The Journal of Systems and Software 131 (2017) 419–441

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

# XTRAITJ: Traits for the Java platform☆

Lorenzo Bettini[a,*], Ferruccio Damiani[b]

[a] Dipartimento di Statistica, Informatica, Applicazioni, Università di Firenze, Italy
[b] Dipartimento di Informatica, Università di Torino, Italy

ARTICLE INFO

*Article history:*
Received 17 June 2015
Revised 6 July 2016
Accepted 24 July 2016
Available online 26 July 2016

*Keywords:*
Java
Trait
IDE
Implementation
Eclipse

ABSTRACT

Traits were proposed as a mechanism for fine-grained code reuse to overcome many limitations of class-based inheritance. A trait is a set of methods that is independent from any class hierarchy and can be flexibly used to build other traits or classes by means of a suite of composition operations. In this paper we present the new version of XTRAITJ, a trait-based programming language that features complete compatibility and interoperability with the JAVA platform. XTRAITJ is implemented in XTEXT and XBASE, and it provides a full Eclipse IDE that supports an incremental adoption of traits in existing JAVA projects. The new version of XTRAITJ allows traits to be accessed from any JAVA project or library, even if the original XTRAITJ source code is not available, since traits can be accessed in their byte-code format. This allows developers to create XTRAITJ libraries that can be provided in their binary only format. We detail the technique we used to achieve such an implementation; this technique can be reused in other languages implemented in XTEXT for the JAVA platform. We formalize our traits by means of flattening semantics and we provide some performance benchmarks that show that the runtime overhead introduced by our traits is acceptable.



## 1. Introduction

The problems of class-based inheritance and in particular its poor support for code reuse were emphasized by Schärli et al. (2003) (see also Ducasse et al. (2006)): both single and multiple class-based inheritance are often inappropriate as a reuse mechanism. The main reason is that classes play two competing roles: a class is both a *generator of instances* and a *unit of reuse*. To accomplish the first role, a class must provide a *complete* set of basic features, and to accomplish the second role it must provide a *minimal* set of sensibly reusable features. Schärli et al. (2003) also observed that *mixins* (Hendler, 1986; Bracha and Cook, 1990; Limberghen and Mens, 1996; Flatt et al., 1998; Bettini et al., 2003a; Ancona et al., 2003), which are subclasses parametrized over their superclasses, are not necessarily appropriate for composing units of reuse. Indeed, mixin composition is linear, because it is still based on the ordinary single inheritance operator—note that the formulation of mixins given by Bracha in JIGSAW (Bracha, 1992) does not suffer from this problem, but most of the subsequent formulations of the mixin construct do.

For the above reasons, *traits* were proposed by Schärli et al. (2003) as pure units of behavior, aiming to support fine-grained reuse. The goal of traits is to provide a flexible solution to the problems of class-based inheritance with respect to code reuse, avoiding the two traditional competing roles of classes as object generators and units of code reuse mentioned above (see also Ducasse et al., 2006; Murphy-Hill et al., 2005; Cassou et al., 2009 for discussions and examples). A trait provides a set of methods that is completely independent of any class hierarchy. The rationale is that the common methods of a set of classes can be factored into a trait. The distinguishing features of traits are that:

- Traits can be composed in an arbitrary order (leading to a class or another trait); and
- The resulting composite unit has complete control over the conflicts that may arise in the composition, and must solve these conflicts explicitly.

These features make traits simpler and more flexible than *mixins*—the "trait" construct incorporated in SCALA (Odersky, 2007) is indeed a form of mixin. The original proposal of traits (Schärli et al., 2003; Ducasse et al., 2006) was given in SQUEAK/SMALLTALK, that is, in a dynamically typed setting. Various formulations of traits in a JAVA-like statically typed setting can be found in the

☆ This work has been partially supported by: project HyVar (www.hyvar-project.eu), which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644298; by ICT COST Action IC1402 ARVI (www.costarvi.eu); and by Ateneo/CSP D16D15000360005 project RunVar.

* Corresponding author. Fax: +39 055 2751525.
*E-mail addresses:* lorenzo.bettini@unifi.it (L. Bettini), ferruccio.damiani@unito.it (F. Damiani).

http://dx.doi.org/10.1016/j.jss.2016.07.035
0164-1212/

literature (see, e.g., Quitslund, 2004; Smith and Drossopoulou, 2005; Nierstrasz et al., 2006; Bono et al., 2007; Reppy and Turon, 2007; Bono et al., 2008; Liquori and Spiwack, 2008; Bettini et al., 2013d; 2013b).

In most of the above proposals, trait composition and class-based inheritance live together. In some formulations (Smith and Drossopoulou, 2005; Nierstrasz et al., 2006; Liquori and Spiwack, 2008) trait names are types, just like class names and interface names in Java—this choice limits the reuse potential of traits, since the role of *unit of reuse* and the role of *type* are competing (see, e.g., Snyder (1986) and Cook et al. (1990)). This does not happen in *pure trait-based programming languages* (Bono et al., 2007, 2008; Bettini et al., 2013d), where:

- Class-based inheritance is not present, and
- Traits are not types.

The rationale for these choices is that pure trait-based programming languages aim to maximize the opportunity for reuse: class-based inheritance is ruled out in order to prevent programmers from writing code that might be difficult to reuse, and traits are not types to rule out the interplay between the competing roles of *unit of reuse* and *type* that would restrict traits' flexibility. These design choices do not reduce the expressivity and usability of the language. In fact, even though class-based inheritance is not present, type subsumption is still supported by Java-like interfaces. Moreover, not using trait names as types in the source program does not prevent us from analyzing each trait definition in isolation from the classes and the traits that use it. This way, it is not necessary to reanalyze a trait whenever it is used by a different class.

In previous work (Bettini and Damiani, 2013; 2014) we introduced the prototype implementation of XTraitJ, a language for pure trait-based programming interoperable with the Java type system without reducing the flexibility of traits (Bettini and Damiani, 2013), and extended XTraitJ and its implementation with full support for Java generics and Java annotations (Bettini and Damiani, 2014). Such extensions allowed us to implement generic traits, classes and generic trait methods. XTraitJ programs are compiled into Java programs, which can then be compiled with a standard Java compiler.

XTraitJ is implemented with Xtext (2015), Bettini (2013). Xtext is a *language workbench* (such as MPS (Voelter, 2011) and Spoofax (Kats and Visser, 2010)): it takes as input a grammar definition and it generates a parser, an abstract syntax tree, and a full Eclipse-based IDE. Thus, by using XText we implement not only the compiler of XTraitJ, but also its Eclipse integration. Furthermore, for the syntax of our trait method bodies, we use XBase (Efftinge et al., 2012), a reusable Java-like expression language that facilitates full interoperability with the Java type system. Since XTraitJ code can coexist with Java code, single parts of a project can be refactored to use traits, without requiring a complete rewrite of the whole code-base. This allows incremental adoption of traits in existing Java projects.

In spite of the nice integration of XTraitJ with Eclipse and Java, the implementation of XTraitJ (Bettini and Damiani, 2014) still suffered from a crucial issue that would prevent the adoption of XTraitJ in a production environment: all the XTraitJ sources have to be available in a project that uses XTraitJ. This leads to the following drawbacks:

- All XTraitJ source files have to be loaded in a XTraitJ program. While this does not prevent us from type checking traits in isolation, it still forces us to compile XTraitJ sources that are provided as libraries.
- Connected to the previous issue, trait libraries cannot be provided in a binary only format.

These are in contrast with the very concept of library. In particular, library artifacts should not be recompiled when used in a program. In industry, shipping libraries with sources might not be acceptable.

*Contributions of the paper.* We present a new version of XTraitJ that addresses the above limitations. We rewrote most of the implementation of XTraitJ in order to achieve full integration of traits with the Java platform, including accessibility of traits in byte-code only format. This removes the above limitations, allows trait libraries to be provided in a binary only format, and makes XTraitJ effectively usable in production. Besides the increased usability of XTraitJ, we believe that the technique that we use to achieve full integration with Java could be easily re-used in other languages that aim at such integration, using XText/XBase. To the best of our knowledge, XTraitJ is the first DSL with non trivial linguistic features that uses such technique.[1] Since XText is the de-facto standard for implementing languages in the Eclipse eco-system, and since XBase is a powerful framework for implementing languages interoperable with Java, we think that our implementation could be useful to XText/XBase users. Furthermore, we formally specify the semantics of XTraitJ by means of a flattening translation (Ducasse et al., 2006; Nierstrasz et al., 2006). The flattening translation specifies that the semantics of a class that uses traits is equivalent to the semantics of the class obtained by inlining into the body of the class the methods provided by the traits that it uses. Finally, we evaluate XTraitJ in terms of the overhead introduced by method forwarding, which is used in the generated Java code to implement traits. The performance tests show that the overhead introduced is an acceptable tradeoff with respect to the code reuse of traits. We also evaluate the performance of the compiler of this new version of XTraitJ, which is improved with respect to the previous versions.

A preliminary version of some of the material presented in this paper appeared in Bettini and Damiani (2013, 2014). The specification of the semantics of XTraitJ (Section 3) and the technique to achieve binary level accessibility of traits (Section 4) are completely new, and both the description of the implementation (Section 5) and the evaluation of the achieved benefits (Section 6) have been revised and extended to reflect the new implementation, to provide more details, and (in Section 6.2) to illustrate performance results.

The implementation is available as an open source project and ready-to-use update site at http://xtraitj.-sf.net. We also provide pre-configured Eclipse distributions with XTraitJ installed, for several architectures. Moreover, XTraitJ programs can be processed with typical Java build tools, like Maven and Gradle, by relying on the Maven integration provided in recent versions of XText (Oehme, 2015). XTraitJ has been developed with *Test Driven Development* technologies, with almost 100% code coverage, using *Continuous Integration* systems (*Jenkins* and *Travis-CI*) and code quality tools, such as *SonarQube*.

*Organization of the paper.* Section 2 illustrates the syntax and, informally, the semantics of the XTraitJ programming language through examples. Section 3 formally specifies the semantics of XTraitJ by means of a translation that compiles traits away. Section 4 describes how XTraitJ has been fully integrated with Java and how binary only accessibility of traits has been achieved.

[1] Indeed, during the development of this new version we found a few issues with some internals of XBase, in particular, related to the implementation of generics under certain circumstances, which had not been considered. In our implementation, we solved them by customizing many parts of the XBase type system concerning generics, but we are also working on fixing these issues in the XBase code base as well—see https://bugs.eclipse.org/bugs/show_bug.cgi?id=468174.

Section 5 describes our implementation: discusses the main design choices, the strategy used to generate the JAVA code, the support for XTRAITJ code validation and the integration of XTRAITJ in Eclipse. Section 6 discusses the pros and cons of the implementation and presents some performance tests, concerning both the runtime and the compiler of XTRAITJ. Related work is discussed in Section 7. Section 8 concludes the paper by outlining possible directions for future work.

## 2. The XTRAITJ programming language

In this section we describe the main features of XTRAITJ by examples. An XTRAITJ program consists of trait declarations and class declarations. As a demonstration of the complete integration with JAVA, in the implementation of XTRAITJ we did not include interface specifications: XTRAITJ programs can seamlessly use existing JAVA interfaces.[2]

In XTRAITJ, method headers, field declarations, method declarations, and class constructors have a syntax similar to JAVA, but ignoring visibility modifiers and checked exception declarations, which we currently do not support. As shown later, all method declarations are **public** by default, if not declared as **private**. All type references can contain type arguments, and traits, classes and method headers can specify type parameters. The syntax of generic types and type parameter declarations is exactly the same as in JAVA (including bounded quantifications and wildcards).

In XTRAITJ a *trait* consists of *provided methods* (the methods defined in the trait), *required methods* (abstract methods assumed to be available in a trait or a class using the trait) and *required fields* (fields assumed to be available in a class using the trait—traits do not provide fields). A method `m` is *declared* by a trait `T` if and only if `m` is either required or provided by `T`. A field `f` is *declared* by `T` if and only if `f` is required by `T`. The declared fields and the declared methods of a trait can be directly accessed in the body of the trait's provided methods. For example,

```
trait T1 {
    int f1; // required field
    int m(); // required method
    int m1() { // provided method
        // can access required fields and required methods
        return f1 + m();
    }
}
```

Qualifying a method `m` as **private** in a trait, hides the name `m` and statically binds the method `m` to the trait. Since the name of a **private** method is bound, the actual name of a **private** method is immaterial. When two or more traits are summed, the names of **private** methods never generate a conflict. It would not make sense (and hence it is forbidden) to declare a required method as **private**. Currently, apart from **private**, there are no other qualifiers in XTRAITJ. We plan to add other qualifiers in the future.

A class in XTRAITJ can implement JAVA interfaces by using traits, and can define fields (possibly with initialization expressions) and constructors (possibly overloaded), but it cannot define methods. For example,

```
class C implements I1, I2 uses T1, T2 {
    int f1 = ...; // declared field
    C() { } // default constructor
    C(int f1) { this.f1 = f1; } // constructor with parameter
}
```

Traits can be used to compose classes and other traits by means of **uses** clauses, which implement the *trait sum* operation described in Section 2.1, and a suite of trait alteration operations (described in Section 2.2). Note that traits do not introduce any state, thus, a class has to provide all the fields required by the traits it uses. In a trait, any field that is used by a provided method must be declared and any method that is used by a provided method must be either required or provided. However, a trait can also require fields and methods that are not used by any of its provided methods. Currently, there is no method overloading in XTRAITJ. We plan to add method overloading in future releases (see also Section 8).

As stated in Section 1, XTRAITJ is a pure trait-based programming language, thus, class-based inheritance is not present, so classes play only the roles of object generators and types. Traits play only the role of units of code reuse and are not types.

XTRAITJ programs are compiled into JAVA source code, which then must be compiled using a standard JAVA compiler. Since we provide Eclipse IDE tooling, when editing XTRAITJ programs from Eclipse, using the XTRAITJ editor, the generated JAVA sources will be automatically compiled inside Eclipse, using the standard automatic building mechanism of Eclipse: saving an XTRAITJ source file will automatically trigger the generation of the corresponding JAVA code and this, in turn, will automatically trigger the compilation of such JAVA code into byte-code. This automatic building mechanism also takes care of recompiling other possible dependencies of the XTRAITJ files.

The semantics of traits can be specified in terms of the so called *flattening principle* (Ducasse et al., 2006; Nierstrasz et al., 2006) that prescribes that the semantics of a class that uses traits is equivalent to the semantics of the class obtained by inlining into the body of the class the methods provided by the traits that it uses. A *flattening semantics* provides a specification of the semantics of traits according to the flatting principle by describing a way to transform classes and traits that use other traits into equivalent formulations that do not use traits. In the rest of this section we will informally use the flattening principle in the examples to explain the features of XTRAITJ. The flattening semantics of XTRAITJ is formally described in Section 3. Note that a flattening semantics aims to provide a specification and it is not an especially effective implementation technique. In fact, our compilation into JAVA (Section 5) does not implement flattening directly and this has several advantages, described in Section 6.

The method bodies in XTRAITJ are not written in JAVA: we use XBASE for that. XBASE (Efftinge et al., 2012) is an extendable and reusable expression language developed with XTEXT, which integrates with the JAVA platform and JDT (Eclipse JAVA development tools). In particular, XBASE reuses the JAVA type system (including generics) without modifications, thus, when a language uses XBASE it can automatically and transparently access any JAVA type. XBASE removes much "syntactic noise" from JAVA (e.g., types of variable declarations can be inferred by XBASE itself) and provides advanced features (e.g., lambda expressions and extension methods). In this paper, the XBASE syntax that we will use is that fragment that is exactly the same as JAVA syntax.[3]

XTRAITJ code can seamlessly refer to any existing JAVA type. Moreover, since XTRAITJ programs are compiled into JAVA source code, we can also seamlessly use XTRAITJ classes in standard JAVA programs. This means that we can use existing frameworks like JUNIT, and write JAVA unit tests for our XTRAITJ code (we previously showed (Bettini and Damiani, 2014) that we can write JUNIT tests directly in XTRAITJ).

[2] The syntax and semantics of interfaces in XTRAITJ is exactly the same as JAVA interfaces, thus, if we added them to the implementation we would have duplicated effort without any further benefits.

[3] Note that XBASE is neither a superset nor a subset of JAVA: only a few statements and expressions are common to both languages.

### 2.1. Trait sum operation and **uses** clause

The symmetric sum operation merges two traits to form a new trait. The summed traits must be disjoint (i.e., they must not provide identically named methods) and consistent (i.e., identically named declared fields and identically named required methods must have the same type).

Traits can be used to build another trait via the **uses** clause. All the traits that are listed in a **uses** clause are added to the body of the trait declaration. For example, given these two trait declarations:

```
trait T1 {
    int f1; // required field
    int m(); // required method
    int m1() { return f1; } // provided method
}
trait T2 {
    int f2; // required field
    int m(); // required method
    int m2() { return f2; } // provided method
}
```

the following trait declaration creates a new trait by merging the two traits above

```
trait T3 uses T1, T2 { }
```

This means that `T3` is equivalent to the "flattened" trait declaration:

```
trait T3 {
    int f1; // required field
    int f2; // required field
    int m(); // required method
    int m1() { return f1; } // provided method
    int m2() { return f2; } // provided method
}
```

Note that this trait sum is well-formed: both `T1` and `T2` require the method `m` with the same signature.

All required fields and required/provided methods of any of the traits listed in the **uses** clause are visible in the body of the trait `T3`. Therefore it would be useless (and hence forbidden) to declare any of them as required in `T3`.

### 2.2. Trait alteration operations

Consider the task of developing a class `CStack` that implements the generic interface:[4]

```
public interface IStack<T> {
    boolean isEmpty();
    void push(T o);
    T pop();
}
```

In a trait-based programming language like XTRAITJ, we implement a generic trait, `TStack` that requires a field (to store the actual stack data in memory) and provides all the methods of the interface:

```
import java.util.List;

trait TStack<T> {
    List<T> collection; // required field

    boolean isEmpty() { return collection.size() == 0; }
    void push(T e) {
        // inserts at position 0
        collection.add(0, e);
    }
    T pop() {
        if (isEmpty())
            return null;
        return collection.remove(0);
    }
}
```

We then define the generic class `CStack` that implements the interface `IStack` using the above trait as follows:

```
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

class CStack<T> implements IStack<T> uses TStack<T> {
    List<T> collection = new ArrayList();

    CStack() {}

    CStack(Collection<T> c) {
        collection.addAll(c);
    }
}
```

The class defines the field `collection` required `TStack` and two constructors.

Let us now suppose that we need to develop a class implementing the following interface:

```
public interface ILifo<T> {
    boolean isNotEmpty();
    void push(T o);
    void pop();
    T top();
}
```

If we implemented the previous interface `IStack` directly in a JAVA class there would be no straightforward way to reuse the code in such class, as it would not be possible to override the `pop` method changing the return type to `void`—this would not type-check. The same problem would arise with the formulations of traits where traits are types (see, e.g., Smith and Drossopoulou, 2005; Nierstrasz et al., 2006; Liquori and Spiwack, 2008), with JAVA 8 interfaces with default methods (see the programming patterns proposed by Bono et al., 2014), and with the "trait" construct of SCALA—c.f. the discussion in Section 1. In the following we show how we can employ XTRAITJ trait alteration operations to reuse code and avoid method conflicts.

[4] Whether we are using `T` to refer to a trait name or to a type parameter should be clear from the context. We prefer to use `T` as mush as possible since it is the standard letter for trait names and, in JAVA, for type arguments.

*Method hiding.* In XTRAITJ e can write a new trait `TLifo` as follows:

```
trait TLifo<T> uses TStack<T>[hide pop] {
    void pop() {
        if (!isEmpty())
            collection.remove(0);
    }
    T top() {
        if (isEmpty())
            return null;
        return collection.get(0);
    }
    boolean isNotEmpty() { return !isEmpty(); }
}
```

This trait `uses` the trait `TStack`, but it "hides" the method `pop`: the expression `T[hide m]`, where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making the method `m` **private** to the trait. By hiding that method we avoid a conflict with the method `pop` provided in this new trait (the two methods have different return types). Note that a trait "inherits" all the field and method requirements of the used traits.

It is now straightforward to write a class `CLifo` implementing `ILifo` using the trait `TLifo`:

```
class CLifo<T> implements ILifo<T> uses TLifo<T> {
    ... // similar to CStack
}
```

*Method/field rename.* We can make further improvements to the implementation of `TLifo`. In the above implementation of `TLifo`'s `pop` method, we do not reuse the implementation of `TStack`'s `pop`.[5] Another thing that does not look right is that `isNotEmpty` does not make sense in `TLifo` (the only reason we put it there is because we need it to declare a class implementing the interface `ILifo`). We will fix these issues in the following.

We can reuse the implementation of `TStack`'s `pop` by using the **rename** operation: The expression `T[rename m1 to m2]`, where the method `m1` must be declared by `T` and the method `m2` must not be declared by `T`, denotes the trait obtained from `T` by replacing all the references to `m1` with (possibly implicit) receiver **this** by references to `m2` and by changing the declaration of `m1` into a declaration of `m2`. Thus, we write this alternative version:

```
trait TLifo<T> uses TStack<T>[rename pop to old_pop] {
    void pop() {
        old_pop(); // ignore returned value
    }
    // ...as above
}
```

With this implementation of `TLifo` we reuse the implementation of `TStack`'s provided method `pop`, after renaming it to `old_pop`. Note that, since **rename** acts both on method declarations and on method references, possible recursive occurrences in the original `pop` implementation will be renamed too.

When we declare the class implementing `ILifo` using the trait `TLifo` we can even **hide** the renamed version of `pop` as follows:

```
class CLifo<T> implements ILifo<T> uses TLifo<T>[hide old_pop] { ... }
```

[5] While this might not be a real problem in this simple example, in more complex scenarios reusing entire methods would really improve code maintainability.

The rename operation can be applied also to fields; the additional keyword **field** is used to distinguish it from the corresponding operation on methods. This distinction provides better tooling support and allows users to be more specific about their intentions.

To further demonstrate XTRAITJ programming features, we now factor the pattern of negating a boolean method into a trait:

```
trait TNegate {
    boolean op(); // required
    boolean notOp() { return !op(); }
}
```

This trait requires a boolean method `op` and provides the method `notOp` that simply returns the negation of the result of `op`.

We can now remove the implementation of `isNotEmpty` from `TLifo`, and in the class `CLifo` we use the trait `TNegate` after renaming `op` to `isEmpty` and `notOp` to `isNotEmpty`:

```
trait TLifo<T> uses TStack<T>[rename pop to old_pop] {
    void pop() {
        old_pop(); // ignore returned value
    }
    T top() {
        if (isEmpty())
            return null;
        return collection.get(0);
    }
}

class CLifo<T> implements ILifo<T>
    uses TLifo<T>[hide old_pop], TNegate[rename op to isEmpty, rename notOp to isNotEmpty] {
    ...
}
```

This example shows the flexible compositional nature of XTRAITJ operations:

- The method `op` is required by `TNegate`; after renaming, the required method `isEmpty` is provided by the other trait in the **uses** clause, `TLifo`;
- `TNegate` provides the method `notOp`, but since we rename it to `isNotEmpty`, the class is able to implement all the `ILifo`'s methods;
- Since **rename** acts both on method declarations and on method references, `isNotEmpty` (i.e., the original `notOp`) is effectively implemented in terms of `isEmpty`.

*Method alias.* The expression `T[alias m1 as m2]`, where the method `m1` must be provided by `T` and the method `m2` must not be declared by `T` (i.e., neither required nor provided), denotes the trait obtained from `T` by adding a new provided method that is a copy of `m1` with name `m2`. When the aliased method `m1` is recursive, its recursive invocation within `m2` refers to the original method. This is the main difference between **alias** and **rename**.

*Method restrict.* The expression `T[restrict m]`, where the method `m` must be provided by `T`, denotes the trait obtained from `T` by making `m` a required method.

*Method/field redirect.* The expression The expression `T[redirect m1 to m2]`, where both the distinct methods `m1` and `m2` must be declared by `T`, denotes the trait obtained from `T` by removing the declaration of `m1` and replacing all the references to `m1` with (possibly implicit) receiver **this** by references to `m2`. Similarly, for **redirect field**.

*Method override.* XTRAITJ does not provide an operation to implement method override, since the same result can be achieved using **restrict**. For instance,

**Table 1**
XTRAITJ syntax.

| | | | |
|---|---|---|---|
| GT | ::= | GI \| GC \| X \| **int** \| ⋯ | *types* |
| GI | ::= | I[<$\overline{\text{GA}}$>] | *interface reference* |
| GC | ::= | C[<$\overline{\text{GA}}$>] | *class reference* |
| GA | ::= | ⋯ | *type arguments* |
| GP | ::= | ⋯ | *type parameters* |
| RT | ::= | GT \| **void** | *method return type* |
| TD | ::= | **trait** T[<$\overline{\text{GP}}$>] [**uses** $\overline{\text{TA}}$] { $\overline{\text{D}}$ } | *trait* |
| TA | ::= | T[<$\overline{\text{GA}}$>] [[$\overline{\text{ao}}$]] | *trait alteration expression* |
| ao | ::= | **alias** m **as** m \| **restrict** m \| **hide** m | *trait alteration operation* |
| | | \| **rename** m **to** m \| **redirect** m **to** m | |
| | | \| **rename field** f **to** f \| **redirect field** f **to** f | |
| D | ::= | F; \| H; \| Q | *trait member declaration* |
| F | ::= | GT f | *field* |
| H | ::= | [<$\overline{\text{GP}}$>] RT m($\overline{\text{GT}}$ $\overline{\text{x}}$) | *method header* |
| Q | ::= | [**private**] M | *possibly qualified method* |
| M | ::= | H MB | *non-qualified method* |
| MB | ::= | ⋯ | *method body* |
| CD | ::= | **class** C[<$\overline{\text{GP}}$>] [**implements** $\overline{\text{GI}}$] [**uses** $\overline{\text{TA}}$] { $\overline{\text{FI}}$; $\overline{\text{K}}$ } | *class* |
| FI | ::= | F[ = ⋯] | *field initialization* |
| K | ::= | ⋯ | *constructor* |

```
trait T1 {
  int m() { ... }
}

trait T2 uses T1[restrict m] {
  int m() { ... }
}
```

This can be seen as T2 overriding m. Indeed, this also implies standard JAVA semantics for method invocation with dynamic binding: if a method in T1 invokes m, then, in the flattened code, the version of the method defined in T2 will be invoked. The original implementation can still be accessed in the new version of the method either by aliasing or by renaming. The programmer will decide which operation to use, taking into consideration how **alias** and **rename** act differently on possible recursive invocations.

## 3. Flattening semantics

This section formally specifies the semantics of XTRAITJ by means of a flattening translation (Ducasse et al., 2006; Nierstrasz et al., 2006). The flattening translation specifies that the semantics of a class that uses traits is equivalent to the semantics of the class obtained by inlining into the body of the class the methods provided by the traits that it uses. It is described by a function that looks up the named traits listed in the **uses** clause and compiles traits away by evaluating all trait composition operations.

The syntax of XTRAITJ classes and traits is given in Table 1, where the big square brackets '[' and ']' are part of the Extended BNF notation, and the overline notation for (possibly empty) sequences is borrowed from the definition of FEATHERWEIGHT JAVA (Igarashi et al., 2001). The empty sequence is denoted by •. Since the syntax and semantics of interfaces in XTRAITJ is exactly the same as in JAVA, the syntax of interfaces is not included (cf. the discussion at the beginning of Section 2). The syntax of type arguments (denoted by GA) and parameter declarations (denoted by GP), which is exactly the same as in JAVA (including bounded quantifications and wildcards), and the syntax of method bodies (denoted by MB) and class constructors (denoted by K) as well as the syntax of expressions used to initialize fields, which is the same as in XBASE, are omitted.

Method headers are equated modulo renaming of declared parameters and type parameters, e.g., we consider <X> **int** m(List<X> x) and <X′> **int** m(List<X′> x′) as equal.

Following FEATHERWEIGHT JAVA (Igarashi et al., 2001), we assume that sequences $\overline{\text{D}}$ of field declarations and method declarations do not contain two (or more) declarations for the same member (field or method), and we use a set-based notation for operators over sequences of declarations. For instance, M = <$\overline{\text{GP}}$> RT m($\overline{\text{GT}}$ $\overline{\text{x}}$ MB) $\in \overline{\text{D}}$ means that the method definition M occurs in $\overline{\text{D}}$. Let ***names***($\overline{\text{D}}$) denote the sequence of the names of the members declared in $\overline{\text{D}}$, let ***keep***($\overline{\text{D}}$, $\overline{\text{n}}$) denote the subsequence of $\overline{\text{D}}$ that contains only the declarations for the names $\overline{\text{n}}$, and let ***drop***($\overline{\text{D}}$, $\overline{\text{n}}$) denote the subsequence of $\overline{\text{D}}$ obtained by removing the declarations for the names $\overline{\text{n}}$. In the union, in the intersection and in the difference of sequences, denoted by $\overline{\text{D}} \cup \overline{\text{D}}'$, $\overline{\text{D}} \cap \overline{\text{D}}'$ and $\overline{\text{D}} - \overline{\text{D}}'$, respectively, it is assumed that if n ∈ ***names***($\overline{\text{D}}$) and n ∈ ***names***($\overline{\text{D}}'$) then ***keep***($\overline{\text{D}}$, n) = ***keep***($\overline{\text{D}}'$, n). In the disjoint union of sequences, denoted by $\overline{\text{D}} \uplus \overline{\text{D}}'$, it is assumed that ***names***($\overline{\text{D}}$) ∩ ***names***($\overline{\text{D}}'$) = •.

The flattening function, $[\![\cdot]\!]$, is given in Table 2. It transforms an XTRAITJ class declaration by inlining into the body of the class the methods provided by the traits that the class uses. Flattening transforms a sequence of trait alteration expressions that occurs in a **uses** clause (i.e., that describes how some traits are used by a class or trait) into a sequence of trait member declarations (the required fields, the required methods, and the provided methods). Note that in Table 2 (and in Table 3) there are no explicit clauses for dealing with classes, traits, interfaces and methods without type parameters, since they are implicitly handled by using the empty list of type parameters/argument—i.e., by considering C<•>, T<•>, I<•> and <•>H denoting C, T, I and H, respectively.

**Table 2**
Flattening XTRAITJ to JAVA (the auxiliary functions *fields*, *sum*, *replaceTypeParameters*, *renameMethod*, *headers* and *renameField* are defined in Table 3).

$$[\![\textbf{class C<}\overline{\texttt{GP}}\textbf{> implements } \overline{\texttt{GI}} \textbf{ uses } \overline{\texttt{TA}} \ \{\ \overline{\texttt{FI;}}\ \ \overline{\texttt{K}}\ \}\!] \quad = \quad \textbf{class C<}\overline{\texttt{GP}}\textbf{> implements } \overline{\texttt{GI}}\ \{\ \overline{\texttt{FI;}}\ \ \overline{\texttt{K}}\ \ \overline{\texttt{Q}}\ \}$$
$$\text{if} \quad [\![\overline{\texttt{TA}}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{Q}}$$
$$\overline{\texttt{F;}} \subseteq \mathit{fields}(\overline{\texttt{FI;}})$$

$$[\![\texttt{TA}_1, \cdots, \texttt{TA}_n]\!] \quad = \quad \mathit{sum}(\cdots \mathit{sum}(\mathit{sum}(\texttt{TA}_1, \texttt{TA}_2), \texttt{TA}_3) \cdots \texttt{TA}_n), \quad \text{if } n = 0 \text{ or } n \geq 2$$

$$[\![\texttt{T<}\overline{\texttt{GA}}\texttt{>}]\!] \quad = \quad \mathit{sum}(\overline{\texttt{D}}', \overline{\texttt{D}}'')$$
$$\text{if} \quad \textbf{trait T<}\overline{\texttt{GP}}\textbf{>}\ \ \textbf{uses } \overline{\texttt{TA}}\ \{\ \overline{\texttt{D}}\ \}$$
$$\overline{\texttt{D}}' = [\![\mathit{replaceTypeParameters}(\overline{\texttt{TA}}, \overline{\texttt{GP}}, \overline{\texttt{GA}})]\!]$$
$$\overline{\texttt{D}}'' = \mathit{replaceTypeParameters}(\overline{\texttt{D}}, \overline{\texttt{GP}}, \overline{\texttt{GA}})$$

$$[\![\texttt{TA}[\textbf{alias } \texttt{m} \textbf{ as } \texttt{m}']]\!] \quad = \quad \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}\ \ \texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}'(\overline{\texttt{GT x}})\ \texttt{MB}$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}(\overline{\texttt{GT x}})\ \texttt{MB} \in \overline{\texttt{Q}}$$
$$\texttt{m}' \notin \textbf{\textit{names}}(\overline{\texttt{H;}}\ \ \overline{\texttt{Q}})$$

$$[\![\texttt{TA}[\textbf{restrict } \texttt{m}]]\!] \quad = \quad \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}(\overline{\texttt{GT x}})\texttt{;}\ \ \textbf{\textit{drop}}(\overline{\texttt{Q}}, \texttt{m})$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}(\overline{\texttt{GT x}})\ \texttt{MB} \in \overline{\texttt{Q}}$$

$$[\![\texttt{TA}[\textbf{hide } \texttt{m}]]\!] \quad = \quad \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}'\ \ \texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}(\overline{\texttt{GT x}})\ \texttt{MB}\ \ \overline{\texttt{Q}}''$$
$$\texttt{m}' \text{ is fresh}$$
$$\overline{\texttt{Q}} = \mathit{renameMethod}(\overline{\texttt{Q}}'\ \ \textbf{private } \texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}(\overline{\texttt{GT x}})\ \texttt{MB}\ \ \overline{\texttt{Q}}'', \texttt{m}, \texttt{m}')$$

$$[\![\texttt{TA}[\textbf{rename } \texttt{m} \textbf{ to } \texttt{m}']]\!] \quad = \quad \overline{\texttt{F;}}\ \ \mathit{renameMethod}(\overline{\texttt{H;}}\ \ \overline{\texttt{Q}}, \texttt{m}, \texttt{m}')$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\texttt{m} \in \textbf{\textit{names}}(\overline{\texttt{H;}}\ \ \overline{\texttt{Q}})$$
$$\texttt{m}' \notin \textbf{\textit{names}}(\overline{\texttt{H;}}\ \ \overline{\texttt{Q}})$$

$$[\![\texttt{TA}[\textbf{redirect } \texttt{m} \textbf{ to } \texttt{m}']]\!] \quad = \quad \overline{\texttt{F;}}\ \ \mathit{renameMethod}(\textbf{\textit{drop}}(\overline{\texttt{H;}}\ \ \overline{\texttt{Q}}, \texttt{m}), \texttt{m}, \texttt{m}')$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\texttt{m} \neq \texttt{m}'$$
$$\texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}(\overline{\texttt{GT x}}),\ \texttt{<}\overline{\texttt{GP}}\texttt{>}\ \texttt{RT m}'(\overline{\texttt{GT x}}) \in \overline{\texttt{H;}} \uplus \mathit{headers}(\overline{\texttt{Q}})$$

$$[\![\texttt{TA}[\textbf{rename field } \texttt{f} \textbf{ to } \texttt{f}']]\!] \quad = \quad \mathit{renameField}(\overline{\texttt{F;}}, \texttt{f}, \texttt{f}')\ \ \overline{\texttt{H;}}\ \ \mathit{renameField}(\overline{\texttt{Q}}, \texttt{f}, \texttt{f}')$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\texttt{f} \in \textbf{\textit{names}}(\overline{\texttt{F}})$$
$$\texttt{f}' \notin \textbf{\textit{names}}(\overline{\texttt{F}})$$

$$[\![\texttt{TA}[\textbf{redirect field } \texttt{f} \textbf{ to } \texttt{f}']]\!] \quad = \quad \textbf{\textit{drop}}(\overline{\texttt{F;}}, \texttt{f})\ \ \overline{\texttt{H;}}\ \ \mathit{renameField}(\overline{\texttt{Q}}, \texttt{f}, \texttt{f}')$$
$$\text{if} \quad [\![\texttt{TA}]\!] = \overline{\texttt{F;}}\ \ \overline{\texttt{H;}}\ \ \overline{\texttt{Q}}$$
$$\texttt{f} \neq \texttt{f}'$$
$$\texttt{GT f}, \texttt{GT f}' \in \overline{\texttt{F}}$$

The clauses in Table 2 assume, without loss of generality, that the names of the **private** methods occurring in the XTRAITJ program to be flattened satisfy the Barendregt convention (Barendregt, 1984) (i.e., they are distinctly named),[6] and that the body of each class declaration and trait declaration do not contain multiple declarations for the same field or method.[7] Flattening fails (meaning that the source program is ill formed) whenever a non applicable operation is encountered, e.g.:

1. A trait alteration operation that tries to alias a non-existent method;
2. A trait alteration operation that tries to alias a method to a name that is already declared as provided;

[6] The Barendregt convention can be enforced by a straightforward preprocessing step.

[7] This condition can be checked by a straightforward preprocessing step.

**Table 3**
Auxiliary functions used by flattening translation (given in Table 2).

| | | |
|---|---|---|
| $fields(\overline{\mathtt{F}[\ =\cdots]\,;})$ | $=$ | $\overline{\mathtt{F};}$ |
| $sum(\overline{\mathtt{F}};'\ \overline{\mathtt{H}};'\ \overline{\mathtt{Q}}',\ \overline{\mathtt{F}};''\ \overline{\mathtt{H}};''\ \overline{\mathtt{Q}}'')$ | $=$ | $\overline{\mathtt{F};}\ \overline{\mathtt{H};}\ \overline{\mathtt{Q}}$ if $\overline{\mathtt{Q}} = \overline{\mathtt{Q}}' \uplus \overline{\mathtt{Q}}''$, $\overline{\mathtt{H};} = (\overline{\mathtt{H}};' \cup \overline{\mathtt{H}};'') - headers(\overline{\mathtt{Q}})$, $\overline{\mathtt{F};} = \overline{\mathtt{F}};' \cup \overline{\mathtt{F}};''$ |
| $replaceTypeParameters(\cdots, \overline{\mathtt{GP}}, \overline{\mathtt{GA}})$ | returns | the variant of its first argument $\cdots$ obtained by replacing the type parameter names declared in $\overline{\mathtt{GP}}$ by the corresponding type arguments in $\overline{\mathtt{GA}}$ |
| $renameMethod(\mathtt{D}_1 \cdots \mathtt{D}_n, \mathtt{m}, \mathtt{m}')$ | $=$ | $renameMethod(\mathtt{D}_1, \mathtt{m}, \mathtt{m}') \cdots renameMethod(\mathtt{D}_n, \mathtt{m}, \mathtt{m}')$ if $n = 0$ or $n \geq 2$ |
| $renameMethod(\mathtt{H};, \mathtt{m}, \mathtt{m}')$ | $=$ | $renameMethod(\mathtt{H}, \mathtt{m}, \mathtt{m}')$; |
| $renameMethod(\mathtt{<}\overline{\mathtt{GP}}\mathtt{>}\ \mathtt{RT}\ \mathtt{m}''(\overline{\mathtt{GT}}\ \overline{\mathtt{x}}), \mathtt{m}, \mathtt{m}')$ | $=$ | $\mathtt{<}\overline{\mathtt{GP}}\mathtt{>}\ \mathtt{RT}\ \mathtt{m}'''(\overline{\mathtt{GT}}\ \overline{\mathtt{x}})$ where $\mathtt{m}''' = \mathtt{m}'$ if $\mathtt{m}'' = \mathtt{m}$, and $\mathtt{m}''' = \mathtt{m}''$ otherwise |
| $renameMethod(\mathtt{H\ MB}, \mathtt{m}, \mathtt{m}')$ | $=$ | $renameMethod(\mathtt{H}, \mathtt{m}, \mathtt{m}')\ renameMethod(\mathtt{MB}, \mathtt{m}, \mathtt{m}')$ |
| $renameMethod(\textbf{private}\ \mathtt{H\ MB}, \mathtt{m}, \mathtt{m}')$ | $=$ | **private** $renameMethod(\mathtt{H}, \mathtt{m}, \mathtt{m}')\ renameMethod(\mathtt{MB}, \mathtt{m}, \mathtt{m}')$ |
| $renameMethod(\mathtt{MB}, \mathtt{m}, \mathtt{m}')$ | returns | the variant of MB obtained by replacing all the references to m with (possibly implicit) receiver **this** by references to $\mathtt{m}'$ |
| $headers(\mathtt{Q}_1 \cdots \mathtt{Q}_n)$ | $=$ | $headers(\mathtt{Q}_1) \cdots headers(\mathtt{Q}_n)$ if $n = 0$ or $n \geq 2$ |
| $headers(\mathtt{H\ MB})$ | $=$ | $\mathtt{H};$ |
| $headers(\textbf{private}\ \mathtt{H\ MB})$ | $=$ | $\bullet$ |
| $renameField(\mathtt{D}_1 \cdots \mathtt{D}_n, \mathtt{f}, \mathtt{f}')$ | $=$ | $renameField(\mathtt{D}_1, \mathtt{f}, \mathtt{f}') \cdots renameField(\mathtt{D}_n, \mathtt{f}, \mathtt{f}')$ if $n = 0$ or $n \geq 2$ |
| $renameField(\mathtt{GT\ f}'';, \mathtt{f}, \mathtt{f}')$ | $=$ | $\mathtt{GT\ f}''';$ where $\mathtt{f}''' = \mathtt{f}'$ if $\mathtt{f}'' = \mathtt{f}$, and $\mathtt{f}''' = \mathtt{f}''$ otherwise |
| $renameField(\mathtt{H\ MB}, \mathtt{f}, \mathtt{f}')$ | $=$ | $\mathtt{H}\ renameField(\mathtt{MB}, \mathtt{f}, \mathtt{f}')$ |
| $renameField(\textbf{private}\ \mathtt{H\ MB}, \mathtt{f}, \mathtt{f}')$ | $=$ | **private** $\mathtt{H}\ renameField(\mathtt{MB}, \mathtt{f}, \mathtt{f}')$ |
| $renameField(\mathtt{MB}, \mathtt{f}, \mathtt{f}')$ | returns | the variant of MB obtained by replacing all the references to f with (possibly implicit) receiver **this** by references to $\mathtt{f}'$ |

3. A trait alteration operation that tries to alias a method to a name that is already declared as required.

The flattening translation specified by the clauses in Table 2 also models, by means of the parts highlighted in gray, the effect of the trait operations on required field and required method declarations—this makes the specification more useful to both XTRAITJ users and developers. If we ignore the parts highlighted in gray, we obtain a variant of the flattening translation where the flattening of a trait expression just returns the set of method definitions provided by the trait (as is done by Nierstrasz et al. (2006) and Bettini et al. (2013d)). Such a variant of the flattening translation models less errors. For instance, the third error listed above is not modeled. Both versions of the flattening translation will succeed and produce the same result whenever applied to a well-typed XTRAITJ program (the XTRAITJ type system is briefly discussed in Section 5.3).

The clauses in Tables 2 and 3 should be mostly self-explanatory—note that the resulting flattened code can be straightforwardly transformed into JAVA code by translating the bodies of methods from XBASE to JAVA.

## 4. JAVA integration

In this section we describe how XTRAITJ is completely integrated with JAVA. In particular, we describe the technique we used to achieve such complete integration. This implementation technique can be re-used to achieve the same level of integration in other DSLs implemented with XTEXT/XBASE. To the best of our knowledge, XTRAITJ is the first DSL with non trivial linguistic features that use this technique. By "non trivial" we mean that for each element of our language, e.g., traits, methods and alteration operations, there exist several generated JAVA artifacts. Instead, most DSLs based on XBASE have a one-to-one mapping in the model inferrer (described in Section 4.1).

In order to describe this technique, it is not necessary to go into the details of our translation (which will be described in

Section 5.2). We first briefly describe the XBASE model inferrer (Section 4.1), which is the main mechanism to use the XBASE type system. Then, after showing the features of XTRAITJ with respect to its integration with JAVA (Section 4.2), we describe the main steps to achieve it, using the XBASE framework (Section 4.3).

### 4.1. The XBASE model inferrer

In this section we sketch the main steps to use XBASE in a language, so that the reader can understand how we implemented XTRAITJ. A language that uses XBASE will inherit, besides the JAVA-like expression syntax, also all its language infrastructure components, like its type system implementation and the JAVA code generator. The XBASE type system is interoperable with the JAVA type system: the language will be able to seamlessly access all the JAVA types, i.e., any existing JAVA library.

In order to reuse the XBASE JAVA type system in XTRAITJ, we have to map the concepts of our language (e.g., traits, required fields, required and provide methods, etc.) into the JAVA model elements of XBASE (e.g., classes, fields, methods, etc.). This mapping is performed by implementing a *model inferrer*. The XBASE expressions used in XTRAITJ, i.e., the body of trait provided methods, will then have to be associated with the corresponding mapped JAVA model method, which becomes the expression's logical container. Such mapping will let XBASE automatically implement type checking for the expressions (XBASE will also be able to define the proper scope for `this`). This means that the whole type system of XBASE, which corresponds to the type system of JAVA, will be automatically part of XTRAITJ.[8]

With this mapping implemented by the model inferrer, XBASE will also be able to automatically generate JAVA code starting from the mapped JAVA model. Thus, the translation of XTRAITJ to JAVA sketched in Section 5.2 is implied by our implementation of the model inferrer for XTRAITJ.

### 4.2. XTRAITJ access to the JAVA type system

The main novelty of the new implementation of XTRAITJ we present in this paper is the way traits are referred to from within an XTRAITJ program. In the previous versions (Bettini and Damiani, 2013, 2014), when we specified that a XTRAITJ trait (or a class) uses a trait, we actually referred to a trait element of the XTRAITJ AST model. In the new version, we refer to the corresponding generated JAVA interface (as described in Section 5.2.1, a trait will correspond to a generated JAVA interface and a generated JAVA class).

This small difference has a huge consequence: we can access traits in any JAVA project or library, even if the original XTRAITJ source is not available. Moreover, we can access traits even in their byte-code format. This way, XTRAITJ libraries can be provided in binary-only format.

This important feature has been achieved by annotating the generated JAVA code with JAVA annotations (Sun Microsystems, Inc., 2007), with *runtime retention policy*. Such annotations are kept in the byte-code. When a trait is referred to in an XTRAITJ program (recall that this actually corresponds to a reference to a JAVA class or interface), our compiler checks that the corresponding JAVA type is annotated with our annotations. This allows us to check that such references are valid trait references. This is possible since XBASE provides an API to access annotation information on JAVA code, even in byte-code format. Also generated JAVA methods are annotated with information about the corresponding methods in the original XTRAITJ source code. For example, in the generated JAVA interface we annotate methods depending on whether they correspond to required or provided methods. We also annotate JAVA interface methods with information to say whether they correspond to renamed and aliased methods. In the presence of renamed and aliased methods, the annotations also contain information about the original method names. This allows us to perform later validation (as described in Section 5.3).

Referring to JAVA types instead of XTRAITJ trait model elements introduces some additional programming tasks in the implementation, especially concerning validation: in the previous implementations, given a trait reference, we used to have direct access to all its elements (i.e., fields, required methods, defined methods). In the current implementation, we have access to the referred JAVA interface methods, and we have to use such methods' annotations to understand which original elements they correspond to. We need such information to check for conflicts, for example.

However, this new technique has an important advantage: in the previous versions we needed to walk up the `uses` relation graph to collect all the visible fields and methods of a trait, and we needed to do that for all the `uses` relations in a program. In the current version, since we are actually referring to JAVA types, we can reuse the XBASE type system infrastructure to collect such elements, since they will correspond to methods in a standard JAVA interface inheritance hierarchy. XBASE already caches such information, so this is also more efficient, besides requiring less programming. Of course, we had to customize a few parts of the XBASE mechanisms for inspecting a JAVA type hierarchy, since we have to treat modifications introduced by the alteration operations. We implement such a customization by using the information we put in the generated JAVA code in the form of the JAVA annotations we described above.

Since we will not have to inspect the `uses` relations repeatedly during the validation, we have a higher level of compositionality, not to mention that we achieve type checking of traits in complete isolation. This improved the performance of the compiler, as shown in Section 6.2.

Note that, if one writes standard JAVA interfaces and classes, using our annotations, XTRAITJ programs could refer to such manually written JAVA code as if they were traits, and our validator would handle them accordingly. Of course, this is not the intended use case, but it is important to stress that trait references in this new versions of XTRAITJ are really completely integrated with JAVA types, since our compiler uses such JAVA annotations only for validating traits.

We can verify the above-mentioned features by performing this experiment (Note that the XTRAITJ compiler generates JAVA files into the source directory `xtraitj-gen`):

- We create an XTRAITJ project with two XTRAITJ files: the first one is to be considered a library trait, `MyLibraryTrait`, and the other one is the client of this trait. Using an `import`, the client can refer to the library trait (which is available in source format), Fig. 1.
- Now we manually copy the generated JAVA source files corresponding to `MyLibraryTrait` (as we will show in Section 5.2.1, for each trait we generate a JAVA interface and a JAVA class), and remove the original XTRAITJ sources. The client trait can still refer to `MyLibraryTrait`, even if the original XTRAITJ sources are not there anymore, Fig. 2.
- Finally, we manually copy the generated `.class` files corresponding to `MyLibraryTrait` into a different folder of the project and set it as part of the classpath, and then we remove the generated JAVA source files corresponding to `MyLibraryTrait`. The client can still refer `MyLibraryTrait`, in binary-only format, Fig. 3.

[8] There are frameworks to easily implement a type system for XTEXT languages, (Bettini, 2015), but when the target platform is JAVA, the best choice is to use XBASE, for the above mentioned reasons.

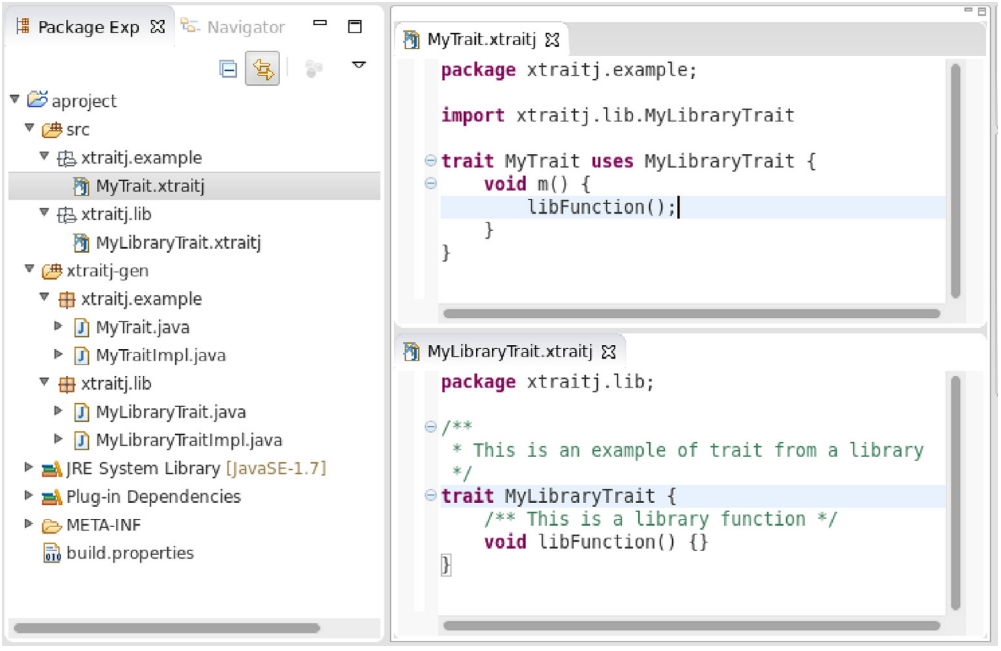


**Fig. 1.** Accessing a trait (in source format).

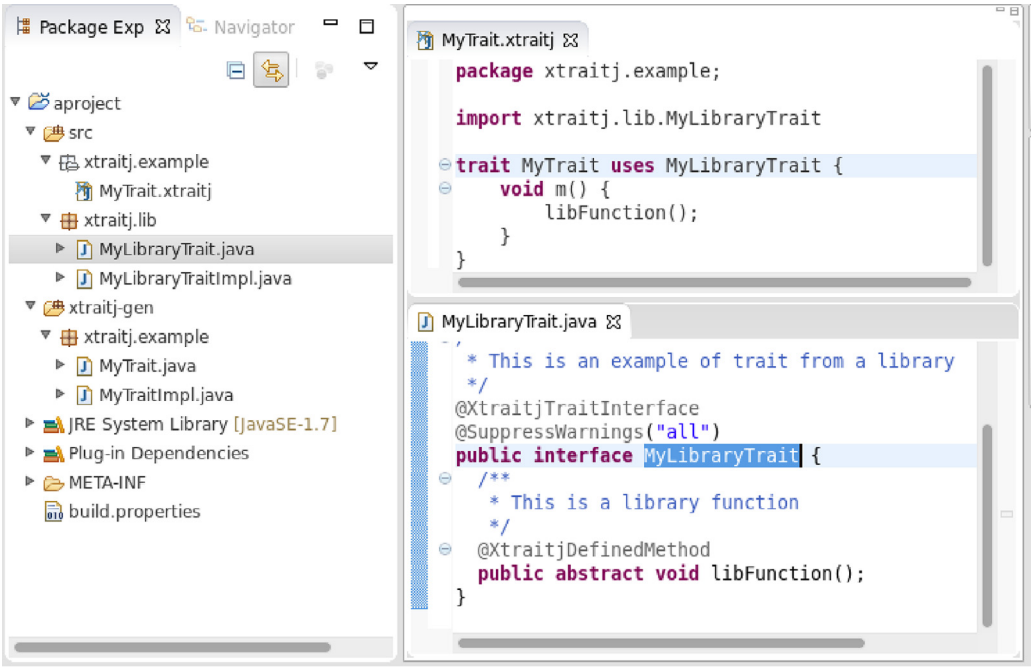


**Fig. 2.** Accessing a trait (in Java source format).

### 4.3. Achieving Java integration

As we anticipated at the beginning of the section, this technique can be reused in other DSLs for achieving the complete level of Java integration, since it is not particularly coupled with our implementation. The main steps can be summarized as follows:

- Annotate the generated Java code with all the information necessary to recover features of the original model elements of your DSL (in our case, traits, fields, required and declared methods, modifications introduced by alteration operations);
- References to other DSL elements should be implemented as references to the corresponding Java types. If, as in our case, a DSL element has more than one Java element associated with it, you should choose one that represents its public "interface", not its implementation (in our case, it is the corresponding Java interface). Note that it is not mandatory that such corresponding Java type is effectively generated and compiled: Xbase is able to refer to Java types inferred in the model inferrer that have not been yet generated into Java files.
- Check that the referred Java types (and possibly the methods therein) are annotated with the information about the original DSL elements;

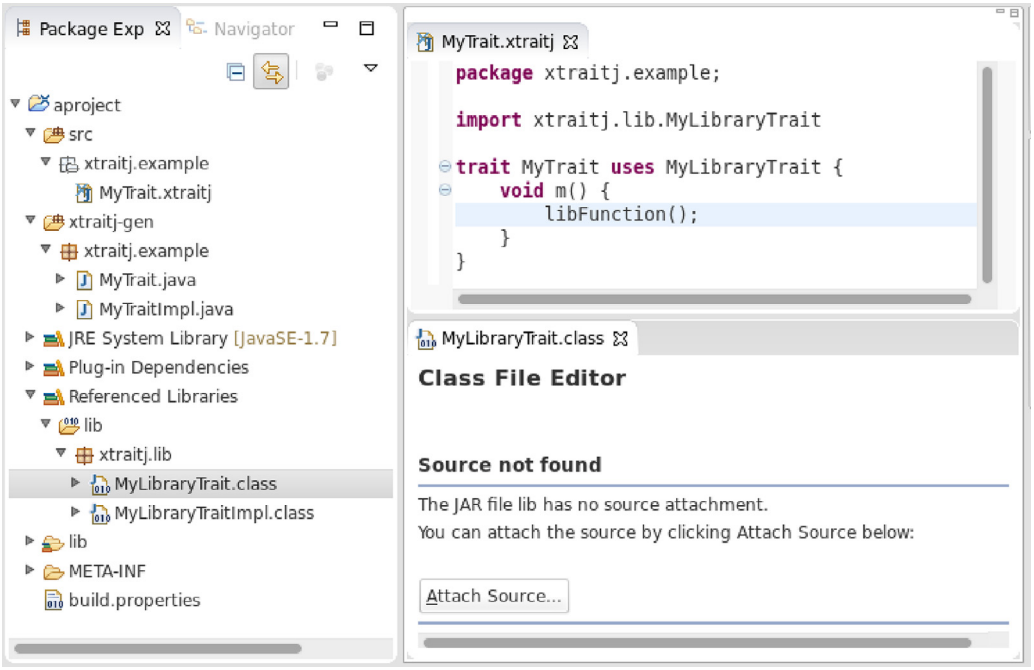


**Fig. 3.** Accessing a trait (in binary format).

- Use such Java annotations for retrieving the information about the original DSL elements;
- Exploit Xbase mechanisms for collecting all the elements of your DSL relations that should be mapped to Java inheritance relations in the generated Java code.

This complete Java integration also brings many benefits in the IDE tooling, as described in Section 5.4.

## 5. Implementation

In this section we describe the main parts of the implementation of Xtraitj, including some design choices and the integration with the Eclipse IDE.

### 5.1. Design choices

Xtraitj traits have been designed with the goals of being compliant to the characteristics of the original formulation of traits (Ducasse et al., 2006), namely complete conflict control on composition of traits, and their lightweight mechanisms with an intuitive semantics. Our main goal is the complete integration with the Java platform. Java is a mainstream language, with a huge ecosystem of libraries and many tools. We believe that it is crucial to be completely compatible and interoperable with the Java platform: this allows us to seamlessly reuse all existing Java libraries and frameworks and to target any JVM compatible platform (including Android). In order to achieve this, we chose Xtext (2015) (see also (Bettini, 2013)) and Xbase (Efftinge et al., 2012).

For similar reasons, we adopted full Java generics. Java Generics are known to have several limitations, especially when compared to C++ templates (we refer to Ghosh (2004) and Batov's work (Batov, 2004) for a broader comparison between Java generics and C++ templates). However, Java generics have already been accepted by a huge community, and we want to target full Java compatibility. Similarly, in Xtraitj we introduced Java annotations, so that we are able to use all the Java frameworks based on Java annotations. A clear example is the possibility to write JUnit tests in Xtraitj, as shown by Bettini and Damiani (2014).

We translate Xtraitj programs into Java source code, which will then be compiled with the standard Java compiler, so there are no backward binary compatibility issues with the resulting output. Our implementation allows for incremental adoption of traits in an existing Java project: single parts of the project can be refactored to use our traits, without requiring a complete rewrite of the whole existing code-base. Actually, it is not even mandatory to use traits everywhere, since Xtraitj code seamlessly coexists with Java code.

Although an IDE is not a strict requirement to develop applications, it helps programmers a lot with features like syntax aware editor, compiler and debugger integration, build automation and code completion, just to mention a few. Indeed, in an agile (Martin, 2003) and test-driven context (Beck, 2003) the features of an IDE like Eclipse become essential and dramatically increase productivity. In this respect, Xtext provides a complete solution for the development of new languages, since it also provides the integration of the language in Eclipse with all the editing and programming tooling. In particular, by using Xbase, our language also supports debugging in Eclipse: one can debug both the generated Java code and the original Xtraitj code (see Section 5.4).

### 5.2. Translation to Java

In this section we sketch the main steps we used to implement Xtraitj. As described in Section 4.1, in order to reuse the Xbase Java type system in Xtraitj, we have to map the concepts of our language into the Java model elements of Xbase by implementing a *model inferrer*. Since Xbase is able to automatically generate Java code starting from the mapped Java model implemented by the model inferrer, the translation of Xtraitj to Java sketched in the following is implied by our implementation of the model inferrer for Xtraitj. Note however that Xbase only deals with typing and validation of expressions: language features like traits, classes, field and method declarations are dealt with directly by Xtraitj by implementing specific validation checks (see Section 5.3).

We observe that the basic idea of the translation is not different from the previous versions of Xtraitj (Bettini and Damiani, 2013, 2014). Thus, the final generated Java code is basically the

same as in the previous implementations, apart from the generated JAVA annotations. What has been completely rewritten is the model inferrer, in order to refer to achieve the complete JAVA integration described in Section 4.

*5.2.1. The basic idea of the translation*

We will now informally sketch the generated JAVA code corresponding to XTRAITJ programs using some examples. Note that the JAVA code is generated only if the XTRAITJ program has passed the validation phase; thus, the generated JAVA code is always well-typed.

The translation of generics from XTRAITJ programs into JAVA programs is rather straightforward. We will use generics in the first example of the translation, and then we drop them in further examples.

Our strategy for generating the JAVA code for traits and classes is based on a few crucial properties:

- There will be exactly one JAVA interface and one JAVA class for each trait;
- There will be exactly one JAVA class for each XTRAITJ class;
- Each method body in each trait will have exactly one corresponding generated JAVA method;
- Trait compositions are implemented through JAVA object composition and method forwarding.

The generated JAVA code then will enjoy *compositionality*.

Using object composition and method forwarding we achieve the semantics of *delegation*. Note that in the literature (e.g., in design patterns by Gamma et al. (1995)), the term *delegation*, originally introduced by Lieberman (1986), is given different interpretations and it is often confused with the term *consultation* (Kniesel, 2000), which corresponds to *method forwarding* in JAVA. When *A delegates* to *B* the execution of a method *m*, `this` is bound to the sender (*A*). Thus, if in the body of the method *m* (defined in *B*) there is a method call `this.n`, then `n` will be executed binding `this` to *A*. In contrast, with standard JAVA method forwarding, `this` is always bound to the receiver *B*. Delegation is a much more powerful mechanism, since it allows us to achieve the semantics of dynamic binding, as we will see in the rest of the section.

Let us consider this trait definition (here we are considering only required fields and provided methods):

```
trait T1<V,U extends List<V>> {
    U f;
    U m(U f1) {
        this.f = f1;
        return this.f;
    }
    V n(U f1) {
        return this.m(f1).get(0);
    }
}
```

From this trait definition the following JAVA interface is generated:

```
public interface T1<V, U extends List<V>> {
  @XtraitjRequiredField public U getF();
  @XtraitjRequiredFieldSetter public void setF(final U f);
  @XtraitjDefinedMethod public U m(final U f1);
  @XtraitjDefinedMethod public V n(final U f1);
}
```

Let us explain the generated code:

- A (required) field in a trait corresponds to the getter and setter methods in the generated JAVA interface.
- This interface also contains the signatures of all the method declarations of the traits, i.e., both provided and required methods. Thus, the generated interface implicitly contains all the requirements of the corresponding trait. Of course, private methods in a trait will not be part of the generated JAVA interface.
- The generated methods are annotated with JAVA annotations, which are part of our XTRAITJ runtime library (required methods, if present in a trait, are annotated with `@XtraitjRequiredMethod` in the generated JAVA interface). Such annotations are crucial to achieve full JAVA integration, as explained in Section 4.

**Convention:** Note that from now on, unless it is strictly necessary for the explanation, we will not show JAVA annotations in the generated code.

Then, a JAVA class is generated implementing such interface:

```
public class T1Impl<V, U extends List<V>> implements T1<V,U> {
  private T1<V,U> _delegator;

  public T1Impl(final T1<V,U> delegator) { this._delegator = delegator; }

  public U getF() { return _delegator.getF(); }
  public void setF(final U f) { _delegator.setF(f); }

  public U m(final U f1) { return _delegator.m(f1); }
  public U _m(final U f1) { // translation into Java of the original method body
    this.setF(f1);
    return this.getF();
  }

  public V n(final U f1) { return _delegator.n(f1); }
  public V _n(final U f1) { // translation into Java of the original method body
    return this.m(f1).get(0);
  }
}
```

The important thing in the generated JAVA class is the `_delegator` field, of type `T1`, i.e., the JAVA interface generated for the trait. Recall that this interface contains all the required methods (including getter and setter methods for fields) and all the provided methods. The actual implementation for this field will be passed to the constructor of this JAVA class. In this class, all the methods defined in `T1` are forwarded to the field `_delegator`, even the ones corresponding to methods provided by the trait. In fact, for each method provided in the trait there will be a method with the same name but prefixed with `_` that contains the translation into JAVA of the original method's body. Both read and write access to fields are translated into calls to getter and setter methods, respectively, in the generated JAVA code. Of course, private methods will be directly translated to corresponding JAVA private methods without any additional method forwarding. Indeed, private methods are always statically bound.

We also forward provided methods to the `_delegator` because this allows a standard JAVA class to override methods and guarantees that the standard JAVA dynamic binding mechanism for overridden methods still works. This will also allow us to implement **restrict** (as shown at the end of Section 2.2, **restrict** can be used to achieve in a trait the same mechanism of standard JAVA method overriding).

Let us now consider the XTRAITJ class definition:

```
class C uses T1<String,ArrayList<String>> {
    ArrayList<String> f;
}
```

First of all, note that the type arguments for `T1` in the **uses** clause respect the bounds of the original trait's type parameters. Similarly, the defined field `f` fulfills the requirements of `T1` (with

the specified type arguments). Thus, the XTRAITJ compiler accepts this class as well-formed.

The JAVA class that is generated from the above XTRAITJ class definition is:

```
public class C implements T1<String,ArrayList<String>> {
 private ArrayList<String> f;

 public ArrayList<String> getF() { return this.f; }
 public void setF(final ArrayList<String> f) { this.f = f; }

 private T1Impl<String,ArrayList<String>> _T1 = new T1Impl(this);

 public ArrayList<String> m(final ArrayList<String> f1) {
  return _T1._m(f1);
 }

 public String n(final ArrayList<String> f1) {
  return _T1._n(f1);
 }
}
```

This shows that the generated JAVA class implements the generated JAVA interface of the used trait (including getter and setter methods for defined fields) with the corresponding type parameters instantiation. The generated JAVA class defines a field for each used trait and creates an object for such field, in this example it is `T1Impl`, passing itself to the trait's class constructor. This is well-typed in JAVA since the class implements the interface `T1`, and `T1Impl`'s constructor expects such a type. Note the use of type arguments for generics and the corresponding type parameters correctly instantiated in the generated class. The class forwards each method defined in the trait to the `T1Impl` instance, in particular, it calls the method with name prefixed with the underscore (recall that such method contains the translation into JAVA of the original method's body).

Summarizing, the main idea of the translation of an XTRAITJ class into a JAVA class is:

- `C` forwards to `T1Impl` the methods defined in the trait `T1`,
- `T1Impl` forwards to `C` the fields required in the trait (actually, the corresponding getter/setter methods).

The generated JAVA class can be used in any JAVA program and can be itself subclassed. In particular, thanks to our method forwarding, dynamic binding will be correctly implemented.

To demonstrate this last point, let us consider a much simpler example:

```
trait T1 {
 String s;
 String m() {
  this.s = this.s + ", Modified";
  return this.s;
 }
 String callM() { return this.m(); }
}

class C uses T1 {
 String s = "a String";
}
```

According to our translation strategy, the generated JAVA class for the trait will be:

```
public class T1Impl implements T1 {
 private T1 _delegator;

 public T1Impl(final T1 delegator) { this._delegator = delegator; }

 public String getS() { return _delegator.getS(); }
 public void setS(final String s) { _delegator.setS(s); }

 public String m() { return _delegator.m(); }
 public String _m() {
  this.setS(this.getS() + ", Modified");
  return this.getS();
 }

 public String callM() { return _delegator.callM(); }
 public String _callM() {
  return this.m();
 }
}
```

If we execute the following JAVA snippet that uses the generated JAVA code corresponding to the above XTRAITJ code:

```
System.out.println(new C().callM());
```

then the string ``a String, Modified'' will be printed on the screen.

Let us now override the original method `m` in a JAVA subclass:

```
public class CExt extends C {
 @Override
 public String m() {
  return super.m() + ", Redefined";
 }
}
```

If we execute the following JAVA snippet

```
System.out.println(new CExt().callM());
```

the string ``a String, Modified, Redefined'' will be printed on the screen. This shows that the use of `_delegator` ensures that standard JAVA classes can override methods and that the standard JAVA dynamic binding mechanism for overridden methods still works.

This is shown in the sequence diagram of Fig. 4.

*5.2.2. Dealing with required methods and Trait Sum*

Let us now consider a trait with a required method:

```
trait T2 {
 String m(); // required
 String useM() { return this.m(); }
}
```

As we saw above, the generated JAVA interface has a method definition for each method declaration in the trait, even required methods:

```
public interface T2 {
 @XtraitjRequiredMethod public abstract String m();
 @XtraitjDefinedMethod public abstract String useM();
}
```

The generated JAVA class follows the same scheme of the previous example: the only exception is that there will be no `_m` in

`T2Impl` since `m` is required. Of course, `m` will be forwarded to `_delegator`:

```
public class T2Impl implements T2 {
  private T2 _delegator;

  public T2Impl(final T2 delegator) { this._delegator = delegator; }
  public String m() { return _delegator.m(); }

  public String useM() { return _delegator.useM(); }
  public String _useM() { return this.m(); }
}
```

Let us now consider the following trait and class:

```
trait T1 {
  String s;
  String m() {
    this.s = this.s + ", Modified";
    return this.s;
  }
  String callM() { return this.m(); }
}

class C uses T1, T2 {
  String s = "a String";
}
```

This class definition is well-formed since all the requirements of the used traits are fulfilled: `T1` provides the method `m` required by `T2` and fields requirements are fulfilled as well.

The corresponding generated Java class is

```
public class C implements T1, T2 {
  private String s = "a String";

  public String getS() { return this.s; }
  public void setS(final String s) { this.s = s; }

  private T1Impl _T1 = new T1Impl(this);

  public String m() { return _T1._m(); }

  public String callM() { return _T1._callM(); }

  private T2Impl _T2 = new T2Impl(this);

  public String useM() { return _T2._useM(); }
}
```

The generated Java class implements both generated Java interfaces of the used traits. This class defines a field for each used traits passing itself to the trait's class constructor. Also in this case, this is well-typed in Java since the class implements both the interface `T1` and `T2`. The class forwards each method defined in a used trait to the corresponding instance (by calling the method prefixed with the underscore).

Running the following Java snippet:

```
System.out.println(new C().useM());
```

will call the Java method corresponding to the one defined in trait `T2`, which will forward the method `m` to `_delegator`; this field refers to the instance of `C`, and `C.m` will call `T1Impl._m`, which corresponds to the method `m` provided by `T1`. This is shown in the sequence diagram of Fig. 5.

Thus, trait method requirements are fulfilled in the generated Java code using method forwarding.

Let us consider an example where a trait uses another trait. This is a variation of the previous example (`T1` and `T2` are the same):

```
trait T3 uses T2 { }

class C uses T1, T3 {
    String s = "a String";
}
```

The generated Java class for `T3` follows the same pattern we have already seen. Moreover, it forwards the methods defined in `T2` to the corresponding instance:

```
public class T3Impl implements T3 {
  private T3 _delegator;

  private T2Impl _T2;

  public T3Impl(final T3 delegator) {
    this._delegator = delegator;
    _T2 = new T2Impl(delegator);
  }

  public String useM() { return _delegator.useM(); }
  public String _useM() { return _T2._useM(); }

  public String m() { return _delegator.m(); }
}
```

Note that when the instance for `T2Impl` is created inside `T3Impl`'s constructor the delegator that is passed to `T3Impl`'s constructor is also passed to `T2Impl`'s constructor. This means that, at runtime, all the fields in a class that correspond to used traits will share the same instance for `_delegator`.

*5.2.3. Dealing with alteration operations*

If a trait or a class uses a trait with some alteration operations, then, in the generated Java code, we cannot simply use the generated Java interface (and class) of the referred trait, since such an interface will be different. Indeed, if we exclude **restrict**, which does not alter the interface, and **alias**, which adds a method in the new interface, all the other alteration operations introduce changes in the interface of the new trait that make it incompatible with the interface of the original trait. This is consistent with the fact that traits are not types in XTraitJ.

We do not implement alteration operations by cloning a trait method and applying source-level modifications: alterations correspond to an additional Java interface and class which act as an adapter between the original used trait and the trait or class that alters the trait.

To give an idea of the translation, let us consider an example that uses **rename**:

```
trait T1 {
    String m() { return "m"; }
    String callM() { return this.m(); }
}
trait T2 uses T1[rename m to renamedM] {
    String callRenamedM() { return this.renamedM(); }
}
```

The generated additional Java class and interface for the adapter and the relations with the other generated Java elements are shown in Fig. 6. The name for the adapter's Java elements is built using the name of the two traits.[9] The additional adapter

[9] In the actual implementation, the name has also an additional suffix, since a trait can use the same trait more than once with different alteration operations.

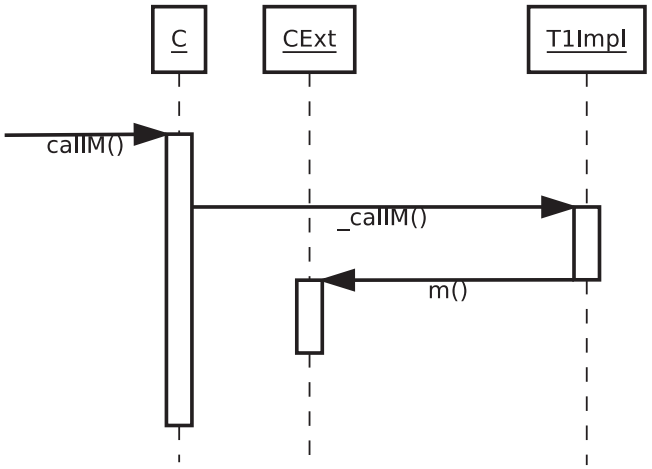


**Fig. 4.** Sequence diagram of the generated JAVA code for classes and traits.

interface corresponds to the original trait's interface after the alteration operations have been applied. However, there is no subtyping relation between the two interfaces. The adapter class implements both the adapter interface and the original trait interface by appropriately forwarding method invocations:

```
public class T2_T1_AdapterImpl implements T2_T1_Adapter, T1 {

  private T2_T1_Adapter _delegator;

  private T1Impl _T1;

  public T2_T1_AdapterImpl(final T2_T1_Adapter delegator) {
    this._delegator = delegator;
    _T1 = new T1Impl(this);
  }

  public String m() { return this.renamedM(); }
  public String renamedM() { return _delegator.renamedM(); }
  public String _renamedM() { return _T1._m(); }

  public String callM() { return _delegator.callM(); }
  public String _callM() { return _T1._callM(); }
}
```

The crucial point in this class is that when it creates the instance for `T1Impl` it passes itself (i.e., `this`) to `T1Impl`'s constructor. Since this class implements `T1`, the constructor invocation is well-typed.

This generated adapter class also implements the semantics of renaming: it forwards the original method to the renamed one. In this example, `_renamedM` is forwarded to `_m`, which represents the translation into JAVA of the original trait method.

Consider this class using `T2`

```
class C uses T2 { }
```

Then executing this JAVA code will print `''m''` twice:

```
C c = new C();
System.out.println(c.callM());
System.out.println(c.callRenamedM());
```

The sequence diagram of the invocation `c.callM()` is shown in Fig. 7.

As hinted above, the additional forwarding to `_delegator` is required in order to implement the dynamic binding semantics for method invocation. We have already showed this behavior by using a standard JAVA derived class with method overriding. The same holds if we use **restrict**. For instance, consider this trait

```
trait T3 uses T2[restrict renamedM] {
    String renamedM() { return "new m"; }
}
```

which "redefines" `renamedM` by restricting it and by providing a new implementation; then consider this class

```
class C uses T3 { }
```

Executing this JAVA code will print `''new m''` twice:

```
C c = new C();
System.out.println(c.callM());
System.out.println(c.callRenamedM());
```

This is another example of the compositionality of our generated code.

### 5.3. Validation

Since we provide a mapping from a trait method to a JAVA method, XBASE is able to automatically type-check the expression of the trait method (e.g., using the return type of the method and the types of the parameters). This works because the types that we use in a XTRAITJ program are actually references to JAVA types. Thus, we completely delegate the type-checking of method bodies to XBASE. Reusing the type system of XBASE is straightforward when the mapping between the language model and the JAVA model is one-to-one, i.e., each element of your language corresponds to exactly one element in the JAVA model. This is not the case for XTRAITJ: as shown in Section 5.2.1, we map each XTRAITJ trait in several JAVA model elements. The introduction of generics in XTRAITJ raised many issues in that respect, since we need to make sure that type parameters and the corresponding type arguments are correctly translated. This required us to tweak the default scoping mechanism of XTEXT in many places to make XBASE's type system work in the presence of generics (see footnote [1] in Section 1.

In Fig. 8 we show some type errors reported by XBASE. Note that XBASE deals only with expressions: language features like traits, classes, field and method declarations are dealt with directly by XTRAITJ; method bodies, in contrast, completely rely on XBASE expressions. Thus, all the checks concerning method conflicts are implemented by us. These also include the check that a class provides all fields and methods required by the used traits (in Fig. 8 we issue an error since the trait requires `String f` while the class defines `int f`). Moreover, we also implement the checks related to the correct usage of trait alteration operations (e.g., required methods and fields cannot be hidden).

### 5.4. IDE

One of our main design choices and goals is the integration of our language in Eclipse (see Section 5.1). In this respect, XTEXT and XBASE enhance the Eclipse tooling for JAVA. For instance, we can navigate to a JAVA type definition directly from an XTRAITJ program, see its type hierarchy, and use other features that are present in the Eclipse JAVA editor. This also holds the other way round: from a JAVA program that uses code generated from a XTRAITJ program we can navigate directly to the original XTRAITJ method definition.

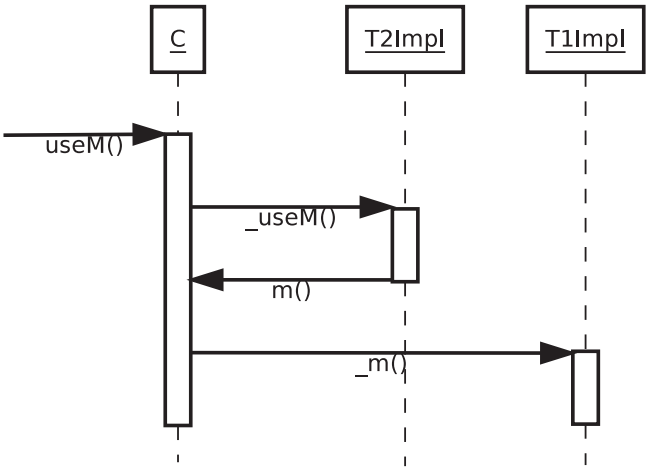


**Fig. 5.** Sequence diagram of the generated Java code for classes and traits in the presence of required methods.

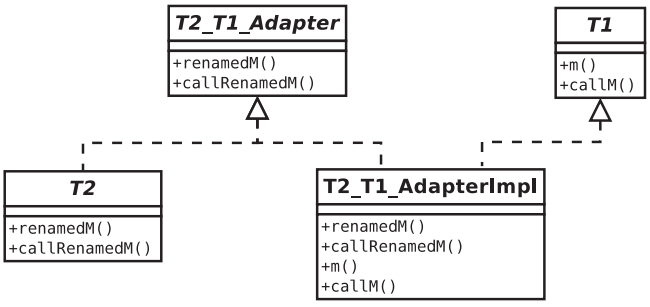


**Fig. 6.** Generated Java interfaces and classes in the presence of **rename**. Interface names are in *italics* font.

In this new implementation of XTraitJ, thanks to the complete integration with Java described in Section 4, we have full support of Java-like packages and automatic import management. The latter feature consists in the editor automatically inserting an import statement during content assist usage. This is shown in Fig. 9: after selection in the content assist, the import statement corresponding to the chosen trait is automatically inserted. This was possible since trait references are actually references to the corresponding generated Java interfaces. Similarly, the "Organize Imports" context menu is available as well. These are mechanisms a Java Eclipse user is accustomed to.

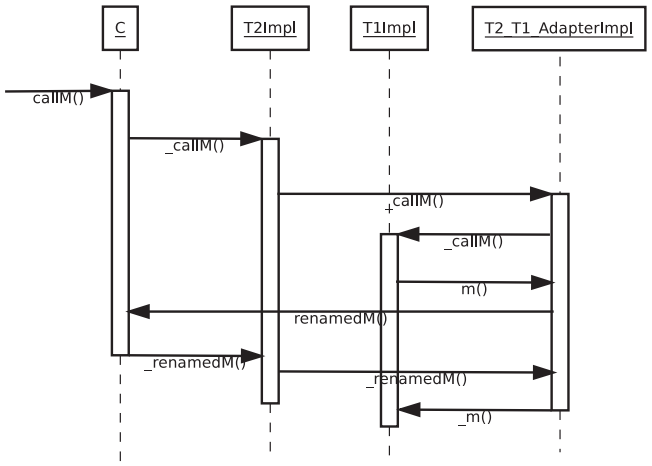


**Fig. 7.** Sequence diagram of the generated Java code for classes and traits in the presence of a renamed method invocation.

Another advantage of this new version of XTraitJ is the complete integration with the Eclipse building infrastructure. Eclipse provides a very nice incremental building mechanism and XText/XBase leverage these mechanisms too. In particular, they rely on the Java types indexing to achieve the incremental building concerning dependencies across programs and, more importantly, across projects. In the previous version, we did not exploit such mechanisms completely, since we relied on references to trait model elements. Since now we refer to traits using Java type references, XTraitJ correctly integrates with the XText Eclipse building mechanisms.

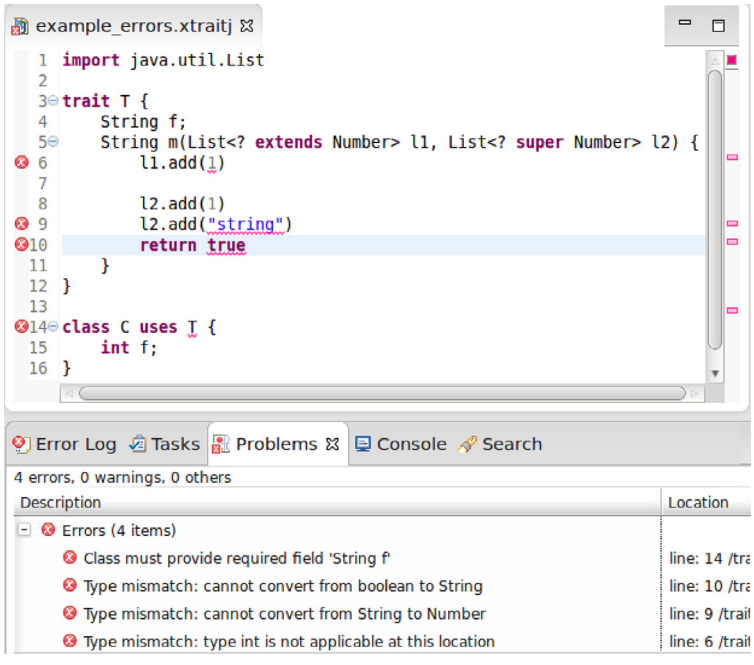


**Fig. 8.** Errors reported in the IDE.

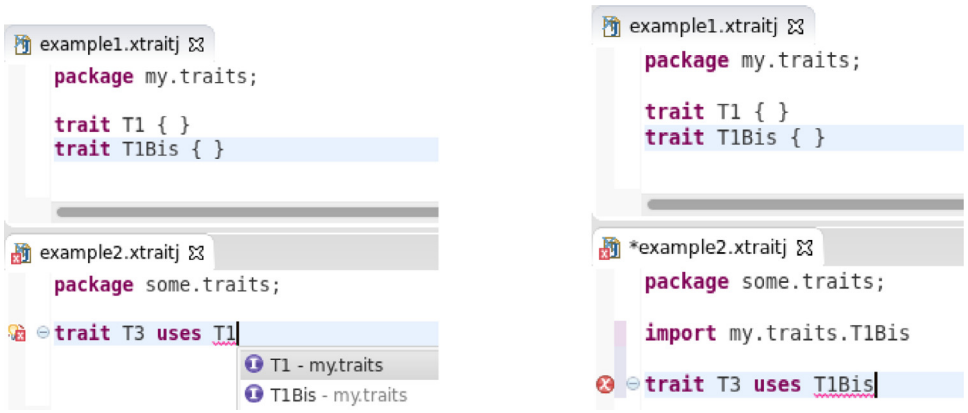


**Fig. 9.** Automatic import insertion during content assist: (right) the content assist proposes trait references for the **uses** clause; (left) the user selects one and the corresponding import statement is automatically inserted.

A well-known problem with implementations that generate Java code is that you can only debug the generated Java code, which is usually quite different from the original program. Our XTraitJ implementation does not have this drawback: thanks to XBase we can debug the original XTraitJ code. In Fig. 10 we show a debug session of a Java program that uses code generated by XTraitJ: we have set a breakpoint on an XTraitJ file, and when the Java program hits the corresponding Java code the debugger automatically switches to the original XTraitJ code (see the file names in the thread stack, the "Breakpoint" view and the "Variables" view). Note that the debugger will automatically skip the additional forward methods generated by our compiler. However, it is always possible to switch between the generated Java code and XTraitJ code. When switching to generated Java code, the programmer can debug the additional forwarding methods.

Currently, there is also refactoring support for names, including names of methods, traits and classes, and generic type parameters renaming. This is the default renaming support provided by XText and XBase; it also works across files. We are investigating adding further refactoring mechanisms to extract methods into separate traits (possibly by integrating such mechanisms with the ones proposed by Bettini et al. (2008), see also Section 8).

## 6. Evaluation

In this section we first discuss the pros and cons of the implementation (Section 6.1); then we present some performance tests, concerning both the runtime and the compiler of XTraitJ (Section 6.2).

### 6.1. Pros and cons of the implementation

The flattening translation presented in Section 3 provides a simple and intuitive way to specify the semantics of traits, but it is not an effective implementation technique. Implementing the flattening semantics directly would lead to a huge amount of duplicated code, increasing the size of the final Java program. Moreover, this would break modularity and traits could not be used to implement libraries, because the clients' code would need to be regenerated. In particular, if the body of a trait's method is changed by the programmer, then all the flattened classes that use that method would need to be regenerated (for instance, this is the approach presented by Quitslund et al. (2004)).

Instead, our implementation is modular in this respect. In fact, as stated in Section 5.2, each method body is translated into exactly one Java method body. Even alteration operations (Section 2.2) do not require us to copy the original method body: an additional adapter class is generated so that the generated Java methods behave according to the semantics of the alteration operations (Section 5.2.3). This implies that our Java code generation is compositional in the presence of alteration operations: we reuse the previously generated Java code and we forward method invocation differently (through an adapter). Note also that copying and modifying the body of a method (i.e., an XBase expression) of a XTraitJ program would not allow us to reuse all the automatic mechanisms of XBase, including the XBase implementation of the Java type system. Moreover, we would not be able to seamlessly reuse the automatic integration in Eclipse provided by XBase, including the debugging mechanisms.

Summarizing, we think that an approach that applies flattening for the implementation would simply not scale: in a project with a big code base a single modification would require the generation of too many artifacts. Our approach does not suffer from this problem, especially in this new implementation that fully exploits the mechanisms already implemented by XText and Eclipse JDT. In fact, dependencies among files and among projects are already handled in an efficient way by XText. On modifications on a file, only the dependent files will be recompiled by XTraitJ. In contrast, a solution based on flattening would not exploit XText's building mechanisms.

The Java code generated by XTraitJ depends on the runtime library of XTraitJ, which includes only the Java annotations that we use in the generated code. Moreover, the generated code depends on the XBase library and on the Google Guava library. Altogether, these dependencies are less than 2 MB in size, so they can be easily deployed together with the generated Java code. Thus the generated Java code can run on any Java platform, and thanks to the reduced size of the required Java libraries, it can be installed on Java devices, such as Android devices.

The only drawback of our translation based on object composition and method forwarding is the overhead of method forwarding. However, to achieve the same flexibility supported by traits in a pure Java application the programmers usually resort to design patterns based on object composition and forwarding (including techniques to simulate multiple inheritance, such as the one presented by Bettini et al. (2003b)). Note that generics in XTraitJ do not introduce any additional overhead: our generics are translated exactly into Java generics, thus they have the same performance as in a standard Java program using generics. In this respect, according to the *type erasure* model (Bracha et al., 1998), generic types are removed during compilation and are not present in the generated byte-code. The performance will thus be the same as a program using raw types.

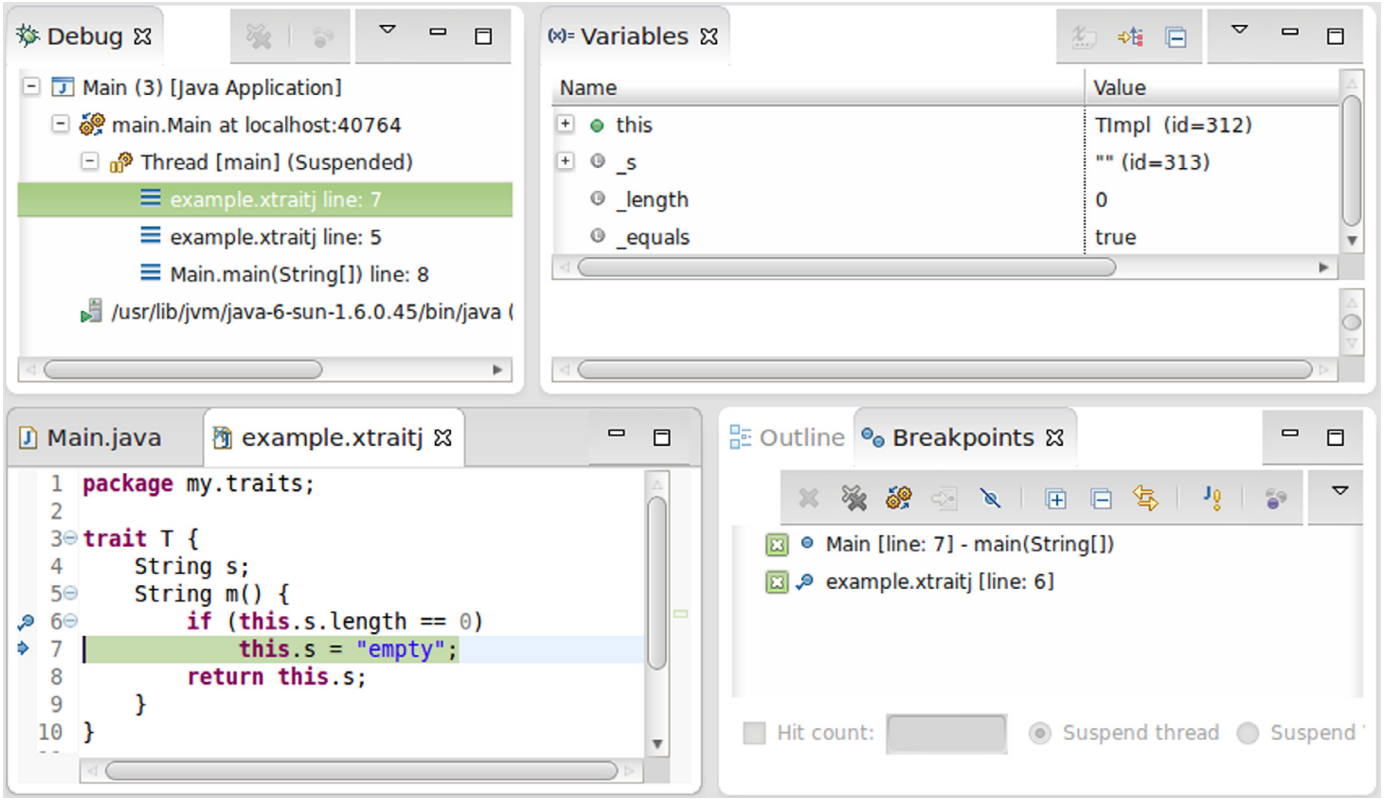


**Fig. 10.** Debugging XTRAITJ code.

### 6.2. Performance tests

To measure the overhead of method forwarding, we ran a few JUnit tests. All these JUnit tests are available in the XTRAITJ source repository, https://github.com/LorenzoBettini/xtraitj. We will show the measured time for executing these tests reported by JUnit itself. The measurements we report in this section were taken on a quad-core 2.20GHz Intel i7-4702HQ CPU, with 16 GiB RAM, running Linux Kubuntu 15.10 64bit, kernel 4.2, Oracle Java 8. We executed such tests several times, to make sure that the divergence of the results of each run was negligible. When we executed the tests there were no other programs running on the computer, apart from Eclipse, from which we ran the JUnit tests (the figures with measurements shown in the following are screenshots of the Eclipse JUnit view).[10]

We ran a few JUnit tests using the `CStack` we implemented in Section 2.2. We ran the same tests using a manually implemented stack, which mimics what we would obtain by flattening the class and traits of Section 2.2. The results are shown in Fig. 11-left. The number in the name of the test represents the number of elements pushed onto the stack and then popped. JUnit reports the execution time for each test case. The test method `aWarmUp` should be ignored; it is used to make sure that both the JVM and the XTEXT internal mechanisms are started and they do not interfere with the time measurements. Recall that in this example, there is no alteration operation. We performed the same tests for the lifo example of Section 2.2 using the `CLifo` class, in particular, the alternate version that uses two traits and several alteration operations. We also compared the performance of `CLifo` with a manually flattened version. The results are shown in Fig. 11-right. Since in this example we use some alteration operations, the number of method forwards is higher than in the previous example.

From the results of the above tests, we see that the slowdown factors gets smaller when the larger number of elements are added/removed. For example, the slowdown factors of `CStack` are 4.0, 3.5 and 1.9 for 1000, 10,000 and 100,000 elements, respectively. Thus, the overhead of single method invocation can be significant. Nevertheless, we believe this overhead is acceptable, and it is compensated by the high degree of code reuse provided by traits.

We performed additional performance tests using our XTRAITJ utility methods for collections that we showed in a previous paper (Bettini and Damiani, 2014); these examples are also available in the XTRAITJ source repository, https://github.com/LorenzoBettini/xtraitj. Such utility methods mimic the methods for collections provided by the *Google Guava* JAVA library. In our performance tests we create a list of $n$ integers and we process that list with our `map` method and then with `join`. We compared the performance of our generated code with code that uses the corresponding methods of the `com.google.common.collect.Iterables` class in the *Google Guava* JAVA library. The results are shown in Fig. 12 (again, the number in the name of the test corresponds to the number $n$ of elements in the list being processed). In this case, the time increases by 26% over that of the Google Guava library, for a big collection.

Finally, we performed some tests for the compiler of the previous version of XTRAITJ (Bettini and Damiani, 2014) compared to

[10] We execute all these performance tests also as part of our Continuous Integration Maven build on the public instance of *Travis-CI*, available at https://travis-ci.org/LorenzoBettini/xtraitj. On Travis-CI we use a *multi-os* build, so that we run the Maven build and all the tests both on Linux and MacOSX virtual machines provided by Travis-CI. At the time of writing the specifications for such virtual machines are Linux Ubuntu 12.04.5 LTS 64bit, kernel 3.13.0, Oracle Java 8 and Mac OS X 10.11, Xcode 8, Oracle Java 8—these are the only details that we found on Travis-CI specifications. The performance results on Travis-CI virtual machines reflect the ones executed on our machine concerning the slowdown factors.

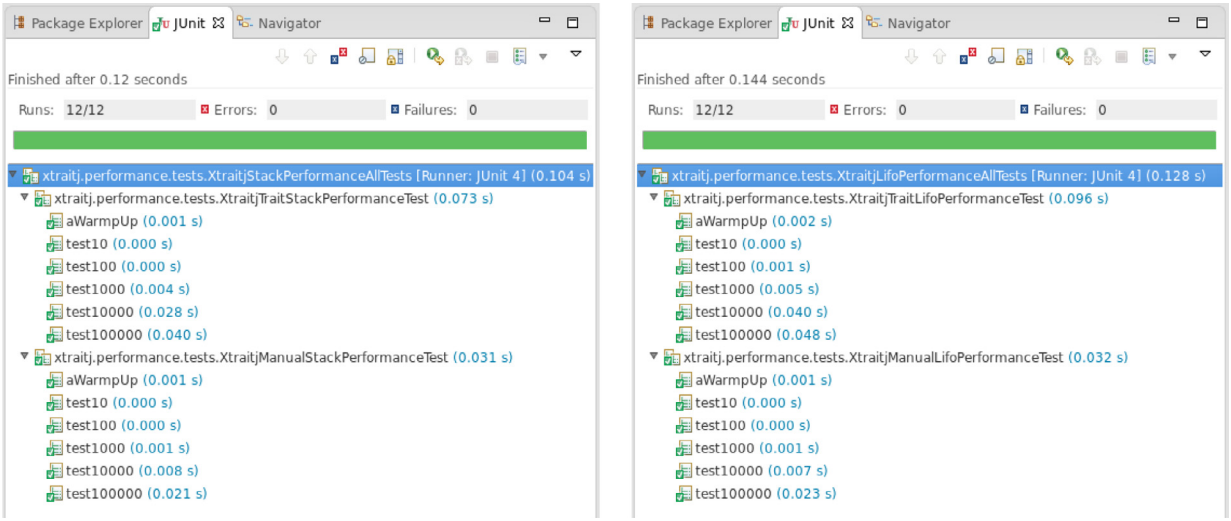


**Fig. 11.** JUnit results for the trait example of stack (left) and of lifo (right).

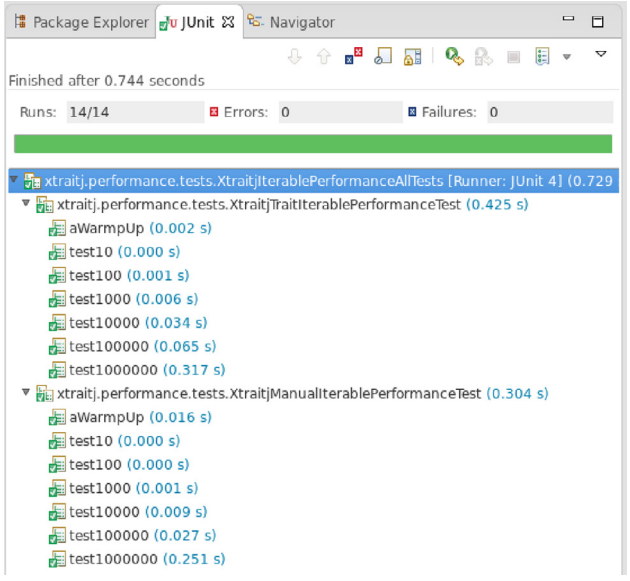


**Fig. 12.** JUnit results for the trait example of utility methods for collections implemented in XTRAITJ and the corresponding ones from the Google Guava library.

the current version of XTRAITJ. The results are shown in Fig. 13:[11] the number $n$ in the name of the test represents the number of traits in the program used as input; each trait $T_i$ uses $T_{i-1}$ and renames a method. The current compilation strategy in XTRAITJ illustrated in this paper performs better than the previous version of the compiler. In fact, we avoid rescanning the whole trait "uses" relation when checking and compiling the traits and we reuse the cached information of the JAVA model elements already created by our model inferrer.

### 6.3. Summary of the evaluation

We believe that the properties of our implementation (see Section 6.1), namely, compositionality, modularity and IDE tooling, compensate for the runtime overhead of method forwarding (see Section 6.2). The implementation described by Smith and Drossopoulou (2005) also uses method forwarding to enjoy the mentioned advantages. The current implementation of Pharo (Black et al., 2010), relies on flattening (i.e, the methods provided by a trait are inlined into the body of the classes that use it). However, most of the drawbacks described above do not apply since in the PHARO environment the IDE and the language system rolled into one in order to provide automatic immediate feedback on any change introduced by the user (i.e., there are no compiling and deploying steps).

## 7. Related work

The literature on traits and JVM-compatible languages has been partially compared throughout the paper. In Section 6 we also compared our implementation strategy with other implementations of traits. Note that we only considered implementations where "traits" refers to constructs that are compliant with the distinguished features listed in the original formulation of traits (Ducasse et al., 2006) (cf. the discussion in Section 1); thus, we have not compared our implementation with languages where "traits" denotes a substantially different construct (e.g., traits in SCALA (Odersky, 2007) or traits in C++). We refer to out previous paper (Bettini and Damiani, 2013) for a comparison with the core language TRAITRECORDJ (Bettini et al., 2013d), which was the starting point for the implementation of XTRAITJ and was implemented by flattening. In the rest of this section we add further discussions and comparisons.

In the original formulation of traits (Ducasse et al., 2006), the methods provided by a trait can access state only by using accessor methods, which become required methods of the trait. A possible way to overcome this limitation is to make traits *stateful* (as proposed by Bergel et al. (2008) for SMALLTALK/SQUEAK) by adding private fields that can be accessed from the clients (possibly under a new name), or merged with other variables.

We believe that keeping traits stateless and introducing required fields, as in the formulation of traits in a structurally-typed setting by Fisher and Reppy (2004), provides a more lightweight way to address the problem. Therefore, we adopted this solution in XTRAITJ: required methods/fields have to be explicitly declared together with their type. This explicit declaration of requirements allows for a better IDE integration and provides the programmer with better tooling experience, which was also one of the main goals of XTRAITJ. We decided to require the programmer to

[11] These JUnit tests are executed in two different versions of Eclipse: one with the old version of XTRAITJ and the other one with the current version. That is why they are reported in two different JUnit executions.

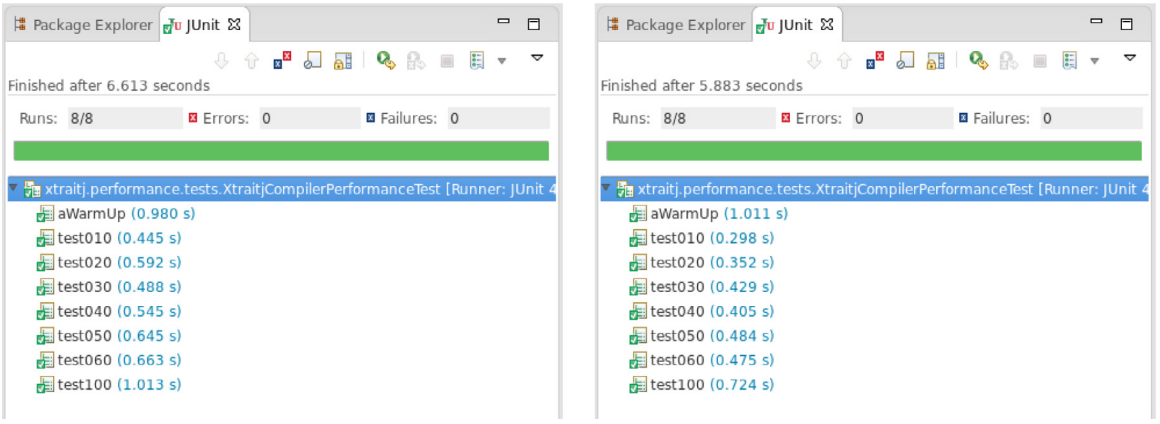


**Fig. 13.** Compiler tests. Left: previous version, right: current version

declare the fields in the classes since, on the one hand, we believe that these field declarations provide a better support for code documentation and, on the other hand, the IDE support of XTRAITJ eliminates any burden on the programmer by a quickfix to automatically generate the declarations of all the fields required by the traits used by a class.

Some of the module composition operations present in Bracha's JIGSAW framework (Bracha, 1992) have been adapted to traits. In particular, an instantiation of the JIGSAW framework within a JAVA-like nominal type system has been proposed by Lagorio et al. (2012). JIGSAW models field and method renaming operations, which are present in XTRAITJ and are not present in most formulations of traits. Method renaming in the context of multiple class-based inheritance is also present in Meyer's EIFFEL language (Meyer, 1997). Method renaming for traits was introduced by Reppy and Turon (2006).

Reppy and Turon (2007) also proposed a variant of traits that can be parametrized by member names (field and methods), types and values. Thus, the programmer can write *trait functions* that can be seen as code templates to be instantiated with different parameters.

*Reverse generics* (Bergel and Bettini, 2011) are a general linguistic mechanism to define a generic type from a non-generic type. For a given set of types, a generic is formed by unbinding static dependencies contained in these types.

In a previous paper (Bettini et al., 2015), we introduced *parametric traits*, that is, traits that are parametrized by interface names and class names. Parametric traits are applied to interface names and class names to generate traits that can be assembled in other (possibly parametric) traits or in classes. This mechanism provides both features similar to trait functions and features similar to reverse generics. Mechanisms like parametric traits and reverse generics could be partially implemented in XTRAITJ, for instance, by introducing an alteration operation to specify which types must become type parameters in the new traits. However, we would not be able to abstract types in method body expressions such as object instantiations: due to the *type erasure* model (Bracha et al., 1998), generic types cannot be instantiated in JAVA. In C++ (using template generic programming) and in dynamically typed languages, adding such mechanisms is easier (see, e.g., Bergel and Bettini (2012, 2013)). Since XTRAITJ targets the JAVA platform, we have to share its limitations.

## 8. Conclusions and future work

In this paper we present XTRAITJ, a trait-based programming language that features complete compatibility and interoperability with the JAVA platform. We chose to implement XTRAITJ by relying on XTEXT since it is the de-facto standard framework for implementing DSLs in the Eclipse ecosystem, it is continuously supported, and it has a wide community. XTEXT is continuously evolving, and the main forthcoming features will be the integration in other IDEs (mainly, IntelliJ), and the support for programming on the Web (i.e., allowing programming directly in a browser) (Efftinge and Zarnekow, 2015).

Bettini et al. (2008) presented a tool for identifying the methods in a JAVA class hierarchy that could be good candidates to be refactored in traits. This tool is an adaptation of the SMALLTALK analysis tool presented by Lienhard et al. (2005) to a JAVA setting. It will be interesting to investigate how to apply this approach for porting and refactoring existing JAVA code to XTRAITJ code, for instance, the JAVA stream library (as in the context of SMALLTALK (Cassou et al., 2009)).

Damiani et al. (2011, 2014a) presented a compositional proof systems for the verification of pure traits. We plan to extend a different verification system, the KeY system (Beckert et al., 2007), for deductive verification of JAVA programs to XTRAITJ by implementing a proof system similar to the one proposed by Damiani et al. (2014a).

We also plan to add method overloading for trait definitions in future releases of XTRAITJ. In the presence of overloaded methods, trait alteration operations could be extended in order to specify the complete signature of methods to avoid ambiguities. Note that, even though in the current version one cannot define overloaded methods in a trait definition, it is still possible to invoke overloaded methods of existing JAVA classes in XTRAITJ method bodies.

Another interesting subject for future work is the expression language. In XTRAITJ we use XBASE, which provides a nice JAVA-like expression language, which is easy to learn for JAVA programmers, has a clean syntax, avoiding much verbosity of JAVA, and has other advanced features. However, XBASE is not JAVA, and this might undermine the usability of XTRAITJ when porting legacy code to traits. Bettini and Crescenzi (2015, 2016) presented a customization of XBASE which has the full JAVA syntax, while keeping all the features provided by XBASE (both from the type system and the IDE tooling point of view). It will be interesting to extract from the implementation presented by Bettini and Crescenzi (2015, 2016) such parts to have an implementation of XTRAITJ that supports the full JAVA expression syntax for the expressions (i.e., the body of methods). This will make it easier to port existing JAVA code to traits, and to adapt the refactoring framework presented by Bettini et al. (2008).

Dynamic trait replacement (Smith and Drossopoulou, 2005; Bettini et al., 2013a) is a programming language feature for changing the objects' behavior at runtime by replacing some of the

objects' methods. In future work we would like to integrate a form of dynamic trait replacement in XTraitJ.

Delta-trait programming (Damiani et al., 2014b) is a recently proposed approach for implementing software product lines (Clements and Northrop, 2001; Pohl et al., 2005) by smoothly integrating the modularity mechanisms provided by delta-oriented programming (Bettini et al., 2013c; Koscielny et al., 2014) and pure trait-based programming (see the discussion in Section 1). In future work we would like to implement delta-trait programming on top of XTraitJ.

## Acknowledgments

We are grateful to Stéphane Ducasse for interesting explanations about the Pharo implementation, and the Xtext committer Sebastian Zarnekow for insightful discussions about the implementation of the Xbase type system. We also thank the JSS anonymous reviewers for many comments and suggestions for improving the presentation.

## References

Pharo, http://pharo.org/.
Xtext, 2015 http://www.eclipse.org/Xtext.
Ancona, D., Lagorio, G., Zucca, E., 2003. Jam—designing a Java extension with mixins. ACM TOPLAS 25 (5), 641–712.
Barendregt, H.P., 1984. The Lambda Calculus Its Syntax and Semantics, vol. 103. North Holland. revised edition
Batov, V., 2004. Java generics and C++ templates. C/C++ Users J. 22 (7), 16–21.
Beck, K., 2003. Test Driven Development: By Example. Addison-Wesley.
Beckert, B., Hähnle, R., Schmitt, P. H. (Eds.), 2007. Verification of Object-Oriented Software: The KeY Approach. Vol. vol. 4334 of LNCS. Springer.
Bergel, A., Bettini, L., 2011. Reverse Generics: Parametrization after the Fact. In: Software and Data Technologies. In: Communications in Computer and Information Science, vol. 50. Springer, pp. 107–123.
Bergel, A., Bettini, L., 2012. Generics and reverse generics for Pharo. In: ICSOFT. SciTePress, pp. 363–372.
Bergel, A., Bettini, L., 2013. Generic programming in Pharo. In: Software and Data Technologies. In: Communications in Computer and Information Science, vol. 411. Springer, pp. 66–79. doi:10.1007/978-3-642-45404-2_5.
Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R., 2008. Stateful traits and their formalization. Comput. Lang. Syst Struct. 34 (2-3), 83–108. doi:10.1016/j.cl.2007.05.003.
Bettini, L., 2013. Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing.
Bettini, L., 2015. Implementing Type Systems for the IDE with Xsemantics. J. Logical Algebraic Methods Progr. doi:10.1016/j.jlamp.2015.11.005. In press
Bettini, L., Bono, V., Likavec, S., 2003. A core calculus of higher-order mixins and classes. In: Types for Proofs and Programs. Springer, pp. 83–98.
Bettini, L., Bono, V., Naddeo, M., 2008. A trait based re-engineering technique for Java hierarchies. In: PPPJ. ACM, pp. 149–158. doi:10.1145/1411732.1411753.
Bettini, L., Capecchi, S., Damiani, F., 2013. On flexible dynamic trait replacement for Java-like languages. Sci. Comput. Progr. 78 (7), 907–932. doi:10.1016/j.scico.2012.11.003.
Bettini, L., Crescenzi, P., 2015. Java– meets Eclipse (An IDE for teaching Java following the object-later approach). In: ICSOFT. SciTePress, pp. 31–42. doi:10.5220/0005512600310042.
Bettini, L., Crescenzi, P., 2016. An eclipse IDE for teaching Java–. In: Software Technologies. Springer, pp. 63–78. doi:10.1007/978-3-319-30142-6_4.
Bettini, L., Damiani, F., 2013. Pure trait-based programming on the Java platform. In: PPPJ. ACM, pp. 67–78. doi:10.1145/2500828.2500835.
Bettini, L., Damiani, F., 2014. Generic traits for the Java platform. In: PPPJ. ACM, pp. 5–16. doi:10.1145/2647508.2647518.
Bettini, L., Damiani, F., Geilmann, K., Schäfer, J., 2013. Combining traits with boxes and ownership types in a Java-like setting. Sci. Comput. Progr. 78 (2), 218–247. doi:10.1016/j.scico.2011.10.006.
Bettini, L., Damiani, F., Schaefer, I., 2013. Compositional type checking of delta-oriented software product lines. Acta Informatica 50, 77–122. doi:10.1007/s00236-012-0173-z.
Bettini, L., Damiani, F., Schaefer, I., 2015. Implementing type-safe software product lines using parametric traits. Sci. Comput. Progr. 97 (Part 3), 282–308. doi:10.1016/j.scico.2013.07.016.
Bettini, L., Damiani, F., Schaefer, I., Strocco, F., 2013. TraitRecordJ: a programming language with traits and records. Sci. Comput. Progr. 78 (5), 521–541. doi:10.1016/j.scico.2011.06.007.
Bettini, L., Loreti, M., Venneri, B., 2003. On multiple inheritance in Java. In: Technology of Object-Oriented Languages, Systems and Architectures. In: The Kluwer International Series in Engineering and Computer Science, vol. 732. Springer, pp. 1–15. doi:10.1007/978-1-4615-0413-9_1.
Black, A.P., Ducasse, S., Nierstrasz, O., Pollet, D., 2010. Pharo by Example (Version 2010-02-01). Square Bracket Associates.
Bono, V., Damiani, F., Giachino, E., 2007. Separating type, behavior, and state to achieve very fine-grained reuse. In: Electronic Proceedings of FTfJP. http://ftfjp.bitbucket.org/
Bono, V., Damiani, F., Giachino, E., 2008. On traits and types in a Java-like setting. In: TCS (Track B). Springer, pp. 367–382. doi:10.1007/978-0-387-09680-3_25.
Bono, V., Mensa, E., Naddeo, M., 2014. Trait-oriented programming in Java 8. In: PPPJ. ACM, pp. 181–186. doi:10.1145/2647508.2647520.
Bracha, G., 1992. The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance. University of Utah Ph.D. thesis.
Bracha, G., Cook, W., 1990. Mixin-based inheritance. In: OOPSLA. ACM, pp. 303–311.
Bracha, G., Odersky, M., Stoutamire, D., Wadler, P., 1998. Making the future safe for the past: Adding genericity to the Java programming language, 10. In: OOPSLA. In: ACM SIGPLAN Notices, vol. 33, pp. 183–200.
Cassou, D., Ducasse, S., Wuyts, R., 2009. Traits at work: The design of a new trait-based stream library. Comput. Lang. Syst. Struct. 35 (1), 2–20. doi:10.1016/j.cl.2008.05.004.
Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. Addison Wesley Longman.
Cook, W., Hill, W., Canning, P., 1990. Inheritance is not subtyping. In: POPL. ACM, pp. 125–135.
Damiani, F., Dovland, J., Johnsen, E.B., Schaefer, I., 2011. Verifying traits: A proof system for fine-grained reuse. In: FTfJP. ACM, pp. 8:1–8:6. doi:10.1145/2076674.2076682.
Damiani, F., Dovland, J., Johnsen, E.B., Schaefer, I., 2014. Verifying traits: an incremental proof system for fine-grained reuse. Formal Aspects of Computing 26 (4), 761–793. doi:10.1007/s00165-013-0278-3.
Damiani, F., Schaefer, I., Schuster, S., Winkelmann, T., 2014. ISoLA 2014, Proceedings, Part I. Springer, pp. 289–303. doi:10.1007/978-3-662-45234-9_21. Delta-Trait Programming of Software Product Lines
Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A., 2006. Traits: a mechanism for fine-grained reuse. ACM TOPLAS 28 (2), 331–388.
Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M., 2012. Xbase: implementing Domain-Specific Languages for Java. In: GPCE. ACM, pp. 112–121.
Efftinge, S., Zarnekow, S., 2015. The Future of Xtext. XtextCon.
Fisher, K., Reppy, J., 2004. A typed calculus of traits. FOOL.
Flatt, M., Krishnamurthi, S., Felleisen, M., 1998. Classes and mixins. In: POPL. ACM, pp. 171–183. doi:10.1145/268946.268961.
Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.
Ghosh, D., 2004. Generics in Java and C++: a comparative model. ACM SIGPLAN Notices 39 (5), 40–47.
Hendler, J., 1986. Enhancement for multiple-inheritance. In: Object-Oriented Programming Workshop. ACM, pp. 98–106. doi:10.1145/323779.323748.
Igarashi, A., Pierce, B., Wadler, P., 2001. Featherweight Java: a minimal core calculus for Java and GJ. ACM TOPLAS 23 (3), 396–450.
Kats, L.C.L., Visser, E., 2010. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: OOPSLA. ACM, pp. 444–463.
Kniesel, G., 2000. Dynamic Object-Based Inheritance with Subtyping. University of Bonn Ph.D. thesis.
Koscielny, J., Holthusen, S., Schaefer, I., Schulze, S., Bettini, L., Damiani, F., 2014. DeltaJ 1.5: delta-oriented programming for Java 1.5. In: PPPJ'14. ACM, pp. 63–74. doi:10.1145/2647508.2647512.
Lagorio, G., Servetto, M., Zucca, E., 2012. Featherweight Jigsaw - Replacing inheritance by composition in Java-like languages. Inf. Comput. 214 (0), 86–111. doi:10.1016/j.ic.2012.02.004.
Lieberman, H., 1986. Using prototypical objects to implement shared behavior in object oriented systems. ACM SIGPLAN Notices 21 (11). 214–214
Lienhard, A., Ducasse, S., Arévalo, G., 2005. Identifying traits with formal concept analysis. In: ASE. IEEE, pp. 66–75. doi:10.1145/1101908.1101921.
Limberghen, M., Mens, T., 1996. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. Object Oriented Syst. 3 (1), 1–30.
Liquori, L., Spiwack, A., 2008. FeatherTrait: a Modest Extension of Featherweight Java. ACM TOPLAS 30 (2), 11:1–11:32.
Martin, R.C., 2003. Agile Software Development: Principles, Patterns, and Practices. Prentice Hall.
Meyer, B., 1997. Object-Oriented Software Construction, 2nd ed. Prentice-Hall.
Murphy-Hill, E.R., Quitslund, P.J., Black, A.P., 2005. Removing duplication from java.io: a case study using traits. In: OOPSLA. ACM, pp. 282–291.
Nierstrasz, O., Ducasse, S., Schärli, N., 2006. Flattening traits. JOT 5 (4), 129–148.
Odersky, M., 2007. The Scala Language Specification. Technical Report. Programming Methods Laboratory, EPFL. Version 2.4
Oehme, S., 2015. Continuous Integration for Xtext languages. XtextCon.
Pohl, K., Böckle, G., van der Linden, F., 2005. Software Product Line Engineering - Foundations, Principles, and Techniques. Springer.
Quitslund, P.J., 2004. Java Traits — Improving Opportunities for Reuse. Technical Report. OGI School of Science & Engineering, Beaverton, Oregon, USA. CSE-04-005
Quitslund, P.J., Murphy-Hill, E.R., Black, A.P., 2004. Supporting Java traits in Eclipse. In: ETX. ACM, pp. 37–41.
Reppy, J., Turon, A., 2006. A foundation for trait-based metaprogramming. FOOL/WOOD.
Reppy, J., Turon, A., 2007. Metaprogramming with traits. In: ECOOP. Springer, pp. 373–398.

Schärli, N., Ducasse, S., Nierstrasz, O., Black, A., 2003. Traits: composable units of behavior. In: ECOOP. Springer, pp. 248–274.
Smith, C., Drossopoulou, S., 2005. *Chai*: Traits for Java-like languages. In: ECOOP. Springer, pp. 453–478.
Snyder, A., 1986. Encapsulation and inheritance in object-oriented programming languages. In: OOPSLA. ACM, pp. 38–45.
Sun Microsystems, Inc., 2007. JSR 308: Annotations on java types. http://jcp.org/en/jsr/detail?id=308.
Voelter, M., 2011. Language and IDE modularization and composition with MPS. In: GTTSE. In: LNCS, vol. 7680. Springer, pp. 383–430. doi:10.1007/978-3-642-35992-7_11.

**Lorenzo Bettini**, PhD in in Computer Science, is an Associate Professor in Computer Science at Dipartimento di Statistica, Informatica, Applicazioni, University of Firenze, Italy. His research interests cover design, theory and implementation of programming languages. He is also a consultant for itemis Schweiz. He has been using Xtext since version 0.7. He has used Xtext and Xtend for implementing many Domain Specific Languages and Java-like programming languages. He is the author of about 60 papers on these topics. He is also the author of the book Implementing Domain-Specific Languages with Xtext and Xtend (Packt Publishing). Home page: http://www.lorenzobettini.it.

**Ferruccio Damiani**, PhD in 1998 in Computer Science from the University of Torino, is an associate professor in Computer Science at the Computer Science Department of the University of Torino since 2005. There, he founded and coordinates the System Modelling, Verification and Reuse (MoVeRe, http://di.unito.it/movere/) research group. Previously he has been assistant professor (1999-2005) at the same Department. His research interests include: computational models and languages; concurrent, distributed, and mobile systems; domain specific languages; static and dynamic analysis techniques; systems evolution and dynamic software updates; variability modeling and software product lines. Home page: http://www.di.unito.it/~damiani.