GENERIC TRAITS IN STATICALLY TYPED LANGUAGES
How to do It?

Andreas Svendsen[1,2] and Birger Mller-Pedersen[2]
[1]*SINTEF, Oslo, Norway*
[2]*Department of Informatics, University of Oslo, Oslo, Norway*

Keywords: Traits, statically typed languages, generics, templates, virtual types.

Abstract: Traits have been proposed as a code reuse mechanism for dynamically typed languages. The paper addresses the issues that come up when introducing traits in statically typed languages, such as Java. These issues can be resolved by introducing generic traits, but at some cost. The paper studies three different generic mechanisms: Java Generics, Templates and Virtual Types, by implementing them all by a preprocessor to Java. The three different approaches are tested by a number of examples. Based on this, the paper gives a first answer to which of the three generic mechanisms that are most adequate.

1 INTRODUCTION

The development of complex programs requires a good way of structuring source code. Modern object-oriented languages are often based on classes as the building blocks for code structure and for combined classification/code-reuse through class based mechanisms like inheritance. As demonstrated in (Ducasse et al., 2006) these mechanisms, however, prove to be inadequate in certain situations of code reuse, and traits are proposed as a complementary mechanism for structuring and reusing code.

A trait is defined as a collection of methods. A class can be defined by a combination of inheritance, variables and methods, traits and glue code to glue all pieces together (see Figure 1). In addition a trait may also be based upon other traits and thereby form a trait hierarchy.

Traits have three important semantic rules ((Nierstrasz et al.,), (Ducasse et al., 2006)):

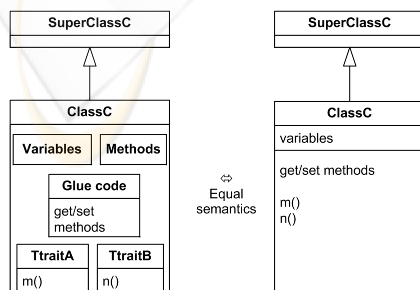


Figure 1: Class composition and flattening property.

- Methods in a class take precedence over methods in traits of the class, and methods in a trait take precedence over methods in the traits of this trait.
- Flattening property: A method has equal semantic properties whether it comes from a trait (including traits being used by the trait) or is directly defined in the class.
- The order of traits and trait methods is insignificant.

Flattening is the most important semantic rule for traits. It tells us that we can remove all trait-specific syntax and move every trait method directly into the class without changing the semantics of the program (illustrated in Figure 1). This makes traits ideal to be handled by a preprocessor, and it does not call for a major change to the Java language.

In case of large trait hierarchies, there may be conflicts between methods from different traits. The precedence rule is one solution, by introducing a new method that excludes the ones in conflict. If this is inadequate, i.e. if the functionality of one of the conflicting methods is needed, special operators for alias and exclusion can be used so that the method is available.

Methods of a trait may need to access methods that are not defined in the trait, and in order to guarantee that these methods are present (either in the class or in other traits used by the same class), these can be

Svendsen A. and Mller-Pedersen B. (2008).
GENERIC TRAITS IN STATICALLY TYPED LANGUAGES - How to do It?.
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 39-46
DOI: 10.5220/0001870700390046
Copyright © SciTePress

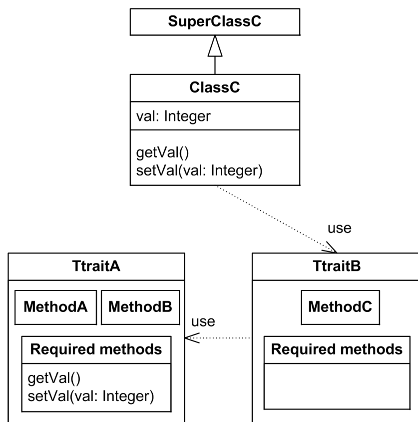


Figure 2: Trait composition forming a trait hierarchy.

specified as required methods. A simple trait hierarchy is illustrated in Figure 2. In this case the methods required from TraitA are defined in ClassC.

We have seen a general description of traits, and to illustrate the concept of traits further, concrete syntax in extended Java is given in the following code example. The trait block `TtraitA` from Figure 2 shows how methods and dependencies can be expressed in a trait.

```
trait TtraitA{
  use {}
  requires {
    getVal();
    setVal(Integer val);
  }
  void methodA(String a){...}
  void methodB(){...}
}
```

Traits were originally proposed for dynamically typed languages, such as Squeak, an open source dialect of Smalltalk. Introducing traits in statically typed languages raises some issues. There have been some work on how to resolve these issues ((Nierstrasz et al.,), (Reichhart, 2005)), even with generic mechanisms, such as type parameters. We shall look more closely into and compare the application of three different generic mechanisms: Java Generics, Templates and Virtual Types, and see how well these mechanisms work in combination with traits in Java. For each of these we shall also examine two different strategies for implementation of traits.

Our investigation is based on a preprocessor. An implementation directly in the language by means of a compiler could be a better solution. (Quitslund, 2004) has looked into the possibility of adding traits to Java as a compiler, but generic mechanisms are not considered. We want to illustrate how well generic mechanisms work with traits, and a preprocessor can give us a good indication.

To give a better understanding of the issues, we will first give an introduction to the issues with traits and statically typed languages, by means of an example. We will then discuss the need for generic traits and the choices we have, before we give a brief review of a possible implementation of generic traits in Java. Test runs show that there are issues with some of the generic mechanisms, which we will discuss before we conclude.

2 ISSUES WITH TRAITS IN STATICALLY TYPED LANGUAGES

In this section we introduce an example which will be used to explain how traits are used, and the issues that arise when we have traits in a statically typed language. We shall then see how generic traits can resolve these issues.

2.1 Example

Imagine that we have two types of lists, a list of `Process` objects and a list of `Dictionary` objects (see Figure 3). A list as such is represented by the first element (object of class `Process` or `Dictionary`), and each object of both `Process` and `Dictionary` will have a reference to the next element in the list, in addition to the `Process`- and `Dictionary`-specific properties. In addition each object will have methods to get and set both its properties and the reference to the next object in the list. This example has been inspired by (Nierstrasz et al.,).

Assume that the classes `Process` and `Dictionary` have their separate superclasses and therefore not any common superclass except `Object` (given that we only have single inheritance). Suppose we need some common functionality in objects in these lists, e.g. methods for reversing the list, copying the list and finding an element in the list. Since the objects do not have any common superclass, we cannot express the common functionality through single inheritance, but have to duplicate the code in each of the classes `Process` and `Dictionary`. Note that this is just a subset of all the functionality we would want in these kinds of lists, and it is used solely to illustrate how traits work.

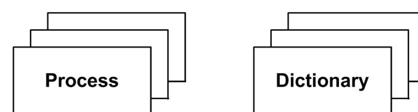


Figure 3: `Process` and `Dictionary` linked lists.

The trait solution is to place all common functionality in a trait, which then can be used by all classes that need it. In this case both the `Process` and `Dictionary` objects need some common functionality, and it will be a good choice to have their classes, `Process` and `Dictionary`, use the trait, TList, with this functionality. This is illustrated in Figure 4.

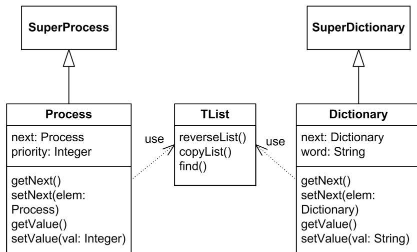


Figure 4: Two classes use functionality in a trait.

2.2 Issues

The `copyList` method in the trait `TList` will need to make a copy of every element in the list. This requires instantiation of new elements (of the right class), in which we are able to set values to variables and to set the next references. We also need to extract these values from the existing elements in the list to be copied. The method `copyList` will use the set/get methods offered by the elements of the list themselves. These methods are therefore listed as required methods by the trait.

There will be just one `copyList` method (in the common trait), and it has to work for both `Process` lists and `Dictionary` lists. As an example, the return type of `copyList` should therefore obviously be either `Process` or `Dictionary`. In a statically typed language, this type shall be specified in advance, but that is not possible, because the trait may be used by different list classes. We have similar problems with the other two trait methods as well.

The following code example illustrates how the `copyList` method would be for a `Process` list. This illustrates that we need a generic way of specifying the return type (`Process`), the type of the variables `newList`, `oldElem`, `temp` and `newElem`, so the trait method `copyList` can be used by `Dictionary` as well. Notice that we need to instantiate new objects of type `Process`, which indicate that the solution has to support instantiation of new objects of a generic type.

```
Process copyList(){
  Process newList = new Process();
  newList.setValue(getValue());
  Process oldElem = newList;
  Process temp = getNext();
  while (temp != null){
     Process newElem = new Process();
     newElem.setValue(temp.getValue());
     oldElem.setNext(newElem);
     oldElem = newElem;
     temp = temp.getNext();
  }
  return newList;
}
```

To overcome this obstacle (Nierstrasz et al.,) and (Reichhart, 2005) have suggested either to use interfaces as types instead of classes, to give traits types, or even to use generic mechanisms. As they have pointed out, some of these solutions have shortcomings. One is the extra work for the programmer in making new interfaces every time he or she wants a method to accept an element of more than one type. Another is that we cannot instantiate new objects from an interface, so return types cannot be specified by means of interfaces. These issues suggest that generic mechanisms may be the most promising solution.

3 GENERIC TRAITS

The above discussion on the issues of traits in a statically typed langauge proves that we need a powerful mechanism for expressing variability of trait methods.

The main aspects we want to vary in the trait methods are types of the parameters, types of temporary variables, and return types. Since we want to vary types, the use of type parameters is an obvious choice. This is a well known concept and is included in Java1.5 and many other statically typed languages. The three generic mechanisms we shall consider are the following:

- Java Generics: Since we develop a solution for Java, this is an obvious alternative.
- Templates: Replace type parameters by the actual type in the preprocessor. Similar to C++ templates.
- Virtual Types: Possibility to redefine the type of a virtual type. This allows us to specify the type of an element further when we have more information about that element.

In the following we provide a brief description of the three generic mechanisms and the differences between them.

3.1 Java Generics

Java Generics has been a part of the Java language since version 1.5. The main purpose of Java Generics is to offer type security by compile time and make

casts redundant (Mahmoud, 2004). Java Generics was designed with the Collection classes in mind, where `Object` is used as the common type of elements in collections. To maintain backward compatibility to older Java libraries, Java Generics was designed and implemented by type erasure. This results in equal runnable code for both traditional Java syntax and Java with generics. The Java compiler accomplishes this by replacing the type parameters by `Object` and inserting casts whenever necessary (Naftalin and Wadler, 2007).

Implementation by erasure gives some restrictions (Bracha, 2004). Since we do not have any representation of the generic types on runtime, we cannot instantiate new objects of type parameters, or make arrays of type parameters. We shall see later that these restrictions can be a problem if we extend traits with Java Generics.

3.2 Templates

Templates (as found in C++)are similar to Java Generics both in syntax and w.r.t. the representation at runtime. When using templates, type parameters will be replaced by the actual types by a preprocessor. Since this is done before we run the Java compiler, we have more flexibility on how this replacement is done. For instance we are able to instantiate new objects from type parameters, because type parameters are replaced with actual types before the instantiation is done.

One may question how well the semantics (or lack) of C++-like templates conform with Java. However, we believe that templates can solve many of the issues, and there this an viable alternative.

3.3 Virtual Types

The concepts of virtual types was introduced in the language Beta as virtual patterns (Lehrmann Madsen and Mller-Pedersen, 1989), and later considered for Java (Thorup, 1997). The semantic rules of virtual types are similar to the rules of virtual methods. A virtual method is first defined as general as possible in a general class, and can then in subclasses of this general class be redefined when we have more information about what functionality we need. The concept of a virtual type is similar: We first define a general type, and redefine it in subclasses to a more specific type.

Since we do not have support for virtual types in Java, we have to replace all virtual types with their redefinition types during the flattening process. This results in a mechanism which is practically similar to templates. The main difference is how and when the actual types are specified. This will be discussed further in section 5.2.

4 IMPLEMENTATION OF GENERIC TRAITS IN JAVA

To be able to compare the three generic mechanisms and to illustrate the differences between them, we have developed a preprocessor for Java with generic traits. The preprocessor is divided into three parts: Parsing, flattening and Java code generation.

To handle the parsing process and to automatically generate an abstract syntax tree (AST), we used AntLR (http://www.antlr.org/). We used an already available grammar (Studman, 2005), and extended this grammar with the necessary syntax for traits.

We found it convenient to develop a data structure that models the trait hierarchy. This structure contains functionality for getting information about the traits and relations between them. This makes it easier to perform the flattening process, especially since we want to do this in two ways.

The code generator is basically a pretty printer which outputs standard Java code. It traverses the syntax tree, but does not print any node that is part of the annotated trait syntax.

4.1 Two Flattening Strategies

We have examined two strategies for flattening (see Figure 5). The first one, which we have called Classic Flattening, is the strategy described in (Ducasse et al., 2006), (Nierstrasz et al.,), (Shärli et al., 2002) and (Reichhart, 2005), and involves moving every method of trait into the class that use the trai. This means that the trait blocks will be redundant, and are thus removed from the processed program.

The other strategy, which we have named Object Flattening, is a fairly different approach. Instead of moving every trait method into the class, we keep the trait blocks (as objects) and make references from the class to the trait block. To be able to keep the trait block in traditional Java syntax, we convert the traits into classes, and instantiate objects of these trait classes in the classes that uses the trait. This instantiation can be done hierarchically.

Every call of a trait method has to be updated to call the method of the correct object (the instantiated trait class). We search the syntax tree for every node which contains a method call, and update the reference according to a table of references to trait objects. A couple of problems raise in this process. To support

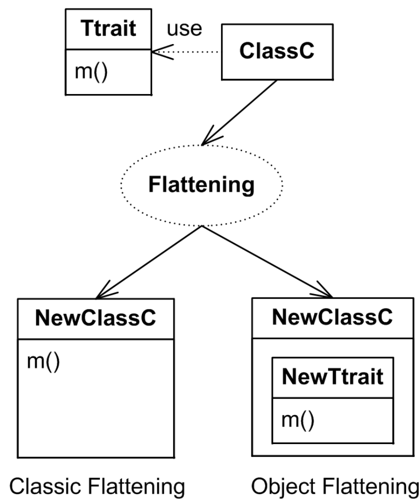


Figure 5: Two flattening strategies.

overloading of methods, we have to compare parameter types. This requires a static semantic check. We also notice that the trait methods can be called from other places than from within the classes which uses them, and to cover this we have to either make the preprocessor more sophisticated or to include a local method in every class which contains a method call to the right method. The first alternative is preferable, because of the extra overhead an extra method call creates. We will, however, not describe this any further.

Note that when we want to use Java Generics with Classic Flattening we might end up with a problem. We want to maintain the generic mechanism, but not the trait block which contains the type parameters. This can be solved by moving every type parameter from a trait hierarchy into the class which use the hierarchy. We also have to move the definition of the actual types to the instantiation of the class. It is also possible to require the programmer to include every type parameter in the class definition as well as in the trait that the class uses. This can to some extent simplify the processing.

4.2 Test Runs

To see how well the three generic mechanisms integrate with traits, and to illustrate the differences between Classic Flattening and Object Flattening, we have run the list example through the preprocessor. The preprocessor supports six options for flattening: (Java Generics, Templates, Virtual Types) x (Classic Flattening, Object Flattening).

With these six options, we can see how each generic mechanism performs with each flattening strategy. Another list example based on the Collection classes and an arithmetic example have also been processed. The test runs provide equal results w.r.t. the problems and benefits of the generic mechanisms and flattening strategies. This gives an indication of which generic mechanism and flattening strategy are preferable.

5 ISSUES WITH THE GENERIC MECHANISMS

Since all three generic mechanisms allow us to specify a general type and later redefine this to a more specific type, some of the problems we saw in section 2.2 can be solved. However, there are some differences between the generic mechanisms. For Templates and Virtual Types the type parameters are bound by the preprocessor, while for Java Generics this is done by the Java compiler.

The test results show that Java Generics may not be so well suited for this purpose as the other two mechanisms.

5.1 Java Generics

One of the purposes of Java Generics is to make casts redundant. As mentioned, Java Generics was designed with the Collection classes in mind, where `Object` is used as the common type of elements in collections. This becomes more flexible with Java Generics. However, Java Generics may not be so well suited for making traits more general. Our test runs indicate a couple of problems with traits and Java Generics:

- Unable to make new objects and arrays according to a type parameter.
- Necessary to add casts after the flattening process.

The first problem is encountered in the `copyList` method (see section 2.2), which needs to instantiate new objects according to a type parameter.

To illustrate the second problem, the need to add casts after the flattening process, we can look at a flattened `Process` class below.

```
class Process <T> {
  Process next;
  int priority;

  // [..] get and set methods
  // [..] copyList and find

  T reverseList(){
    T temp = this;          //missing cast
    T prev = null;
```

```
    T next = null;
    while(temp != null){
      next = temp.getNext(); //missing casts
      temp.setnext(prev);    //missing casts
      prev = temp;
      temp = next;
    }
    return prev;
  }
}
```

The reverseList method generates errors when compiled by the Java compiler. Casts between the type T and the type Process are missing. The correct reverseList method with the necessary casts added is shown below.

```
T reverseList(){
  T temp = (T) this;
  T prev = null;
  T next = null;
  while(temp != null){
    next = (T) ((Process) temp).getNext();
    ((Process) temp).setnext((Process) prev);
    prev = temp;
    temp = next;
  }
  return prev;
}
```

A more sophisticated preprocessor can resolve some of these issues. Casts can be automatically added by the preprocessor, and methods with special expressions (newExpr) that are based on type parameters can have the type parameters automatically resolved and copied to the respective class. However, this requires the preprocessor to perform semantic checks of the program, and it requires a complex implementation of the preprocessor. By resolving the type parameters and copying the methods into the class we obtain a template-like mechanism, and this indicates that Templates may be a better mechanism for this purpose.

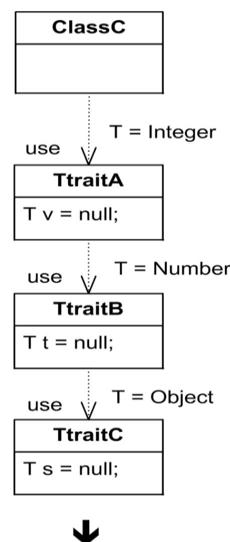
5.2 Templates vs. Virtual Types

Since there is no support for either Templates or Virtual Types in Java, we have to translate the generic trait syntax into plain Java. By binding the type parameters (template parameters) and virtual types in the preprocessor we also get some extra flexibility. Since the preprocessor substitutes the types during flattening, we do not have the same issues as with Java Generics. This is illustrated in the following reverseList method below, which is generated with Templates-alternative.

```
Process reverseList(){
  Process temp = this;
  Process prev = null;
  Process next = null;
  while(temp != null){
    next = temp.getNext();
    temp.setnext(prev);
    prev = temp;
    temp = next;
  }
  return prev;
}
```

Since the use of both Templates and Virtual Types implies type substitution by the preprocessor, the two generic mechanisms may seem similar. However, there are some differences:

- Semantic Difference. A Template type parameter can have any type as actual type, even any primitive type. A virtual type has to be redefined to a subtype of the original (virtual) type.
- Different Binding. Templates keep the bindings at all levels of the trait hierarchy, while for virtual types the outermost binding has effect for all uses of the virtual type in the trait hierarchy. This is illustrated in Figure 6. The type parameter (or virtual type) T is bound to Object, Number and Integer. If T is a template parameter, these bindings have affect on the different uses of T. If T is a virtual type, the outermost binding (T = Integer) will affect all uses of T in the trait hierarchy.



Templates:	Virtual Types:
s: Object	s: Integer
t: Number	t: Integer
v: Integer	v: Integer

Figure 6: Type binding for Templates and Virtual Types.

For a Template type parameter we have to specify an actual type at each trait level, even if the type is equal to the binding at the previous level. For Virtual Types we only redefine the virtual type if the actual type is not equal to the previous actual type.

Since we redefine virtual types outwards in the hierarchy, one should be careful about the names of virtual, in order to avoid redefining a not-intended virtual type. Assume for instance that a trait `A` uses two other traits `B` and `C`, which both defines a virtual type `T`. Trait `A` needs to redefine the `T` from `B`, but wants to keep `T` from `C` as it is. We are unable to express this with Virtual Types without changing the name of `T` in either `B` or `C`. Since Templates redefine type parameters inwards the hierarchy this is not an issue. However, with a strict naming policy we can avoid these issues with Virtual Types.

Since templates can bind a type parameter to any type, including primitive types, we will get a type safety issue. Since our solution is based on a preprocessor that generates Java code, static semantic errors will be handled by the Java compiler. However, if we want to implement generic traits into the language, this may be a large issue that requires extensive static semantic checking. Virtual Types may therefore be a better solution if we want to integrate generic traits into Java.

Redefinition of Virtual Types is restricted to subtypes of the original virtual type. This imposes that the original virtual type has to be general enough, and this may limit the usability of Virtual Types. However, since every type in Java is a subtype of `Object`, this is a viable alternative.

5.3 Classic Flattening vs. Object Flattening

Our test results indicate that Classic Flattening works without any problems with both Templates and Virtual Types. Some issues arise when we want to use Classic Flattening together with Java Generics, since the trait blocks are removed by the preprocessor. Object Flattening addresses this concern by keeping the trait blocks by converting them to classes.

Since the trait blocks are maintained, high level programming concepts can be applied, e.g. reflection. More advanced flattening can be the result since we have more information about the traits available at runtime.

However, there are some difficulties with Object Flattening:

- More complex than Classic Flattening.
- Different semantics for the keywords 'this' and 'super'.
- More overhead due to extra references.

Object Flattening does not remove the trait blocks and does not copy the methods into the classes that use the traits. This requires an update of method calls since we want them to call methods of the objects corresponding to the trait blocks. This leads to a more complex preprocessor that generates more complex Java code. Another factor is more overhead since the method cannot be looked up locally when Java performs the method call. These issues are illustrated in Figure 5 where we have to refer to different objects, `NewClassC` (Classic Flattening) and `NewTrait` (Object Flattening) objects, when we need to perform a call to m.

Since trait methods by Object Flattening is contained in a trait block object, the keywords 'this' and 'super' do not refer to the class that uses the trait, but refer to the trait block object instead. This may require a rewrite of source code or an even more complex implementation of the preprocessor.

6 CONCLUSIONS

We have developed a preprocessor for generic traits in Java in order to compare three approaches to generic traits and two strategies for flattening. One of the main concerns with generic traits is that they should be easy to use and implement. Our test results indicate the following:

- Java Generics, as it is today, is not sufficient to make traits more general. In other languages, for instance C#, where generics is not implemented by erasure, the conclusion may be different.
- Templates and Virtual Types solve the issues introduced by static typing. Virtual Types probably conforms better to Java semantics than Templates do.
- Classic Flattening is both easier to use and to implement than Object Flattening. However, there are some advantages with Object Flattening, i.e. easier integration into Java since trait blocks are not removed, but this area requires more research.

Our results are based on a preprocessor that generates Java code that can be compiled by a Java compiler. This does not require any changes of the JVM, and is one of the advantages with traits. However, a real implementation of generic traits in Java may still be a better approach.

Virtual Types implemented with traits in Java is probably one of the best approaches and is an area that requires more investigation.

7 RELATED WORK

(Reichhart, 2005) describe an implementation of traits in C♯. The difficulties and possible solutions of typing and parameterizing traits are shown. By presenting a simple prototype, the possibilities and difficulties of integrating traits in statically typed languages are discussed.

(Nierstrasz et al.,) realize that traits can offer clear benefits for statically typed languages. The issues of integrating traits into such languages are summarized. Traits are examined in the context of the languages Java, C♯ and C++.

An implementation of traits as a native concept of a programming language (Fortress) is described by (Allen et al., 2007). The Fortress Programming Language is a general-purpose, statically typed programming language. There are two basic concepts in Fortress: object and trait. Much of the type system is based on trait types, and every trait extends the `Object` trait. Static parameters include type parameters.

8 FUTURE WORK

Generic traits with virtual types as a language concept in Java is an area that requires more investigation.

An implementation of the preprocessor based on NextGen http://japan.cs.rice.edu/nextgen/, which is a version of Java that supports runtime representation of generic types, would be an interesting approach to the problems with Java Generics.

We have seen that Object Flattening has many issues and is more complex than Classic Flattening. However, a more sophisticated implementation of the preprocessor with Object Flattening may give some benefits, since we keep the trait blocks after flattening.

By using the representation of generic traits presented in this article, a study of how large impact generic traits will have on the Java library should be initiated.

ACKNOWLEDGEMENTS

The work reported in this paper has been done within the context of the SWAT project and funded by The Research Council of Norway, grant no. 167172/V30. It is based on a master thesis by Andreas Svendsen.

REFERENCES

Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G. L., and Tobin-Hochstadt, S. (2007). The Fortress Language Specification. http://research.sun.com/projects/plrg/fortress.pdf.

Bracha, G. (2004). Generics in the Java Programming Language. http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

Ducasse, S., Nierstrasz, O., Schrli, N., Wuyts, R., and Black, A. P. (2006). Traits: A Mechanism for Fine-grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388.

Lehrmann Madsen, O. and Mller-Pedersen, B. (1989). Virtual Classes - A Powerful Mechanism in Object-Oriented Programming. In *OOPSLA '89: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*, pages 397–406, New York, NY, USA. ACM.

Mahmoud, Q. H. (2004). Using and Programming Generics in J2SE 5.0. http://java.sun.com/developer/technicalArticles/J2SE/generics/.

Naftalin, M. and Wadler, P. (2007). *Java Generics and Collections*. O'Reilly Media.

Nierstrasz, O., Ducasse, S., Reichhart, S., and Schrli, N. Adding Traits to (Statically Typed) Languages. Technical Report IAM-05-006.

Quitslund, P. J. (2004). Java Traits - Improving Opportunities for Reuse. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA.

Reichhart, S. (2005). Traits in CSharp. http://www.iam.unibe.ch/s̃cg/Archive/Projects/Reic05a.pdf.

Shärli, N., Ducasse, S., Nierstrasz, O., and Black, A. (2002). Traits: Composable Units of Behavior. Technical report.

Studman, M. (2005). Java 5 (aka 1.5) grammar. http://www.antlr.org/grammar/1090713067533/index.html.

Thorup, K. K. (1997). Genericity in Java with Virtual Types. In *ECOOP'97 - European Conference on Object-Oriented Programming*, volume 1241, pages 444–471. Springer Verlag.