Interfaces with traits

Kjetil Østerås[1,2] and Birger Møller-Pedersen[1]
[1]Department of Informatics, University of Oslo
[2] TANDBERG R&D, Oslo

Abstract

In order to provide for reuse of methods independent of and across different class/subclass hierarchies, the notion of trait has recently been proposed. A trait is a set of methods that a class may have in addition to a superclass and in addition to properties defined for the class: the class will get the methods of the trait as if they were defined in the class.
Traits were in the first place introduced for Squeak/Smalltalk, and recently there have been some efforts introducing traits in typed languages like Java and C#. This paper describes a prototype implementation of traits for Java with a trait syntax that is more intuitive than ordinary trait syntax. As part of this work a brand new idea emerged: while it is common to let traits have interfaces (with the obvious implication that a class that gets this trait will also implement these interfaces), the paper introduces the opposite facility: that an interface may have traits. It is shown that this has interesting possibilities, and it is also shown how this is implemented.

1. Introduction

In order to extend reuse beyond what can be achieved by single inheritance and without the problems of multiple inheritance and mixin inheritance, the notion of trait [3] has been introduced. A trait is a collection of methods, not tied to any class and independent of any class hierarchy.

As an example consider the class hierarchy of Figure 1.

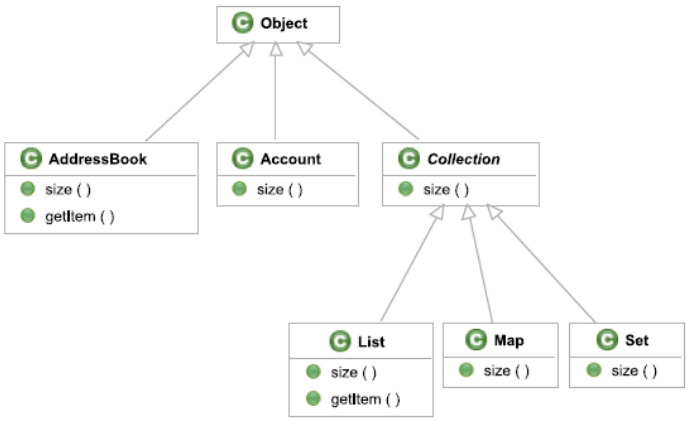

Figure 1 Class hierarchy

Assume that we need to extend some of these classes with two new methods `isEmpty` and `iterator`. The method `isEmpty` shall check if `Collections` and `AddressBooks` are empty, and `iterator` shall iterate over elements of `AddressBooks` and `Lists`. The methods required to implement these two methods are given in the classes. We can find out if something is empty by use of the `size` method, and we can accomplish iteration over `AddressBooks` and `Lists` by the use of the `size` method in combination with the `getItem` method:

```
boolean isEmpty() {
  return size() == 0;
}
public Iterator iterator() {
  return new Iterator() {
         int current =0;
         public boolean hasNext() {
                return current < size();
         }
         public Object next() {
                return getItem(current++);
         }
  };
}
```

The usual place for such common methods is in a common superclass. However, in this case the only common superclass is `Object`, and it is not a good idea to put the `isEmpty` and `iterator` methods in class `Object`. We will therefore have to duplicate the methods in the actual classes, see Figure 2.

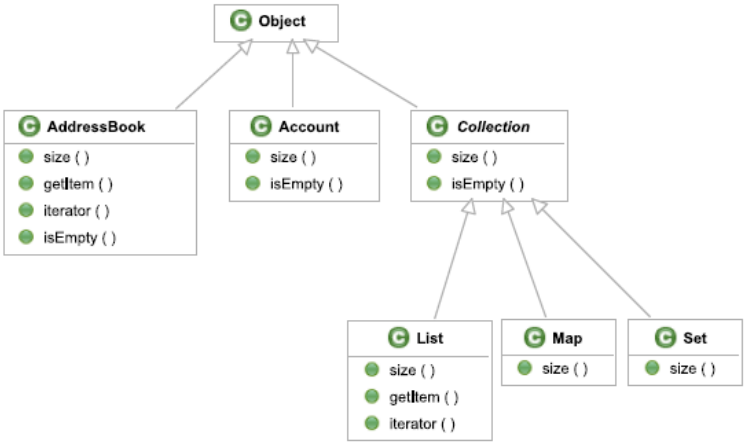

Figure 2 Class hierarchy extension without traits

The duplication of code in this hierarchy implies a maintenance problem. The methods with exactly the same body are pasted into several different classes and must be maintained separately.

Traits remedy this situation simply by defining the traits `Emptiness` and `ItemIterable` with the methods `isEmpty` and `iterator` respectively, and then let the actual classes have these traits, see Figure 3 for an illustration. There is no standard way of modeling traits, so traits are simply depicted as UML classes (even though traits are not types) with a *trait* stereotype. A dependency arrows with the *with* stereotype indicates that a class has a certain trait.

As illustrated in Figure 3, when using traits, classes are still organized in a single inheritance hierarchy, and traits can be used to specify the incremental difference in behavior with respect to the applied classes [3].

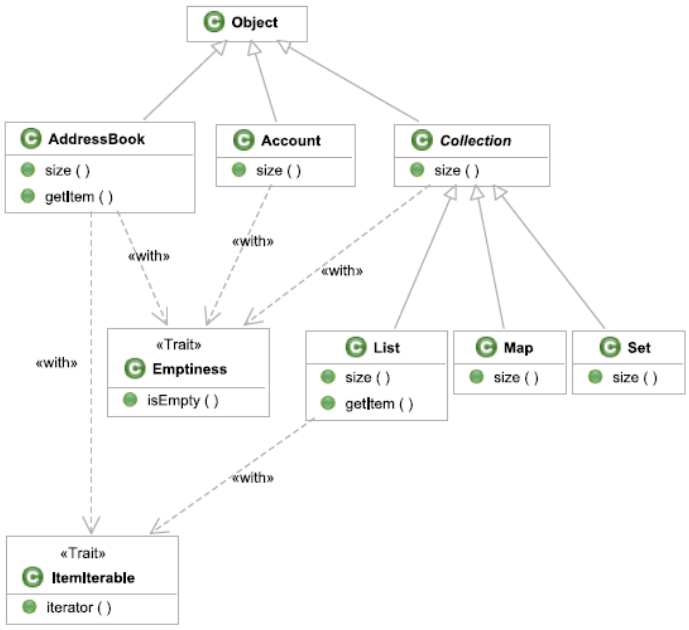

Figure 3 Class hierarchy extension with traits

The paper is organized as follows: Chapter 2 introduces traits and their implementations in un-typed languages. Chapter 3 describes the implementation of traits in Java, while Chapter 4 describes the idea of letting interfaces and not just classes have traits.

The work reported in this paper is based upon a Master's Thesis within the field of the SWAT project (The Research Council of Norway grant no. 167172/V30).

2. Traits: semantics and implementations

A trait can only define methods and not variables, but in order for the trait methods to do anything useful they need to manipulate variables of the classes that get the trait. This manipulation is done by calling methods that the trait *requires* the classes to have.

In the example above the `Emptiness` trait requires the method `size`, and the `ItemIterable` trait requires the methods `size` and `getItem`.

In order to make traits more usable, both traits and classes have the ability to be composed of any number of traits. The composition order is irrelevant. When traits are combined in this form, the requirements of all the traits are combined to form a new set of requirements. Requirements posted by one trait can possibly be implemented by some other trait, so together they can form a composed trait with fewer requirements. Composition can for example be used to collect associated traits into one larger trait. As an example consider a collection hierarchy. Assume that we want to use traits to define the properties of an ordered collection. An ordered collection in our case should have the ability to be sorted and reversed. We create the trait `Ordered` and we use composition to apply the traits `Sortable` and `Reversable` to the `Ordered` trait. When a class applies the `Ordered` trait it gets the methods from `Sortable` and `Reversable` as well as the methods from `Ordered`.

Composition of classes by using the traits model can be described with the equation Class = Superclass + State + Traits + Glue [3]. The methods required by the traits can either be supplied by other traits or implemented by the Glue-code of the class. The Glue-code may also override methods provided a traits.

The idea of traits is that they should behave as if the methods of the trait were defined directly in the class that has the trait. This property is called the *flattening* property of traits. This means that the semantics of a class defined using traits is exactly the same as that of a class constructed directly from all the non-overridden methods of the trait. Similarly, trait composition does not affect the semantics of a trait, a trait defined using traits is exactly the same as the trait constructed directly from all of the non-overridden methods of the component traits.

Since traits can be composed arbitrarily there can be cases where several traits provide the same methods. We say that a method *conflict* occurs when some unit, either a trait or a class applies several traits that contain methods with the same name and signature. To grant access to conflicting or hidden trait methods traits support the *alias* operation. The alias operation makes a trait method available under another name. So if for instance two traits `t1` and `t2` are used in a class and they both provide the method `m`, a method conflict occurs. To resolve this conflict the class can provide its own `m` method and override the `m` methods of both traits. However if the class wants to use one of the `m` methods of `t1` or `t2` it has to be aliased to make it available under another name.

Trait composition also supports the *exclusion* operation. Using the exclusion operation a trait can exclude a given method from a composite trait. The exclusion operation allows a trait to avoid a conflict.

Traits has currently been implemented in the Squeak programming language [8] a dialect of the Smalltalk programming language [4], and it has been used with success to re-factor the Smalltalk-80 collection hierarchy where the amount of code was reduced by approximately 10% [1] .

3. Traits in Java

The implementation of traits in Squeak is rather straight forward: the checking of whether a required method is present or not in the class that gets a trait is done at runtime when the actual method lookup is made

In addition, Squeak is a dynamically typed programming language, where type checking is deferred until run-time; therefore one did not need to worry about type checking issues at compile-time like the typing of method arguments and return types of methods provided by a trait.

Implementation of traits in Java will have to take into account that the use of required methods within a trait has to be resolved at compile-time; in addition type checking of e.g. method arguments and return types has to be done at compile-time. [11] describes some issues when including traits into a statically typed language. The paper focuses on a trait implementation in C#, but many of the same issues exist when including traits in the Java language.

Traits in Java is not a new idea. There have been attempts to add traits to the Java language before. One of the earlier implementations of traits in Java [2] simulates traits by the use of AspectJ [9], which provides an aspect-oriented extension to Java. Another trait implementation has been done in a subset of the Java language called mini-Java [12].

3.1 Issues

3.1.1 *Trait type.*

The relationship between type and trait has to be considered when implementing traits in a statically typed language. For instance if we have a trait `Ta`, do we also have a type

`Ta` as a valid reference type to be used e.g. as type of formal parameters. The answer is no: A trait does not introduce a type.

Another issue that arises when traits are implemented in a statically typed language is concerned with type-checking of requirements. A strategy for how required methods should be type-checked must be implemented in a preprocessor. The strategy must enable the trait to be reused across multiple classes correctly, and it should not cause unnecessary restrictions of use.

3.1.2 *Argument type.*

Consider a trait method and its arguments. What if we wanted a method to be able to take as input-argument a reference typed with a class that has a specific trait. More formally we want to define that parameter p_i has the type `t` where `t` is any class with trait `T`. If we extend the type system of the language so that each trait defines its own type and create a subtype relation whenever a trait is applied in a class, we would be able to use the trait type as parameter p_i:

```
trait TLinkable {
  public boolean include s (TLinkable linkable){
        . . .
  }
}
```

If the trait does not have a corresponding type then we use an interface to specify the type of parameter p_i. That interface would then have to be implemented by all classes that are used as types of actual parameters. This is in the following illustrated by a trait `TLinkable` which offers a method public `boolean includes(...)`. The method `includes` needs to take a reference typed by a class which applies the trait `TLinkable` as input-argument, and it does so by typing this argument by an interface `ILinkable`:

```
trait TLinkable {
  public boolean includes(ILinkable linkable){
        . . .
  }
}
```

3.1.3 *Return value type.*

Return value type of trait methods is another issue. A problem occurs when a return value of a trait method is supposed to be the same type as the class which gets the trait. It is not sufficient to solve this by typing traits because the returned object will only have the methods contained in the specific trait. The solutions are either to use generics to give the trait its class as a parameter or one could introduce a new keyword that signifies that this method returns the type of the class that gets the trait. The keyword proposed is `selftype` or `ThisType`. The following is a small example of a trait which provides linked list functionality. In a linked list we typically want a method for getting the next node in the linked list chain. This is implemented in the trait so it can be reused in different classes.

```
trait TLinkable{
  public ThisType getNext() {
        . . .
  }
}
```

3.2 Implementation

Java with traits is implemented by means of a preprocessor that performs parsing of Java with traits, performs trait-specific type checking and produces ordinary Java. The preprocessor uses JavaCC [6] to generate the parser and JJTree [7] to generate the abstract syntax tree. The Java grammar has been extended to include the traits syntax described in section 3.3.

The reason for using abstract syntax trees when operating with traits is that they are easy to work with. A trait inclusion consists of copying the methods from the trait into the class definition. This is done with abstract syntax trees by just copying a whole subtree from one branch to another, see Figure 4. The class nodes and trait nodes have child nodes which represents method declarations. Solid arrows indicate references, and dotted arrows indicate node copying.

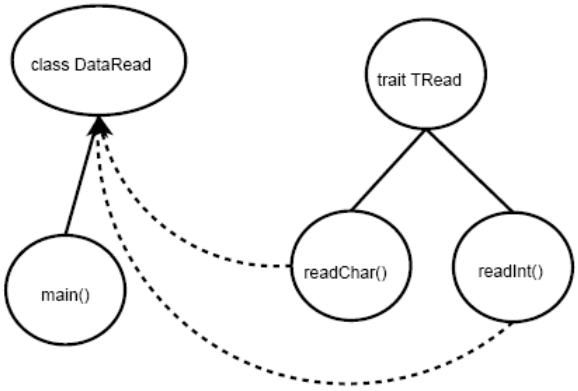

Figure 4 AST transformation

The preprocessor implements the traits model as described above, with adoptions to make traits fit into the Java programming language. The preprocessor supports:

- Definition of traits.
- Application of traits in classes.
- Composition of traits.
- Type checking of traits
- Aliasing and exclusion of trait methods.

As part of the application of traits in classes and composition of traits, flattening of traits is implemented.

3.3 Trait Syntax and Semantics

The grammar for the Java with traits implementation is based upon the Java language reference grammar [5]. When a rule below is not listed, it will be found in the Java language reference.

The grammar below uses the same BNF-style conventions as in the Java language specification, with the following rules:

- `[x]` denotes zero or one occurrences of `x`.
- `{x}` denotes zero or more occurrences of `x`.
- `x|y` means one of either `x` or `y`.

The traits syntax is rather standard, however, we have deliberately introduced the keyword **`with`** in order to specify that a class has a trait. The standard keyword in other

implementations is **uses**, however, that does not work well with the original/everyday meaning or use of the term ‘trait’.

3.3.1 Trait declaration

An ellipsis in TypeDeclaration is used in order to signify that the grammar rule is extended and the rest can be found in the Java language specification.

```
TraitDeclaration :
  trait Identifier [implements TypeList] TraitBody
TraitBody :
  { [TraitRequirements] {TraitMemberDecl} }
```

A trait declaration consists of a keyword trait followed by an identifier and an optional list of interface names that are implemented by the trait. The trait body consists of two parts, the first part contains required methods used by methods inside the trait and the second part contains trait members. Trait members can be method declarations and composite traits.

The requirement part consists of a set of method signatures. InterfaceMethodDeclaratorRest refers to a rule defined in the Java language specification.

```
TraitRequirements :
  requires { {TraitMethodRequirement} }

TraitMethodRequirement :
  {Modifier} Type Identifier InterfaceMethodDeclaratorRest
```

Note that requirements are explicitly listed opposed to the way requirements are handled in the original Smalltalk implementation. In the Smalltalk implementation a requirement is generated by a call to an undefined method inside a trait method. The reason for explicitly adding method requirements when adding traits to Java is that the type-checker part of the preprocessor should be able to control if a trait can be flattened or not. Declaring a requirement inside a trait creates a precondition stating that flattening can only happen if a method with the same signature or with any compatible signature exists in the class that gets the trait.

A trait member can be one of two elements, either a with-block where other traits are used, or a method declaration which will add a provided method to the enclosing trait entity.

```
TraitMemberDecl : WithTrait TraitMethodDecl

TraitMethodDecl :
  {Modifier} Type Identifier MethodDeclaratorRest
```

There is nothing special about the TraitMethodDecl it is just like any ordinary method declaration in a Java class. The reason for creating yet another rule for method declaration and not refer to the Java method declaration is because the Java language specification has divided the method declaration in a way that makes it harder to reuse.

3.3.2 Trait application

In order for traits to be useful we have to supply a way to include the declared traits inside a class body. Inclusion of traits is done by creating a with-block within a class or interface body. A with-block can also be within a trait body.

```
ClassBodyDeclaration :
  ...
  WithTrait

InterfaceBodyDeclaration :
  ...
  WithTrait

WithTrait : with { {WithDeclaration;} }

WithDeclaration : Identifier [TraitOperations]
```

A with-block inside a class will signify that this class needs to be flattened before it can be compiled by an ordinary Java compiler. The preprocessor will locate these with-blocks and execute the flattening. With-blocks can contain any number of traits, and the order of inclusion is not important.

3.3.3 Trait operations

Whenever traits are applied optional operations can be used on the included trait methods. Operations are added in order to resolve method conflicts that occur when composing traits. There are two different operations: aliasing and exclusion. Aliasing is an operation which takes two method signatures `m1` and `m2` as parameters. The effect of the alias operation is that the body of `m1` is reachable by calling `m2` as well as by calling the original `m1`.

Exclusion is an operation which takes a single method signature as parameter. The effect of the exclusion operation is that the method with the associated signature is excluded from the composing trait entity. Use of exclusion operation is needed when two traits with the same method signature are supposed to be combined/flattened and we do not want to cause a method conflict. By exclusion the programmer can choose one of the conflicting methods.

```
TraitOperations : { {TraitOperation;} }
TraitOperation : Aliasing  | Exclusion
Aliasing :MethodSignature -> MethodSignature
Exclusion : ^ MethodSignature
MethodSignature : {Modifier} Type Identifier MethodSignatureRest
MethodSignatureRest : FormalParameters [throws QualifiedIdentifierList]
```

3.4 Synchronized wrapper example

A common traits example is the synchronized read-write example. In this case we have two classes that both want a synchronized read-write method. Read and Write are traits with an implementation of the read and write methods:

```
trait Read {
  int read(){...}
}
trait Write {
  void write (int b){...}
}
```

Classes using these traits need only include them to get their functionality. When making the synchronized version of read and write, we make a new trait which overrides the read and write methods and synchronizes calls on the methods. This shows how the super keyword can be used to generate wrapper functions.

```
trait SyncRead {
  synchronized int read(){
        return super.read();
  }
}
trait SyncWrite {
  synchronized void write(int b){
        return super.write(b);
  }
}

class A {
  with{Read; Write;}
}
class B {
  with {Read; Write;}
}
class SyncA extends A {
  with {SyncRead; SyncWrite;}
}
class SyncB extends B {
  with {SyncRead; SyncWrite;}
}
```

4. Interfaces and traits

4.1 Trait implementing interfaces

A natural extension of traits when inserting them into the Java language is to make traits implement interfaces. This is a simple extension of the trait syntax and has also been done in a traits prototype for C# [13]. When a Java class implements an interface it means that the class provides methods corresponding to the signatures in the interface. When a trait implements an interface it would mean that the trait provides bodies for methods that are defined as signatures in the interface. When a class applies a trait that implements a certain interface `i`, this means that the class also provides the methods that are defined in the interface `i` and hence the class now also implements interface `i`. However, there are exceptions to this behavior. Since the trait model provides the class with the exclusion operation on traits, there is a possibility of the class excluding the method needed to implement a certain interface.

Here we will present a clarifying example. Ordering in Java is often implemented by using the `Comparable` interface. The comparable interface contains a single method with the signature `int compareTo(Object o)`. This method returns a negative integer, zero or a positive integer as the object is less than, equal to or greater than the specified object o. The `Comparable` interface is often used when sorting a collection of objects. The following example

```
trait TEqual implements Comparable {
  public int compareTo(Object o) {
        return 0;
  }
}
```

defines a trait that can be used by a class which should be evaluated to be equal to any object with respect to the `compareTo` method of the `Comparable` interface. We define a trait `TEqual` which implements the `Comparable` interface.

Now we want to define a class with this trait to make the class equal to anything else. We define a class `UniversalEqual` with the trait `TEqual`:

```
class UniversalEqual {
  with {TEqual;}
}
```

The end result when we combine these two is:

```
class UniversalEqual implements Comparable {
  public int compareTo( Object o ) {
         return 0;
  }
}
```

We see that when the trait `TEqual` is used, all its methods are included and therefore all its implemented interfaces are added to the class.

4.2 Interface with traits

Traits can implement interfaces, but what about the idea of interfaces with traits. This idea would enable interfaces to contain with-blocks that are inserted into classes implementing the interface before flattening is done.

In the preprocessor this special possibility has been implemented. The way it works is that the syntax is extended so that interface bodies can contain with-blocks in the same way as classes. Now we can naturally not flatten interfaces in the same way as we flatten classes, because it would lead to illegal code. So interface flattening is implemented in the preprocessor like this: when an interface is implemented by a class the with-blocks that are contained inside this interface is injected into the class as if the with-blocks were written directly into the class body.

An interface with traits can be thought of like a carrier of with-blocks, and hence we can have multiple inheritance of method bodies only by extending the interface syntax. We will show that this can be a good idea when we want to provide a default implementation of the methods in a certain interface. Assume that we want to implement a `MouseListener`, which is an interface with five methods: `mouseClicked`, `mouseEntered`, `mouseExited`, `mousePressed`, `mouseReleased` and assume that we only want to catch the event where the mouse is clicked, the other methods are supposed to be empty.

This can be solved by using the new trait feature. First we define a trait containing the empty mouse methods:

```
trait EmptyMouseListener {
  public void mouseClicked (MouseEvent e)
  public void mouseEntered (MouseEvent e)
  public void mouseExited (MouseEvent e)
  public void mousePressed (MouseEvent e)
  public void mouseReleased (MouseEvent e)
}
```

We could have decided that the default implementations of the listener methods should have some side effect for instance logging events, but right now we keep the method bodies empty. Then we define the `MouseListener` interface with `EmptyMouseListener`:

```
interface MouseListener {
  with {
        EmptyMouseListener;
  }
  public void mouseClicked (MouseEvent e);
  public void mouseEntered (MouseEvent e);
  public void mouseExited (MouseEvent e);
  public void mousePressed (MouseEvent e);
  public void mouseReleased (MouseEvent e);
}
```

Finally we define our class that should listen to the mouse clicked event. The class contains one simple method which prints out the number of times a mouse click event has happened:

```
class PrintClickCount implements MouseListener {
  private int numClicks = 0;
  public void mouseClicked (MouseEvent e) {
        numClicks++;
        System.out.println(”Number of clicks ” + numClicks);
  }
}
```

This problem could also be solved by using single inheritance. We could have created a superclass of `PrintClickCount` that provides us with empty methods of the `MouseListener` methods, and overriding only the `mouseClicked` method in `PrintClickCount`. This solution would, however, manipulate the class hierarchy and that might not always be a good idea.

Interface with traits can be useful in cases where interface methods can be implemented by using other methods within the same interface. For instance the `List` interface of the `java.util` package contains as many as 25 method declarations as of Java 1.5, and many of the methods are easy to implement using either one or more the other methods in the same interface. A couple of examples of this are as pointed out in section 1 where `isEmpty` can be implemented based on `size`. Another example is the the `add(Object)`, `addAll(Collection)` and `addAll(int, Collection)` can all be implemented based on the `add(int, Object)`. In fact the whole `List` interface could be implemented based on the four methods `size`, `remove(int)`, `get(int)` and `add(int, Object)`.

This interface with trait feature is similar to something called interfaces with default implementations [10], where an interface can contain a default implementation of any method signature inside the interface. The class implementing this interface can choose to use the default implementation of the interface method or provide its own.

Another practical use for default implementations is when an interface provides optional methods. Many of the methods of the `List` interface are optional. This could be implemented by using interface with traits in a way that the optional methods unless overridden throw an `UnsupportedOperationException`.

5. Conclusion

We have described the issues in designing and implementing traits in a statically scoped, strongly typed language, and we have described a prototype implementation of Java with traits. The grammar for the trait extension to Java has been covered and we have described how to implement that both classes and traits themselves may have traits, including flattening and type checking of requirements. In addition to the obvious feature that traits may have interfaces, we have described a new feature: interfaces with traits.

6. References

[1] Black, A., Schärli, N., and Ducasse, S.: *Applying traits to the Smalltalk collection hierarchy*, OOPSLA'03 - 17th International Conference on Object-Oriented Programming Systems, Languages and Applications, 2003.

[2] Denier, S.: *Traits programming with AspectJ*, Actes de la Premiére Journée Francophone sur le Développement du Logiciel par Aspects (JFDLPA'04), Paris, France, 2004.

[3] Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A.: *Traits: A mechanism for fine-grained reuse*, ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 28(2), pp. 331–388, 2006.

[4] Goldberg, A. and Robson, D.: *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983.

[5] Gosling, J., Joy, B., and Steele, G.: *The Java (TM) Language Specification*: Addison-Wesley, 1999.

[6] http://javacc.dev.java.net/: *JavaCC, Java Compiler Compiler - The Java Parse Generator*.

[7] http://javacc.dev.java.net/doc/JJTree.html: *JJTree ReferenceDocumentation*.

[8] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., and Kay, A.: *Back to the future: the story of squeak, a practical smalltalk written in itself*, OOPSLA '97: 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, 1997, ACM Press.

[9] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J.: *Aspect-orientedprogramming*, ECOOP'97 - European Conference on Object-Oriented Programming, 1997, Springer-Verlag.

[10] Mohnen, M.: *Interfaces with default implementations in Java* PPPJ '02/IRE '02: The inaugural conference on the Principles and Practice of programming, and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, National University of Ireland. Maynooth, County Kildare, Ireland, 2002.

[11] Nierstrasz, O., Ducasse, S., Reichhart, S., and Schärli, N.: *Adding Traits to (statically typed) languages*, Institut für Informatik, Universität Bern, Switzerland Technical Report IAM-05-006, 2005.

[12] Quitslund, P. J.: *Java traits — improving opportunities for reuse*, OGI School of Science & Engineering, Beaverton, Oregon, USA Technical Report CSE-04-005, 2004.

[13] Reichhart, S.: *A prototype of Traits for C#*. Informatikprojekt, University of Bern, 2005.