  object BidServer { ... }
}
```

Broker System

The final system:

- literally mix everything together

Final system

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

Broker System
Granularity

Granularity of the required services:

- currently at component level
- not necessarily appropriate for all systems
- can be made coarser
- or more fine grained

Broker System
Granularity

Coarser granularity

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

trait AskServers
  requires BrokerSystem { ... }

trait BidServers
  requires BrokerSystem { ... }
```

Broker System
Granularity

Finer granularity

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

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

  object MatchServer { ... }
}
```

Conclusion

The four language features that enable composition:

- nested classes
- abstract type members
- mixin composition
- selftype annotations

Bonus: the Scala language

Bibliography

[1] Odersky, M., 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.

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

[3] Dumitriu, B. Component Composition with Scala. `http://losser.labs.cs.uu.nl/~bdumitri/article.pdf`

Questions?

Q & A Session