A Prototypical Java-like Language with Records and Traits

Lorenzo Bettini\textsuperscript{1} Ferruccio Damiani\textsuperscript{1} Ina Schaefer\textsuperscript{2} Fabio Strocco\textsuperscript{1}
\textsuperscript{1}Dipartimento di Informatica, Universit\`a di Torino, C.so Svizzera, 185 - 10149 Torino, Italy
\textsuperscript{2}Chalmers University of Technology, 421 96 Gothenburg, Sweden

Abstract
Traits have been designed as units of fine-grained behavior reuse in the object-oriented paradigm. In this paper, we present the language \textsc{Sugared Welterweight Record-Trait Java} (\textsc{SWRTJ}), a \textsc{Java} dialect with \textit{records} and \textit{traits}. Records have been devised to complement traits for fine-grained state reuse. Records and traits can be composed by explicit linguistic operations, allowing code manipulations to achieve fine-grained code reuse. Classes are assembled from (composite) records and traits and instantiated to generate objects. We present the prototypical implementation of \textsc{SWRTJ} using \textsc{XT}ext, an Eclipse framework for the development of programming languages as well as other domain-specific languages. Our implementation comprises an Eclipse-based editor for \textsc{SWRTJ} with typical IDE functionalities, and a stand-alone compiler, which translates \textsc{SWRTJ} programs into standard \textsc{Java} programs.

Categories and Subject Descriptors
D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.6 [Programming Environments]: Integrated environments; D.3.3 [Programming Languages]: Language Constructs and Features; F.3.3 [Studies of Programming Constructs]: Type Structure

General Terms Design, Languages

Keywords Java, Trait, Type System, Implementation, Eclipse

1. Introduction
The term \textit{trait} was used by Ungar et al. [51] in the dynamically-typed prototype-based language \textsc{Self} to denote a parent object to which an object may delegate some of its behavior. Subsequently, traits have been introduced by Schiörl et al. [23, 45] in the dynamically-typed class-based language \textsc{Squeak/Smalltalk} to play the role of \textit{units for behavior fine-grained reuse}, in order to counter the problems of class-based inheritance with respect to code reuse (see, e.g., [19, 23, 36] for discussions and examples). A trait is a set of methods, completely independent from any class hierarchy. The common behavior (i.e., the common methods) of a set of classes can be factored into a trait, and traits can be composed to form other traits or classes. Two distinctive features of traits are that traits can be composed in arbitrary order and that composed traits or classes must resolve possible name conflicts explicitly. These features make traits more flexible and simpler than mixins [6, 17, 25, 28, 33]. Various formulations of traits in a \textsc{Java}-like setting can be found in the literature (see, e.g., [14, 15, 34, 37, 44, 46]). The recent programming language \textsc{Fortress} [5] incorporates a form of trait construct, while the \textit{“trait”} construct incorporated in \textsc{Scala} [38] is indeed a form of mixin.

Records have been proposed in [14] as the counterpart of traits, with respect to state, to play the role of \textit{units for state fine-grained reuse}. A record is a set of fields, completely independent from any class hierarchy. The common state (i.e., the common fields) of a set of classes can be factored into a record.

In this paper, we present the programming language \textsc{Sugared Welterweight Record-Trait Java} (\textsc{SWRTJ}), a \textsc{Java} dialect using traits and records as composable units of behavior and state reuse, respectively, and aiming at interface-based polymorphism. \textsc{SWRTJ} is based on the calculus presented in [11] whose complete formalization and proof of type safety is available as a technical report in [10]. In \textsc{SWRTJ}, the declarations of object type, state, behavior and instance generation are completely separated. \textsc{SWRTJ} considers:

\begin{itemize}
  \item \textit{Interfaces}, as pure types, defining only method signatures.
  \item \textit{Records}, as pure units of state reuse, defining only fields.
  \item \textit{Traits}, as pure units of behavior reuse, defining only methods.
  \item \textit{Classes}, as pure generators of instances, implementing interfaces by using traits and records, and defining constructors.
\end{itemize}

In \textsc{SWRTJ}, like in \textsc{Fortress}, there is no class hierarchy and consequently no class-based inheritance. Multiple inheritance with respect to methods is obtained via the trait construct, and multiple inheritance with respect to fields is obtained via the record construct. Thus, multiple inheritance is subsumed by ensuring that, in the spirit of the original trait proposal in \textsc{Squeak/Smalltalk}, the composite unit has complete control over the composition and must resolve conflicts explicitly.

Although \textsc{SWRTJ} rules out class-based inheritance and relies on traits and records as the reuse mechanisms, class-based inheritance could be encoded with the existing concepts of \textsc{SWRTJ} as a syntactic sugar construct, as we will explain in Section 5. However, this “shortcut” might undermine the code reuse potential provided by traits and records leading back to the pitfalls of standard class...
Based inheritance. Furthermore, the introduction of class-based inheritance in SWRTJ is not necessary, since a SWRTJ program can be used in conjunction with any standard Java program. SWRTJ is a Java-dialect, and the SWRTJ compiler generates standard Java classes that do not depend on a specific library. Thus, a Java program can reuse SWRTJ interfaces and create standard Java objects using the code generated by the SWRTJ compiler. This technique can be used to combine code written in SWRTJ with Java libraries whose sources cannot be changed. For instance, SWRTJ objects can be passed to standard Java code provided the respective interfaces are compatible. The possibility of combining SWRTJ generated Java code with standard Java code can also be used to incrementally refactor Java class hierarchies into trait and record-based code. The refactoring can be done along the lines of [9], where we present an approach for identifying the methods in a Java class hierarchy that can be good candidates to be refactored in traits.

SWRTJ is equipped with a Java-like nominal type system where the only user-defined types are interface names. This supports typechecking traits in isolation from traits and classes using them. In Scala [38] and Fortress [5], each trait, as well as each class, also defines a type. But the role of unit of reuse and the role of type are competing, as already pointed out by Snyder [47] and Cook et al. [22]. In order to be able to define the subtyping relation on traits such that a trait is always a subtype of the component traits, both Scala and Fortress rule out the method exclusion operation. (Method exclusion forms a new trait by removing a method from an existing trait.) However, without method exclusion, the reuse potential of traits is restricted. In SWRTJ, traits (as well as records and classes) do not define types. Thus, in SWRTJ method exclusion is supported and enhances the reuse potential of traits. This will be illustrated in Example 1 of Section 2.

We present the implementation of SWRTJ using ĀTEXT [4], a framework for the development of programming languages as well as other domain-specific languages (DSLs). The syntax of SWRTJ is defined in XTEXT using an EBNF grammar. The XTEXT generator creates a parser, an AST-meta model (implemented in EMF [48]) as well as a full-featured Eclipse-based editor for SWRTJ. Using the SWRTJ compiler, a SWRTJ program is translated into a standard Java program. The SWRTJ compiler can also be used as a command-line program, outside of the Eclipse IDE. The implementation described in the paper is based on [50]. The implementation of SWRTJ is available as an open source project at http://swrtj.sourceforge.net.

Organization of the Paper. In Section 2 we illustrate record and traits as supported in SWRTJ. In Section 3 and Section 4 we describe SWRTJ and its implementation. In Section 5 we discuss the features of our language and its impact on programming, and report on our experience with XTEXT. Related work is discussed in Section 6. We conclude by outlining some future work.

2. Traits and Records in SWRTJ

In this section we illustrate record and traits as supported in SWRTJ. We refer to Section 6 for a comparison of our features with respect to other trait constructs present in the literature.

In SWRTJ, a trait consists of provided methods (i.e., methods defined in the trait), of required methods which parametrize the behavior, and of required fields that can be directly accessed in the body of the methods, along the lines of [11, 15]. Traits are building blocks to compose classes and other, more complex, traits. A suite of trait composition operations allows the programmer to build classes and composite traits. A distinguished characteristic of traits is that the composite unit (class or trait) has complete control over conflicts that may arise during composition and must solve them explicitly. Traits do not specify any state. Therefore a class composed by using traits has to provide the required fields through records, as explained in the following. The trait composition operations considered in SWRTJ are as follows:

- A basic trait defines a set of methods and declares the required fields and the required methods.
- The symmetric sum operation, +, merges two traits to form a new trait. It requires that the summed traits must be disjoint (that is, they must not provide identically named methods) and have compatible requirements (two requirements on the same method/field name are compatible if they are identical).
- The operation exclude forms a new trait by removing a method from an existing trait.
- The operation aliasAs forms a new trait by giving a new name to an existing method. When a recursive method is aliased, its recursive invocation refers to the original method (as in [23]).
- The operation renameTo creates a new trait by renaming all the occurrences of a required field name or of a required/provided method name from an existing trait. Therefore, in SWRTJ, the actual names of the methods defined in a trait (and also the names of the required methods and fields) are irrelevant, since they can be changed by the renameTo operation.

Records are building blocks to compose classes and other, more complex, records by means of operations analogous to the ones described above for traits. The record composition operations considered in SWRTJ are as follows:

- A basic record defines a set of fields.
- The symmetric sum operation, +, merges two disjoint records to form a new record.
- The operation exclude forms a new record by removing a field from a record.
- The operation renameTo creates a new record by renaming a field in a record.

SWRTJ fosters the programming methodology based on design by interface, starting from declaring the services (methods) that our implementation components provide using a separate mechanism (the interface) which has only that aim; this naturally turns the interface in the only means to declare a type. The state and the implementations of such services are once again split into two different entities which can be reused separately, i.e., records and traits, respectively. Note that these blocks, records and traits, are pure units of reuse, and cannot be instantiated directly (this would undermine their usability). However, they can be used by the classes, which are the only means to assemble records and traits in order to implement interfaces and to instantiate objects.

In the following, we illustrate the record and traits constructs of SWRTJ through examples (inspired by [15]) concerning the implementation of simple data structures.

**Example 1. Consider the task of developing a class Stack that implements the interface:**

```java
interface IStack { boolean isEmpty(); void push(Object o); Object pop(); }
```

In Java, the corresponding implementation would be a class as follows:

```java
class Stack implements IStack {  
    List l;  
    Stack() { l = new ArrayList(); }  
    boolean isEmpty() { return l.size() == 0; }  
    void push(Object o) { l.addFirst(); }  
    Object pop() { Object o = l.getFirst(); l.removeFirst(); return o; }  
}
```

Suppose that afterward a class implementing the following interface should be developed:
interface IList {  
boolean isEmpty();  
void push(Object o);  
void pop();  
Object top();  
}

In JAVA there is no straightforward way to reuse the code in class Stack, as it would not be possible to override the pop method changing the return type from Object to void.

In a language with traits and records, the fields and the methods of the class can be defined independently from the class itself. If the class Stack was originally developed in SWRTJ, it would have been written as follows (the interface IList and the class CArrayList, not shown here, have all the standard methods of lists and are part of the library of the language; they correspond to List and ArrayList in JAVA):

record RElements is { IList l; }

trait TStack is {  
IList l; /\ requires field */  
boolean isEmpty() { return (l.size() == 0); }  
void push(Object o) { l.addFirst(o); }  
Object pop() { Object o = l.getFirst(); l.removeFirst(); return o; }  
}

class Stack implements IStack by RElements and TStack {  
Stack() { l = new CArrayList(); }  
}

The record REElements and the trait TStack are completely independent from the code of class Stack. SWRTJ extends method reusability of traits to state reusability of records and fosters a programming style relying on small components that are easy to reuse. Because of the trait and record operations to exclude and rename methods and fields, they can be reused to develop completely unrelated classes. Based on this, a programmer would be able to write a class Lifo implementing the interface ILifo by reusing the record REElements and by defining a trait TLifo that reuses the trait TStack by exploiting the method exclusion operation:

trait TLifo is (TStack[exclude pop]) + { IList l; /\ requires field */  
boolean isEmpty(); /\ requires method */  
boolean isNotEmpty() { return !isEmpty(); }  
void pop() { l.removeFirst(); }  
Object top() { return l.getFirst(); }  
}

class Lifo implements ILifo by RElements and TLifo {  
Lifo() { l = new CArrayList(); }  
}

The body of trait TLifo satisfies the requirements of trait sum operation described at the beginning of the section: the method pop is excluded thus it does not generate a conflict and the field requirement is identical to the one of TStack.

This is a paradigmatic example of trait composition that does not preserve structural subtyping. If traits were types and composed traits were subtypes of the component traits (as in SCALA and FORTRESS), then the declaration of the trait TLifo would not typecheck, since:

• the trait TLifo should provide all the methods provided by TStack, and
• a method with signature Object pop() and a method with signature void pop() could not belong to the same trait/class.

EXAMPLE 2. A class Queue that implements the interface

interface IQueue {  
boolean isEmpty();  
void enqueue(Object o);  
Object dequeue();  
}

can be written by defining a trait TQueue that reuses the record REElements and the trait TStack by renaming the method pop, excluding the method push and providing the methods enqueue and isNotEmpty:

trait TQueue is (TStack[pop renameTo dequeue, exclude push]) + { IList l; /\ requires field */  
boolean isEmpty(); /\ requires method */  
boolean isNotEmpty() { return (!isEmpty()); }  
void enqueue(Object o) { l.addLast(o); }  
Object dequeue() { Object o = l.getFirst(); l.removeFirst(); return o; }  
}

class Queue implements IQueue by REElements and TQueue {  
Queue() { l = new CArrayList(); }  
}

Again, if traits were types and composed traits were subtypes of the component traits, then the declaration of the trait TQueue would not typecheck.

SWRTJ traits and records satisfy the so called flattening principle, that has been introduced in the original formulation of traits in SQUEAK/SMALLTALK [23] (see also [30, 31, 37]) in order to provide a canonical semantics for traits. According to the flattening principle, the semantics of a method/field introduced in a class by a trait/record is identical to the semantics of the same method/field defined directly within the class. For instance, the semantics of the class Queue of Example 2 is identical to the semantics of the JAVA class:

class Queue implements IQueue { 
List l;  
Queue() { l = new CArrayList(); }  
boolean isEmpty() { return (l.size() == 0); }  
boolean isNotEmpty() { return (!isEmpty()); }  
void enqueue(Object o) { l.addLast(o); }  
Object dequeue() { Object o = l.getFirst(); l.removeFirst(); return o; }  
}

We conclude this Section showing some possible alternative implementations of the data structures of Example 1.

EXAMPLE 3. The class TLifo could be implemented, in terms of TStack also using renameTo:

trait TLifo is (TStack[renameTo old_pop]) + { IList l; /\ requires field */  
Object old_pop(); /\ requires method */  
boolean isEmpty(); /\ requires method */  
boolean isNotEmpty() { return !isEmpty(); }  
void pop() { old_pop(); }  
Object top() { return l.getFirst(); }  
}

Alternatively, we could start by implementing TLifo and then implement TStack in terms of TLifo:

trait TLifo is { IList l; /\ requires field */  
boolean isEmpty() { return (l.size() == 0); }  
boolean isNotEmpty() { return !isEmpty(); }  
void push(Object o) { l.addFirst(o); }  
void pop() { l.removeFirst(); }  
Object top() { return l.getFirst(); }  
}

class TStack is (TLifo[renameTo old_pop]) + {  
void old_pop(); /\ requires method */  
}
Note that this alternative implementation is smaller, reuses more code, and makes the trait TStack independent from the actual IList field.

3. The SWRTJ Programming Language

In this section we describe the syntax and semantics of the SWRTJ programming language, and we sketch the main features of its type system.

Syntax. The syntax of SWRTJ is illustrated in Table 1 (without primitive types and file imports that are not relevant for this description). An interface can extend one or more interfaces. A class must implement one or more interfaces. In the syntax, the underline bar indicates a (possibly empty) list, as in Featherweight Java [29]. For instance, $T = \{ U_1, \ldots, U_n \}$. The parameter declarations are denoted by $\bar{F}$ to indicate $F_1 \ldots F_n$. The same notation can be applied to the other lists. In the syntax, $\mathbf{m}$ denotes a method name, $\mathbf{f}$ a field name and $\mathbf{x}$ a local variable or parameter. Note that the grammar contains only the field access $\texttt{this} \cdot \mathbf{f}$ (even for field assignment) because a field can be used only within a method (trait method or constructor) due to its private visibility. Variables and methods can only have an interface type. Trait expressions in trait composition operations do not have to be names of already defined traits: they can also be “anonymous” traits $\{ \bar{F}, \bar{S}, \bar{H} \}$. The same holds for record expressions. This is in particular useful for traits, when a trait expression has to define some “glue” code for other traits used in a trait composition operation. In this way, there is no need to define a trait with a name only for that. An example of such usage is in the stream scenario, shown later, and in the examples of Section 2.

The entry point of an SWRTJ program is specified by

```java
program <name> {
    <expression>
}
```

In the scope of the program, the implicit object $\mathbf{args}$ is the list of the program’s command line arguments.

In SWRTJ traits, records and classes are not types in order to subdivide the roles of the different constructs. Moreover, as illustrated in Example 1, traits and records support exclusion operations that violate subtyping. Therefore, a type can be only an interface or a primitive type (e.g., int, boolean etc.). For simplicity, method overloading is not supported. Constructor overloading allows only the definition of constructors with different numbers of parameters within a class.

Visibility modifiers are ruled out since they are not necessary. The traditional visibility modifiers are implied by the constructs used in SWRTJ as follows:

- **private**: every instance variable is private. Since class is not a type, fields can be accessed only from the this parameter. Every provided method is private if it does not appear in the interfaces implemented by a class. Interfaces are the only way to make a method accessible from the outside.
- **protected**: is not necessary since inheritance is ruled out.
- **public**: every method declared in the interface implemented by a class is public. Fields cannot be public in order to support information hiding.

The system library of SWRTJ provides interfaces such as IObject (implicitly extended by every interface), IInteger, IString etc. Every interface has a corresponding class implementing it, e.g., CObject, CInteger, etc. For synchronization mechanisms, we provide the interface ILock (and the corresponding class CLock) with methods lock and unlock avoiding to introduce specific concurrency features in the language. In order to deal with collections, IList is an interface with typical list operations. CArray is the corresponding implementation class. The interfaces IInputStream and IScanner, and their implicit “global” instances out and in can be used to perform basic operations such as writing to standard output and reading from standard input, respectively. Standard basic types, such as int, boolean, etc., are also provided. Methods can be declared as void with the usual meaning.

Semantics. The semantics of SWRTJ is specified through a flattening function $\llbracket \cdot \rrbracket$, given in Table 2, that translates a SWRTJ class declaration into a Java class declaration, a record expression into a sequence of fields declarations, and a trait expression into a sequence of method declarations. The clauses in Table 2 are self-explanatory. Note that, in the translation of trait expressions, the clause for field renaming is simpler than the clause for method renaming (which uses the auxiliary function $\texttt{mR}$); this is due to the fact that fields can be accessed only on this.

Type System. In this paper, we do not present the meta-theory of SWRTJ. We refer to [10] for the formalization of the FRTJ calculus on which SWRTJ is based. In this section, we provide an introduction to the typing of SWRTJ intended for the programmer. For simplicity, we will not consider void.

The SWRTJ type system supports type-checking records and traits in isolation from classes that use them. Therefore, each trait and record definition has to be typechecked only once, i.e., every class can use a trait or a record without type-checking it again. This is more efficient and convenient in practice, e.g., if the trait/record source is not available. The basic idea of the type system is to collect constraints when checking traits and records, and to establish that these constraints hold when a class is declared, in order to ensure that the pseudo-variable this in all the methods of used traits can be used safely.

In the SWRTJ type system, a nominal type is either a class name or an interface name. Note that in SWRTJ classes are not source types. Class names cannot be used as types in the code written by the programmer, but are used only internally by the compiler to type object creation expressions $\texttt{new C(\ldots)}$. The type of this is an inferred structural type. The syntax for expression types is as follows $\theta ::= \mathbb{I} | \mathbb{C} | \langle \mathbb{F}, \mathbb{S}, \mathbb{H} \rangle$ where $\mathbb{I}$ is an interface name, $\mathbb{C}$ is a class name and the $\langle \mathbb{F}, \mathbb{S}, \mathbb{H} \rangle$ is the structural type for this, which contains all the fields $\mathbb{F}$ and the signatures $\mathbb{S}$ of the methods that can be selected on this in the context where the expression occurs. If the expression is a constructor call, such as $\texttt{new C(\ldots)}$, its type is the class $\mathbb{C}$; if the expression is this, its type is $\langle \mathbb{F}, \mathbb{S} \rangle$; otherwise, the type of the method is an interface, for instance, if the expression is a method call, such as $\mathbf{x}.\mathbf{m}(\ldots)$, the type is the interface that $\mathbf{I}$ declares as return type of the method $\mathbf{m}$.

SWRTJ type system checks all requirements by inferring constraints. A constraint is a triple $\langle \mathbb{F}, \mathbb{S}, \mathbb{H} \rangle$ consisting of required fields, method signatures and interfaces collected while analyzing an expression. The constraints contain the types of every field and method selected on this and the name of every interface used, either as type in the methods parameters to which this is passed as argument or as return type in the methods in which this is returned.

The subtyping relation is the reflexive and transitive closure of the immediate subinterfacing relation declared by the extends clauses in the interface definitions. The subtyping relation for nominal types is the reflexive and transitive closure of the relation obtained by extending subinterfacing with the interface implementa-
In this section, we describe the implementation of SWRTJ using a stream library, together with the corresponding classes and trait definitions.

A classical example to show how traits can deal with situations where other mechanisms such as class inheritance do not fit well for code reuse, is the stream scenario, as in [23]. In particular, during parsing, the AST is generated in the shape of an EMF model (Eclipse Modeling Framework [48]). Thus, the manipulation of the AST can use all mechanisms provided by EMF itself. There is a direct correspondence between the names used in the rules of the grammar and the generated EMF model JAVA classes. For instance, if we have the following grammar snippet:

```
Message : MethodInvocation | FieldAccess;
```

the XTEXT framework generates the following EMF model JAVA interface (and the corresponding implementation class):

```
public interface MethodInvocation extends Message {
    MethodName getMethodName();
    ELList<Expression> getArgumentList();
}
```

Besides, XTEXT generates many other classes for the editor for the language to be defined. The editor contains syntax highlighting, background parsing with error markers, outline view, code completion. Further, XTEXT provides the infrastructure for code generation. Most of the code generated by XTEXT can already be used off the shelf, but other parts can or have to be adapted by customizing some classes used in the framework. The usage of the customized classes is dealt with by relying on Google-Guice [1], so that the programmer does not have to maintain customized abstract factories [26]. In this way it is very easy to insert custom implementations into the framework (“injected” in Google-Guice terminology), with the guarantee that the custom classes will be used consistently throughout the code of the framework.

The validation mechanisms for the language must be provided in the actual type of an AST node. Thus, there is no need to add code.

Table 2. Flattening SWRTJ to JAVA

<table>
<thead>
<tr>
<th>ID</th>
<th>interface I extends I {S,}</th>
<th>interface definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>RD</td>
<td>record R is RE</td>
<td>record definition</td>
</tr>
<tr>
<td>TD</td>
<td>trait T is TE</td>
<td>trait definition</td>
</tr>
<tr>
<td>CD</td>
<td>class C implements I by RE and TE {R}</td>
<td>class definition</td>
</tr>
<tr>
<td>RE</td>
<td>{F</td>
<td>R</td>
</tr>
<tr>
<td>TE</td>
<td>{F, S, R</td>
<td>T</td>
</tr>
<tr>
<td>R0</td>
<td>exclude f</td>
<td>f renameTo f</td>
</tr>
<tr>
<td>T0</td>
<td>exclude m</td>
<td>f renameFieldTo f</td>
</tr>
<tr>
<td>F</td>
<td>if</td>
<td>field definition</td>
</tr>
<tr>
<td>S</td>
<td>I m(T x)</td>
<td>method signature</td>
</tr>
<tr>
<td>M</td>
<td>S [Ix = x, SE, return e;]</td>
<td>method</td>
</tr>
<tr>
<td>K</td>
<td>O(T x) BE</td>
<td>constructor</td>
</tr>
<tr>
<td>BE</td>
<td>[T x = x, SE, ];</td>
<td>block expression</td>
</tr>
<tr>
<td>SE</td>
<td>this f = e</td>
<td>e.m(x) if (e) BE else BE</td>
</tr>
<tr>
<td>e</td>
<td>null</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 2. Flattening SWRTJ to JAVA

relation declared by the implements clauses in the class definitions.

An Example. A classical example to show how traits can deal with situations where other mechanisms such as class inheritance do not fit well for code reuse, is the stream scenario, as in [23]. In Listing 1, we show the interfaces, records and traits for implementing a stream library, together with the corresponding classes and the main program. Note that the TReadWriteStream reuses the previously defined traits and resolves conflicts by method renaming. Furthermore, it contains “glue” code (in an anonymous trait) for the initialization of the fields of the composed traits. Similarly, the record ReadWriteResource reuses the resources defined in the records ReadResource and WriteResource by renaming the field resource. Note that, to keep the example simple, in the implementation of init methods, we simply assigned the global instances in and out for standard input and standard output, respectively.

4. Implementing SWRTJ

In this section, we describe the implementation of SWRTJ using the XTEXT [4] framework for Eclipse. Although Eclipse itself provides a framework for implementing an IDE for programming languages, this procedure is still quite laborsious and requires a lot of manual programming. XTEXT eases this task by providing a high-level framework that generates most of the typical and recurrent artifacts necessary for a fully-fledged IDE on top of Eclipse.

The first task in XTEXT is to write the grammar of the language using an EBNF-like syntax. Starting from this grammar, XTEXT generates an ANTLR parser [39]. The generation of the abstract syntax tree is handled by XTEXT as well. In particular, during parsing, the AST is generated in the shape of an EMF model (Eclipse Modeling Framework [48]). Thus, the manipulation of the AST can use all mechanisms provided by EMF itself. There is a direct correspondence between the names used in the rules of the grammar and the generated EMF model JAVA classes. For instance, if we have the following grammar snippet:

```
Message : MethodInvocation | FieldAccess;
```

the XTEXT framework generates the following EMF model JAVA interface (and the corresponding implementation class):

```
public interface MethodInvocation extends Message {
    MethodName getMethodName();
    ELList<Expression> getArgumentList();
}
```

Besides, XTEXT generates many other classes for the editor for the language to be defined. The editor contains syntax highlighting, background parsing with error markers, outline view, code completion. Further, XTEXT provides the infrastructure for code generation. Most of the code generated by XTEXT can already be used off the shelf, but other parts can or have to be adapted by customizing some classes used in the framework. The usage of the customized classes is dealt with by relying on Google-Guice [1], so that the programmer does not have to maintain customized abstract factories [26]. In this way it is very easy to insert custom implementations into the framework (“injected” in Google-Guice terminology), with the guarantee that the custom classes will be used consistently throughout the code of the framework.

The validation mechanisms for the language must be provided in the actual type of an AST node. Thus, there is no need to add code.

2 The reader who is familiar with ANTLR will note that the syntax of XTEXT grammars is very similar to ANTLR’s syntax.
to implement a visitor structure [26]. XTEXT leverages this mechanism by only requiring methods with a @Check annotation, that will be called automatically for validating the model according to the type of the AST node being checked. The validation takes place in the background, together with parsing, while the user is writing a SWRTJ program, so that an immediate feedback is available, as usually in IDEs.

For instance, the following method in the SWTJJavaValidator checks that the this variable is not used in the program context (i.e., that this can be used only in method bodies):

```java
@Check
public void thisCheck(This thisRule) {
    if(!lookup.getOwner(thisRule).instanceof Program)) {
        showError(new ErrorMessage
        ("\"this\" is not allowed in program context", thisRule));
    }
}
```

Note that the only important things in this method definition are the @Check annotation and the parameter: the internal validator of XTEXT will invoke this method when it needs to validate an AST node representing an occurrence of this, which is declared in the grammar as the This non-terminal symbol (remember that given a grammar symbol, XTEXT will generate a corresponding class for the AST with the same name).

Binding the symbols (e.g., the binding of a field reference to its declaration) is important in compiler development. EMF uses “proxies” to represent references. It can delay the resolution (binding) of references when they are accessed. XTEXT already provides an implementation for binding references, which basically binds a reference to a symbol n to the first element declaration with name n occurring in the model. This usually has to be adapted in order to take the visibility of names in a program into account. For instance, a field is visible only in the methods of a class, such that different hierarchies can safely have fields with the same name. XTEXT supports the customization of binding in an elegant way with the abstract concept of “scope”. The actual binding is still performed by XTEXT, but it can be driven by providing the scope of a reference, i.e., all declarations that are available in the current context of a reference. Note that this also permits to filter out elements according to their kind, e.g., in order not to mix field names with method names if we need to resolve a reference to a field.

The programmer can provide a customized AbstractDeclarationScopeProvider. XTEXT will search for methods to invoke, using reflection, according to a convention on method name signatures. Suppose, we have a rule ContextRuleName with an attribute ReferenceAttributeName assigned to a cross reference with TypeToReturn type, that is used by the rule ContextType. You can create one or both of the following two methods

```java
I Scope scope_<ContextRuleName>, <ReferenceAttributeName>
.getSystemService(ctx, EReference ref)
```

The XTEXT binding mechanism looks for the first method (by reflection), if this does not exist, then it looks for the second. If no such method exists, the default linking semantics (see above) is used.

For instance, if we consider the grammar rule for method invocation illustrated at the beginning of this section, we can drive the resolution of the method name in a method invocation statement in any expression where such statement can occur by defining the following method (The code should be understandable without the knowledge of XTEXT):

```java
public IScope scope_METHOD_INVOCATION_METHOD_EXPRESSION_CONTEXT,
Expression context, EReference ref) {
    ExpressionType expressionType =
      ExpressionType.createExpressionType(context.getMethodMember());
    Collection<MethodDeclaration> methodList =
      if (expressionType != null) {
        methodList = expressionType.getMethodDeclarations();
      } else {
        methodList = new LinkedList<>(methodList);
      } return Scopes.scopeFor(methodList);
}
```

The scope provider will be used by XTEXT not only to solve references, but also to implement code completion. Thus, a programmer achieves two goals by implementing the abstract concept
SWRTJ programs may look more verbose than standard class-based programs. However, the degree of reuse provided by records and traits is higher than the reuse potential of standard static class-based hierarchies. The distinction of each programming concept in a separate entity pays off in the long run, since each component is reusable in different contexts, in an unanticipated way. This does not happen so easily with standard class-based OO linguistic constructs. Class hierarchies need to be designed from the start with a specific reuse scenario in mind. In particular, for single
inheritance, precise design decisions should be made from the very beginning. This design might be hard to change, if not impossible, forcing either to refactoring or to code duplication [36].

Our linguistic constructs are lower-level than standard OO mechanisms. However, some syntactic sugar can be added to reduce the amount of code to write in some situations. For instance: fields might also be declared directly in classes (thus encompassing the JAVA-like calculus proposed in [15]), class names might be used as types, etc. Along the same lines, we can simulate class inheritance with our linguistic constructs (thus encoding JAVA dialects, like [46], and programming environments, like [41], where traits coexist with class inheritance). This can be easily achieved by inverting the “flattening” concept. For instance, if we start from the JAVA-style class Stack given at the beginning of Example 1 in Section 2, we can easily separate fields and methods into automatically generated records and traits, and generate the corresponding SWRTJ class declaration. Similarly, inheritance and method overiding can be simulated with inherited interfaces and trait method renaming, respectively. The super call can be simulated with a call to the renamed version of the method. However, this would decrease the level of reuse for such components.

This issue of having these additional constructs in the language is related to the debate of whether it is better to have a pure or hybrid programming language. In this respect, the purity of a language usually imposes more verbose solutions than a hybrid language. For instance, consider the amount of code for writing the main static public method of a public class in JAVA, and compare it to the simple form of the corresponding function in C++ which also provides functions besides classes and methods. The goal of reducing verbosity often led to additional language constructs which may break the pure linguistic features, e.g., the addition of imperative features to a purely functional language (see, e.g., OBJECTIVE CAML [42]). The general debate between pure and hybrid languages is out of the scope of the present paper. Nonetheless, we argue that having linguistic constructs for reusable code development eases adding other high-level linguistic constructs which otherwise may only be possible at the cost of code duplication and of the resulting complexity of code maintenance.

We might also introduce other syntactic sugar constructs to ease the programming with our language; for instance, instead of writing the required methods in a trait, we could group the required methods in a specific interface and let the trait “implement” that interface meaning that the trait requires all the methods specified in the interface. However, this would not make traits types, since as we stressed throughout the paper, not having traits types enhances their reuse potential.

6. Related Work

Traits are well suited for designing libraries and enable clean design and reuse which has been shown using SMALLTALK/SQUEAK e.g., [13, 18]. Recently, [8] pointed out limitations of the trait...
model caused by the fact that methods provided by a trait can only access state by accessor methods (which become required methods of the trait). To avoid this, traits are made stateful (in a SMALLTALK/SQUEAK-like setting) by adding private fields that can be accessed from the clients possibly under a new name or merged with other variables. In SWRTJ traits are stateless. By their required fields, however, it is possible to directly access state within the methods provided by a trait. Moreover, the names of required fields (in traits) and provided fields (in records) are unimportant because of the field rename operation. Since field renaming works synergically with method renaming, exclusion and aliasing, SWRTJ has more reuse potential. Concerning field requirements, they are not present in most formulation of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. They were introduced in the formulation of traits in a structurally typed setting by Fisher and Reppy [24].

SWRTJ requires that the summed traits must be disjoint. The disjoint requirement for composed traits was proposed by Snyder [47] for multiple class-based inheritance (see also the JIGSAW framework [16]). According to other proposals, two methods with the same name do not conflict if they are syntactically equal [23, 37] or if they originate from the same subtrait [34]. When a recursive method is aliased in our language, its recursive invocation refers to the original method (as in [23]). The variant of aliasing proposed in [34] (where, when a recursive method is aliased, its recursive invocation refers to the new method) can be straightforwardly encoded by exclusion, renaming and symmetric sum. Concerning method renaming and required field renaming, they are not present in most formulation of traits in the SMALLTALK/SQUEAK-like and JAVA-like settings. Method renaming has been introduced in the formulation of traits in a structurally typed setting by Reppy and Turon [43]. Renaming operations were already present in the JIGSAW framework [16] in connection with module composition and in the EIFFEL language [35] in connection with multiple class-based inheritance.

In [44], a variant of traits that can be parametrized by member names (field and methods), types and values is proposed. Thus, the programmer can write trait functions that can be seen as code templates to be instantiated with different parameters. This enhances the code reuse provided by traits already. It could be interesting to adapt this approach to our context and to extend the parametrization functionalities also to interfaces, records and classes. This will be the subject of future work. However, an important difference between our proposal and the one in [44] is that, in the latter, traits play also the competing role of type, which is avoided in SWRTJ. Another feature of SWRTJ is that structural types are used only “internally” on this, i.e., the programmer works with nominal types (interfaces) alone. We believe this is an important feature from a practical point of view, as it reduces the distance between the classical JAVA-like languages and our linguistic constructs, from the perspective of the programmer.

In [41] an Eclipse plugin is presented that supports JAVA programmers with using trait-like mechanisms for JAVA classes. JAVA is not extended: traits are modeled as (possibly) abstract classes and trait method requirements as abstract methods. The plugin also aims at helping the programmer to refactor JAVA hierarchies with traits: the programmer can select a set of methods to be extracted into stateless classes that play the role of traits. When a class uses a trait, the methods provided by the trait are copied into that class; the programmer then has to use this plugin to keep track of the actual source of a method, in order not to accidentally change the copy of a method and to be aware of what to change: either the copy of the class or the original method in the trait. Thus the programmer has to rely on this plugin completely, while in our case the SWRTJ compiler can be used also outside from Eclipse.

The work of [41] starts from the one in [40] where traits are implemented directly as an extension of a subset of the JAVA language, and a compiler translates such programs into standard JAVA programs. The authors, however, state that such approach requires a strong effort to be conservative with respect to JAVA since, in order to generate well typed JAVA code, many type checking functionalities for JAVA related code must be implemented in their compiler. Such extension, moreover, lacks a formalization, thus no property of type safety for such language extension is available.

In our approach, since we deal with a JAVA dialect, we can concentrate on the linguistic parts that are characteristic of our own language, without re-implementing JAVA checks. However, the code generated by our compiler is standard JAVA code, and our formalization [10] guarantees that such code will be type correct with respect to the JAVA compiler (starting from a well-typed SWRTJ program).

7. Conclusions and Future Work

In this paper, we presented the programming language SWRTJ and its implementation. SWRTJ is based on the calculus presented in [11]. In that paper, we considered mechanisms for code reuse for implementing Software Product Lines (a set of software systems with well-defined commonalities and variabilities [21]). We explored a novel approach to the development of SPL, which provides flexible code reuse with static guarantees. In order to be of effective use for SPL, the type-checking has to facilitate the analysis of newly added parts without re-checking already existing products. The SWRTJ type system (Section 3) satisfies this requirement since it supports type-checking records and traits in isolation from classes that use them.

A special form of reuse is at the base of the contemporary agile software development methodologies, which are based on an iterative approach, where each iteration may include all of the phases necessary to release a small increment of a new functionality: planning, requirements analysis, design, coding, testing, and documentation. Another example is the use of Extreme Programming [7], where team members work on activities simultaneously. While an iteration may not add enough functionality to guarantee the release of a final product, an agile software project intends to be capable of releasing new software at the end of every iteration. However, this means that the next iteration will reuse the software produced in the previous ones. We believe that an interesting future research direction is to investigate whether the programming language features proposed in this paper may help in writing software following an agile methodology.

In [9], we presented a tool for identifying the methods in a JAVA class hierarchy that could be good candidates to be refactored in traits (by adapting the SMALLTALK analysis tool of [32] to a JAVA setting). It will be interesting to investigate the application of this approach also for detecting possible candidates for records and traits in the context of porting existing JAVA code to SWRTJ code.

Acknowledgments. We are grateful to the developers of XTEXT, in particular, Sven Efftinge and Sebastian Zarniewicz, for their prompt help and support during the development of SWRTJ. We thank Viviana Bono for many discussions on the subject of this paper, Stéphane Ducasse for interesting discussions about traits, and Alexandre Bergel for insightful comments on the technical report [10].

References