Interfaces with Skeletal Implementations in Java

Markus Mohnen

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

Abstract. Java's interfaces allow the definition of class properties independently of class inheritance. Since interfaces allow multiple inheritance, their potential for reusable code is higher than those of abstract classes. We present a technique for augmenting interfaces with skeletal implementations by using Java's inner classes. Using this technique, we propose an extension of Java for providing skeletal implementations in interfaces.

1 Introduction

Multiple inheritance of method implementations from more than one superclass in object oriented languages has been identified as a cause of problems for a long time (see for instance [17, 16]). One of the main problems is that it seems impossible to define a satisfactory general strategy for choosing or combining inherited method implementations.

Newer languages like Ada95 [6] or Java [8, 2] avoid these problems by not allowing multiple inheritance of method implementations at all. In contrast to Ada95, however, which does not support multiple inheritance as a language construct[1], Java allows multiple inheritance of *signatures*: Interfaces allow the definition of properties which can be implemented by classes[2]. 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 would be useful to equip interfaces with *skeletal implementations* of methods, which can be used by a class implementing the interface. Skeletal implementations are useful since they reduce the effort required to implement the 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 skeletal implementations of the other methods.

[1] In [4] it is argued that typical uses of multiple inheritance can be simulated in Ada95 using other language constructs.

[2] Interfaces are similar to what is called type classes in the functional language Haskell [9].

The usefulness of skeletal implementations has also been seen by the Java developers. In the Java 2 API there are several abstract classes which provide skeletal 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`.

The approach of adding abstract classes containing skeletal implementations has drawbacks for the implementation of the interface and for the implementation of the abstract classes:

– To use the skeletal 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 skeletal implementations for all interfaces in an interface hierarchy with multiple inheritance, it is unavoidable to duplicate code in the abstract classes.

We suggest a different approach for providing skeletal implementations of interfaces. Java's inner classes can be used to add default implementations to an interface without creating a separate abstract class. Using this approach, we avoid the problems mentioned in the last paragraph. Based on this technique, we propose an extension of Java, which allows to insert skeletal implementations directly in interfaces.

Note that our extension is not intended to be an simulation of full-featured multiple inheritance. In our view, interfaces are an orthogonal language element whose major purpose is to group kindred classes. Hence, the introduction of skeletal implementations in interfaces does not contradict the design decision of the Java developers to disallow multiple inheritance of classes.

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. Section 4 reviews the approach used in the Java 2 API for skeletal implementations. We present our approach for augmenting interfaces with skeletal implementations in Section 5. In Section 6 we propose a language extension of Java for skeletal implementations, using our method as implementation technique. Section 7 concludes.

2 Related Work

The approaches studied for multiple inheritance of method implementations from more than one superclass in object oriented languages are numerous. 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 [10, 11]. 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 [13] 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 [18] resulting from the linearisation in CLOS.

While multiple inheritance is still allowed in C++ [19, 20], 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.

Object-oriented languages which disallow multiple inheritance are Ada95 [6, 4], and Java [8]. Java still allows multiple inheritance of interfaces.

Interfaces are not novel to Java. Similar constructs are known for instance in the context of design patterns (*template methods* in [7]), and have been used in other languages: In the object-oriented language Actor [22], the corresponding construct is named *protocol*. The functional language Haskell [9] 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*.

3 Interfaces in Java

In this section, we introduce our running example and use it to demonstrate the usefulness of Java's 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. This package is used in (1) a program for computing Hasse diagrams of abstract domains and (2) in a program for data flow analysis of Java classes. The structures we consider here are the following:

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 antisymmetric, 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

```
le(e1,e2)==(lt(e1,e2)||e1.equals(e2))
```

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

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) the value of `le(e1,e2)` should be equal to the value of `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. Again, the following additional conditions, the absorption laws, cannot be expressed in Java:

`e1.equals(meet(join(e1,e2),e1))`

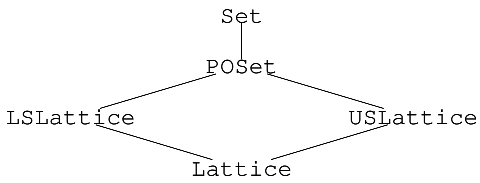
and `e1.equals(join(e1,meet(e1,e2)))`.

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.
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.

Fig. 2 Interface Hierarchy staring from `Set`:



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.

Fig. 3 An Implementation of `Set`

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

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.

4 Skeletal Implementations using Abstract Classes

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

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 skeletal implementations which ensure that these conditions are met.

In this section, we demonstrate how skeletal 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.

4.1 Basic Approach

Fig. 4 shows skeletal 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 the class `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 skeletal implementations for both `le` and `lt`, each in terms of the other. Consequently, using both skeletal implementations *at the same time* would result in an nonterminating recursion. To prevent that `AbstractPOSet` is used directly without overriding one of the methods, the modifier `abstract` was added.

Fig. 4 Skeletal Implementations using Abstract Classes

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

(a) Skeletal 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) Skeletal Implementation of `POSet`

4.2 Disadvantages

Using abstract classes to provide skeletal 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 `VectorSet` class 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 an abstract class `AbstractLattice` which implements the interface `Lattice` by extending both `AbstractLSLattice` and `AbstractUSLattice` is not possible. Instead, an abstract class `AbstractLattice` can be defined either by extending `AbstractLSLattice` or `AbstractUSLattice`; In either cases, the skeletal implementation of one the method `join` and `meet` must be duplicated from the missing class.
3. The skeletal implementations are located in a separate class, which might even be in a separate compilation unit.

5 Skeletal Implementations using Inner Classes

To avoid the disadvantages of skeletal implementations using top level abstract classes, we propose to use *inner classes*. Inner classes were introduced in the transition from Java 1.0 to Java 1.1 (see [2, Appendix D]).

5.1 Basic Approach

The basic idea of our proposal is to move the skeletal implementations from a top level class to an *inner class of the interface*. For example, Fig. 5 shows the updated definitions for the interfaces `Set` and `POSet`. Both interfaces now contain an abstract inner class `Skeleton`, which contain the skeletal implementation previously located in the classes `AbstractSet` and `AbstractPOSet` (see Fig. 4). Note that the class `POSet.Skeleton` extends the class `Set.Skeleton`.

Just like in the approach using top level abstract classes, a class implementing an interface can use the skeletal implementation by extending the skeleton. For instance, a class implementing `Set` could have the following form:

```
class SomeSet implements Set
              extends Set.Skeleton {
  ...
}
```

Fig. 5 Skeletal Implementations using Inner Classes

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

(a) Sets

```
interface POSet extends Set {
  boolean le(Object e1, Object e2);
  boolean lt(Object e1, Object e2);
  abstract class Skeleton
  extends Set.Skeleton 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) Partially Ordered Sets

In this sense, our approach is a conservative extension of the approach from Section 4.

To use the skeletal implementation in the presence of an unrelated superclass, a class can create an inner class extending the skeleton. The methods of the (outer) class can then use the methods of the inner class to perform their task. In addition, the inner class must contain stubs for those methods which are implemented by the (outer) class.

This is necessary, since the inner class extends the (abstract) skeleton. Through these stubs, the methods from the skeleton can access the correct method of the implementing class.

An example for this technique is shown in Fig. 6. It contains a new version of the `VectorSet` class from Section 3. This time, we can use the skeletal implementations provided by the new version of the interface `Set`. To access the methods of the inner class, an object of the inner class is created when initialising the class. Vice versa, the stubs inside the inner class can access the methods of the enclosing instance of the outer class through the `VectorSet.this` reference. In this case, we have one stub method, the method `elements`.

Fig. 6 An Implementation of `Set` using Skeletons in Inner Classes

```
import java.util.Enumeration;
class VectorSet extends java.util.Vector
                implements Set {
  protected class Implementation
  extends Set.Skeleton {
    public Enumeration elements() {
      return VectorSet.this.elements();
    }
  }
  protected Implementation imp
                = new Implementation();
  public boolean isElement(Object e) {
    return imp.isElement(e);
  }
}
```

This method imposes only a minimal runtime overhead: To gain access to the local class, a new object `imp` must be created. It is not possible to avoid this by using `static` methods, since (for obvious reasons) `static` methods have no access to non-`static` fields or methods.

5.2 Multiple Inheritance

Essentially the same approach can be used in the context of multiple inheritance of interfaces. Here we can distinguish two variations:

1. To define a class which implements multiple interfaces, we include one inner class for each interface implemented and insert stubs like before.
2. For an interface which extends multiple interfaces, we apply the same technique for the *inner skeleton class*. Since this class can extend one of the skeleton classes of the inherited interfaces, it contains $n-1$ implementation classes (if n is the number of inherited classes).

In Fig. 7, an example for this case is shown. This version of the interface `Lattice` contains an inner skeleton class, which extends the skeleton class of the interface `USLattice.Skeleton`. Hence, it inherits an implementation of the method `meet`. In addition, it uses the skeleton implementation of the method `join` in `LSLattice.Skeleton`.

With multiple inheritance of interfaces, a method can be inherited more than once. Consequently, more than one skeletal implementation might be available. Of course, we have to be careful to choose that skeletal implementation we need at this class or interface.

Fig. 7 Skeletons vs. Multiple Inheritance

```
import java.util.Enumeration;
interface Lattice extends LSLattice,
                          USLattice {
  abstract class Skeleton
  implements Lattice
  extends USLattice.Skeleton  {
    protected class Implementation
    implements Lattice
    extends LSLattice.Skeleton {
      public Object join(Object e1, Object e2)
        throws IllegalArgumentException {
        return Skeleton.this.join(e1,e2);
      }
      public Enumeration elements() {
        return Skeleton.this.elements();
      }
    }
    protected Implementation imp
              = new Implementation();
    public Object meet(Object e1, Object e2)
      throws IllegalArgumentException {
      return imp.meet(e1,e2);
    }
  }
}
```

5.3 Discussion

We have used inner classes in two ways: (1) To provide a skeletal implementation inside an interface as an abstract class, and (2) to define a complete implementation inside a class implementing the interface.

The second kind of use is crucial for our approach: It allows to use skeletal implementation without the need to inherit on top level from the abstract class containing the skeletal implementation.

In principle, it would be possible to locate these inner classes on top level also. However, there is really no point in this variation:

- On the conceptual level, this would create top level classes where each one would only be important (1) for the definition of default implementations of exactly one interface, or (2) for the implementation of exactly one other class only. Obviously, this would contradict the principle of encapsulation. Using Java's other encapsulation mechanism, packages, to encapsulate the classes would not help either. On the contrary, since a class can be part of at most one package, this approach would again reduce the programmer's freedom to locate the classes where it is necessary.
- On the technical level, this is exactly the same technique used by Java compilers to implement inner classes. Why take the effort if compiler technology has already solved the problem?

However, only our approach has the advantage of locating both interface definition and skeletal implementation in the same compilation unit. In addition, it can be used as a technique for implementing the language extension we present in the next section.

6 An Extension of Java

Several extensions of Java have been considered in literature, e.g. functional constructs in Pizza [14] (now superseded by GJ [5], which is no longer promoted as functional), virtual types [21], and parametric types [3].

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

1. We avoid the well-known 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 skeletal implementations for some of its methods. If a class implementing the interface has no references to the skeletal implementation, then its semantics is the same with and without our extension. If we would choose to use skeletal implementations of methods automatically, then we would effectively change the semantics of existing classes.

6.1 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 skeletal implementations. Following the syntactical structure of Java defined in [8][3], we extend the definition of *AbstractMethodModifier* [8, §9.4]:

AbstractMethodModifier: one of
`public abstract default`

To allow the use of skeletons, we change the definition of *MethodBody* [8, §8.4.5] in the following way:

MethodBody:
  Block
  SkeletonUse
  `;`

SkeletonUse:
  = *SkeletonSpec* `;`

SkeletonSpec:
  `default`
  TypeName `. default`

The syntactic alternative *SkeletonUse* 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. A skeleton declaration, i.e. one with the modifier `default`, must either be accomplished by a body or by a *SkeletonUse*. This changes the condition in [8, §9.4], which states that "Every method declaration in the body of an interface is implicitly abstract".
3. A *SkeletonUse* may occur both in interface definitions and class definitions. In addition, the condition in [8, §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 *SkeletonUse* in these cases.
4. The unqualified form of *SkeletonUse* 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 *SkeletonSpec*, 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 *SkeletonUse* occurs in a method of an interface, then it operates as a skeleton declaration of the method. This is useful for passing skeletons from a superinterface, since skeletons are not inherited automatically. Note that skeletons coming from superinterfaces which are more than one step higher in the interface hierarchy may not be used. Hereby, we avoid that large parts of the interface hierarchy are used. Furthermore, we achieve a behaviour similar to the one of `super`.

[3] We use this reference although inner classes are not included there but in [2, Appendix D]. However, our extension does not interfere with the syntactical extension imposed by the introduction of inner classes.

Fig. 8 Skeletal Implementations using Language Extension

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

(a) Sets

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

(b) An Implementation of `Set` using Skeletons

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

(c) Skeletons vs. Multiple Inheritance

For instance, a variation of the interface `Set` might look like the one in Fig. 8(a). Here, we define the skeleton implementations by using the extended syntax. The class `VectorSet` can then be written by using the unqualified skeleton use (see Fig. 8(b)). The qualified use of skeletons is useful in the context of multiple superinterfaces, as demonstrated in Fig. 8(c). Because the methods `meet` and `join` both are inherited from exactly on superinterface, it would not be necessary to qualify the skeleton 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 even if the implementing class has redefined it. This is similar to the use of `super` in ordinary methods. Therfore, we also change the definition of *MethodInvocation* [8, §15.11]:

MethodInvocation:
  MethodName `(` $ArgumentList_{opt}$ `)`
  Primary `.` *Identifier* `(` $ArgumentList_{opt}$ `)`
  `super` `.` *Identifier* `(` $ArgumentList_{opt}$ `)`
  SkeletonSpec `.` *Identifier* `(` $ArgumentList_{opt}$ `)`

The last syntactical alternative is new. In contrast to the use of default implementations as declarations of methods in *SkeletonUse*, 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.

6.2 Implementation

To simplify the presentation, we characterise an implementation of the new language constructs as a translation to (standard) Java. Of course, a direct generation of JVM code is preferable in a real system. So far, we have not implemented the extension, but we are planning to do so in a forthcoming project.

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 skeleton declarations, *SkeletonUse*, and the new alternative of *MethodInvocation*.

Item 2 can be solved essentially by using the techniques from Section 5; A full specification is beyond the scope of this paper.

For Item 1, we have to realise that context sensitive conditions in Java are checked by reading the class files resulting from compilation of the classes on which the conditions depend. Hence, we must ensure that the presence or absence of default implementations of methods can be determined by examining the resulting class file. Here, we are constrained by two conditions: (1) We cannot change the class file format as specified in [12], and (2) we must avoid that our extension becomes incompatible with existing class files. Therfore, we cannot use an inner class with a name which may be used in Java. Fortunately, the Java specification allows mechanical translators the use of a dollar sign in names. For instance, the recently added inner classes use this feature. Therefore, we can translate default implementations by using an inner class named `$Default`. Consequently, the presence or absence of skeletal implementations of a method can be checked by querying for the method inside the inner class `$Default`.

6.3 Discussion

The language extension we propose introduces skeletal implementations in Java interfaces. We have deliberately defined the extension in such a way that skeletal 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" [23], we see interfaces as an orthogonal language element. Its major purpose is to group kindred classes. Hence, the introduction of skeletal 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 brings.

7 Conclusions

We have presented a new approach of using skeletal implementations of methods in Java interfaces. Our approach uses inner classes, a recently added Java construct. In contrast to providing skeletal implementations by using abstract classes, which is the current approach for providing skeletal implementations, our approach has three major advantages:

– A class implementing an interface is free to extend other classes.
– No code duplication occurs in the presence of multiple inheritance of interfaces.
– The skeletal implementations are located in the interface.

The method we have presented can be used as a programming technique. However, to reduce the effort of using method skeletons, we have proposed an extension of Java. It is future work to implement this extension.

References

1. AMERICA, P., AND VAN DER LINDEN, F. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *Proceedings of the Conference on Object-Oriented Programming,*

Systems, Languages, and Applications (OOPSLA) and European Conference on Object-Oriented Programming (ECOOP) (1990), vol. 25, 10 of *ACM SIGPLAN Notices*, ACM, pp. 161–168.
2. ARNOLD, K., AND GOSLING, J. *The Java Programming Language*. Addison-Wesley, 1996.
3. BANK, J. A., MYERS, A. C., AND LISKOV, B. Parameterized types for Java. In POPL'97 [15], pp. 132–145.
4. BARNES, J., Ed. *Ada 95 Rationale*. No. 1247 in Lecture Notes in Computer Science. Springer–Verlag, 1997.
5. BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. 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)* (1998), vol. 33, 10 of *ACM SIGPLAN Notices*, ACM, pp. 183–200.
6. DUFF, R. A., AND TUCKER TAFT, S., Eds. *Ada 95 Reference manual*. No. 1246 in Lecture Notes in Computer Science. Springer–Verlag, 1997.
7. GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.
8. GOSLING, J., JOY, B., AND STEELE, G. *The Java Language Specification*. The Java Series. Addison Wesley, 1996.
9. HUDAK, P., PEYTON JONES, S. L., AND WADLER, P. Report on the Programming Language Haskell — A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices 27*, 4 (1992).
10. KEENE, S. E., AND GERSON, D. *Object-oriented programming in Common LISP: A programmer's guide to CLOS*. Addison-Wesley, 1989.
11. LAWLESS, J. A., AND MILLER, M. M. *Understanding CLOS: The Common Lisp Object System*. Digital Press, 1991.
12. LINDHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*. The Java Series. Addison Wesley, 1997.
13. MEYER, B. *Eiffel: The Language*. Object-Oriented Series. Pretive Hall, 1992.
14. ODERSKY, M., AND WADLER, P. Pizza into Java: Translating theory into practice. In POPL'97 [15], pp. 146–159.
15. *Proceedings of the 24th Symposium on Principles of Programming Languages (POPL)* (Jan. 1997), ACM.
16. SAKKINEN, M. A Critique of the Inheritance Principles of C++. *Computing Systems 5*, 1 (1992), 69–110.
17. SAKKINEN, M. The Darker Side of C++ Revisited. *Structured Programming 13*, 4 (1992), 155–177.
18. SNYDER, A. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (1986), vol. 21, 11 of *ACM SIGPLAN Notices*, ACM, pp. 38–45.
19. STROUSTRUP, B. Classes: an abstract data type facility for the C language. *ACM SIGPLAN Notices 17*, 1 (Jan. 1982), 42–51.
20. STROUSTRUP, B. *The C++ Programming Language*, third ed. Addison-Wesley, 1997.
21. THORUP, K. K. Genericity in Java with Virtual Types. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)* (1997), M. Aksit and S. Matsuoka, Eds., no. 1241 in Lecture Notes in Computer Science, Springer–Verlag, pp. 444–471.
22. WU, C. T. Improving reusability with Actor 4.0's protocol mechanism. *Journal of Object-Oriented Programming 5*, 1 (1992), 49–51.
23. WU, C. T. Protocol vs. Multiple Inheritance. *Journal of Object-Oriented Programming 5*, 3 (1992), 72–75.