Generic traits for the Java platform

Generic Traits for the Java Platform [*]

Lorenzo Bettini Ferruccio Damiani

University of Torino, Italy
{bettini,damiani}@di.unito.it

Abstract

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. Traits were proposed as a mechanism for fine-grained code reuse to overcome many limitations of class-based inheritance. In this paper we present the extended version of XTRAITJ, a trait-based programming language that features complete compatibility and interoperability with the Java platform. XTRAITJ provides a full Eclipse IDE that aims to support an incremental adoption of traits in existing JAVA projects. This new version fully supports JAVA generics: traits can have type parameters just like in JAVA, so that they can completely interoperate with any existing JAVA library. Furthermore, XTRAITJ now supports JAVA annotations, so that it can integrate with frameworks like JUNIT 4.

Categories and Subject Descriptors D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.2.6 [*Programming Environments*]: Integrated environments; D.2.3 [*Coding Tools and Techniques*]: Object-oriented programming, Program editors; D.3.2 [*Language Classifications*]: Object-oriented languages, Java; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Java, Trait, IDE, Implementation, Eclipse

1. Introduction

Traits were proposed by Schärli et al. [30, 52] as pure units of behavior, aiming to support fine-grained reuse. The main goal of traits is providing a flexible solution to the problems of class-based inheritance with respect to code reuse, since the two traditional roles of classes as object generators and units of code reuse are competing (see, e.g., [26, 30, 45] for discussions and examples). A trait provides a set of methods that is completely independent from any class hierarchy; the rationale is that the common methods of a set of classes can be factored into a trait. Traits can be composed in an arbitrary order leading to a class or another trait. 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* (a mixin is a subclass parameterized over its superclass) [3, 23, 33, 37, 41]—the "trait" construct incorporated in SCALA [47] is indeed a form of mixin. The original proposal of traits [30, 52] 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 literature (see, e.g., [15, 17, 19, 20, 42, 46, 48, 51, 53]).

In most of the above proposals, trait composition and class-based inheritance live together. In some formulations [42, 46, 53] 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 code reuse and the role of type are competing (see, e.g., Snyder [54] and Cook et al. [27]). This does not happen in *pure trait-based programming languages* [17, 19, 20], which aim to maximize the opportunity for reuse:

- class-based inheritance is not present, in order to prevent writing code that might be difficult to reuse, and
- traits are not types, in order to not restrict the flexibility of traits.

This way, the two traditional roles of classes as object generators and units of code reuse are *completely* separated. Classes only play the role of object generators and the role of units of reuse is played only by traits.

These design choices do not reduce the expressivity and usability of the language. In fact, even though class-based inheritance is not present, still type subsumption is supported by JAVA-like interfaces. Moreover, not using trait names as types in the source program does not prevent to analyze 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 [14] we presented the prototype implementation of XTRAITJ, a language for pure trait-based programming. XTRAITJ provides complete compatibility and interoperability with the JAVA type system without reducing the flexibility of traits. XTRAITJ programs are compiled into JAVA programs, which can then be compiled with a standard JAVA compiler. XTRAITJ is implemented with XTEXT [2, 11] that provides a full Eclipse IDE integration, and XBASE [31], a reusable expression language that facilitates full integration 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 existing code-base. This also allows an incremental adoption of traits in existing JAVA projects.

With a year's experience we came to the conclusion that XTRAITJ was not practical without full support of JAVA generics. Not being able to write traits and classes with type parameters limited the possibility of writing reusable code, forcing us to use `Object`

[*] Work partially supported by MIUR (proj. CINA), Ateneo/CSP (proj. SALT), and ICT COST Action IC1201 BETTY. Authors listed in alphabetical order.

in many places (e.g., field declarations and method signatures) and to resort to explicit type casts (thus undermining the static type safety). Moreover, we could not fully use all existing JAVA libraries that use generics. Another thing that we missed in XTRAITJ was the ability to annotate elements of our language with JAVA annotations. This prevented us from reusing many existing JAVA frameworks that are based on annotations, such as, e.g., JUNIT, mocking frameworks and dependency injection frameworks.

For all the above reasons we decided to extend XTRAITJ with respect to the above mentioned features. Adding annotations was not that difficult, while adding generics required much more effort (indeed, adding type parameters to a language is usually time consuming). XBASE provides a nice integration with the JAVA type system, and takes care of type checking expressions, but it requires a mapping of our language elements to a JAVA type model elements (classes, methods, inheritance relations, etc.). In the existing version of XTRAITJ this mapping was involved due to the several JAVA model elements that we need to create to implement the semantics of traits. When adding type parameters to XTRAITJ elements, we had to make sure that in the corresponding mapped JAVA model elements the binding to the original type parameters was solved correctly, otherwise the JAVA type checking implemented by XBASE would not work. This was the part that required much work. When this part was in place, the translation into JAVA code worked as before, and did not require us to change the main translation strategy that was already implemented.

In this paper we present the new extended version of XTRAITJ, with full support of JAVA generics and JAVA annotations. We will discuss design choices, implementation strategy and rationale, and provide examples of use of XTRAITJ with JAVA generics and JAVA annotations. All examples shown in the paper concentrate on the use of these two main new features, showing that we are now able to implement generic traits, classes and generic trait methods (for instance, as we will see in the paper, libraries of generic collection classes) and that we can annotate such elements (in order to use existing JAVA frameworks such as JUNIT to write test cases directly in XTRAITJ). Note that, since our generics (type parameters and generic type arguments) have the same syntax of JAVA generics and are translated directly into JAVA generics, they do not introduce any overhead in the final program. This new version of XTRAITJ also improved IDE tooling such the "Outline View" that now takes generics into consideration, and "Quickfixes" to help the programmer add required fields into a class that uses traits.

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.

Organization of the paper Section 2 recalls the syntax (and informally the semantics) of the XTRAITJ programming language with generics through examples. Section 3 shows some advanced examples in XTRAITJ using JAVA-like generics, functional programming features and JAVA-like annotations. Section 4 outlines the technical details of 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 (concentrating on the new features). Section 5 briefly discusses the pros and cons of the implementation. Section 6 concludes the paper by discussing some related work and outlining possible directions for future work.

2. The XTRAITJ programming language

In this section we describe the main features of XTRAITJ by examples. The syntax of XTRAITJ is given in Table 1, where the question mark symbol '?' and the big parens '$\big($' and '$\big)$' are part of the Extended BNF notation, and the overline notation for (possibly empty) sequences is borrowed from [38]. A program consists of interface declarations, trait declarations, and class declarations. In XTRAITJ, interface declarations ID, method headers H, field declarations F, method declarations M, and class constructors K have a similar syntax as in JAVA (but ignoring, e.g., visibility modifiers and checked exception declarations). All type references can contain type arguments (denoted by GT), and interfaces, traits, classes and method headers can specify type parameters (denoted by TP). 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). The required fields and the required or provided methods of a trait can be directly accessed in the body of the trait's provided methods. Traits can then be used to compose classes and other traits by means of **uses** (*trait sum* operation) and a suite of trait alteration operations. Note that traits do not introduce any state, thus, a class has to provide all the required fields of the traits it uses. A class in XTRAITJ can implement interfaces by using traits and can define fields (possibly with initialization expression) and constructors (but it cannot define methods).

Qualifying a method m as **private** in a trait T, hides the name m and permanently 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 are automatically managed to avoid name clashes and name captures. Note that 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 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. 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 6).

GS	::=	$\texttt{GI} \mid \texttt{GC}$	*source language type*
GI	::=	$\texttt{I}\big(\texttt{<}\overline{\texttt{GT}}\texttt{>}\big)?$	*generic interface reference*
GC	::=	$\texttt{C}\big(\texttt{<}\overline{\texttt{GT}}\texttt{>}\big)?$	*generic class reference*
ID	::=	$\textbf{interface}\ \texttt{I}\big(\texttt{<}\overline{\texttt{TP}}\texttt{>}\big)?\ \big(\textbf{extends}\ \overline{\texttt{GI}}\big)?\ \{\ \overline{\texttt{H}};\}$	*interface*
H	::=	$\big(\overline{\texttt{<TP>}}\big)?\ \big(\texttt{GT} \mid \textbf{void}\big)\ \texttt{m}(\overline{\texttt{GT}\ \texttt{x}})$	*method header*
TD	::=	$\textbf{trait}\ \texttt{T}\big(\texttt{<}\overline{\texttt{TP}}\texttt{>}\big)?\ \big(\textbf{uses}\ \overline{\texttt{TA}}\big)?\ \{\ \overline{\texttt{D}}\ \}$	*trait*
D	::=	$\texttt{F}; \mid \texttt{H}; \mid \big(\textbf{private}\big)?\ \texttt{M}$	*trait member declaration*
TA	::=	$\texttt{T}\big(\texttt{<}\overline{\texttt{GT}}\texttt{>}\big)?\big([\overline{\texttt{ao}}]\big)?$	*trait alteration expression*
ao	::=	$\textbf{alias}\ \texttt{m}\ \textbf{as}\ \texttt{m} \mid \textbf{restrict}\ \texttt{m} \mid \textbf{hide}\ \texttt{m}$	*trait alteration operation*
		$\mid \textbf{rename}\ \texttt{m}\ \textbf{to}\ \texttt{m} \mid \textbf{redirect}\ \texttt{m}\ \textbf{to}\ \texttt{m}$	
		$\mid \textbf{rename}\ \texttt{f}\ \textbf{to}\ \texttt{f} \mid \textbf{redirect}\ \texttt{f}\ \textbf{to}\ \texttt{f}$	
F	::=	$\texttt{GS}\ \texttt{f}$	*field*
M	::=	$\texttt{H}\ \{\cdots\}$	*method*
CD	::=	$\textbf{class}\ \texttt{C}\big(\texttt{<}\overline{\texttt{TP}}\texttt{>}\big)?\ \big(\textbf{implements}\ \overline{\texttt{GI}}\big)?\ \big(\textbf{uses}\ \overline{\texttt{TA}}\big)?\ \{\ \overline{\texttt{FI}};\ \overline{\texttt{K}}\ \}$	*class*
FI	::=	$\texttt{F}\big(\ = \cdots\big)?$	*field initialization*
K	::=	$\texttt{C}(\overline{\texttt{GS}\ \texttt{x}})\ \{\cdots\}$	*constructor*

Table 1. XTRAITJ syntax (including generics)

As stated in the Introduction, XTRAITJ is a pure trait-based programming language, thus, class-based inheritance is not present, so classes only play the role of object generators and types; traits only play the role of units of code reuse and are not types. This is reflected by the syntax of types and type references in Table 1.

XTRAITJ programs are compiled into JAVA source code, which can then be compiled using a standard JAVA compiler. In particular, since we provide Eclipse IDE tooling, when editing XTRAITJ programs from Eclipse, using the XTRAITJ editor, the generated JAVA sources will be automatically compiled into byte code inside Eclipse.

The semantics of traits can be specified in terms of the so called *flattening principle* [30] that prescribes that the semantics of a class that uses traits is equivalent to the semantics of the class obtained by inlining in the body of the class the methods provided by the traits it uses. In general, a *flattening semantics* aims to specify the semantics of traits by describing a way to transform classes and traits that use other traits into equivalent formulations that does not use other traits.[1] The flattening semantics of XTRAITJ can be formally described by means of a translation function that looks up the named traits listed in the **uses** clause and evaluates all the trait composition operations. For simplicity, we have not included such a formal definition in this paper—we refer to [17] for the presentation of the flattening semantics of TRAITRECORDJ, our previous language proposal for integrating traits in Java (the differences between TRAITRECORDJ and XTRAITJ are discussed in [14]).

2.1 Xbase in a nutshell

The method bodies in XTRAITJ are not written in JAVA: we use XBASE for that. XBASE [31] is an extendable and reusable expression language developed with XTEXT. It integrates tightly 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 more advanced features (e.g., lambda expressions). We refer to XBASE documentation [2], here we only describe the main differences with respect to JAVA, so that the reader can understand the examples shown in the paper.

Variable declarations use a different syntax and do not require to specify the type if it can be inferred from the initialization expression. For example,

```
val l = new ArrayList<String>();
var s = "foo";
```

correspond to the following JAVA variable declarations:

```
final ArrayList<String> l = new ArrayList<String>();
String s = "foo";
```

A cast expression in XBASE is written using the infix keyword `as`, thus, instead of writing "`(C) e`" we write "`e as C`".

The other interesting feature provided by XBASE is *lambda* expressions, which have the shape `[ p1, p2, ... | body ]`. XBASE automatically translates lambda expressions into JAVA anonymous classes. Moreover, if the runtime JAVA library is version 8, then XBASE automatically compiles its lambda expressions into JAVA 8 lambda expressions. XBASE also introduces *function types* for lambda expressions, which have the shape `(param types) => return type`. For example,

```
val (int, int)=>String f = [ x, y | x+y+"" ]
```

[1] A flattening semantics aims to provide a specification, it is not an especially effective implementation technique.

Note that in XBASE type inference works also the other way: since we specified the (function) type in the variable declaration, we do not need to specify the types of lambda parameters.

A lambda expression can be evaluated using the method `apply` and passing all the required arguments, for example

```
val result = f.apply(1, 2)
```

2.2 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 summed 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. Therefore it would be useless (and hence it is forbidden) to declare any of them as required.

2.3 Generic traits and classes

Consider the task of developing a class `CStack` that implements the generic interface[2]:

```
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;
```

[2] Whether we are using `T` to refer to a trait name or to a type parameter should be clear from the context.

```
import static org.junit.Assert.*;
import org.junit.Test;

public class StackTest {
    @Test public void testEmptyStack() {
        IStack<String> stack = new CStack<String>();
        assertTrue(stack.isEmpty());
        assertNull(stack.pop());
    }
    @Test public void testPushAndPop() {
        IStack<String> stack = new CStack<String>();
        stack.push("foo");
        stack.push("10");
        assertEquals("10", stack.pop());
        assertEquals("foo", stack.pop());
    }
}
```

Listing 1: Using JUNIT for testing XTRAITJ generated JAVA code

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

    boolean isEmpty() { return collection.size() == 0; }
    void push(T e) { 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);
    }
}
```

We can then use this XTRAITJ class inside trait methods, for example we can create instances of that, after instantiating type parameters:

```
new CStack<String>().push("my string");
new CStack<Integer>().push(10);
```

Of course, all these expressions will be type checked according to instantiated generic types, as in JAVA.

In the above example we also show how XTRAITJ code can seamlessly refer to any existing JAVA type (in this example we use the standard collections).

Since XTRAITJ programs are compiled into JAVA source code, we can also seamlessly use XTRAITJ classes into standard JAVA programs[3]. This means that we can use existing frameworks like JUNIT, and write JAVA unit tests for our XTRAITJ code, as shown in Listing 1 (in Section 3.3 we will show that we can write JUNIT tests directly in XTRAITJ).

The use of generics in XTRAITJ reflects JAVA generics, so it should be immediately clear to a JAVA programmer how to declare type parameters and type arguments. In the following example we declare a class parameterized on a bounded type variable (note the F-bounded quantification [25]) and pass as type argument to interface and trait references a generic type parameterized on such type variable:

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

class CStackOfSetsOfComparable<T extends Comparable<T>>
            implements IStack<Set<T>> uses TStack<Set<T>> {
    List<Set<T>> collection = new ArrayList();
}
```

2.4 Trait operations by examples

Let us now suppose that a class implementing the following interface should be developed:

```
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 break the type system. The same problem would arise with the formulations of traits where traits are types (see, e.g., [42, 46, 53]), with JAVA 8 interfaces with default methods (see the programming patterns proposed in [21]), 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 operations to reuse code and avoid method conflicts.

We 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> { ... }
```

We can further improve the implementation of `TLifo`. In fact, in the above implementation of `TLifo`’s `pop` method, we do not reuse the implementation of `TStack`’s `pop`[4]. Another thing that does not

[3] Of course, we mean “use the corresponding generated JAVA classes/interfaces”, which have the same name of the XTRAITJ source elements.

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

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** `n1` **to** `n2`], where the method (or field) `n1` must be declared by `T` and the method (resp. field) `n2` must not be declared by `T`, denotes the trait obtained from `T` by replacing all the references to `n1` with (possibly implicit) receiver **this** by references to `n2` and by changing the declaration of `n1` into a declaration of `n2`. 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] { ... }
```

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 shows the high and 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 used in the **uses** clause, `TLifo`;
- `TNegate` provides the method `notOp`, but since we rename it to `isNotEmpty`, then 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`.

```
trait TIterableExtensions<T> {
    Iterable<T> iterable;

    T head() {
        val iterator = iterable.iterator();
        if (iterator.hasNext())
            return iterator.next();
        return null;
    }

    T last() {
        // optimized according to the iterable type
        if (iterable instanceof List<?>) {
            val list = iterable as List<T>
            if (list.isEmpty())
                return null;
            return list.get(list.size() − 1);
        } else if ... // other optimizations
        } else {
            var T result = null;
            for (T t : iterable)
                result = t;
            return result;
        }
    }
}
```

Listing 2: The trait implementing utility methods for `Iterable` JAVA objects.

XTRAITJ provides other alteration operations (see Table 1) that we will not use in this paper. We refer to [14] and `http://xtraitj.sf.net` for the complete list of alteration operations.

3. XTRAITJ at work

In this section we show some examples of the new programming features of XTRAITJ, and how they can be used in a JAVA programming context. We refer to XTRAITJ implementation site for other examples: `http://xtraitj.sf.net`.

3.1 Utility Methods for Collections

Using the new XTRAITJ trait features we can easily implement utility methods for JAVA collections and iterables thanks to the full JAVA generics support. The implementation we show in the following is inspired by the utility static methods for iterable collections provided by the *Google Guava* JAVA library.

In Listing 2 we show the trait implementing utility methods for a JAVA generic `Iterable` object. First, we only show two simple utility methods returning the first element and the last element of an iterable object (these methods are not part of the JAVA `Iterable` interface). In particular, for the method `last` we have specialized and optimized implementations for specific iterable types (again, inspired by the implementation of the Guava library).

We can easily implement a trait, `TList` that acts as a wrapper for a standard JAVA list (i.e., that requires a list field, and forwards to that field all the methods of the JAVA `List` interface); for the lack of space we will not show this trait here.

We then assemble a class implementing the JAVA `List` interface using `TList` and `TIterableExtensions`. This class represents a `List` with the additional methods of `TIterableExtensions`:

```
class StringListWithExtensions implements List<String> uses
        TIterableExtensions<String>, TList<String>
{
    List<String> list; // required by TList
    Iterable<String> iterable; // required by TIterableExtensions

    StringListWithExtensions(String..args) {
        this.list = new ArrayList();
        // ... add args to the list
        this.iterable = this.list;
    }
}
```

Another utility method that we can implement in `TIterable-Extensions` is `join` to return a string representation with a specific separator:

```
// in TIterableExtensions<T>
String join(CharSequence separator) {
    val result = new StringBuilder();
    val iterator = iterable.iterator();
    while (iterator.hasNext()) {
        result.append(iterator.next().toString());
        if (iterator.hasNext())
            result.append(separator);
    }
    return result.toString();
}
```

3.2 Functional Programming

As an example of generic method and functional programming, we also implement `mapToList` that returns a `List` version of the iterable after applying a mapper function (the type of the elements of the returned list might be different from the type of the iterable's elements); note that we use the XBASE syntax for specifying a functional type (see Section 2.1):

```
// in TIterableExtensions<T>
<R> List<R> mapToList((T) => R mapper) {
    val result = new ArrayList<R>();
    for (e : iterable)
        result += mapper.apply(e);
    return result;
}
```

We can also implement a more general version of the popular functional programming concept called *map*, which allows to transform a collection into another one, applying a mapping function to each element. The above implementation of `mapToList` can be seen as an *eager* implementation of mapping; now we implement a *lazy* version of mapping that returns a new `Iterable`: the transformation will actually take place only when iterating over the new iterable.

In order to implement this, we need an implementation of a JAVA `Iterator` in XTRAITJ, that we will use to implement a custom version of the `next` method. We then write a "wrapper" trait:

```
trait TIterator<E> {
    Iterator<E> iterator;

    boolean hasNext() { return iterator.hasNext(); }
    E next() { return iterator.next(); }
    void remove() { iterator.remove(); }
}
```

Then we write a transformer iterator as follows[5]:

```
trait TTransformerIterator<T,R>
            uses TIterator<T>[rename next to origNext] {
    (T) => R function;

    R next() { return function.apply(origNext()); }
}
```

[5] We could have avoided to write the wrapper trait `TIterator` and implement everything in `TTransformerIterator`; however, since we may want to implement other customized iterators, we preferred this solution, which is more code reuse oriented.

Finally, the following class acts as the bridge between the JAVA iterator interface and our implemented traits:

```
class TransformerIterator<T,R>
        implements Iterator<R>
        uses TTransformerIterator<T,R> {

    Iterator<T> iterator;
    (T) => R function ;

    TransformerIterator(Iterator<T> iterator, (T) => R function) {
        this.iterator = iterator
        this.function = function
    }
}
```

This class implements an `Iterator` for the transformed iterables (note the type parameter `R`). We can now implement:

```
// in TIterableExtensions<T>
<R> Iterable<R> map((T) => R mapper) {
    return [| new TransformerIterator(iterable.iterator(), mapper)];
}
```

The above method implementation uses an XBASE feature related to lambda expressions: `Iterable` is a JAVA interface with only one method, i.e., a *SAM* type (Single Abstract Method type), and when such a type is expected, we can instead specify a lambda expression that is meant to implement such abstract method. In this example the abstract method is `Iterator iterator()` and we specify a lambda expression without parameters returning an iterator (note that type inference for generics is automatic). Basically, XBASE will generate the following JAVA code for the above return statement, using an anonymous class:

```
return new Iterable<R>() {
  public Iterator<R> iterator() {
    return new TransformerIterator<T, R>(iterable.iterator(), mapper);
  }
};
```

This useful feature of XBASE predated the use of JAVA 8 lambda expressions in the context of SAM types, also called *functional interfaces*. For example, in JAVA 8, we could have written[6]

```
return () -> new TransformerIterator<T, R>(iterable.iterator(), mapper);
```

Following similar techniques we can implement other utility methods based on lambda expressions, such as, e.f., `filter`, `forEach`, etc.

3.3 Writing JUNIT tests

Together with generics, the new version of XTRAITJ also supports JAVA annotations: one can annotate a trait's provided method and a class' field[7]. Using annotations we can easily implement JUNIT tests directly in XTRAITJ, as shown in Listing 3 (note that XTRAITJ supports *static* imports as in JAVA).

Thanks to traits compositionality features, we can easily separate the actual test cases from test initialization: both traits require the same field used as test fixture[8], and we can write several classes executing the same tests (implemented in `TStackTestCase`) using different fixtures, i.e., implementing variations of setup traits (`TStackAsArrayListSetup`, etc.):

```
class CStackAsArrayListTest
            uses TStackTestCase, TStackAsArrayListSetup {
```

[6] As hinted in Section 2.1, XBASE is fully compliant with JAVA 8.

[7] We did not find any benefit in annotating required methods and fields; not to mention in such cases annotations would make conflict resolution harder to handle, thus we ruled out such situations.

[8] In tests, a *fixture* refers to the fixed state used as a baseline for tests.

```
import static org.junit.Assert.*

trait TStackTestCase {
    IStack<String> fixture;

    @Test void testNotEmpty() {
        assertFalse(fixture.isEmpty());
    }
    @Test void testContents() {
        assertEquals("foo", fixture.pop());
        assertEquals("bar", fixture.pop());
        assertNull(fixture.pop());
    }
}

trait TStackAsArrayListSetup {
    IStack<String> fixture;
    @Before void setup() {
        fixture = new CStack<String>(newArrayList("foo", "bar"));
    }
}

trait TStackAsLinkedListSetup {
    IStack<String> fixture;
    @Before void setup() {
        fixture = new CStack<String>(newLinkedList("foo", "bar"));
    }
}
```

Listing 3: Implementing JUNIT tests in XTRAITJ

```
    IStack<String> fixture;
}
```

The corresponding generated JAVA class can be executed as a JUNIT test (as a future extension, we will implement the feature to run a JUNIT test in Eclipse directly on the original XTRAITJ file).

The *Dependency Injection* pattern [34] is based on "injecting" actual implementation classes into a class hierarchy in a consistent way. This is useful when classes delegate specific functionalities to other classes. For example messages are forwarded to the objects referenced in fields, which abstract the actual behavior. These fields are instantiated through injection mechanisms so that implementation classes' names are not hardcoded in the code. With respect to manual implementation of existing patterns [35], with *dependency injection frameworks* it is much easier to keep the desired consistency, and the programmer needs to write less code. Google Guice [1] uses JAVA annotations, `@Inject`, for specifying the fields that will be injected, and a *module* is responsible for configuring the bindings for the actual implementation classes. We can then use Google Guice also in XTRAITJ code; in particular, we can annotate class' fields with the `@Inject` annotation and have such fields injected using different Guice module implementations in different scenarios.

4. 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.

4.1 Design choices

In [14] we described in details all the design choices we followed when implementing XTRAITJ; here we just summarize the main ones, and we refer the interested reader to [14] for full details (also concerning the comparison with the calculus TRAITRECORDJ [17], which was the starting point for the implementation of XTRAITJ).

XTRAITJ traits have been designed with the goals of being as much as possible compliant to the characteristics of the original formulation of traits [30], namely the complete conflict resolution control on compositionality of traits and their lightweight mechanisms with an intuitive semantics. Moreover, 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 then 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 [2, 11] and XBASE [31].

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 [36] and Batov's work [4] 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 this new version of XTRAITJ we introduced JAVA annotations, so that we are able to use all its benefits; a clear example is the possibility to write JUNIT tests in XTRAITJ, as shown in Section 3.3.

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 an 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. It is not even mandatory to use traits everywhere, since XTRAITJ code seamlessly coexists with JAVA code. As a demonstration of the complete integration with the JAVA type system, in the implementation of XTRAITJ we did not include interface specifications: XTRAITJ programs can seamlessly use existing JAVA interfaces. Indeed, 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.

Although an IDE is not a strict requirement to develop applications, still it surely helps programmers a lot with features like syntax aware editor, compiler and debugger integration, build automation and code completion, just to mention a few. In an agile [43] and test-driven context [5] the features of an IDE like Eclipse become essential and they dramatically increase productivity. With this respect, XTEXT provides a complete solution for the development of new languages, since it also deals with all the artifacts related to 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 4.4).

4.2 Translation to JAVA

In this section we sketch the main steps we used to implement XTRAITJ. By using XBASE we inherit not only the syntax of JAVA-like expressions, but also all its language infrastructure components, like its type checking implementation and the compiler generating JAVA code. XBASE type system is completely integrated with the JAVA type system, thus, also a language using XBASE will be compatible with JAVA and its type system (including generics): the language will be able to seamlessly access all the JAVA types, i.e., any existing JAVA library. However, XBASE only deals with expressions: language features like traits, classes, field and method declarations are dealt with directly by XTRAITJ; method bodies, instead, completely rely on XBASE expressions.

In order to reuse 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 to 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.

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 the following is implied by our implementation of the model inferrer for XTRAITJ.

4.2.1 The basic idea of the translation

We will now informally sketch the generated JAVA code corresponding to XTRAITJ programs using some examples. For a more detailed description we refer the interested reader to [14]. 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.

Our strategy for generating the JAVA code for traits and classes is based on a crucial property: there will be exactly one JAVA interface and one JAVA class for each trait and class declaration; each original method body in each trait will have exactly one corresponding generated JAVA method. The generated JAVA code then will enjoy *compositionality*. Indeed, trait compositions are implemented through object composition and method delegation.

Let us consider this trait definition (here we are only considering 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) {
        val r = this.m(f1);
        return r.get(0);
    }
}
```

From this trait definition the following JAVA interface is generated:

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

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.

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> _delegate;

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

  public U getF() { return _delegate.getF(); }
  public void setF(final U f) { _delegate.setF(f); }
  public U m(final U f1) { return _delegate.m(f1); }
  public U _m(final U f1) {
    this.setF(f1);
    return this.getF();
  }

  public V n(final U f1) { return _delegate.n(f1); }
  public V _n(final U f1) {
    final U r = this.m(f1);
    return r.get(0);
  }
}
```

The important thing in the generated JAVA class is the `_delegate` 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 delegated to the field `_delegate`, 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 `_delegate` because this allows a standard JAVA class to override methods and makes sure that the standard JAVA dynamic binding mechanism for overridden methods still works.

Let us now consider the XTRAITJ class definition:

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

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 is that `C` delegates to `T1Impl` for the methods defined in the trait, and `T1Impl` delegates to `C` for the fields required in the trait.

The generated JAVA class for the class definition in XTRAITJ 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.

4.2.2 Dealing with alteration operations

Alteration operations in trait sums will not correspond to copies and modifications of the original trait methods bodies: we will deal with such operations in the generated JAVA code by generating "adapter" interfaces and classes (see Section 5 for the discussion about trade-offs of our implementation).

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 interface will be different: Alteration operations introduce changes in the interface of the new trait that make it incompatible with the interface of the original trait. We refer the interested reader to [14] for further details about the treatment of alteration operations in our implementation.

4.3 Validation

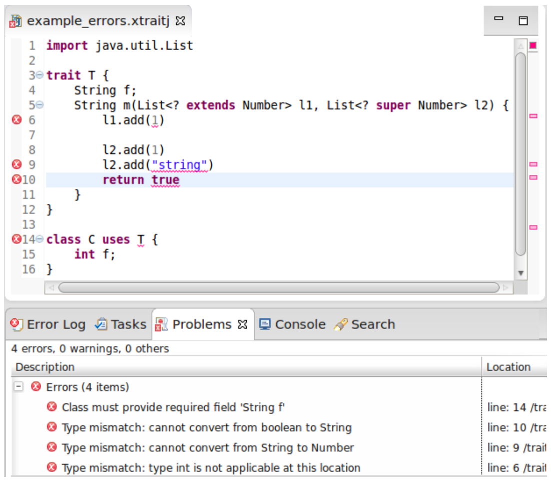
Since we provide a mapping from a trait method to a JAVA method, then XBASE is able to automatically type-check the expression of the trait method (e.g., using the return type of the method and the types of the parameters). This works since the types that we use in a XTRAITJ program are actually references to JAVA types. Thus, we completely delegate the type-checking of method bodies to XBASE. Reusing the type system implementation 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 4.2.1. The introduction of generics in this new version of XTRAITJ raised many issues with 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 parts, in order to make XBASE's type system work in the presence of generics.

In Figure 1 we show some type errors reported by XBASE. All the checks concerning method conflicts are instead implemented by us; these also include the check that a class provides all fields and methods required by the used traits (in Figure 1 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).

4.4 IDE

One of our main design choices and goals is the integration of our language in Eclipse (see Section 4.1). With this respect, XTEXT and XBASE enhance the Eclipse tooling concerning the integration with JAVA. For instance, we can navigate to a JAVA type definition directly from an XTRAITJ program, see its type hierarchy, and 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 (see Figure 2).

A well-known problem with implementations that generate JAVA code is that you can only debug the generated JAVA code that is usually quite different from the original program. Our XTRAITJ

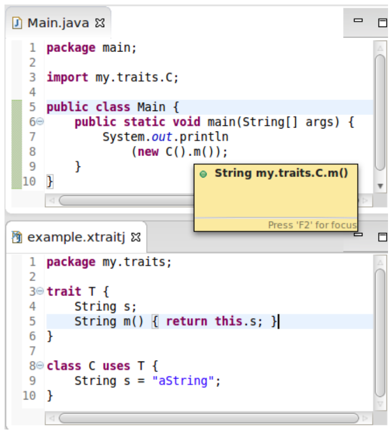


```
 1 import java.util.List
 2
 3 trait T {
 4     String f;
 5     String m(List<? extends Number> l1, List<? super Number> l2) {
 6         l1.add(1)
 7
 8         l2.add(1)
 9         l2.add("string")
10         return true
11     }
12 }
13
14 class C uses T {
15     int f;
16 }
```

4 errors, 0 warnings, 0 others

Description	Location
Errors (4 items)	
Class must provide required field 'String f'	line: 14 /tra
Type mismatch: cannot convert from boolean to String	line: 10 /tra
Type mismatch: cannot convert from String to Number	line: 9 /trai
Type mismatch: type int is not applicable at this location	line: 6 /trai

Figure 1. Errors reported in the IDE.



```
package main;

import my.traits.C;

public class Main {
    public static void main(String[] args) {
        System.out.println
            (new C().m());
    }
}
```

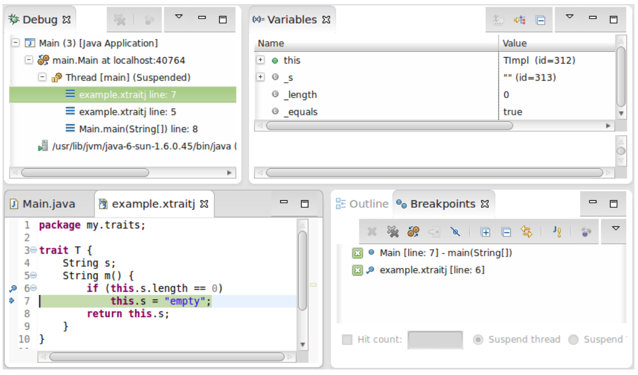
String my.traits.C.m()

```
package my.traits;

trait T {
    String s;
    String m() { return this.s; }
}

class C uses T {
    String s = "aString";
}
```

Figure 2. Accessing to XTRAITJ method definition from JAVA code.

implementation does not have this drawback: thanks to XBASE we can debug the original XTRAITJ code. In Figure 3 we show a debug session of a JAVA program that uses code generated by XTRAITJ: we have set a break point on a 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 also the additional forward methods.

We have customized the "Outline" view for the XTRAITJ editor: besides fields and method declarations, the view also has a special nodes, called "requirements" and "provides", respectively, that show all the required fields and methods and provided methods that come from used traits. This mechanism takes into consideration possible rename operations. The "Outline" view is also synchronized with the editor contents, so that it is updated on-the-fly. Moreover, clicking on any node of the view automatically selects the corresponding element in the editor. Finally, instantiated type parameters are taken into consideration in the outline representation.



Debug ☒

```
Main (3) [Java Application]
  main.Main at localhost:40764
    Thread [main] (Suspended)
      example.xtraitj line: 7
      example.xtraitj line: 5
      Main.main(String[]) line: 8
  /usr/lib/jvm/java-6-sun-1.6.0.45/bin/java (
```

Variables ☒

Name	Value
this	TImpl (id=312)
_s	"" (id=313)
_length	0
_equals	true

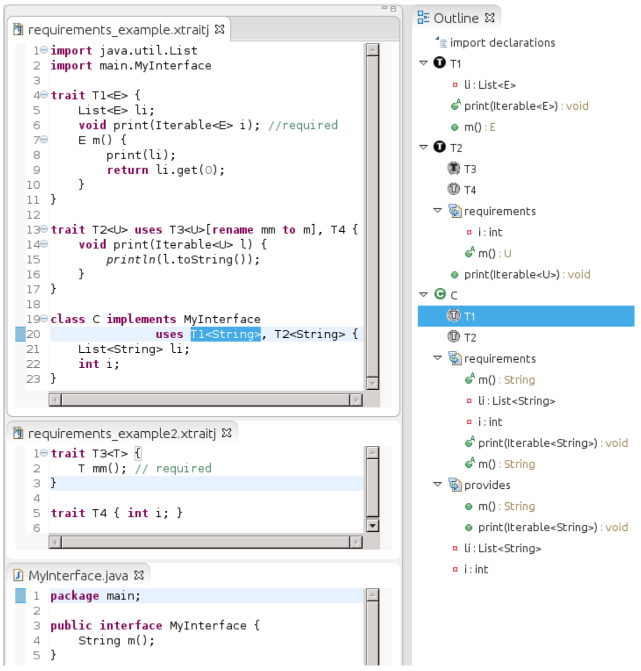
Main.java | example.xtraitj ☒

```
package my.traits;

trait T {
    String s;
    String m() {
        if (this.s.length == 0)
            this.s = "empty";
        return this.s;
    }
}
```

Outline | Breakpoints ☒

```
Main [line: 7] - main(String[])
example.xtraitj [line: 6]
```

Figure 3. Debugging XTRAITJ code.



requirements_example.xtraitj ☒

```
import java.util.List
import main.MyInterface

trait T1<E> {
    List<E> li;
    void print(Iterable<E> i); //required
    E m() {
        print(li);
        return li.get(0);
    }
}

trait T2<U> uses T3<U>[rename mm to m], T4 {
    void print(Iterable<U> l) {
        println(l.toString());
    }
}

class C implements MyInterface
                uses T1<String>, T2<String> {
    List<String> li;
    int i;
}
```

requirements_example2.xtraitj ☒

```
trait T3<T> {
    T mm(); // required
}

trait T4 { int i; }
```

MyInterface.java ☒

```
package main;

public interface MyInterface {
    String m();
}
```

Outline ☒

```
import declarations
T1
    li : List<E>
    print(Iterable<E>) : void
    m() : E
T2
    T3
    T4
    requirements
        i : int
        m() : U
    print(Iterable<U>) : void
C
    T1
    T2
    requirements
        m() : String
        li : List<String>
        i : int
        print(Iterable<String>) : void
        m() : String
    provides
        m() : String
        print(Iterable<String>) : void
    li : List<String>
    i : int
```

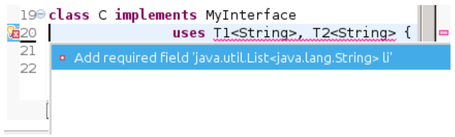
Figure 4. Outline view showing all the requirements.

Figure 4 shows this view in action with some examples. For instance, for `T2` it shows as requirements the field `i` (declared in the used trait `T4`) and the method `m` (declared in the used trait `T3`). In particular, the latter actually refers to the renamed method `mm`, and clicking on the outline node for `m` will actually bring the programmer to the definition of the original method `mm` in `T3` (which is in a different file).

For classes, this special "requirements" node will also list all the methods of the implemented interfaces of the class (see the class `C` in Figure 4). Clicking on such nodes will open the corresponding JAVA editor. Note that since the class in the example provides type arguments for type parameters, the outline shows required methods and provided methods that come from generic traits with type parameters instantiated accordingly.

We believe this is an extremely useful feature, especially when the **uses** or **implements** relations spawn several traits and interfaces, possibly stored in different files.

In this new version of XTRAITJ we also offer quickfixes to automatically insert fields required by the used traits in a class; also



```
19 class C implements MyInterface
20                 uses T1<String>, T2<String> {
21
22
```

Add required field 'java.util.List<java.lang.String> li'

Figure 5. Quickfix for adding missing required fields.

in this case we take into consideration possible instantiated type parameters. For example, let us remove the declaration of the field `li` from the class `C` in Figure 4; An error will be reported about the missing requirement, marking the trait expression that introduces such requirement, and a quickfix is available to add the missing required field (where the generic type is correctly instantiated) as shown in Figure 5.

Currently, there is also refactoring support for names, including names of methods, traits and classes, including generic type parameters renaming. This is the default renaming support provided by XTEXT and XBASE and it works also across files. We are investigating about adding further refactoring mechanisms to extract methods into separate traits (possibly by integrating such mechanisms with the ones proposed in [12], see also Section 6).

5. Evaluation

The flattening principle sketched in Section 2 allows to describe the semantics of traits, but it is not necessarily 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 be hardly used to implement libraries since 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 are using that specific method would need to be regenerated (for instance, this is the approach of [49]).

Instead, our implementation is modular with this respect. In fact, as hinted in Section 4.2, each method body is translated into exactly one single JAVA method body. Even alteration operations do not require 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. This implies that our JAVA code generation is compositional in the presence of alteration operations: we reuse the already generated JAVA code and we delegate 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.

The translation strategy, which has been used since the beginning of the implementation of XTRAITJ, has been adapted to the addition of generics and annotations. As hinted in Section 4.3, the most difficult part in adding generics to XTRAITJ has been to correctly bind generic type references to the type parameters. This usually takes place automatically in languages that use XBASE, because each DSL element is mapped to a single JAVA model element by the model inferrer. This is not the case in XTRAITJ (because of traits methods delegation and alteration operations), so we had to implement a custom scoping mechanism to bind generic types correctly. This is a strict requirement to make XBASE type system work correctly.

Finally, the generated JAVA code has no dependency on XTRAITJ (indeed, we do not need any runtime library that is specific of our traits): it only depends on XBASE library and Google Guava library. These two libraries 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 delegation is due to the overhead of the method forwarding. However, in order 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 delegation (including techniques to simulate multiple inheritance, such as, e.g., [18]). As future work we plan to investigate possible tweaks to the code generation so that the generated JAVA code can be more easily optimized by the JAVA compiler. To this aim, we will use micro benchmarks to test the possible code generation strategies.

Summarizing, we believe that the properties of our implementation described above, namely, compositionality, modularity and IDE tooling, compensate consistently the runtime overhead of method forwarding. With this respect, also the implementation described in [53] uses delegation to enjoy the mentioned advantages.

Finally, 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 programs using generics. With this respect, according to the *type erasure* model [24], generic types are removed during the compilation and are not present in the generated byte-code. The performance will then be the same of a program using raw types.

6. Conclusions, related and future work

In this paper we presented the extended version of XTRAITJ with full support for JAVA generics: traits can have type parameters just like in JAVA. Furthermore, XTRAITJ now supports JAVA annotations, so that it can integrate with frameworks like JUNIT 4. This highly extended the interoperability of XTRAITJ with the JAVA platform with respect to the previous version of XTRAITJ. In the rest of this Section we will discuss some further related work and hints possible future directions.

The literature on traits and JVM-compatible languages has been partially quoted and compared through the paper. In Section 5 we also compared our implementation strategy with other implementations of traits; of course we only considered implementations where "trait" refers to the original formulation of traits [30], that is, we cannot compare our implementation with languages where traits have a different meaning (e.g., traits in SCALA [47] or traits in C++). In the rest of this section we add further discussions and comparisons.

In the original formulation of traits [30] the methods provided by a trait can only access state 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. [10] 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 [32], 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. We decided to require to 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 eliminates any burden to the programmer by a quickfix to automatically generate the declarations of all the fields required by the traits used by a class (as shown in Figure 5, Section 4.4).

Some of the module composition operations present in Bracha's JIGSAW framework [22] 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. [39]. 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 [44]. Method renaming for traits has been introduced by Reppy and Turon [50].

Reppy and Turon [51] also proposed a variant of traits that can be parameterized 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. It could be interesting to add to XTRAITJ a similar feature, which enhances the code reuse already provided by traits. This will be the subject of future work.

Reverse generics [7] is 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 [16] we introduced *parametric traits*, that is, traits that are parameterized 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 parameter 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 [24], 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., [8, 9]). Since XTRAITJ targets the JAVA platform, we have to share its limitations with that respect.

Dynamic trait replacement [13, 53] 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.

In [12], a tool is presented for identifying the methods in a JAVA class hierarchy that could be good candidates to be refactored in traits; this was an adaptation of the SMALLTALK analysis tool of [40] 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, [26]).

In [28, 29] a compositional proof systems for the verification of pure traits is presented. We plan to extend the KeY system [6] for deductive verification of JAVA programs to XTRAITJ by implementing a proof system similar to the one proposed in [29].

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 XBASE expressions.

Acknowledgments

We thank the anonymous PPPJ'14 referees for insightful comments and suggestions for improving the presentation.

References

[1] Google guice. `http://code.google.com/p/google-guice`.

[2] Xtext. `http://www.eclipse.org/Xtext`.

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM TOPLAS*, 25(5):641–712, September 2003.

[4] V. Batov. Java generics and C++ templates. *C/C++ Users Journal*, 22(7):16–21, 2004.

[5] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[6] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

[7] A. Bergel and L. Bettini. Reverse Generics: Parametrization after the Fact. In *Software and Data Technologies*, volume 50 of *Communications in Computer and Information Science*, pages 107–123. Springer, 2011.

[8] A. Bergel and L. Bettini. Generics and Reverse Generics for Pharo. In *ICSOFT*, pages 363–372. SciTePress, 2012.

[9] A. Bergel and L. Bettini. Generic Programming in Pharo. In *Software and Data Technologies*, volume 411 of *Communications in Computer and Information Science*, pages 66–79. Springer, 2013.

[10] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[11] L. Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[12] L. Bettini, V. Bono, and M. Naddeo. A trait based re-engineering technique for Java hierarchies. In *PPPJ*, pages 149–158. ACM, 2008.

[13] L. Bettini, S. Capecchi, and F. Damiani. On flexible dynamic trait replacement for Java-like languages. *Science of Computer Programming*, 78(7):907–932, 2013.

[14] L. Bettini and F. Damiani. Pure trait-based programming on the Java platform. In *PPPJ*, pages 67–78. ACM, 2013.

[15] L. Bettini, F. Damiani, K. Geilmann, and J. Schäfer. Combining traits with boxes and ownership types in a Java-like setting. *Science of Computer Programming*, 78(2):218–247, 2013.

[16] L. Bettini, F. Damiani, and I. Schaefer. Implementing type-safe software product lines using parametric traits. *Science of Computer Programming*, 2013. In press, `http://dx.doi.org/10.1016/j.scico.2013.07.016`.

[17] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. TraitRecordJ: A programming language with traits and records. *Science of Computer Programming*, 78(5):521–541, 2013.

[18] L. Bettini, M. Loreti, and B. Venneri. On Multiple Inheritance in Java. In *Technology of Object-Oriented Languages, Systems and Architectures*, volume 732 of *The Kluwer International Series in Engineering and Computer Science*, pages 1–15. Springer, 2003.

[19] V. Bono, F. Damiani, and E. Giachino. Separating Type, Behavior, and State to Achieve Very Fine-grained Reuse. In *Electronic proceedings of FTfJP*, 2007.

[20] V. Bono, F. Damiani, and E. Giachino. On Traits and Types in a Java-like setting. In *TCS (Track B)*, volume 273 of *IFIP*, pages 367–382. Springer, 2008.

[21] V. Bono, E. Mensa, and M. Naddeo. Trait-oriented programming in Java 8. In *PPPJ*. ACM, 2014. `http://dx.doi.org/10.1145/2647508.2647520`.

[22] G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[23] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA*, volume 25(10), pages 303–311. ACM, 1990.

[24] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *OOPSLA*, volume 33(10) of *ACM SIGPLAN Notices*, pages 183–200, Oct. 1998.

[25] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded Polymorphism for Object-oriented Programming. In *FPCA*, pages 273–280. ACM, 1989.

[26] D. Cassou, S. Ducasse, and R. Wuyts. Traits at work: The design of a new trait-based stream library. *Comput. Lang. Syst. Struct.*, 35(1):2–20, 2009.

[27] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *POPL*, pages 125–135. ACM, 1990.

[28] F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer. Verifying traits: A proof system for fine-grained reuse. In *FTfJP*, pages 8:1–8:6. ACM, 2011.

[29] F. Damiani, J. Dovland, E. B. Johnsen, and I. Schaefer. Verifying traits: an incremental proof system for fine-grained reuse. *Formal Aspects of Computing*, 26(4):761–793, 2014.

[30] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[31] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing Domain-Specific Languages for Java. In *GPCE*, pages 112–121. ACM, 2012.

[32] K. Fisher and J. Reppy. A typed calculus of traits. In *FOOL*, 2004.

[33] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL*, pages 171–183. ACM, 1998.

[34] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://www.martinfowler.com/articles/injection.html`, Jan. 2004.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[36] D. Ghosh. Generics in Java and C++: a comparative model. *ACM SIGPLAN Notices*, 39(5):40–47, May 2004.

[37] J. Hendler. Enhancement for multiple-inheritance. In *Object-Oriented Programming Workshop*, pages 98–106. ACM, 1986.

[38] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[39] G. Lagorio, M. Servetto, and E. Zucca. Featherweight Jigsaw - Replacing inheritance by composition in Java-like languages. *Information and Computation*, 214(0):86 – 111, 2012.

[40] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *ASE*, pages 66–75. IEEE, 2005.

[41] M. Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: A solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[42] L. Liquori and A. Spiwack. FeatherTrait: A Modest Extension of Featherweight Java. *ACM TOPLAS*, 30(2):11:1–11:32, 2008.

[43] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2003.

[44] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[45] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. Removing duplication from java.io: a case study using traits. In *OOPSLA*, pages 282–291. ACM, 2005.

[46] O. Nierstrasz, S. Ducasse, and N. Schärli. Flattening traits. *JOT*, 5(4):129–148, 2006.

[47] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

[48] P. J. Quitslund. Java Traits — Improving Opportunities for Reuse. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, Sept. 2004.

[49] P. J. Quitslund, E. R. Murphy-Hill, and A. P. Black. Supporting Java traits in Eclipse. In *ETX*, pages 37–41. ACM, 2004.

[50] J. Reppy and A. Turon. A Foundation for Trait-based Metaprogramming. In *FOOL/WOOD*, 2006.

[51] J. Reppy and A. Turon. Metaprogramming with traits. In *ECOOP*, volume 4609 of *LNCS*, pages 373–398. Springer, 2007.

[52] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *ECOOP*, volume 2743 of *LNCS*, pages 248–274. Springer, 2003.

[53] C. Smith and S. Drossopoulou. *Chai*: Traits for Java-like languages. In *ECOOP*, volume 3586 of *LNCS*, pages 453–478. Springer, 2005.

[54] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA*, volume 21(11), pages 38–45. ACM, 1986.