Theoretical and Applied Informatics
Vol. 28 (2016), no. 3, pp. 1–7
DOI: 10.20904/283001

A Modularity Bug in Java 8*

Simon KRAMER[1,2†]

[1]Bern University of Applied Sciences
[2]SK-R&D Ltd liab. Co

Abstract We demonstrate a modularity bug in the interface system of Java 8 on the practical example of a textbook design of a modular interface for vector spaces. Our example originates in our teaching of modular object-oriented design in Java 8 to undergraduate students, simply following standard programming practices and mathematical definitions. The bug shows up as a compilation error and should be fixed with a language extension due to the importance of best practices (design fidelity).

Keywords component-based software construction, interface specifications, object-oriented programming, programming-language pragmatics

Received 24 JAN 2017 **Revised** 24 FEB 2017 **Accepted** 01 MAR 2017

1 Introduction

We demonstrate a modularity bug in the interface system of Java 8 [2] on the practical example of a textbook design of a modular interface for vector spaces, which are very important for many computer applications of linear algebra. Our design is textbook in the sense that we simply follow standard programming practices and mathematical definitions, as any mathematically-inclined teacher of object-oriented programming in Java would do. That is, we construct our design from its standard algebraic components, thus ending up with not only *a* but actually *the* natural modular design (by the algebraic definition of vector spaces), which however—and to our great surprise and embarrassment to our students—fails compilation due to an important expressiveness limitation of the Java 8 language. This limitation resides in its (not so) generic type system and should be remedied with an appropriate language extension, for the sake of keeping best practices best and saving our faces to our students of Java.

Nota bene We do *not* claim that it be impossible to model vector spaces in Java 8; it *is* possible. What we do claim is that it is impossible to specify with interfaces and implement with corresponding classes such spaces in Java 8 *in the natural modular way,* whereby we mean *reflecting their algebraic structure with its definitional components* (design fidelity). So the great modularity promise of component-based software construction with object-oriented programming in Java has not been fulfilled yet, even after more than 20 years. In particular (counterexample, bug report),

*See also technical report [1].
†E-mail: simon.kramer@a3.epfl.ch

it cannot be fulfilled in Java 8 when *combining* such basic components as a field of scalars (e.g., rational numbers) and an additive group of vectors of such scalars so as to form a vector space (e.g., over rationals). Unfortunately, not even the planned Java 9 module system [3]

- representing "a fundamental shift to modularity as a first-class citizen for the whole Java platform" and
- in which "interfaces play a central role"

offers the required basic modularity, perhaps because of prioritising packages (coarser-grained modularity through import and export control) over interfaces (finer-grained modularity through multiple-inheritance-based composition). Of course, modularity transcends grain size by its very definition. The fact that it does not do so in Java makes it *ad hoc* and broken in Java, strictly speaking.

2 The bug

We demonstrate the modularity bug in the interface system of Java 8 by constructing our interface (specification) for vector spaces bottom-up from its standard algebraic (interface) components, combining them through multiple inheritance when needed. Luckily, Java 8 offers multiple inheritance from interfaces. Recall that from classes, Java 8 only offers single inheritance. Our following interface components specify the algebraic operations up to our sought vector spaces, however without stipulating their corresponding algebraic (equational) laws (constraints). It would be nice if also this could become possible in a future release of Java in which interfaces could carry such algebraic structure, *à la* algebraic specifications [4]. In particular, the commutativity of commutative rings—including the commutativity of their two constituting components of an additive group and a multiplicative monoid—is left implicit (unspecified) in the following corresponding interface (Page 4). So here, "additive" and "multiplicative" implicitly imply "commutative" (like for numerical operations).

Our interfaces are presented as divided up into non-problematic and problematic components. The non-problematic interface components are meant to illustrate important modular (algebraic) interface specifications that *are* possible in Java 8 (though unfortunately still without algebraic constraints). The problematic interface components are meant to demonstrate an important modular (algebraic) interface specification that is *not* possible in Java 8, namely the one for vector spaces (our bug). The frontier between the non-problematic and the problematic is our empirical evidence for our claimed expressiveness limitation in Java 8 and motivates our suggested language extension therefor.

2.1 Non-problematic interfaces

Our first (atomic) interface component for vector spaces is the following one for (commutative) additive semigroups. These carry one binary operation, here called *plus*, specified with an explicit parameter *right*, with the corresponding parameter *left* left implicit, as is typical in object-oriented programming languages. (We can refer to this implicit parameter with the Java-keyword this.)

algebra.AdditiveSemigroup

```java
package algebra;

interface AdditiveSemigroup <T> {
  T plus (T right);
}
```

A typical example of this strucure are the natural numbers (naturals) with addition. They actually form also a (commutative) multiplicative semigroup (and more), which is our second atomic interface component for vector spaces.

`algebra.MultiplicativeSemigroup`

```java
package algebra;

interface MultiplicativeSemigroup <T> {

  T times (T right);

}
```

Our next (compound) interface component for vector spaces is the following one for additive monoids, which we obtain through single inheritance from our interface component for additive semigroups and through the addendum of a getter method for the additionally required additive neutral element.

`algebra.AdditiveMonoid`

```java
package algebra;

interface AdditiveMonoid    <T>
  extends AdditiveSemigroup <T> {

  T getZero(); // the additive neutral element

}
```

As a typical example, the naturals form (also) an additive monoid. Moreover, they form a multiplicative monoid, which is similar to its additive counterpart.

`algebra.MultiplicativeMonoid`

```java
package algebra;

interface MultiplicativeMonoid    <T>
  extends MultiplicativeSemigroup <T> {

  T getOne(); // the multiplicative neutral element

}
```

Our next, further compound interface component for vector spaces is the following one for additive groups, which we obtain through single inheritance from our interface component for additive monoids and through the addendum of a getter method for the additionally required additive inverse elements.

`algebra.AdditiveGroup`

```java
package algebra;

interface AdditiveGroup  <T>
  extends AdditiveMonoid <T> {

  T getAddInv(); // the additive inverse element

}
```

As a typical example, the integer numbers (integers) form an additive group. Moreover, they form a commutative ring, which we obtain through multiple (double) inheritance from our (two) interface components for additive groups on the one hand and for multiplicative monoids on the other hand.

`algebra.CommutativeRing`

```java
package algebra;

interface CommutativeRing      <T>
  extends AdditiveGroup        <T>,
          MultiplicativeMonoid <T> {

}
```

Our next compound interface component for vector spaces is the following one for multiplicative groups, which we obtain through single inheritance from our interface component for multiplicative monoids and through the addendum of a getter method for the additionally required multiplicative inverse elements.

`algebra.MultiplicativeGroup`

```java
package algebra;

interface MultiplicativeGroup  <T>
  extends MultiplicativeMonoid <T> {

  T getMultInv(); // the multiplicative inverse element

}
```

Our final non-problematic, further compound interface component for vector spaces is the following one for fields, which we obtain through

- multiple (double) inheritance from our (two) interface components for additive groups on the one hand and for multiplicative groups on the other hand, and
- the overriding of the getter method for the multiplicative inverse elements from the inherited multiplicative group with an exception for division by zero. (Of course, we could define a more general, algebraic exception instead of getting by with the predefined arithmetic exception.) Luckily, such an exceptional overriding is possible with Java 8 multiple inheritance.

`algebra.Field`

```java
package algebra;

interface Field               <T>
  extends AdditiveGroup       <T>,
          MultiplicativeGroup <T> {

  T getMultInv() throws ArithmeticException; // div by Zero!

}
```

A typical example of a field are the rational numbers (rationals) with addition and multiplication, of course including subtraction and division through the inherited additive and the inherited multiplicative inverse elements, respectively.

Table 1 `algebra.VectorSpace` (required version, but with compilation errors)

```java
package algebra;

interface VectorSpace   <Vector<Scalar>>
  extends AdditiveGroup <Vector<Scalar>>,
          Field                <Scalar> {

  Vector<Scalar> timesScalar (Scalar s);

}
```

```
$javac algebra/VectorSpace.java
algebra/VectorSpace.java:3: error: > expected
interface VectorSpace     <Vector<Scalar>>
                                 ^
algebra/VectorSpace.java:3: error: <identifier> expected
interface VectorSpace     <Vector<Scalar>>
                                        ^
algebra/VectorSpace.java:3: error: ';' expected
interface VectorSpace     <Vector<Scalar>>
                                          ^
algebra/VectorSpace.java:4: error: <identifier> expected
    extends AdditiveGroup <Vector<Scalar>>,
                                          ^
algebra/VectorSpace.java:4: error: ';' expected
    extends AdditiveGroup <Vector<Scalar>>,
                                           ^
algebra/VectorSpace.java:5: error: illegal start of type
            Field                <Scalar> {
                                          ^
algebra/VectorSpace.java:7: error: '(' expected
    Vector<Scalar> timesScalar(Scalar s);
    ^
algebra/VectorSpace.java:7: error: <identifier> expected
    Vector<Scalar> timesScalar(Scalar s);
                              ^
8 errors
```

2.2 Problematic interfaces

The desideratum of an interface for vector spaces in Java is the one displayed in the upper part of Table 1. It is simply the one that the mathematical (algebraic) definition requires. We obtain it through multiple (double) inheritance from our (two) interface components for an additive group of vectors on the one hand and for a field of scalars on the other hand, and through the addendum of a method for the scalar multiplication of vectors. Very unfortunately, the desired interface produces the compilation errors displayed in the lower part of Table 1, which cannot be remedied in Java 8 due to the limited expressiveness of its (not so) generic type system. Even the simpler, less desirable, because type-wise less precise interface for vector spaces displayed in the upper part of Table 2 produces an intrinsic compilation error, which is displayed in the lower part.

An even simpler interface would be a simplistic specification for vector spaces.

Hence, multiple inheritance from interfaces with different generic types is problematic in Java 8. This state of affairs has preempted—and still preempts, even in the future Java 9 [3]—the modularity of many interface specifications and thus has worsened best practices in Java for more

Table 2 `algebra.VectorSpaceAH` (*ad hoc* version, also with compilation error)

```
package algebra;

interface VectorSpaceAH <Vector, Scalar>
  extends AdditiveGroup <Vector>,
          Field         <Scalar> {

  Vector timesScalar (Scalar s);

}
```

```
$javac algebra/VectorSpaceAH.java
algebra/VectorSpaceAH.java:3: error: AdditiveGroup cannot \
be inherited with different arguments: <Vector> and <Scalar>
interface VectorSpaceAH   <Vector, Scalar>
^
1 error
```

than 20 years.

Note that industrial hard- and software systems typically have a much more complex structure of combined components — and thus require a much higher degree of modularity — than vector spaces. *A fortiori,* our reported modularity bug negatively impacts such industrial systems — much more than computer algebra systems — programmed in Java.[1] Given that modularity is a key criterion and perhaps even a necessary condition for good hard- and software systems, present Java presents no net benefit but only a net cost for the principled programmer and code reviewer of such systems. In sum, the benefit of the backwards compatibility of so-far Java-generics as implemented by *type erasure* (of different types to one and the same most general type `Object`) is more than erased by the greater cost of broken (unscalable) modularity. Big many-component systems cannot in general have modular specifications as Java-interfaces, which of course defeats the whole purpose of modular specification in Java and thus Java-code-designing software engineering. The present Java-interface system does not scale, because of its broken modularity, which in turn is due to the insufficiently generic type system of Java. Fixing Java-modularity implies breaking its backwards-compatibility, which however is a future net gain.

3 Conclusion

We have demonstrated a modularity bug in the interface system of Java 8, which is due to an important inadequacy in the Java generic type system, by producing reproducible evidence in the form of non-correctable compilation errors. The type system of Java 8 should be generalised (with type constructors, like in the programming language Scala, for example), in order to allow the modular specification with interfaces of such basic and important objects as vector spaces. Currently, modular interface specifications only up to fields are possible. Java-interfaces should also allow the stipulation of algebraic (and other constraints) on their specified methods, at least to some extent. Currently, Java-interfaces roughly correspond to mere algebraic signatures.

The modular specification of vector spaces could serve as a minimal benchmark for programming language modularity. On a different bench, poor modularity of mainstream programming languages such as Java could be a mark of the usually poor modularity of the mathematics teaching in

[1]In ordered structures, we try to construct counter-examples that are as small as possible.

computer science. There, linear algebra (vector spaces) is usually the first and only algebra subject taught in the curriculum, and its students are usually served all the axioms of its rich axiomatic structure as a single unstructured lot on a single plate in a single course, so to say. A more palatable because modular alternative is to split up the teaching of linear algebra to computer science students into its natural constituting components (field, ring, groups, monoids, semigroups), as to students of mathematics. In our opinion and own course practice, object-oriented design (evolving later into software engineering) should first be taught with the distinguished structural components of applied abstract algebra (elementary equational logic), which are sufficiently general and rich in structure to serve as instructive examples *par excellence.*

To conclude with an analogy, just as

1. the "spaghetti programming" mindset is intertwined with the "spaghetti proving" mindset via the famous Curry-Howard isomorphism and

2. the "spaghetti proving" mindset is intertwined with the "spaghetti axioms" mindset via the mentioned infamous pedagogical modularity bug,

3. the "spaghetti programming" mindset is intertwined with the "spaghetti axioms" mindset by the transitivity of intertwining mental spaghetti,

so to say.

References

[1] S. Kramer. A modularity bug in Java 8. *preprint arXiv:1701.02189*, 2017.

[2] Java$^{\mathrm{TM}}$ Platform. `https://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`.

[3] S. Mak and P. Bakker. *Java 9 Modularity Patterns and Practices for Developing Maintainable Applications.* O'Reilly Media. August 2016. Early Release: RAW & UNEDITED.

[4] D. Sannella and A. Tarlecki. *Foundations of Algebraic Specification and Formal Software Development.* Springer. DOI: 10.1007/978-3-642-17336-3.