I-Java: an extension of Java with incomplete objects and object composition*

Lorenzo Bettini, Viviana Bono, and Erica Turin

Dipartimento di Informatica, Università di Torino, Corso Svizzera, 185 – 10149 Torino, Italy

Abstract. Object composition is often advocated as a more flexible alternative to standard class inheritance since it takes place at run-time, thus permitting the behavior of objects to be specialized dynamically. In this paper we present I-Java, an extension of the Java language with a form of incomplete objects, i.e., objects with some missing methods which can be provided at run-time by composition with another (complete) object. Therefore, we can exploit object composition as a linguistic construct, instead of resorting to a manual implementation. This work builds on our theoretical model of incomplete objects, which was proved type-safe.

Keywords: Object-oriented programming, Language extension, Object composition, Class-based languages, Java.

1 Introduction

Design patterns were introduced as "programming recipes" to overcome some of the limitations of class-based object-oriented languages. Most of the design patterns in [17] rely on object composition as an alternative to class inheritance, since it is defined at run-time and it enables dynamic object code reuse by assembling existing components. Patterns exploit the programming style consisting in writing small software components (units of reuse), that can be composed at run-time in several ways to achieve software reuse. However, design patterns require manual programming, which is prone to errors.

Differently from class-based languages, object-based languages use object composition and *delegation* to reuse code (see, e.g., the languages in [33, 12], and the calculi in [15, 1]). Every object has a list of *parent* objects: when an object cannot answer a message it forwards it to its parents until there is an instance that can process the message. However, a drawback of delegation in the absence of any type discipline is that run-time type errors (*message-not-understood*) can arise when no delegates are able to process the forwarded message [34]. We refer to Kniesel [23] for an overview of problems when combining delegation with static type discipline.

We wanted to design a form of object composition with these goals in mind: maintaining the class-based type discipline, gaining some of the flexibility of the object-based paradigm, preventing the message-not-understood error statically, and preserving the nominal type system of Java. In [8, 6] we presented two possible solutions for

* This work has been partially supported by the MIUR project EOS DUE and by EU Project Software Engineering for Service-Oriented Overlay Computers (SENSORIA, contract IST-3-016004-IP-09).

Java-like languages: the first combines incomplete objects with *consultation*, and the second with *delegation* [26]. In both cases an object *A* has a reference to an object *B*. However, when *A* forwards to *B* the execution of a message *m*, two different bindings of the implicit parameter `this` can be adopted for the execution of the body of *m*: with *delegation*, `this` is bound to the sender (*A*), while with *consultation*, during the execution of the body the implicit parameter is always bound to the receiver *B*. Delegation is more powerful as it allows dynamic method redefinition. Both proposals are type-safe, therefore they allow to capture statically message-not-understood errors. In particular, we formalized two versions of *Incomplete Featherweight Java* (IFJ), as extensions of Featherweight Java [21, 30]. The programmer, besides standard classes, can define *incomplete* classes whose instances are *incomplete* objects that can be composed in an object-based fashion. Hence, in our calculi it is possible: (*i*) to instantiate standard classes, obtaining fully-fledged objects ready to be used; (*ii*) to instantiate incomplete classes, obtaining *incomplete objects* that can be composed (by *object composition*) with complete objects, thus yielding new complete objects at run-time; (*iii*) in turn, to use a complete object obtained by composition for composing with other incomplete objects. This shows that objects are not only instances of classes (possibly incomplete classes), but they are also prototypes that can be used, via the object composition, to create new objects at run-time, while ensuring statically that the composition is type-safe. We then can use incomplete and complete objects as our re-usable building blocks to assemble at run-time, on the fly, brand new objects.

The goal of this paper is to present an implementation of the incomplete objects with consultation [8] by means of I-Java, an extension of the Java language with incomplete objects and object composition. We implemented a preprocessor that, given a program that uses our language extension, produces standard Java code (the preprocessor is available at `http://i-java.sf.net`). The transformation implemented by the preprocessor (presented in Section 3) models the incomplete classes, the method composition, the method call via consultation, and integrates well the Java subtyping within the incomplete object model subtyping. Our prototype implementation has one limitation with respect to the field treatment, that will be discussed in the conclusions (see Section 5). However, being the emphasis of incomplete object extension on a more flexible method addition, fields treatment can be seen as orthogonal and postponed to future work.

We describe briefly our proposal, borrowing from our previous work on the calculus IFJ with consultation [8]. Classes, besides standard method definitions, can declare some methods as "incomplete" (these can be seen as *abstract* methods); these missing methods must be provided during object composition by complete objects. Thus, incomplete objects represent the run-time version of inheritance and dynamic binding mechanisms. This is a sort of dynamic inheritance since it implies both substitutivity (that is, a composed object can be used where a standard object is expected) and dynamic code reuse (since composition permits supplying at run-time the missing methods with those of other objects).

One of the key design choices is the nominal subtyping mechanism that is typical for mainstream languages like Java and C++. This feature makes the extension conservative with respect to the core Java, since it does not affect those parts of the programs

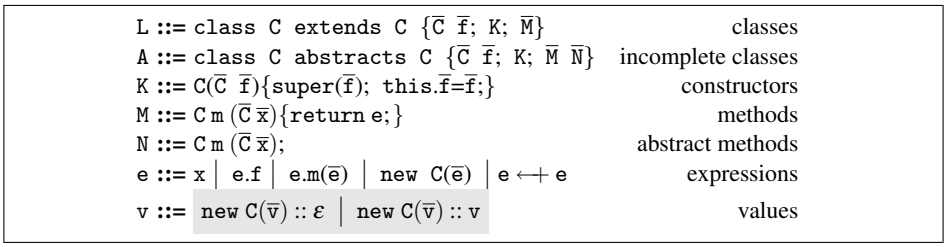

Fig. 1: IFJ syntax; run-time syntax appears shaded.

that do not use incomplete objects. Furthermore, incomplete classes can rely on standard class inheritance to reuse code of parent classes (although this kind of inheritance does not imply subtyping in our setting). Thus incomplete objects provide two forms of code reuse: *vertical* (i.e., the code reuse achieved via standard class inheritance) and *horizontal* (i.e., the one achieved via object composition). Finally, in order to enhance run-time flexibility in composing objects we implicitly use structural subtyping during composition: an incomplete object can be composed with any object providing all the requested methods (the signatures must match) independently of the classes of these objects.

Therefore, the language extension we propose is not a simple automatic implementation of the object composition that one might implement manually. In fact, any object providing the required methods can be used in object composition, no matter what its class is. In case of a manual implementation, instead, the object should be stored in a class field, thus forcing it to belong to a specific class hierarchy.

We observe that, although the preprocessor performs a static code generation (by using also static type information), the code generated is thought to perform object composition dynamically (without relying on static type information anymore); thus, the preprocessor generates statically code that has all the features to provide dynamic object composition.

2 Overview of the IFJ calculus

This section is devoted to presenting some details of the calculus with consultation (we refer the reader to [8] for the complete formalization). The calculus IFJ (*Incomplete Featherweight Java*) is an extension of FJ (*Featherweight Java*) [21, 30] with incomplete objects. We assume the reader is familiar with the main features of FJ (however, this is not strictly necessary for the general comprehension of the paper).

The abstract syntax of IFJ constructs is given in Figure 1 and it is just the same as FJ extended with incomplete classes, abstract methods and object composition (and some run-time expressions that are not written by the programmer, but are produced by the semantics, that we will discuss later). As in FJ, we will use the overline notation for possibly empty sequences (e.g., "$\overline{\texttt{e}}$" is a shorthand for a possibly empty sequence "$\texttt{e}_1, \ldots, \texttt{e}_n$") and for sequences of pairs in a similar way, e.g., $\overline{\texttt{C}}\ \overline{\texttt{f}}$ stands for $\texttt{C}_1\ \texttt{f}_1, \ldots, \texttt{C}_n\ \texttt{f}_n$. The empty sequence is denoted by $\bullet$. A class declaration `class C extends D` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\}$ is as in FJ. An incomplete class declaration `class C`

`abstracts D` $\{\overline{\texttt{C}}\ \overline{\texttt{f}};\ \texttt{K};\ \overline{\texttt{M}}\ \overline{\texttt{N}}\}$ inherits from a standard (or incomplete) class `D` and, apart from adding new fields and adding/overriding methods, it can declare some methods as "incomplete" (we will call these methods also "abstract" or "expected"). On the other hand, standard classes cannot inherit from incomplete classes (this is checked by the typing). The main idea of our language is that an incomplete class can be instantiated, leading to *incomplete objects*. Method invocation and field selection cannot be performed on incomplete objects.

An incomplete object expression $\texttt{e}_1$ can be composed at run-time with a complete object expression $\texttt{e}_2$; this operation, denoted by $\texttt{e}_1 \longleftarrow\!\!+\ \texttt{e}_2$, is called *object composition*. The key idea is that $\texttt{e}_1$ can be composed with a complete object $\texttt{e}_2$ that provides all the requested methods, independently from the class of $\texttt{e}_2$ (of course, the method signatures must match). Then, in $\texttt{e}_1 \longleftarrow\!\!+\ \texttt{e}_2$, $\texttt{e}_1$ must be an incomplete object and $\texttt{e}_2$ must be a complete object expression (these requirements are checked by the type system); indeed, $\texttt{e}_2$ can be, in turn, the result of another object composition. The object expression $\texttt{e}_1 \longleftarrow\!\!+\ \texttt{e}_2$ represents a brand new (complete) object that consists of the sub-object expressions $\texttt{e}_1$ and $\texttt{e}_2$; in particular, the objects of these sub-expressions are not modified during the composition.

Finally, values, denoted by `v` and `u`, are fully evaluated object creation terms. However, the object representation of IFJ is different from FJ in that fully evaluated objects can be also compositions of other objects. Thus, objects are represented as lists of terms `new` $\texttt{C}(\overline{\texttt{v}})$ (i.e., expressions that are passed to the constructor are values too). For instance, `new` $\texttt{C}(\overline{\texttt{v}})$ :: `new` $\texttt{D}(\overline{\texttt{u}})$ :: ε represents the composition of the incomplete object of class `C` with a standard complete object of class `D` (ε denotes the empty list). We represent a standard object with a list of only one element, `new` $\texttt{C}(\overline{\texttt{v}})$ :: ε.

In the type system we distinguish between the type of an incomplete object and the type of a composed object (i.e., an incomplete object that has been composed with a complete object). If `C` is the class name of an incomplete object, then $\langle\texttt{C}\rangle$ is the type of an incomplete object of class `C` that has been composed. The types of the shape $\langle\texttt{C}\rangle$ are only used by the type system for keeping track of objects that are created via object composition. Indeed, the programmer cannot write $\langle\texttt{C}\rangle$ explicitly; this is consistent with Java-like languages' philosophy where the class names are the only types that can be mentioned in the program (apart from basic types).

The subtype relation $<:$ on classes (types) is induced by the standard subclass relation extended in order to relate incomplete objects. First of all, we consider an incomplete class `class C abstracts D` $\{\ldots\}$; if `D` is a standard class, since `C` can make some methods of `D` incomplete, then it is obvious that an incomplete object of class `C` cannot be used in place of an object of class `D`. Thus, `abstracts` implements subclassing without subtyping. Instead, when the incomplete object is composed with a complete object (providing all the methods requested by `C`), then its type is $\langle\texttt{C}\rangle$, and it can be used in place of an object of class `D`. Since, as said above, we do not permit object completion on a complete object, then a complete object can never be used in place of an incomplete one.

Typing rules are adapted from those of FJ in order to handle incomplete objects and object composition. In particular, field selection and method selection are allowed only on objects of concrete types, where a *concrete* type is either a standard class `C` or

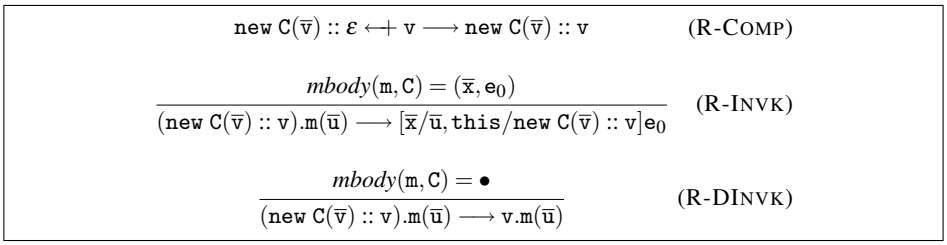

Fig. 2: Semantics of IFJ

$\langle \texttt{C} \rangle$. A key rule for dealing with object composition is introduced. It checks that the left expression in a composition is actually an incomplete object and that the right one is a complete object that provides all the methods needed by the incomplete object. (This also shows that the typing of $\leftarrow\!\!+$ is structural, which is a key feature of the system, since it enhances the flexibility of object composition.) The final type is the concrete type based on the original class of the incomplete object. Also typing rules for methods and classes of FJ were adapted to deal with incomplete classes. All the typing rules of IFJ that are related to incomplete objects, therefore not checked by the Java language compiler, have been implemented as checks in the I-Java preprocessor (Section 3.3).

The (deterministic call-by-value) operational semantics is shown partly in Figure 2. The main idea of the semantics of method invocation is to search for the method definition in the (class of the) head of the list using a *mbody* lookup function (this is defined, together with the function *fields*, in [8]). If this is found, rule (R-INVK), then the method body is executed; otherwise, rule (R-DINVK), the search continues on the next element of the list (of course, in a well-typed program, this search will succeed eventually). Method and field selections within a method body expect to deal with an object of the class where the method (or field) is defined (or a subclass). Thus, it is sensible to substitute `this` with the sublist whose head $\texttt{new C}(\overline{\texttt{v}})$ is such that $\mathit{mbody}(\texttt{m},\texttt{C})$ is defined. This is also consistent with the concept of *consultation*. Thus, the list implements the scope of `this` inside a method body: the scope is restricted to the visibility provided by the class where the method is defined. Note that this solves also possible ambiguities due to name clashes, even for methods hidden by subsumption [31].

The key point is that, when objects are composed, the resulting object consists of a list of sub-objects; in particular, these sub-objects are not modified. Thus, the states of the objects within an object composition never change and the object composition produces a brand new object (see (R-COMP) rule in Figure 2) where all the sub-objects are actually shared. In an imperative setting, this mechanism will assure that there will not be problems when an object is pointed to by references in different parts of the program. Finally, not modifying incomplete objects directly also makes them more re-usable especially in cases when object composition may not have to be permanent: the behavior of an object may need to evolve many times during the execution of a program and the missing methods provided during an object composition might need to be changed, e.g., because the state of the incomplete object has changed. Since the original incomplete object is not modified, then it can be re-used in many object compositions during the evolution of a program.

```
class Action {
  void run() { }
  void display() {}
}

class Button abstracts Action {
  void run(); // incomplete method
  void display() {
    // redefined to draw the button
  }
}

class SaveActionDelegate {
  void run() {
    // implementation
  }
}

class MenuItem abstracts Action {
  void run(); // incomplete method
  void display() {
    // redefined to show the item
  }
}

class KeyboardAccel abstracts Action {
  void run(); // incomplete method
  void display() {
    // redefined to hook key combination
  }
}

class Frame {
  void addToMenu(Action a) {...}
  void addToToolbar(Action a) {...}
  void setKeybAcc(Action a) {...}
}

SaveActionDelegate deleg =
  new SaveActionDelegate();
myFrame.addToMenu
  (new MenuItem("save") <- deleg);
myFrame.addToToolbar
  (new Button("save") <- deleg);
myFrame.setKeybAcc
  (new KeyboardAccel("Ctrl+S") <- deleg);
```

Listing 1: The implementation of action and action delegates with incomplete objects and object completion.

2.1 A case study

In this section, we show a programming example written in a slightly Java-sugared syntax (we consider all methods as public and we will denote object completion operation with `<-`). We consider the development of an application that uses widgets such as graphical buttons, menus, and keyboard shortcuts. These widgets are usually associated to an event listener (e.g., a callback function), that is invoked when the user sends an event to that specific widget (e.g., one clicks the button with the mouse or chooses a menu item).

The design pattern *Command* [17] is useful for implementing these scenarios, since it permits parametrization of widgets over the event handlers, and the same event handler can be reused for similar widgets (e.g., the handler for the event "save file" can be associated with a button, a menu item, or a keyboard shortcut). Thus, they rely on this object for the actual implementation of the action semantics, while the action widget itself abstracts from it. This decouples the action visual representation from the action controller implementation.

We can implement directly this scenario with incomplete objects, as shown in Listing 1: the class `Action` and `SaveActionDelegate` are standard Java classes (note that they are not related). The former is a generic implementation of an action, and the latter implements the code for saving a file. We then have three incomplete classes implementing a button, a menu item, and a keyboard accelerator; note that these classes inherit from `Action`, make the method `run` incomplete, and override the method `display`.

We also assume a class `Frame` representing an application frame where we can set keyboard accelerators, menu items, and toolbar buttons. An instance of class `Button` is an incomplete object (it requires the method `run`) and, as such, we cannot pass it to `addToToolbar`, since `Button` $\not<:$ `Action` (subclassing without subtyping). However, once we composed such instance (through object completion operation, `<-`) with an instance of `SaveActionDelegate`, then we have a composed complete object (of type $\langle$`Button`$\rangle$) that can be passed to `addToToolbar` (since $\langle$`Button`$\rangle$ $<:$ `Action`). Note that we compose `Button` with an instance of `SaveActionDelegate` which provides the requested method `run`, although `SaveActionDelegate` is not related to `Action`.

Moreover, we use the same instance of `SaveActionDelegate` for the other incomplete objects. Thus, modifying the internal state of `deleg` will assure that all the actions are updated, too. For instance, if the `SaveActionDelegate` had logging features, we could enable them and disable them during the execution of our program, and all the actions resulting from the object compositions will use logging consistently.

We now discuss some possible manual implementations in Java of this scenario, showing that our proposal is not simply syntactic sugar. With standard Java features, one could write the `Button` class with a field, say `deleg`, on which we call the method `run`. This approach requires `deleg` to be declared with a class or interface that provides such method, say `Runnable`. However, this solution would not be as flexible as our incomplete objects, since one can then assign to `deleg` only objects belonging to the `Runnable` hierarchy. This might not be optimal in case of reuse of existing code (that cannot be modified); in particular, this scenario is not ideal for unanticipated code reuse. The solution can be refined to deal with these problems by introducing some *Adapters* [17], but this require additional programming. On the other hand, if one wanted to keep the flexibility, one should declare `deleg` of type `Object`, and then call the method `run` by using Java Reflection APIs, (e.g., `getMethod`); however, this solution is not type safe, since exceptions can be thrown at run-time due to missing methods.

2.2 Incomplete objects and design patterns

As we have already mentioned in the introduction, design patterns propose interesting solutions to recurrent programming problems in specific contexts, and they often rely to object composition and method forwarding as a more flexible alternative to class-based inheritance. However, in spite of their usefulness, they still require manual programming: the programmer cannot use directly linguistic constructs and must prepare the structure of classes and their interaction relations from scratch, and follow the "informal" scheme of the patterns to be implemented. Furthermore, as also stated in [17], some design patterns can be seen as an ad-hoc solutions to fulfill the lacks of a programming language. For instance, the Visitor's mechanism of visit/accept methods would not be required in a language with dynamic overloading or multi-methods: only the visit methods would be enough, by relying on their dynamic overloading semantics ("CLOS has multi-methods, for example, which lessen the need for a pattern such as Visitor" [17]).

With incomplete objects, we tried to shorten the distance between the language features and the design patterns, so that their implementation can be smoother [10]. We

do not aim at replacing design patterns completely: design patterns will keep on providing "a high-level language of discourse for programmers to describe their systems and to discuss common problems and solutions" [11]. Language features should make the application of a pattern easier and possibly more efficient [11]. For instance, implementations of patterns were proposed using *aspect-oriented programming* [19] and *collaboration-based design* [25].

Summarizing, we are not claiming that incomplete objects and their compositionality features allow the programmers to implement some development scenarios which would not be implementable simply by using design patterns. We claim instead that, by using the proposed linguistic features, the programmer can express the structure and the interaction relations in a specific context directly, with less additional manual programming; furthermore, as linguistic features with their own additional typing system and operational semantics (which was proved sound), incomplete objects represent a flexible and safe programming mechanism. After all, one could do even without inheritance and dynamic binding, but one would need to implement these mechanisms from scratch in many development scenarios, and such manual implementations, besides being tedious and time consuming, would not enjoy the same safety properties of those provided by the linguistic constructs themselves ("Our patterns assume Smalltalk/C++-level language features, and that choice determines what can and cannot be implemented easily. If we assumed procedural languages, we might have included design patterns called Inheritance, Encapsulation, and Polymorphism." [17]). Finally, incomplete objects and object composition can be used to implement other design patters that rely on object composition and forwarding [17]: *Adapter*, *Bridge*, *Decorator*, *Proxy*, *Strategy* and *Chain of Responsibilities*.

Some works, such as [16, 18], provide programming tools to help the programmer implement and use design patterns: it is possible to describe patterns, instantiate them (possibly starting from templates), merge them into the existing programs, check their correct implementation and usage, and also recognize them in the source code. Thus, these tools act at a meta-level with respect to the programming language, while we wanted to have object composition at the linguistic level. However, we believe that the value of these tools would still be profitable in order to have a higher level view also of incomplete objects and object composition, moreover they could still deal with patterns even when implemented via incomplete objects.

3 The Java extension I-Java

In this section we describe the Java extension I-Java and its implementation, in particular we describe the classes generated by our preprocessor and the checks performed by it. We tried to minimize the changes to the original Java syntax in designing our linguistic extension of Java. Actually, our solution does not need any extension of the Java grammar at all, it relies instead on *Java annotations*: we will use two specific annotations to specify incomplete classes and complete classes. The preprocessor will then transform the annotated classes and generate the appropriate additional code.

In Java, the linguistic construct that is closer to our incomplete classes is the *abstract class*. Both incomplete classes and abstract classes allow to define the imple-

```
@Incomplete
public abstract class IncC extends SuperC {
   private int i = 0;

   public void myCompleteMeth() {
      ++i;
      myIncompleteMeth();
   }

   // an incomplete method
   public abstract void myIncompleteMeth();
}
```

```
@Complete
public class SuperC {
      private f = false;

      public void myMeth() {
            f = true;
            myIncompleteMeth();
      }

      public void myIncompleteMeth() {
            ...
      }
}
```

Listing 2: An incomplete class and a complete class in I-Java (get/set methods are not shown)

mentation of some methods only, and to delay the implementation of the remaining methods (specifying only their signatures). However, abstract classes cannot be instantiated, while incomplete classes are the generators of incomplete objects. The basic idea is that the programmer will use abstract classes to specify incomplete classes, and the I-Java preprocessor will generate additional (non-abstract) Java classes that will be used to instantiate objects.

Instead of adding a new keyword to the language, an incomplete class can be defined by writing a standard abstract class with the annotation `@Incomplete`. An example of incomplete class declaration is shown in Listing 2 (left). As for declaring incomplete methods, we simply reuse the standard Java syntax to declare abstract methods. Thus, basically, the code for defining incomplete classes is standard Java code with a specific annotation; this code will be transformed by the preprocessor, generating further classes. Note that instead of using a keyword `abstracts`, we simply use the Java keyword `extends`; however, for incomplete classes, this is to be intended as subclassing without subtyping. Thus, it must not be possible to assign an object of class `IncC` to a variable of type `SuperC`. This constraint is checked by the preprocessor (Section 3.3). As for object composition, we do not add a new operator to the syntax, instead, we simply use a method generated by the compiler: `compose`. The choice of not introducing new keywords in the language also allowed to re-use existing Java editors and IDEs to write I-Java code.

Another annotation, `@Complete`, is used to denote those (complete) classes that are exploited to build incomplete class-based hierarchies (see Listing 2, right). This is needed in order to enforce by the preprocessor some constraints on the fields of such classes (see Section 3.3): we require that `@Incomplete` and `@Complete` classes have only private fields, and have public get/set methods for the fields[1]. Therefore, the superclass `SuperC` (and its ancestors) must be annotated with `@Complete`. Note that: (*i*) `@Complete` classes, with respect to the Java compiler and with respect to our object

[1] Following the standard convention that if the field is called `myField`, the corresponding accessor methods are `getMyField` and `setMyField`.

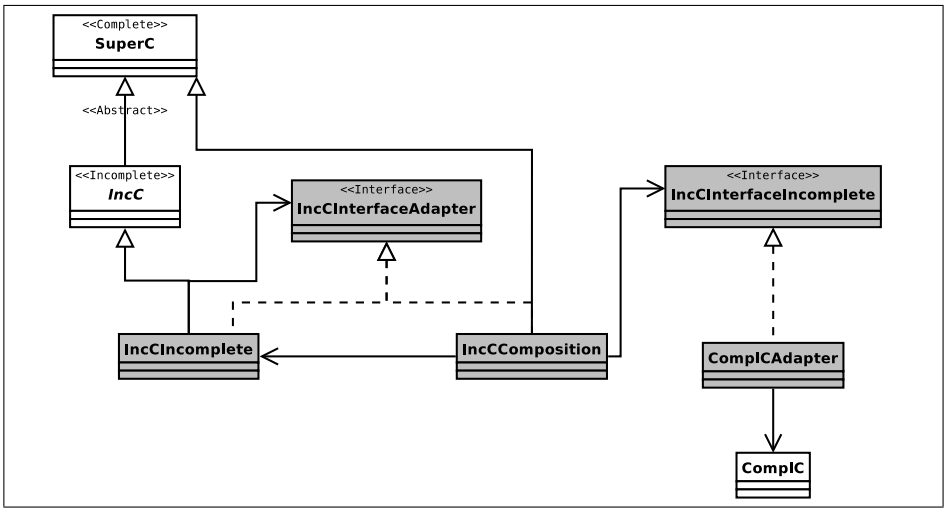

Fig. 3: The original and generated classes (in gray).

composition, are treated as standard Java classes, i.e., they can be extended, instantiated, and their (complete) objects can be used within any object composition with incomplete objects; (*ii*) objects of both `@Complete` classes and normal Java classes can be exploited in the object composition, making possible a form of reuse.

3.1 Code generated by the preprocessor

As said above, the preprocessor will generate some additional classes, starting from an incomplete class. In the following we describe in detail the role of the generated classes. The name of these classes will be formed using the original incomplete class name and adding a suffix. In the following, we assume that the incomplete class is named `IncC` (we will use the example of Listing 2). Note that the generated classes will not be used by the programmer directly (actually the programmer is not aware of these additional generated classes). In Figure 3, we show the UML diagram containing the original classes (in white) and the generated classes and interfaces (in gray).

Since we use standard Java abstract classes to define incomplete classes, we would not be able to create objects of incomplete classes since we would need to be able to instantiate an abstract class. Thus, the preprocessor will generate the class `IncCIncomplete`: this is the "instantiable" version of the incomplete class `IncC`. The generated class extends the original incomplete class and provides a definition for the corresponding abstract methods which simply throws a run-time exception. Note that these methods will never be invoked in the code, since the preprocessor checks that no methods are invoked on an incomplete object and we assume that the programmer never uses directly the generated code; however, these methods can be discoverable and invoked in regular Java code by reflection and we preferred to raise an error in case of invocation. The generated `IncCIncomplete` will also contain the method `compose` that will be used to perform an object composition; in particular `x.compose(y)` returns an

```
public class IncCIncomplete extends IncC implements IncCInterfaceAdapter {
    private IncCInterfaceAdapter myThis = this;

    public void setMyThis(IncCInterfaceAdapter myThis) { this.myThis = myThis; }

    // copied from IncC with some adjustments
    public void myCompleteMeth() {
        setI(getI()+1);
        myThis.myIncompleteMeth();
    }

    // copied from SuperC with some adjustments
    public void myMeth() {
        setF(true);
        myThis.myIncompleteMeth();
    }

    // will never be called
    public void myIncompleteMeth() {
     throw new RuntimeException("call of incomplete method myIncompleteMeth");
    }

    public IncCComposition compose(Object ob) {
        IncCInterfaceIncomplete aii = (IncCInterfaceIncomplete)ob;
        return new IncCComposition<IncCInterfaceIncomplete>(aii, this);
    }
}
```

Listing 3: The generated class `IncCIncomplete`

object which corresponds to the composed object `x` $\leftarrow\!\!+$ `y`. The generated class `IncCIncomplete` corresponding to the incomplete class in Listing 2 is shown in Listing 3. Before getting into details of `IncCIncomplete` and of the generated class `IncCComposition`, we first need to explain other generated classes and interfaces.

One of the main features of the calculus of [8] is that the methods that are incomplete can be provided, within an object composition, by any (complete) object that has those methods, independently from the complete object class. However, in Java, an object used in a composition must implement a specific interface with all the methods requested by the incomplete object. For this reason, the preprocessor generates the interface `IncCInterfaceIncomplete`; this interface contains all the incomplete methods of the class `IncC`, i.e., all the methods that must be provided during object composition. The basic idea is that this interface represents any complete object in an object composition that will provide all the missing methods. As an alternative implementation, we could have used the class `Object` and then use reflection to invoke the requested method or use casts; however, this would have increased the overhead in the generated code.

```
public class IncCComposition implements IncCInterfaceAdapter
        extends SuperC {
  private IncCIncomplete head;
  private IncCInterfaceIncomplete complete;
  // constructor omitted...

  public void myCompleteMeth() {
    head.setMyThis(this);
    head.myCompleteMeth();
    head.setMyThis(head);
  }

  public void myIncompleteMeth() {
    complete.myIncompleteMeth();
  }
}
```

Listing 4: `IncCComposition` generated class.

Suppose we use an object of class `ComplC`, which provides all the methods required by the incomplete object of class `IncC`. Our translated code would require that `ComplC` also implements `IncCInterfaceIncomplete`, but since this is a generated interface this would require to modify the code of `ComplC`, which is quite unacceptable. We solve this issue by generating, for each class `ComplC` whose objects are used in an object composition with an incomplete object of class `IncC`, an additional class `ComplCAdapter`, which implements `IncCInterfaceIncomplete` by forwarding to a wrapped instance of class `ComplC`. Of course, if `ComplC` is used in several object composition, the generated `ComplCAdapter` will implement all the corresponding generated `InterfaceIncompletes`.

The generated class `IncCComposition`, Listing 4, is crucial since it implements the actual composition between an incomplete object and a complete object. In the calculus of incomplete objects that we are considering, [8], we can only compose an incomplete object with a complete one; thus the incomplete object can only be of type `IncCIncomplete`, but the complete one can be, in turn, the result of another object composition. For this reason, the class `IncCComposition` declares the complete sub-object as `IncCInterfaceIncomplete`, and the implementation of an incomplete method simply consists in forwarding the method invocation to the complete object. The implemented interface `IncCInterfaceAdapter` will be clear in the following. Note that `IncCComposition` extends the original class "abstracted" by the incomplete class. This respects the fact that in the calculus, if `C abstracts D`, then a composed object of type $\langle$`C`$\rangle$ is such that $\langle$`C`$\rangle <:$ `D`.

Indeed, the hardest part is the implementation of a complete method. If we simply forwarded to the `IncCIncomplete` instance, then, upon the invocation of an incomplete method from within `IncCIncomplete` we would simply call the empty implementation (i.e., the one throwing a run-time exception), which is wrong: we need to "go back" to the original message receiver, which is the object of type `IncCComposition`. To solve

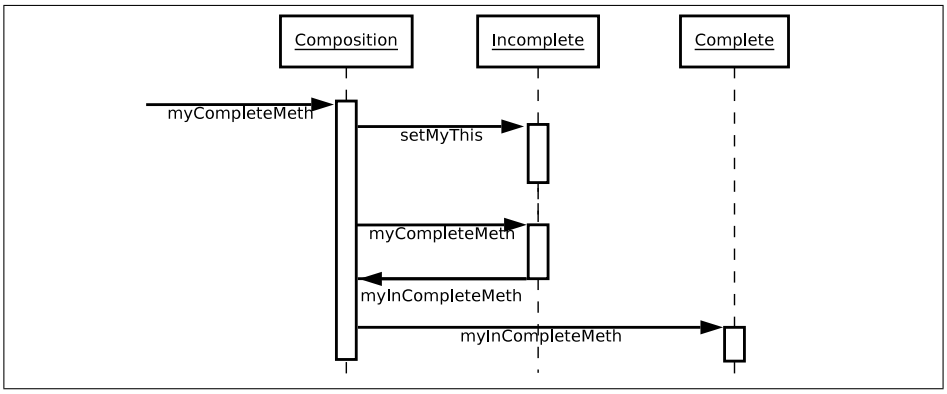

Fig. 4: The sequence diagram of a complete method invocation.

this issue, we need to simulate the binding of `this`; we do it by introducing "another `this`" in the class `IncCIncomplete` (Listing 3), that we call `myThis`. Note that the complete methods of the original incomplete classes are copied into `IncCIncomplete`, and all the occurrences of method invocations on `this` are replaced with invocations on `myThis`. The implementation of a complete method in `IncCComposition` takes care of setting `myThis`, through method `setMyThis`, to the `IncCComposition` instance itself. In Figure 4 we show the sequence diagram of an invocation of `myCompleteMeth` (which also calls `myIncompleteMeth`).

This will guarantee that upon the invocation of an incomplete method on a composed object, the right implementation will be selected (using `myThis`). This holds also for inherited methods (which are neither redefined, nor made incomplete, such as `myMeth`): they are copied and adjusted so that method invocations are performed on `myThis` and field accesses are transformed into invocations of the get/set corresponding methods.

Finally, the preprocessor generates the interface `IncCInterfaceAdapter`, containing all the methods in the original incomplete class (both complete and incomplete). Note that this interface is implemented both by `IncCIncomplete` and `IncCComposition`, since it represents the type for `myThis`.

3.2 Code modified by the preprocessor

The preprocessor needs also to modify the client code that uses incomplete objects. However, due to the way the classes and interfaces generated by the processor are related through subtyping relations, we only need to preprocess incomplete object instantiation expressions. Furthermore, complete object instances used in object compositions need to be wrapped using the generated adapter class. Thus, if we consider the following code:

```
IncC inc = new IncC();
ComplC compl = new ComplC();
SuperC c = inc.compose(compl);
c.myIncompleteMeth();
```

the preprocessor will only replace the incomplete object instantiation and the composition statement as follows:

```
IncC inc = new IncCIncomplete();
ComplC compl = new ComplC();
SuperC c = inc.compose(new ComplCAdapter(compl));
c.myIncompleteMeth();
```

3.3 Checks performed by the preprocessor

The preprocessor of I-Java is not a simple program translator: it also performs some type checks. These checks guarantee that the corresponding generated Java program will be not only well-typed, but also type-safe, in the sense that no method invocation on an incomplete object will take place at run-time (of course, this relies on the fact that the programmer does not modify the generated code). All these checks correspond to the type checking rules of the incomplete object calculus of [8].

First of all, it checks that no method invocation (but `compose`) is performed on an incomplete object. Remember that the Java type checker would allow such invocations since an incomplete object in I-Java corresponds to an instance of a preprocessor-generated class; that is why our preprocessor must check this. Then, it checks that the argument of a `compose` invocation is correct, i.e., that it is a complete object providing all the required methods. Furthermore, it checks that an incomplete class is not used as the base class of a standard Java class. Finally, it checks that the subtyping rules of the calculus are respected: the only one non-standard in Java is that an incomplete class is not a subtype of a complete class (or of another different incomplete class).

Another check is performed on all `@Incomplete` and `@Complete` classes: their fields must be private and all the related accessory methods must be provided. If not, the preprocessor rises an error. This is necessary because of two (orthogonal) issues: (*i*) all fields, including the inherited ones, must be accessible from all methods of the `IncCIncomplete` class, since the preprocessor copies there all complete methods that the incomplete class inherits directly and indirectly from its `SuperC`; (*ii*) at the same time, we want to forbid that the wrong copy of the fields be modified, that is, the one of the composition object, that inherits from `SuperC` (we recall that in our formal model [8] fields do not belong to the composition, but to the single objects, part of the composition, as the former is not an object in the model and the field access is orthogonal to the method access). It would be possible to provide a different implementation that gives total freedom to the programmer in choosing the preferred visibility modifiers for all classes by building a preprocessor that: (*i*) adds by default the appropriate accessory methods to all classes belonging to an incomplete hierarchy, regardless the visibility modifiers; (*ii*) and modifies the client code by substituting to all field accesses on instances of classes of an incomplete hierarchy the appropriate accessory methods' calls. However, we believe that our restriction on having private fields and public accessory methods for all classes of an incomplete hierarchy is a reasonable one, as the direct access to fields is often seen as a not totally convenient design choice. Note also that the classes of the complete objects used to be composed with the incomplete ones do not have any constraint to be subjected to.

4 Related Work

Incomplete objects can be seen as *wrappers* for the objects used in object composition. However, they differ from decorator-based solutions such as the language extension presented in [9]: incomplete objects provide a more general-purpose language construct and the wrappers of [9] could be actually implemented through incomplete objects. Another form of wrapping of methods is the one offered by the *delegates* of C#. Delegates are objects pointing to one method or to a set of methods, that will be executed when invoked appropriately on the delegate. Therefore, it is possible to treat methods as anonymous functions, implementing a form of reuse. Delegates can be then seen as complementary to incomplete objects, as the latter implements a different form of reuse, allowing to customize a prototype (i.e., an incomplete object) in more than one way via object composition.

A further construct of C# that deals with some form of incompleteness is the one of *partial classes*, that makes it possible to subdivide a class definition among two or more files. This mechanism is useful to implement a form of reuse oriented to the design of large scale projects, as a class distributed over distinct files allows more programmers to work on it, moreover it helps the addition of new code to a class without modifying the source. Moreover, the mechanism of partial classes is a static one, and it does not take place at run-time like our object-composition. Once again, the form of reuse offered by partial classes is complementary to the one implemented by incomplete objects.

Traits [13] are composable units containing only methods, and they were proposed as an add-on to traditional class-based inheritance, allowing a higher degree of code reuse. Traits (in particular the ones with fields [5]) and incomplete objects share an important feature, composition, which permits composing sets of methods "at the right level", for instance not too high in a hierarchy for traits, and "when needed" for incomplete objects. There is a main difference between traits and incomplete objects: traits are more oriented towards software reuse, in the sense they help avoiding code duplication and other bad design choices, while incomplete objects are a device to model certain forms of unanticipated evolution. This seems to make the two approaches orthogonal and complementary, thus, it might be interesting to investigate the coexistence and cooperation of traits and incomplete objects in the same language. An issue to pursue as a further research may be the use of incomplete objects as an exploratory tool to design traits: experiments made at run-time without modifying a class hierarchy might give indications on where to put a method in a new version of the hierarchy. In [7] we presented a tool for identifying in a Java class hierarchy the methods that could be good candidates to be refactored in traits (this is the adaption of the Smalltalk analysis tool of [27] to a Java setting). It would be interesting to investigate the application of this approach also for detecting possible candidates for incomplete classes.

There are some relations between aspects [22] and our incomplete objects. Both are used to combine features taken from different sources. In the aspect case, the main idea is to factorize into aspects some cross-cutting functions (such as logging services or concurrency primitives) that are needed globally by a library, instead of duplicating and scattering them into the business code. In our case, we consider objects as building blocks that can be used to combine features on the fly, in order to obtain and experiment with multi-function objects whenever it is desired. In a sense, the role of incomplete

objects is orthogonal to the one of aspects, because the former play a local role, while the latter a more global one.

In [2], a general model (Method Driven Model) for languages supporting object composition is proposed: this is based on the design of classes in an aspect-oriented style. The authors do not formalize their model within a calculus, but it is possible to see that the main feature of a language based on this model would be to compose dynamically the overall behavior of an object from the multiple "aspects" that abstract the variant behavior, as discussed in [3]. The main difference between their proposal and ours is that for them the run-time behavior is codified in aspects, whereas we internalize it in Java by exploiting incomplete classes and object composition.

The language *gbeta* [14] supports a mechanism called "object metamorphosis", which is a mechanism to specialize dynamically an existing object, by applying to it a class as a constraint in such a way the object becomes an instance of that class. The main difference between the gbeta specializing objects and our incomplete objects is that the former maintain the object identity, while the latter are used to create dynamically new objects which are not instances of any classes present in the program. Both proposals are proved type-safe, but a more direct comparison is not straightforward, as the type system of gbeta exploits concepts such as virtual classes which are not present in a Java-like setting like ours. The language gbeta also supports dynamic class composition [28] (classes are first class values and existing classes may be composed dynamically to yield new classes), while in our language we act on object composition. It is important to remark that one of our main design decision was that our extension must integrate seamlessly in a Java-like language as a conservative extension.

Roles [24] are a conceptual abstraction that can be used in object-oriented systems to implement specific entities within a domain. *ObjectTeams/Java* (OT/J) [20] is a programming language that provide roles in a Java context, following the criteria defined in [32]. Although the inherent compositional nature of roles looks similar to incomplete objects, the two approaches are rather different both for features and for intention. First of all, incomplete objects are a low level linguistic feature: they represent a dynamic and object-based implementation of inheritance. Thus, once objects are composed, they cannot be de-composed (although we might consider studying such an operation and how this affects the static type system). On the contrary, roles can be attached to base objects and detached; in particular, upon removal, a role is also destroyed. This is another important difference with respect to our object composition: objects in our language keep their own identity and life-cycle (and they can be used in many object composition), while roles can be attached to one base object only and they "live" only when they are part of such a base object. Role definitions also specify the class of their base objects, and this couples them to these classes (this coupling can be reduced, but not removed, by using *unbound* roles, similar to abstract classes, and subroles), while in our approach the type of objects in composition is not known in advance. Moreover, OT/J implements a "flattening" semantics concerning fields, while in our composed objects all the subobjects have their own fields. Summarizing, roles are a higher level feature which is somehow complementary to incomplete objects, and these two linguistic constructs aim at solving different programming contexts. These differences seem inherent of the role approach and can be found in other implementations (see, e.g., [4]).

5 Conclusions

In this paper we presented I-Java, which enhances the Java language by adding to it some features related to dynamic object evolution in a type-safe way. In particular, some behavior that was not foreseen when the class hierarchy was implemented might be supplied dynamically by making use of already existing objects, thus generating an unanticipated reuse of code and a sharing of components. The code generated by the preprocessor does not contain much overhead with respect to manual implementations (e.g., of patterns): first of all, if get/set methods are final they can be inlined by the Java compiler itself; moreover, method forwarding for incomplete methods would be present also in standard design pattern implementations. The only additional overhead is the one due to `myThis`; note, however, that this is due to the enhanced flexibility of incomplete objects that can be composed with objects independently from their types, and to the "navigation" capabilities when invoking methods on a composed object (again if these capabilities were to be implemented manually, the overhead would be the same).

The extension of I-Java with a delegation mechanism (as studied in [6]) is under development; the structure of the generated classes, in particular the usage of `myThis`, should make the extension with delegation reasonably easy, with respect to the generated code. However, the preprocessor will have to perform some more involved checks and the code for method invocation in the generated classes will have to take care of avoiding possible name clashes (we refer to [6] for further details about the formal treatment of delegation in our calculus of incomplete objects).

Acknowledgments. We warmly thank Betti Venneri for her work on the calculus IFJ. The anonymous referees provided many important suggestions to improve the quality and the clarity of the paper. We are also grateful to Alexandre Bergel for his suggestions on the paper.

References

1. C. Anderson, F. Barbanera, M. Dezani-Ciancaglini, and S. Drossopoulou. Can Addresses be Types? (a case study: Objects with Delegation). In *WOOD'03*, volume 82(8) of *ENTCS*, pages 1–22. Elsevier, 2003.
2. C. Babu and D. Janakiram. Method Driven Model: A Unified Model for an Object Composition Language. *ACM SIGPLAN Notices*, 39(8):61–71, 2004.
3. C. Babu, W. Jaques, and D. Janakiram. DynOCoLa: Enabling Dynamic Composition of Object Behaviour. In *Proc. of RAM-SE*, 2005.
4. M. Baldoni, G. Boella, and L. W. N. van der Torre. Interaction between Objects in powerJava. *Journal of Object Technology*, 6(2), 2007.
5. A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits. In *Advances in Smalltalk — ISC 2006*, volume 4406 of *LNCS*, pages 66–90. Springer, 2007.
6. L. Bettini and V. Bono. Type Safe Dynamic Object Delegation in Class-based Languages. In *Proc. of PPPJ*, pages 171–180. ACM Press, 2008.
7. L. Bettini, V. Bono, and M. Naddeo. A trait based re-engineering technique for Java hierarchies. In *Proc. of PPPJ*, pages 149–158. ACM Press, 2008.
8. L. Bettini, V. Bono, and B. Venneri. Object incompleteness and dynamic composition in Java-like languages. In Paige and Meyer [29], pages 198–217.

9. L. Bettini, S. Capecchi, and E. Giachino. Featherweight Wrap Java: wrapping objects and methods. *Journal of Object Technology*, 7(2):5–29, 2008.
10. J. Bishop. Language features meet design patterns: raising the abstraction bar. In *ROA '08*, pages 1–7. ACM, 2008.
11. J. Bishop and R. N. Horspool. On the Efficiency of Design Patterns Implemented in C# 3.0. In Paige and Meyer [29], pages 356–371.
12. C. Chambers. Object-Oriented Multi-Methods in Cecil. In *Proc. of ECOOP*, volume 615 of *LNCS*, pages 33–56. Springer, 1992.
13. S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.
14. E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Dep. of Computer Science, Univ. of Århus, 1999.
15. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *LNCS*, pages 42–61. Springer, 1995.
16. G. Florijn, M. Meijers, and P. van Winsen. Tool Support for Object-Oriented Patterns. In *Proc. of ECOOP*, pages 472–495, 1997.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
18. Y. Guéhéneuc. Three Musketeers to the Rescue – Meta-Modeling, Logic Programming, and Explanation-based Constraint Programming for Pattern Description and Detection. In *Workshop on Declarative Meta-Programming at ASE 2002*, 2002.
19. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. of OOPSLA*, volume 37 of *ACM SIGPLAN Notices*, pages 161–173. ACM Press, 2002.
20. S. Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.
21. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
22. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *LNCS*, pages 220–242. 1997.
23. G. Kniesel. Type-Safe Delegation for Run-Time Component Adaptation. In *Proc. of ECOOP 99*, volume 1628 of *LNCS*, pages 351–366. Springer, 1999.
24. B. B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Sytems*, 2(3):143–160, 1996.
25. M. Kuhlemann, S. Apel, M. Rosenmüller, and R. E. Lopez-Herrejon. A Multiparadigm Study of Crosscutting Modularity in Design Patterns. In Paige and Meyer [29], pages 121–140.
26. H. Lieberman. Using prototypical objects to implement shared behavior in object oriented systems. *ACM SIGPLAN Notices*, 21(11):214–214, 1986.
27. A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proc. of ASE*, pages 66–75. IEEE, 2005.
28. A. B. Nielsen and E. Ernst. Optimizing Dynamic Class Composition in a Statically Typed Language. In Paige and Meyer [29], pages 161–177.
29. R. Paige and B. Meyer, editors. *Proc. of TOOLS*, volume 11 of *LNBIP*. Springer, 2008.
30. B. C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA, 2002.
31. J. Riecke and C. Stone. Privacy via Subsumption. *Inf. and Computation*, (172):2–28, 2002.
32. F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowledge Engineering*, 35(1):83–106, 2000.
33. D. Ungar and R. B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, 1987.
34. J. Viega, B. Tutt, and R. Behrends. Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages. Technical Report CS-98-03, 1998.