RWTH Aachen

Department of Computer Science
Technical Report

Interfaces with Default Implementations in Java

Markus Mohnen

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2002-09

RWTH Aachen · Department of Computer Science · April 2002

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Interfaces with Default Implementations in Java

Markus Mohnen

Lehrstuhl für Informatik II, RWTH Aachen, Germany
mohnen@informatik.rwth-aachen.de

Abstract. With the interface construct, Java features a concept with high potential for producing reusable code: Java's interfaces allow the definition of class properties independently of class inheritance. We propose an extension of Java for providing default implementations in interfaces. Default implementations are useful since they reduce the effort required to implement an interface. They are especially interesting if there is a canonical way to implement methods of the interface in terms of some other methods. In these cases, an implementation can be obtained by implementing the base methods and use the default implementations of the other methods. We discuss the rationale for our design and show that the extension can be implemented both efficiently and conservatively, i.e. without modification of the Java virtual machine.

1 Introduction

The interface construct in Java allows the definition of properties which can be implemented by classes. An interface in Java just contains names and signatures of methods and fields, but no method implementations. If a class *implements* in interface, it must provide implementations for the methods.

Since interfaces can be used just like classes in declarations and signatures, it is possible to base programs on *properties of classes* instead of classes. Furthermore, the inheritance hierarchy of interfaces is independent of the class inheritance tree. Therfore, this language feature gives a higher potential to produce reusable code than *abstract classes*, i.e. classes where the implementation of some methods is omitted. The Java 2 API makes extensive use of interfaces. For instance, the package `java.util` contains six interface hierarchies.

In many cases, it is useful to equip an interface with a set of *default implementations* of methods since they reduce the effort required to implement an interface. They are especially interesting if there is a canonical way to implement functions of an interface in terms of some other functions. In these cases, an implementation can be obtained by implementing the base methods and use default implementations of the other methods.

The usefulness of default implementations has also been seen by the Java developers. In the Java 2 API there are several abstract classes which provide default implementations for some of the interfaces. For instance, in the package `java.util`, the class `AbstractCollection` is an abstract class implementing `Collection` except for the methods `iterator` and `size`.

However, the approach of adding abstract classes containing default implementations has drawbacks for the implementation of the interface and for the implementation of the abstract classes:

– To use the default implementation of the interface, a class must extend the abstract class. Consequently, the programmer is no longer free to create an inheritance hierarchy matching the needs of the problem at hand. Altogether, this abolishes the advantages of interfaces over abstract classes.
– To provide default implementations for all interfaces in an interface hierarchy with multiple inheritance, it is unavoidable to duplicate code in the abstract classes.

The contribution of this paper is a proposal for an extension of Java which allows the direct definition of *default implementations* in interfaces and avoids the drawbacks imposed by their simulation in terms of abstract classes. We provide a complete design of this extension and discuss the interaction of default implementations with Java's inheritance mechanisms.

An explicit goal of our work was to propose a conservative extension. This is important in two aspects: (1) The semantics of all existing programs remain unchanged. (2) The extension can be implemented efficiently by using the existing Java virtual machine.

This paper is the successor of [16] and improves the work presented there in terms of a more efficient and simpler implementation and in terms of better syntax of the proposed extension.

The paper is organised as follows. In the next section we give an overview of related work. An example of the usefulness of interfaces is presented in Section 3. In addition, this section describes how default implementations are simulated in the Java 2 API. In Section 4 we describe our extension to the Java language. Section 5 describes how we propose to implement the language extension in terms of a translation to standard Java. Section 6 concludes.

2 Related Work

Interfaces are not novel to Java. Similar constructs are known for instance in the context of design patterns (*template methods* in [9]), and have been used in other languages: In the object-oriented language Actor [24], the corresponding construct is named *protocol*. The functional language Haskell [11] features *type classes*. In contrast to these approaches, however, Java lacks the feature of providing default implementations in interfaces. Another object-oriented language with an interface-like construct without default implementations is POOL-I [1]. Here, the construct is called *type*.

Since Java allows multiple inheritance of interfaces, the introduction of default implementations is related to *multiple inheritance* of method implementations from more than one superclass. Consequently, it might be argued that our proposal is weaker than extending Java with full multiple inheritance. However, we belive that the Java language design excluded this for good reasons. A lot of research has been done on full multiple inheritance and it turned out that the main problem is how to define a satisfactory general strategy for choosing or combining inherited method implementations, which is at the same time intuitive and easy to use.

- One of the most general approaches was taken in CLOS [12, 13]. Here, a method can be declared to be either `primary`, `before`, `around`, or `after`. These attributes control if, and in which order the corresponding methods of the immediate superclasses are called. In addition, the programmer has control over how the results of the methods calls are combined to the final result. While this approach grants almost total control over method combination, it also introduces a high degree of complexity in the language. This might be the reason why newer languages resign from this approach in favour of simpler solutions.
- Eiffel [15] also allows multiple inheritance of methods, but differs from CLOS in the way this is implemented. While CLOS requires linearisation of the inheritance hierarchy, Eiffel implements it directly. The approach avoids the problems of the encapsulation violation [20] resulting from the linearisation in CLOS.
- While multiple inheritance is still allowed in C++ [21, 22], this language has a more restrictive way of handling inherited method implementations: If a method is inherited from more than one superclass, calls to this method must be qualified with the name of the superclass. The disadvantage of this approach lies in the resulting dependencies between unrelated classes: To use a class with methods inherited from more than one superclass, the programmer is forced to *encode parts of the inheritance hierarchy*. Consequently, resulting programs become harder to maintain.
- The newest object–oriented languages Ada95 [7, 3], and Java [10] disallow multiple inheritance of method implementations.

Another related thread of research are *mixins* [5, 8]: While inheritance allows for the creation of one new class, based on zero or more superclasses and a specification of an increment, mixins liberate the increment such that it can be re-used and applied to different superclasses. Of course, this is also a restricted form of multiple inheritance, where the increment is used as an (abstract) superclasses. Not surprisingly, default implementations of interfaces can be simulated by placing the default implementations in a mixin. However, we consider this to be a pollution of the mixin concept, since the increment represented by the mixin is not a freely reusable component: The mixin can only be used when the interface is also used.

3 Interfaces in Java

In this section, we introduce our running example and use it to demonstrate the usefulness of Java's interfaces.

3.1 Applications of Interfaces

In general, interfaces are useful to avoid that related classes have to share a common (abstract) superclass. Instead, classes can support multiple common behaviours by implementing multiple interfaces. For instance, the code in Fig. 1 is taken from a package we developed for modelling mathematical structures. The structures we consider here are the following:

Fig. 1 Modelling Mathematical Structures

```
interface Set {
  boolean isElement(Object e);
  java.util.Enumeration elements();
}
```

(a) Sets

```
interface POSet extends Set {
  boolean le(Object e1, Object e2) throws IllegalArgumentException;
  boolean lt(Object e1, Object e2) throws IllegalArgumentException;
}
```

(b) Partially Ordered Sets

```
interface LSLattice extends POSet {
  Object meet(Object e1, Object e2) throws IllegalArgumentException;
}
```

(c) Lower Semi Lattice

```
interface USLattice extends POSet {
  Object join(Object e1, Object e2) throws IllegalArgumentException;
}
```

(d) Upper Semi Lattice

```
interface Lattice extends LSLattice, USLattice { }
```

(e) Lattice

Sets (Fig. 1(a)) in the mathematical sense (not to be confused with the collection data structure `java.util.Set`) can be used to check if an element belongs to this set by using the method `isElement`. Furthermore, a set "knows" all its elements and makes the accessible as a `java.util.Enumeration` through the method `elements`.

Partially Ordered Sets (Fig. 1(b)) have an additional binary relation "less-or-equal", represented as method `le`. The relation must be reflexive, transitive, and anti–symmetric, but of course these properties cannot be guaranteed in this context. Often it is useful to have the strict portion "less-than-but-not-equal" separately as method `lt`. Obviously, it should hold that the value of `le(e1,e2)` is equal to the value of `lt(e1,e2)||e1.equals(e2)`.

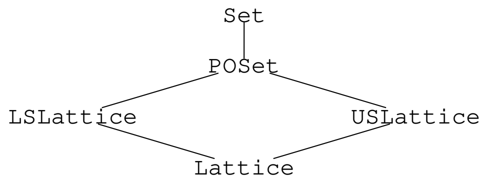
Lower Semi Lattices (Fig. 1(c)) are partially ordered sets with the additional property that the greatest lower bound (`meet`) of any two elements exists. Again, conditions which cannot be expressed here are: (1) `meet` should be commutative and associative and (2) `le(e1,e2)` should be equal to `e1.equals(meet(e1,e2))`.

Upper Semi Lattices (Fig. 1(d)) are dual to lower semi lattices: The operation `meet` for greatest lower bound is replaced with the operation `join` for least upper bound.

Lattices (Fig. 1(e)) are both lower and upper semi lattices. The absorption laws, i.e. `e1` must be equal to both `meet(join(e1,e2),e1)` and `join(e1,meet(e1,e2))` cannot be expressed in Java.

Since the methods `lt`, `le`, `meet`, and `join` are applicable only to elements of the corresponding structure, these methods throw a `IllegalArgumentException` if this condition is violated. In fact, the `throws` clause would not be necessary since any method can throw this exception in Java; However, we included it anyway.

Fig. 2 Interface Hierarchy staring from `Set`:



In this example we can see two variations of the of interfaces:

1. As a definition of properties, like the interfaces `Set`, `POSet`, `LSLattice`, and `USLattice`.
2. As a collection of properties, like the interface `Lattice`.

These interfaces form the following inheritance hierarchy in Fig. 2.

A class `VectorSet` implementing the interface `Set` in shown in Fig. 3. Since the class `java.lang.Vector` already contains an appropriate `elements` method, we can avoid implementing one by using `Vector` as superclass. The implementation of the method `isElement` is *generic* it the sense that it works for any class implementing `Set`.

It can be argued that this example is not the best possible, since we can easily avoid extending `Vector` by encapsulating a field of type `Vector` in an implementation of `Set`. The missing `elements` method can easily be provided as a stub which redirects to the `elements` method of the encapsulated field. With this approach, an implementation of `Set` would be free to extend other classes.

However, in principle, this argument is always possible. Replacing inheritance by membership and adding stubs for relevant inherited methods is a well known technique. Obviously, the resulting class is a simulation of the original class with a similar interface. However, it also clear that the resulting class is not equivalent to the original class, simply because it cannot be used in a context where the superclass is needed. In a certain sense, it is the opposite of object-orientation, since we have to add stubs where inheritance could be used to reuse existing solutions. Whether this technique is appropriate or not always depends on the complexity of the classes involved and on the intended use of the classes. In our example, it would be applicable, since the classes are simple and there is only one stub.

Nevertheless, in view of these objections, we still favour this example. In contrast to other examples used in literature (which typically involve classes representing documents, employees, or aircrafts) our example is fully self contained and demonstrates all effects. Its major flaw is that it might be considered too small to need the attention.

3.2 Default Implementations using Abstract Classes

In general, default implementations of methods can reduce the amount of effort required to implement an interface. Furthermore, there are two special situations where default implementations are useful:

Fig. 3 An Implementation of `Set`

```
class VectorSet extends java.util.Vector implements Set {
  public boolean isElement(Object e) {
    for (java.util.Enumeration es=elements();es.hasMoreElements();) {
      if (e.equals(es.nextElement())) return true;
    }
    return false;
  }
}
```

1. There is a standard way to implement a method using other methods. For instance, the method `isElement` from interface `Set` in Fig. 1(a) can be implemented using the (generic) method `elements` as shown in the class `VectorSet` from Fig. 3.
2. Additional conditions must be fulfilled. For example, reconsider the conditions for the methods `le` and `lt` in the interface `POSet` from Fig. 1(b): `le` must be reflexive, transitive, and anti-symmetric, and it should hold that `le(e1,e2) == (lt(e1,e2)||e1.equals(e2))`. Obviously, it is appealing to provide default implementations which ensure that these conditions are met.

We demonstrate how default implementations can be provided by using abstract classes and we show the deficiencies of this approach. The Java 2 API uses this method to a large extent.

Fig. 4 shows default implementations in this style for two of the interfaces from the previous section: The abstract class `AbstractSet` in Fig. 4(a) contains the generic implementation of the method `isElement` which we already used in `VectorSet` from Fig. 3. A full implementation of the interface `Set` can be obtained by extending `AbstractSet` with a method `elements`. The modifier `abstract` in the declaration of the class `AbstractPOSet` in Fig. 4(b) is actually not required from the Java type system, since neither does `AbstractPOSet` contain abstract methods nor does it omit to implement methods from the interface `POSet`. However, it provides default implementations for both `le` and `lt`, each in terms of the other. Consequently, using both default implementations *at the same time* would result in a nonterminating recursion. To prevent that `AbstractPOSet` is used directly without overriding one of the methods, the modifier `abstract` was added.

Using abstract classes to provide default implementations has several severe disadvantages:

1. A class implementing an interface by extending the corresponding abstract class is *no longer free to extend other classes*. Consequently, the class `VectorSet` from Fig. 3 cannot be expressed in terms of `AbstractSet`.
2. Since abstract classes cannot inherit from more than one abstract class, this approach leads to *code duplication* in the case that the interfaces use multiple inheritance. For instance, assume that we have abstract classes `AbstractLSLattice` and `AbstractUSLattice` both extending the abstract class `AbstractPOSet` and implementing the interfaces `LSLattice` and `USLattice`, respectively. Since multiple inheritance of classes is prohibited, the straightforward way of defining a class `AbstractLattice` implementing the interface `Lattice` by extending both the class `AbstractLSLattice` and the class `AbstractUSLattice` is

Fig. 4 Default Implementations using Abstract Classes

```
abstract class AbstractSet implements Set {
  public boolean isElement(Object e) {
    for (java.util.Enumeration es=elements();es.hasMoreElements(); ) {
      if (e.equals(es.nextElement())) return true;
    }
    return false;
  }
}
```

(a) Default Implementation of `Set`

```
abstract class AbstractPOSet extends AbstractSet implements POSet {
  public boolean lt(Object e1, Object e2) throws IllegalArgumentException {
    if (!isElement(e1) || !isElement(e2)) throw new IllegalArgumentException();
    return le(e1,e2) && !e1.equals(e2);
  }
  public boolean le(Object e1, Object e2) throws IllegalArgumentException {
    if (!isElement(e1) || !isElement(e2)) throw new IllegalArgumentException();
    return lt(e1,e2) || e1.equals(e2);
  }
}
```

(b) Default Implementation of `POSet`

not possible. Instead, an abstract class `AbstractLattice` can only be defined by extending either `AbstractLSLattice` or `AbstractUSLattice`; In both cases, the default implementation of one the method `join` and `meet` must be duplicated from the missing class.

3. The default implementations are located in a separate class, which might even be in a separate compilation unit.

4 An Extension of Java

Several extensions of Java have been considered in literature, e.g. functional constructs in Pizza [17] (now superseded by GJ [6], which is no longer promoted as functional), virtual types [23], and parametric types [2].

Having identified default implementations as a useful language feature, we propose an extension of Java. With this language extension, interfaces can be augmented with default implementations, and classes and interfaces can refer to those. We deliberately define default implementations of methods such that they are *not automatically used* in the absence of an explicit implementation. To use a default implementation, a class or interface must explicitly state this. There are two main reasons for this decision:

1. We avoid the problems with method combination and method selection which would occur otherwise in the context of multiple inheritance of interfaces.
2. Our extension is backward compatible with the existing Java 2 in the following sense: Assume we have an interface which provides default implementations for some of its methods. If a class implementing the interface has no references to the default implementation, then its semantics is the same with and without our extension. If we would choose to use default implementations of methods automatically, then we would effectively change the semantics of existing classes.

4.1 Specification

We reuse the Java keyword `default` for our purposes. Our extension allows this keyword to be used as an additional modifier of method definitions in interfaces. In this way, we allow the declaration of default implementations. Following the syntactical structure of Java defined in [10], we extend the definition of *AbstractMethodModifier* [10, §9.4]:

AbstractMethodModifier: one of
    `public abstract default`

To allow the use of default implementations, we change the definition of *MethodBody* [10, §8.4.5] in the following way:

MethodBody:
    Block
    DefaultUse
    `;`

DefaultUse:
    = *DefaultSpec* `;`

DefaultSpec:
    `default`
    TypeName `. default`

The syntactic alternative *DefaultUse* in the rule for *MethodBody* is new.

Furthermore, we require the following additional context sensitive conditions to be fulfilled:

1. A declaration may not contain both the modifier `abstract` and the modifier `default`.
2. The declaration of a default implementation, i.e. one with the modifier `default`, must either be accomplished by a body or by a *DefaultUse*. This changes the condition in [10, §9.4], which states that "Every method declaration in the body of an interface is implicitly abstract".
3. A *DefaultUse* may occur both in interface definitions and class definitions. In addition, the condition in [10, §8.4.5] that "The body of a method must be a semicolon if and only if the method is either `abstract` or `native`" remains valid. Consequently, the body cannot be *DefaultUse* in these cases.
4. The unqualified form of *DefaultUse* may be used if and only if there is exactly one direct superinterface which has a default implementation for the method.
5. For the qualified form of *DefaultSpec*, the *TypeName* must be the name of one of the direct superinterfaces and this direct superinterface must contain a default implementation for the method.

If a *DefaultUse* occurs in a method of an interface, then it operates as a declaration of a default implementation for the method. This is useful for passing default implementations from a superinterface, since default implementations not inherited automatically. Note that default implementations coming from superinterfaces which are more than one step higher in the interface hierarchy may not be used. Hereby, we avoid that large

Fig. 5 Default Implementations using Language Extension

```
interface Set {
  default boolean isElement(Object e) {
    for (java.util.Enumeration es=elements();es.hasMoreElements();) {
      if (e.equals(es.nextElement())) return true;
    }
    return false;
  }
  java.util.Enumeration elements();
}
```

(a) Interface `Set` with Default Implementation

```
class VectorSet extends java.util.Vector implements Set {
  public boolean isElement(Object e) = default;
}
```

(b) An Implementation of `Set` using Default Implementation

```
interface Lattice extends LSLattice, USLattice {
  default Object meet(Object e1, Object e2) throws IllegalArgumentException
    = USLattice.default;
  default Object join(Object e1, Object e2) throws IllegalArgumentException
    = USLattice.default;
}
```

(c) Default Implementations vs. Multiple Inheritance

parts of the interface hierarchy are used. Furthermore, we achieve a behaviour similar to the one of `super`.

For instance, a reformulation of the interface `Set` using this extension is shown in Fig. 5(a). It combines the pure interface from Fig. 1(a) with the generic implementation of the method `isElement` from Fig. 3. The class `VectorSet` can then be written by using the unqualified use. The qualified use is needed in the context of multiple superinterfaces, as demonstrated in Fig. 5(c). Because the methods `meet` and `join` both are inherited from exactly on superinterface, it would not be necessary to qualify the use in this example. However, we added the qualifiers to increase readability of this example.

In addition to the use of default implementations as declarations of methods, we allow default implementations to be invoked explicitely. This is similar to the use of `super` in ordinary methods. Therfore, we also change the definition of *MethodInvocation* [10, §15.11]:

MethodInvocation:
  MethodName (*ArgumentList*$_{opt}$)
  Primary **.** *Identifier* (*ArgumentList*$_{opt}$)
  `super` **.** *Identifier* (*ArgumentList*$_{opt}$)
  DefaultSpec **.** *Identifier* (*ArgumentList*$_{opt}$)

The last syntactical alternative is new. In contrast to the use of default implementations as declarations of methods in *DefaultUse*, we require that the name of the method is appended. This is useful, since it allows the use of default implementations in all methods. The additional context sensitive conditions are essentially the same we introduced above.

4.2 Discussion

The language extension we propose introduces default implementations in Java interfaces. We have deliberately defined the extension in such a way that default implementations of methods are not automatically used in the absence of an explicit implementation. Therfore, we can avoid the method combination/selection problems of multiple inheritance of classes. Furthermore, this decision makes our extension *conservative*, in the sense that the semantics of all existing Java programs remains unchanged. Consequently, this decision is the one which fits Java best.

It is important to see that our extension is not intended to be an simulation of full-featured multiple inheritance. In contrast to the developers of Actor, who promoted protocols as "safe multiple inheritance" [25], we see interfaces as an orthogonal language element. Its major purpose is to group kindred classes. Hence, the introduction of default implementations in interfaces does not contradict the design decision of the Java developers to disallow multiple inheritance of classes.

On the other hand, it would be possible to allow the definition of default implementations in *classes* also. Here, the presence of `default` would act opposing to the qualifier `abstract`, resulting in four instead of three modes:

`abstract`: Overriding and hiding is allowed and inheriting happens automatically. If a `abstract` method is not overridden, the extending class must be declared as `abstract`.

`default`: Overriding and hiding is allowed and inheriting does not happen automatically. If a `default` method is neither chosen nor overridden, the extending class must be declared as `abstract`.

normal: Overriding and hiding is allowed and inheriting happens automatically.

`final`: Overriding and hiding is prohibited and inheriting happens automatically.

However, we are not sure if this approach is really interesting: In classes, the established mechanism of providing an implementation is obviously the definition of methods. If a subclass chooses not to use this implementation, it can simply override it. Consequently, although this further extension would do no harm, it is unclear which benefits it would bring.

5 Implementation

In this section we demonstrate the basic concepts involved in implementing the extension. We are able to implement the additional features such that no change to the design of the Java virtual machine is needed. Hence, code generated from programs using the extension can be executed on any implementation of the Java virtual machine.

Apart from the obvious extension of lexical and syntactical analysis, translating the new language constructs involves two major tasks:

1. Checking the additional context sensitive conditions imposed by our extension.
2. Generating code for the declaration of default implementations, for *DefaultUse*, and for the new alternative of *MethodInvocation*.

We start by discussing Item 2 since the solution for Item 1 depends on the way the code is generated.

5.1 Translation to Standard Java

We characterise the code generation for the new language constructs by a translation to (standard) Java. Using this approach, we accomplish two targets: Firstly, the implementation is conservative in the sense that it does not need an extended JVM. Secondly, the translation is much simpler than a direct generation of JVM code.

Translation of Method Bodies in Interfaces

To translate an interface I with default bodies to a standard Java interface, we recall that in Java all methods reside inside classes; Standalone top–level methods do not exist. Consequently, we must locate all default implementations of I inside a new class. To emphasise the binding with I we create the class as an *inner class* of the translated interface. The new inner class is named `$default` to avoid name clashes with other inner classes of I (The Java specification allows mechanical translators the use of a dollar sign in names). By placing the class inside the interface, we also avoid clashes between translations of different default implementations. Of course, we also could have avoided these by including the name of the interface in the name of a new top level class. However, this is exactly the way the Java compilers translates inner classes: By translation to top–level classes with qualified names not usable by programs. Therefore, we can rely on the Java compiler to resolve the names correctly.

Since we use the inner class `$default` only as container for the method implementations, we will never create objects of this class or create subclasses. Hence, it is a direct subclass on `java.lang.Object`. Being a member class of an interface also means that `$default` is implicitly declared `static`, i.e. the nesting is only relevant to scoping and has no effect on instances of I.

For each method m in I with a default body B, we create a `static` method m with body B'in `$default`. Since m is `static`, we can use it without having to instantiate `$default`. However, in order to allow B' to call other methods of I, we have to have a reference to the current instance of I. Therefore, we have to explicitely pass the current instance in an additional parameter of m. Since this is exactly what the standard translation of virtual (non–`static`) methods does by the implicit `this` parameter, we name the new parameter `thiss` and it is of type I. Consequently, we obtain B' from B by prefixing all unqualified method invocations with `thiss`.
Formally, we define the translation from extended Java to Java for such an interface

$$
\begin{array}{l}
\mathit{mod}\ \texttt{interface}\ I\ \texttt{extends}\ \mathit{sup}\ \texttt{\{} \\
\quad M \\
\quad \texttt{default}\ \mathit{mod_1}\ T_1\ m_1(S_1)\ \texttt{\{}B_1\texttt{\}} \\
\quad \dots \\
\quad \texttt{default}\ \mathit{mod_n}\ T_n\ m_n(S_n)\ \texttt{\{}B_n\texttt{\}} \\
\texttt{\}}
\end{array}
$$

Fig. 6 Translating the Extension

```
interface Set {
  boolean isElement(Object e);
  java.util.Enumeration elements();
  class Default {
    static boolean isElement(Set thiss, Object e) {
      for (java.util.Enumeration es=thiss.elements();es.hasMoreElements();) {
        if (e.equals(es.nextElement())) return true;
      }
      return false;
    }
  }
}
```

(a) Translation of `Set`

```
class VectorSet extends java.util.Vector implements Set {
  public boolean isElement(Object e) {
    return Set.Default.isElement(this, e);
  }
}
```

(b) Translation of `VectorSet`

with modifiers *mod*, super interface list *sup*, non–default members M, and default methods $m_1,\ldots,m_n$, each m_i with modifiers mod_i, result type T_i, signature S_i,and body B_i as:

$$
\begin{array}{l}
mod\ \texttt{interface}\ I\ \texttt{extends}\ sup\ \{\\
\quad M\\
\quad mod_1\ T_1\ m_1(S_1)\texttt{;}\\
\quad \ldots\\
\quad mod_n\ T_n\ m_n(S_n)\texttt{;}\\
\quad \texttt{public class \$default \{}\\
\quad\quad \texttt{static}\ mod_1\ T_1\ m_1(I\ \texttt{thiss,}S_1)\ \{B'_1\}\\
\quad\quad \ldots\\
\quad\quad \texttt{static}\ mod_n\ T_n\ m_n(I\ \texttt{thiss,}S_n)\ \{B'_n\}\\
\quad \}\\
\}
\end{array}
$$

Here, we assume that the new syntactic alternative *DefaultUse* is not used in I. However, this is just to allow a separate presentation of this step and has no deeper consequences. We will give the translation of this construct later. Fig. 6(a) contains an example of result of this translation for the interface `Set` from Fig. 5(a).

Translation of *DefaultUse* and *MethodInvocation*

A method declaration with a *DefaultUse* occurring in a class is translated by creating a new body for the method. The body mainly consists of a single invocation of the according `static` method in the according super interface. As additional first argument in this invocation, we provide the reference `this` to the current instance. If the method returns a result, then the method invocation is prefixed by `return`.

To define the translation, we assume that all occurrences of *DefaultUse* are in qualified form. We define the translation from extended Java to Java for a class

$$
\begin{array}{l}
\mathit{mod}\ \texttt{class}\ C\ \texttt{extends}\ \mathit{sup}_C\ \texttt{implements}\ \mathit{sup}_I\ \{\\
\quad M\\
\quad \mathit{mod}_1\ T_1\ m_1(T_{1,1}\ p_{1,1},\ldots,T_{1,k_n}\ p_{1,k_n})\ = I_1\texttt{.default;}\\
\quad \ldots\\
\quad \mathit{mod}_n\ T_n\ m_n(T_{n,1}\ p_{n,1},\ldots,T_{n,k_n}\ p_{n,k_n})\ = I_n\texttt{.default;}\\
\}
\end{array}
$$

with modifiers *mod*, super class list sup_C, implemented interface list sup_I, non–*DefaultUse* members M, and default using methods $m_1,\ldots,m_n$, each m_i with modifiers mod_i, result type T_i, signature $T_{n,1}\ p_{n,1},\ldots,T_{n,k_n}\ p_{n,k_n}$, and default location interface I_i:

$$
\begin{array}{l}
\mathit{mod}\ \texttt{class}\ C\ \texttt{extends}\ \mathit{sup}_C\ \texttt{implements}\ \mathit{sup}_I\ \{\\
\quad M\\
\quad \mathit{mod}_1\ T_1\ m_1(T_{1,1}\ p_{1,1},\ldots,T_{n,k_1}\ p_{n,k_1})\ \{\\
\quad\quad \texttt{return}^?\ I_1\texttt{.\$default.}m_1(\texttt{this},p_{1,1},\ldots,p_{1,k_1});\\
\quad \}\\
\quad \ldots\\
\quad \mathit{mod}_n\ T_n\ m_n(T_{n,1}\ p_{n,1},\ldots,T_{n,k_1}\ p_{n,k_1})\ \{\\
\quad\quad \texttt{return}^?\ I_n\texttt{.\$default.}m_n(\texttt{this},p_{n,1},\ldots,p_{n,k_n});\\
\quad \}\\
\}
\end{array}
$$

The `return` statement is omitted in the body of m_i iff $T_i = \texttt{void}$. Fig. 6(b) shows the translation for the class `VectorSet` from Fig. 5(b).

The new alternative of *MethodInvocation* which may occur inside method bodies of classes is translated in the same way: A method invocation of the from

$$I\texttt{.default.}m(a_1,\ldots,a_n)$$

is translated to

$$I\texttt{.\$default.}m(\texttt{this},a_1,\ldots,a_n)$$

In interfaces, both *DefaultUse* and the new alternative *MethodInvocation* may occur at methods with the modifier `default`. Here, the translation is similar, except that the additional parameter `thiss` is used instead of `this`. After this translation, the method bodies are removed from the interface as described above.

Example Execution

To demonstrate the interaction of the various parts, we consider an instance o of the class `VectorSet` from Fig. 5(b). A method invocation

$$o\texttt{.isElement}(a)$$

with any argument a enters the body created by translating *DefaultUse*. Hence, the following method invocation is executed:

$$\texttt{Set.Default.isElement}(o,a)$$

Here, the `for` loop is entered and the `Enumeration` is created by the following method invocation:

$$o\texttt{.elements()}$$

Hence, we have achieved exactly what we wanted.

5.2 Checking Context Sensitive Conditions

Context sensitive conditions in Java are checked by reading the class files resulting from compilation of the classes on which the conditions depend. Hence, the only real task for checking the newly introduced context sensitive conditions is to ensure that the presence or absence of default implementations can be determined by examining the class file resulting from compilation of the interface. In principle, we could insert additional attributes with the necessary information in the class files. The specification [14] allows this under the restriction that additional attributes do not change the execution of a class. Since these information would be needed only by the compiler and not by the JVM, we would not violate this restriction.

However, we do not need to follow this approach. By translating default implementations using an inner class named `$default` we are able to check the presence or absence of default implementations of a method by simply querying for methods inside the inner class `$default` of the interface. If either the class or the method inside the class is missing, then there is no default implementation.

5.3 Discussion

The approach for translating our extension to standard Java uses the static methods in the inner classes for the simulation of virtual methods. Passing the current instance as explicit first argument is exactly the same as the code generated by a standard Java does for virtual methods of classes. Apart from the additional method invocations needed to cross the border between default and explicit methods, our approach suffers no penalty with respect to a textual code duplications. Fig. 7 contains the byte codes generated for the method `isElement` by textual duplication and our approach. We can see that they differ only in the way the method `elements` in invoked: For the class `VectorSet`, the method is invoked as virtual method of `java.util.Vector` and for the interface `Set` with default implementation it is invoked as method of the interface.

We have implemented a first version of a source to source compiler. It is based on Barat [4], an open source front-end for Java. The implementation is available at http://www-i2.informatik.rwth-aachen.de/~mohnen/JDI/.

Fig. 7 Comparison of Generated Byte Codes for `isElement()`

```
Method boolean isElement(java.lang.Object)
   0 aload_0
   1 invokevirtual #7 <Method java.util.Enumeration elements()>
   4 astore_2
   5 goto 23
   8 aload_1
   9 aload_2
  10 invokeinterface #10 <InterfaceMethod java.lang.Object nextElement()>
  15 invokevirtual #8 <Method boolean equals(java.lang.Object)>
  18 ifeq 23
  21 iconst_1
  22 ireturn
  23 aload_2
  24 invokeinterface #9 <InterfaceMethod boolean hasMoreElements()>
  29 ifne 8
  32 iconst_0
  33 ireturn
```

(a) Byte Code for `VectorSet` in Fig. 3

```
Method boolean isElement(Set, java.lang.Object)
   0 aload_0
   1 invokeinterface #6 <InterfaceMethod java.util.Enumeration elements()>
   6 astore_2
   7 goto 25
  10 aload_1
  11 aload_2
  12 invokeinterface #9 <InterfaceMethod java.lang.Object nextElement()>
  17 invokevirtual #7 <Method boolean equals(java.lang.Object)>
  20 ifeq 25
  23 iconst_1
  24 ireturn
  25 aload_2
  26 invokeinterface #8 <InterfaceMethod boolean hasMoreElements()>
  31 ifne 10
  34 iconst_0
  35 ireturn
```

(b) Byte Code for `Set` in Fig. 5(a)

6 Conclusions

We have presented a new approach of using default implementations of methods in Java interfaces. In contrast to providing default implementations by using abstract classes, which is the current approach for providing default implementations, our approach has three major advantages: (1) A class implementing an interface is free to extend other classes. (2) No code duplication occurs in the presence of multiple inheritance of interfaces. (3) The default implementations are located in the interface.

The paper gives a complete design of the extension language and discusses the rationale for the decisions taken. We show that the extension is small and conservative.

In addition, we explain how the extension can be implemented efficiently. Our design is based on a translation to standard Java. Consequently, we were able to avoid extending the Java virtual machine. Programs from our proposed extended Java can be executed by any existing JVM implementation.

References

[1] P. America and F. van der Linden. A Parallel Object-Oriented Language with Inheritance and Subtyping. In OOPSLA/ECOOP'90 [18], pages 161–168.

[2] J. A. Bank, A. C. Myers, and B. Liskov. Parameterized types for Java. In POPL'97 [19], pages 132–145.

[3] J. Barnes, editor. *Ada 95 Rationale*. Number 1247 in Lecture Notes in Computer Science. Springer–Verlag, 1997.

[4] B. Bokowski and A. Spiegel. Barat - A Front-End for Java. Technical Report B-98-09, FU Berlin, FB Mathematik und Informatik, 1998.

[5] G. Bracha and W. Cook. Mixin-Based Inheritance. In OOPSLA/ECOOP'90 [18], pages 303–311.

[6] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 33, 10 of *ACM SIGPLAN Notices*, pages 183–200. ACM, 1998.

[7] R. A. Duff and S. Tucker Taft, editors. *Ada 95 Reference manual*. Number 1246 in Lecture Notes in Computer Science. Springer–Verlag, 1997.

[8] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proceedings of the 25th Symposium on Principles of Programming Languages (POPL)*. ACM, January 1998.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[10] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison Wesley, 2nd edition, 2000.

[11] P. Hudak, S. L. Peyton Jones, and P. Wadler. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices*, 27(4), 1992.

[12] S. E. Keene and D. Gerson. *Object-oriented programming in Common LISP: A programmer's guide to CLOS*. Addison-Wesley, 1989.

[13] Jo A. Lawless and Molly M. Miller. *Understanding CLOS: The Common Lisp Object System*. Digital Press, 1991.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, 2nd edition, 1999.

[15] B. Meyer. *Eiffel: The Language*. Object-Oriented Series. Pretive Hall, 1992.

[16] M. Mohnen. Interfaces with Skeletal Implementations in Java. In *Object-Oriented Technology – ECOOP 2000 Workshop Reader*, number 1964 in Lecture Notes in Computer Science, pages 295–296. Springer–Verlag, 2000.

[17] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In POPL'97 [19], pages 146–159.

[18] *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP)*, volume 25, 10 of *ACM SIGPLAN Notices*. ACM, 1990.

[19] *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL)*. ACM, January 1997.

[20] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 21, 11 of *ACM SIGPLAN Notices*, pages 38–45. ACM, 1986.

[21] B. Stroustrup. Classes: an abstract data type facility for the C language. *ACM SIGPLAN Notices*, 17(1):42–51, January 1982.

[22] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[23] K. K. Thorup. Genericity in Java with Virtual Types. In M. Aksit and S. Matsuoka, editor, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, pages 444–471. Springer–Verlag, 1997.

[24] C. T. Wu. Improving reusability with Actor 4.0's protocol mechanism. *Journal of Object-Oriented Programming*, 5(1):49–51, 1992.

[25] C. T. Wu. Protocol vs. Multiple Inheritance. *Journal of Object-Oriented Programming*, 5(3):72–75, 1992.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: biblio@informatik.rwth-aachen.de

95-11 *	M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases
95-12 *	G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
95-13 *	M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views
95-14 *	P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work
95-15 *	S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems
95-16 *	W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming
96-1 *	Jahresbericht 1995
96-2	M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees
96-3 *	W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
96-4	K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability
96-5	K. Pohl: Requirements Engineering: An Overview
96-6 *	M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
96-7	O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs
96-8 *	S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
96-9	M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming
96-10	R. Conradi / B. Westfechtel: Version Models for Software Configuration Management
96-11 *	C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement
96-12 *	R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
96-13 *	K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools

96-14 * R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

96-15 * H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases

96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet

96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation

96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations

96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems

96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto

96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design

97-01 Jahresbericht 1996

97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization

97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting

97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets

97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

98-01 * Jahresbericht 1997

98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes

98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen

98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 * Jahresbericht 1998

99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge / Marcin Jurdzinski: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 Andreas Becks / Stefan Sklorz / Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 Mareike Schoop: Cooperative Document Management

2000-06 Mareike Schoop / Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus

2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC Languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark / Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

2002-05 Horst Lichter / Thomas von der Maßen / Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines

2002-06 Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.