Non-Conforming Inheritance: the SmartEiffel Experiment of a High-Level Mechanism

Frederic Merizen
frederic.merizen@loria.fr

Dominique Colnet
dominique.colnet@loria.fr

LORIA
Campus Scientifique
BP 239
54506 Vandoeuvre-lès-Nancy Cedex

Philippe Ribet
philippe.ribet@loria.fr

Cyril Adrian
cyril.adrian@laposte.net

ABSTRACT

Non-conforming inheritance (NC-inheritance) is a mechanism recently introduced in the new Eiffel definition. The NC-inheritance mechanism is similar to traditional inheritance but it disallows polymorphism. This simple mechanism appears to be useful in many situations because it allows the designer to capture more design decisions in the source code itself. Furthermore this mechanism helps compilers to statically remove more dynamic dispatch code. NC-inheritance incurs no type-system soundness problems even when arguments are redefined covariantly or when the exportation is restricted in the subclass. Out of the Eiffel world, the NC-inheritance mechanism can be useful to add a no-penalty and no-risk multiple-inheritance-like facility. For instance the Java language, initially designed for simple-inheritance, could be a possible candidate for an NC-inheritance extension.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Classes and objects, Inheritance, Polymorphism*

General Terms

Design, Languages

Keywords

Non-conforming inheritance

1. INTRODUCTION

As members of the ECMA Eiffel normalization group [1] we have implemented and experimented non-conforming inheritance (NC-inheritance) into our SmartEiffel compiler. Surprisingly, this simple mechanism had a substantial impact on our programming practice. NC-inheritance offers useful design options at no run-time cost, without requiring a whole-system analysis and is not inherently Eiffel-specific. This paper is not meant as reference material for NC-inheritance in Eiffel, and we will not go into Eiffel-specific peculiarities of syntax or semantics. Instead, this paper is our attempt to document the NC-inheritance mechanism in a mostly language-neutral way, in the hope that our experiment will be useful for other language designers.

The idea behind NC-inheritance is simple: complement the traditional inheritance mechanism with a new one, which keeps the code reuse aspects but discards the subtyping relationship. As a matter of fact, the traditional inheritance mechanism of object-oriented languages is perfectly suitable to represent the "*is a*" relationship between two types. For example, when the class APPLE inherits from the class FRUIT, the traditional inheritance mechanism is perfect just because an APPLE *is a* kind of FRUIT. When the "*is a*" relationship does exist between two types, all methods and all attributes can be inherited without any major problems. Furthermore, in such a situation, polymorphism can be used safely because the subclass is a more specific version of its superclass. All methods which are exported by the ancestor are likely to be exported by the descendant because of the very nature of the "*is a*" relationship.

Still, in some situations, a class has some methods and/or attributes in common with another class although they are not related by the "*is a*" relationship in the sense of the Liskov [11] substitution principle. Having one class plainly inherit from the other is not a good idea: in addition to the implementation, the child gets conformance and thus (unwanted) polymorphism. The developer then often has to choose between duplicating code or using awkward composition relationships. This is where NC-inheritance can help.

Section 2 presents a typical example where the NC-inheritance mechanism is really better than traditional inheritance. Section 3 describes the NC-inheritance mechanism and its interaction with other langugage mechanisms. Section 4

shows some common uses of the NC-inheritance mechanism in the SmartEiffel libraries. Section 5 explains the impact of NC-inheritance for design pattern translation. Section 6 presents the related work and section 7 concludes.

2. A MOTIVATING EXAMPLE

As a first motivating example of the NC-inheritance mechanism, let us take again the traditional stack class example. The implementation of the Java class `Stack` relies on traditional inheritance: the class `java.util.Stack` simply extends the class `java.util.Vector`, where the class `Vector` is a resizable ordered collection of elements. Obviously, this made `Stack` very simple to implement. One may argue that a `Stack` is not really *a kind of* `Vector`, but the purpose of this article is not to to discuss this. Still, it is a matter of fact that the implementation decision made here allows us to use all methods of the class `Vector` on `Stack` objects as well. The less we can say is that a `Stack` obtained by simply inherit `Vector` is not a pure `Stack` because one can directly peek or put elements whatever their position in the `Stack`.

The traditional Eiffel language allows the class `Stack` to hide methods and attributes inherited from the class `Vector`, but this protection is quite weak with traditional inheritance. It can be bypassed using polymorphic assignments of `Stack` objects to `Vector` variables: first assign your `Stack` object into a local `Vector` variable and then just apply any method of `Vector` you want on your `Stack` object. If `Stack` inherits from `Vector` with the traditional inheritance mechanism, such polymorphic assignments are allowed. Thus, changing the exportation status of some operations from `Vector` into `Stack` does not really prevent the usage of those operations.

Thanks to NC-inheritance, this is no longer possible because it is exactly the purpose of this new mechanism: the NC-inheritance mechanism is similar to traditional inheritance but it disallows polymorphism. In the Eiffel language, this new mechanism allows the designer of the class `Stack` to inherit the class `Vector` in a non-conforming way. Once this decision is made, it is no longer possible to assign an expression of type `Stack` into a `Vector` variable. Actually, if `Stack` does not explicitly inherit from at least one class in a conforming way, then `Stack` is at the top of a local subgraph of the conformance graph. If the `Stack` is actually such a local top, then `Stack`s cannot be assigned into variables of any other type, not even into variables of the universal type ANY (equivalent to Java's `Object`). The compile-time protection cannot be bypassed simply because all assignments and all argument passing sites are statically checked. Because the exportation status of inherited methods and attributes can be altered in the subclass, the list of available methods can be actually limited to the *pure* point of view one can have on a `Stack`.

Since we have implemented the NC-inheritance mechanism in our SmartEiffel compiler we have added such a class, named STACK, with a limited set of available methods [6]. We also used the very same NC-inheritance to implement the class QUEUE, which NC-inherits from our class RING_ARRAY.

3. THE NC-INHERITANCE MECHANISM

3.1 Design goals and graphical convention

Originally, we introduced NC-inheritance in the Eiffel language to allow for *implementation inheritance* that does not incur the possibility of unwanted polymorphic calls. We reach the goal of implementation inheritance without polymorphic calls by forbidding polymorphic assignments. If an object can never be polymorphically assigned to a variable of an ancestor type, then that ancestor will not get substituted by the heir for any call. This is the behaviour appropriate to implementation purpose.

Of course, as a useful mechanism to represent the *is a* relationship,traditional inheritance is still part of the programming tools we need. With NC-inheritance to contrast, we now identify traditional inheritance as *Conforming inheritance*. The vocabulary may be new, but the mechanism is kept unchanged. As a consequence, we now have two kinds of inheritance, C-inheritance and NC-inheritance. Figure 1 shows a graphical representation for the C-inheritance and the NC-inheritance relationships.

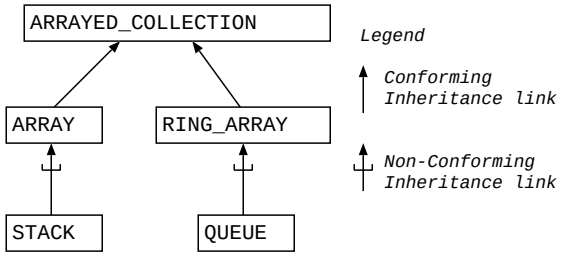

Figure 1: Two kinds of inheritance, traditional C-inheritance or the new NC-inheritance mechanism.

3.2 Definitions and terminology

Once there are two inheritance mechanisms—C-inheritance and NC-inheritance—in the same language, we have to be careful when using notions such as *parent* or *ancestor* and their opposites *child* and *heir*. Using the class hierarchy represented in figure 2, let's refine those words.

The *parent* relationship does not take the conforming or non-conforming nature of inheritance into account. For instance, the classes B and C both inherit A, and A is the parent of both. The *ancestor* relationship is the reflexive transitive closure of the *parent* relationship as usual [13], and it does not take conformance into account. As an example, still on figure 2, A is an ancestor of D.

The *conforming parent* relationship is the *parent* relationship restricted to conforming inheritance. The class B has a conforming inheritance link to A, so A is a *conforming parent* of B. The *conforming ancestor* relationship is the reflexive transitive closure of the *conforming parent* relationship. For instance, A is a conforming ancestor of both C and B. While the NC-inheritance link from C to A does not provide conformance to A, the path through B does; NC-inheritance does not disallow C-inheritance or prevent it from providing conformance to any class.

We say that a class is a *non-conforming ancestor* if it is an

ancestor but not a conforming ancestor. In figure 2, the class D has two non-conforming ancestors, A and B, and it is the only class that has non-conforming ancestors. Finally, as it is sometimes more convenient to use verbs than nouns, we say that a class *inherits* from its ancestors, that it *NC-inherits* from its non-conforming ancestors, and that it *conforms to* its conforming ancestors.

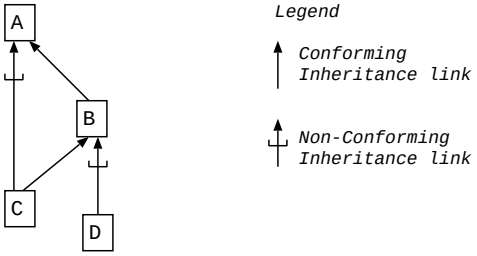

Figure 2: The class A is a conforming ancestor of B and C. The class A is a non-conforming ancestor of D.

The syntax of the NC-inheritance mechanism is not the most important point to be discussed here; the normalization work [1] is not completed and the point is still discussed. In traditional Eiffel, the list of parents is arranged in an *inherit clause* introduced by the `inherit` keyword. At time being, the SmartEiffel implementation keeps this clause for traditional (conforming) inheritance and adds a new clause, named *insert clause* listing the non-conforming parents. Syntactically, an *insert clause* only differs from an *inherit clause* by the fact that it is introduced by the new `insert` keyword.

3.3 Semantics of NC-inheritance

As said previously, the only distinctive feature of NC-inheritance is to allow a class to *inherit* from another class without *conforming* to it. In other words, NC-inheritance does not give you the permission to make polymorphic assignments.

The NC-inheritance mechanism is relevant only for statically checked object-oriented languages. Indeed, in a statically typed language, an assignment from a source expression of type S to a destination variable of type D is valid if and only if S *conforms to* D. Obviously, the same restriction that applies to assignments also applies to method parameter passing: an actual argument of type S matches a formal argument of type D if and only if S conforms to D. As we already know, methods and attributes are inherited in the same way through NC-inheritance as through C-inheritance. Multiple inheritance is equally possible for conforming parents or for non-conforming parents. Also, regardless of conformance, a class can rename or redefine methods and attributes it has inherited, and it can change their exportation status. Again, the only difference is that the child does not gain conformance to its parent through an NC-inheritance link. The semantics of NC-inheritance are thus very simple.

The small class hierarchy portrayed on figure 1 features both conforming and non-conforming inheritance relationships. While ARRAYs and RING_ARRAYs can be assigned to ARRAYED_COLLECTIONs, STACKs cannot be assigned to ARRAYs nor to ARRAYED_COLLECTIONs. Polymorphic assignments are not possible for QUEUEs either. Of course, one can still write a class and make it conform to STACK. Let's assume that we write the class STACK_WITH_BELLS_AND_WHISTLES and that it has a conforming inheritance link to STACK, but no conforming inheritance link to a class that conforms to ARRAY. Then objects of type STACK_WITH_BELLS_AND_WHISTLES can be polymorphically assigned to STACKs, but not to ARRAYs.

3.4 Impact on static checks

Let's now see how NC-inheritance interacts with the rest of the language. The bad news is that all the NC-inherited code must be checked in the subclass even if it was correct in the ancestor. Actually, this is due to the static type of the `this` expression, the receiver, also called `Current` in Eiffel or `self` in some other object-oriented languages.

As a typical example, let's have a look at a simple piece of code that is perfectly valid in its original class. In the class A, assuming that the variable `my_a` is of type A, the following code is obviously correct[1]:

```
my_a = this;
```

This performs polymorphic assignments of the `this` object to a variable of an ancestor type—in this case, the ancestor happens to be the class A itself. This code becomes invalid when NC-inherited by a class B since `this`, whose type is B, does not conform to A and cannot be assigned to `my_a`. We could have solved this problem by allowing NC-heirs to perform polymorphic assignments of themselves to variables or method parameters of an NC-ancestor type like C++ does, but in doing this we would have allowed polymorphic calls to escape (see 6.2).

We only know a few other situations than NC-inheritance where code can become invalid through inheritance. This phenomenon forces the compiler not only to check the validity of the code in the parent class, but also to check it again, as NC-inherited code, in the child class. While a whole-system analysis is not needed, this does require access to the NC-ancestors' source code. We suspect that sufficient information could be gathered from JVM or CLI bytecode but have not investigated the issue in full depth. We can think of no practical way to implement NC-inheritance if the body of an ancestor's method is only available in a pure binary object-code form.

3.5 Type system soundness

Eiffel has had covariance and export restrictions from the beginning. We find these two features useful design tools, but they are also well-known for making a type system unsound [4, 3, 5]. But NC-inheritance does not create a subtyping relationship and it is not possible to polymorphically assign an object to a variable of an NC-ancestor type or to pass it as an argument to a function that expects an NC-ancestor.

[1] In Eiffel, we would write `my_a := Current`.

Widening the availability of an inherited method or attribute in a child is harmless for the subtyping relationship: to the outside world, the child simply can do *more* than the parent could. Conversely, restricting the availability of a feature in a child *breaks* the *is a* relationship. The `Stack` from section 2 is not really a `Vector`, because a `Stack` does not have the random access facilities a `Vector` has. As we have seen, when an export restriction is applied in a C-inheritance link, it can be bypassed. Just assign the child with restrictive exportation to a variable of a less restrictive parent type, and access the method or attribute through that variable. This trick is the only way to break exportation restrictions, and a number of solutions have been proposed to address this issue, some of them involing a whole-system anlysis. If the exportation restriction is applied in an NC-inheritance link instead of a C-inheritance link, the trick does not work since assignments to NC-ancestors are not allowed. Exportation restrictions are safe if performed through NC-inheritance.

Similarly, calls to methods with a covariant redefinition only become unsafe when polymorphism enters the picture [12]. Covariantly redefining a method's argument restricts the set of correct types for this argument.

This restriction can be circumvented with exactly the same trick used for exportation restrictions: assign the child to a less restrictive parent variable. Through this variable, you can now call methods with arguments that are invalid in the covariant redefinition, but valid in the ancestor. But, as we already know, the polymorphic assignment that is central to this trick is not possible if the covariant redefinition was made in a NC-child. We conclude that covariant redefinitions are safe if performed through NC-inheritance.

3.6 The universal ancestor

Before the introduction of the NC-inheritance mechanism in Eiffel, a class with no explicit parent was, by default, a descendant of the universal class ANY, which is the equivalent of the Java class `Object`. An *is a* relationship between two classes is a strong design element that should result from a conscious design decision. For this reason, *is a* relationships should be explicitly stated in the source code an never result from a default value in the langugage. For that reason, classes that have no explicit parent are now given ANY as a default *non-conforming* parent by the SmartEiffel compiler. Of course, this default parent can be overriden by the developer. Listing ANY as an explicit conforming parent is all they have to do to get the equivalent of the old default behaviour. The new default is consistent with normal programming practice, where ANY is mainly used as a pool of utility methods. There are hardly any variables of such a general type as ANY: such variables are bound to be downcast sometime later. One could argue that programs that have a lot of ANY variables should be written in a dynamically typed language rather than a statically typed one.

To sum it up, NC-inheritance gives developers all features of C-inheritance, plus it allows them to get (unwanted) polymorphic assignments rejected. That way, our goal of preventing polymorphic calls is clearly attained.

3.7 NC-inheritance and encapsulation

The Eiffel exportation mechanism allows the developer to be quite selective about which classes are granted access to a given method or attribute. A method or attribute can be equipped with a *possibly empty* list of classes that are allowed to access them as clients. This mechanism works in conjunction with the inheritance mechanism: if a class is listed as a valid client for a given method or variable, then all its heirs are also valid clients.

One gets the Eiffel equivalent of Java's `public` exportation level by granting access to the universal class ANY. This way, because ANY is the ancestor of all classes, the permission is passed down from parent to child, making the method or the attribute available everywhere. Conversely, a method or variable with an empty list of valid clients can never be used in qualified calls. It is only available to the class defining it and its heirs using unqualified calls. This is equivalent to the `protected` level of C++. In the Eiffel langugage, this is the most restrictive encapsulation level available. There is no equivalent to the `private` level of C++.

The exportation mechanism of Eiffel we just recalled here is not new and has not changed since a long time. The introduction of NC-inheritance has not changed the exportation rules. The permissions are passed down from parent to child through conforming and non-conforming inheritance links alike. We initially envisionned a more restrictive rule: permissions would only by passed down through conforming inheritance links. This rule actually proved very impractical. The classes in figure 3 illustrate the problem. The class

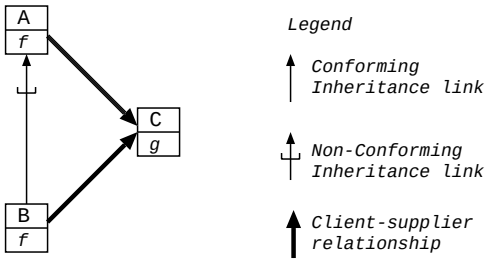

Figure 3: The method `f` of the class A uses the method `g` of the class C. When `f` is NC-inherited by B, it should still be allowed to use `g`.

B NC-inherits from the class A, and the method `f` of A uses the method `g` of C. Suppose we use the restrictive rule. To keep the code of `f` valid in the context of B, you would have to add B to the export-list of `g` in C. In other words, we would enforce a strong coupling between a class and the suppliers of its NC-parents. That's hardly a way to favor code reuse, and not even feasible if C is a third-party class that cannot be modified.

3.8 NC-inheritance for Java?

The Java language [2] has interface inheritance and class inheritance. Interface inheritance is about conformance but not about code reuse, while class inheritance is about both conformance *and* code reuse. While we understand that the term *implementation inheritance* is used to emphasize the difference between the two kinds of inheritance, we also feel it is a bit of a misnomer since class inheritance is not

only about implementation. Actually, misusing the class inheritance mechanism for mere code reuse purposes leads to the "a `Stack` *is (not quite) a* `Vector`" issues depicted in section 2. Holub goes as far as considering the use of class inheritance as a strong indication of bad design [9].

We claim that NC-inheritance would bring type-safe reuse inheritance, and even multiple reuse inheritance to the Java language. No run-time penalty is incurred, and no change to the virtual machine is required. As we have seen, NC-inheritance allows code reuse without the cost of spurious conformance. Of course, disallowing the use of NC-heirs where NC-ancestors are expected makes for cleaner and more precise design, but it also helps performance. Since NC-inheritance does not create any conformance link, methods or attributes that are acquired through an NC-inheritance link can be treated by the compiler just as if they were plainly written in the child. Most importantly, this means that no special treatment is required to handle the virtual function table, even in the case of multiple NC-inheritance. Actually, a great part of the power of NC-inheritance lies in the possibility to mix and match implementations. Just as interface inheritance, NC-inheritance *needs* multiple inheritance to reach its full potential.

4. NC-INHERITANCE AT WORK

4.1 Sharing constants with NC-inheritance

The first usage we made of NC-inheritance was probably to share constants amongst various classes. Indeed, it is straightforward to group a topic-related bunch of constants definitions in a dedicated class. Other classes that have to access those constants just need to NC-inherit the dedicated class. All constant names are thus directly part of the new class definition exactly as other local definitions are. A typical example is the class COLOR_LIST in our Vision graphical library. As one may guess, this class is a list of color definitions. When a class needs to use this bunch of color names, this class just has to NC-inherit from the class COLOR_LIST. In such a situation, it is better to use NC-inheritance to emphasize the fact that we just want to `insert` the list of definitions and there is no "*is a*" relationship between this class and COLOR_LIST.

In Java, there seems to be a common idiom of sharing constant definitions by using `static member` definitions. This new possibility offered by NC-inheritance avoids the need to prefix each constant usage with the name of the class where the static definition is located. As an example, a class that needs to manipulate ASCII constants a lot, just has to NC-inherit the ASCII_CONSTANT class to make all ascii constant names part of the new class.

Actually, Eiffel does not offer the possibility to define `static` attributes or functions. To implement a Java-like solution in Eiffel, one would actually need to create an instance of the class that holds the constant definitions just to access these constants. The NC-inheritance solution does not require the creation of an extra object and is quite attractive in Eiffel. This way, it is also possible to share the access not only for simple constants, but also for singleton objects to be shared. Shared singleton access is achieved in Eiffel by using functions which are executed only once, hence the next section which is about code sharing *via* NC-inheritance.

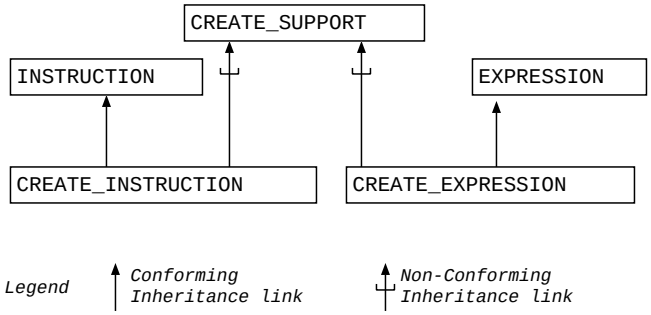

Figure 4: A routine storage example.

4.2 Sharing methods and attributes with NC-inheritance

Also very natural is the need to share some code between classes which are not related by a common conforming ancestor. As an example, in our SmartEiffel compiler, as shown on figure 4, the class CREATE_EXPRESSION is a specialization of the abstract class EXPRESSION (a CREATE_EXPRESSION *is a* EXPRESSION). Similarly, the class CREATE_INSTRUCTION is a specialization of the abstract class INSTRUCTION (similarly a CREATE_INSTRUCTION *is a* INSTRUCTION). Because the implementation of the class CREATE_EXPRESSION is similar to the implementation of the class CREATE_INSTRUCTION, common instance variables as well as many common methods are defined once in CREATE_SUPPORT. The usage of NC-inheritance (figure 4) emphasizes the fact that CREATE_SUPPORT is just a useful place to share common code which is necessary to implements both CREATE_EXPRESSION and CREATE_INSTRUCTION and that there is no possible substitution of CREATE_EXPRESSION with CREATE_INSTRUCTION (or conversely).

Actually, the classes EXPRESSION and INSTRUCTION already have a common ancestor named CODE. Thus, one may argue that there is no need for a CREATE_SUPPORT class since common code of the CREATE_EXPRESSION and CREATE_INSTRUCTION classes could be stored into the CODE class itself. This is not possible because EXPRESSION and INSTRUCTION have a lot of heirs that don't use the implementation of CREATE_SUPPORT. If the contents of CREATE_SUPPORT were fused into the class CODE, its methods would pollute the name space of other classes that don't need them. Even worse, its attributes would needlessly bloat the representation of all those classes.

Still for the example of figure 4, it is interesting to notice that methods which are defined in CREATE_SUPPORT and that are inherited using an NC-inheritance link can be triggered due to dynamic dispatch. Indeed a shared method of CREATE_SUPPORT can be for example the implementation of an abstract method of the class CREATE_EXPRESSION. The class EXPRESSION has a lot of heirs and dynamic dispatch does work as expected. The fact that some method is inherited via some NC-inheritance link does not prevent to use dispatch on this method.

4.3 Hiding some universal properties

As we have seen previously for the STACK example of figure 1, it is sometimes better to hide some inherited operations. By using NC-inheritance, we can safely hide operations or attributes that were visible in the non-conforming parent class.

This remark even applies to the universal class ANY. Among other operations, ANY contains a comparison method `is_equal`, the equivalent of the Java `equals` method and the traditional `clone` operation to duplicate objects. Generally a class is supposed to provide those methods, but there are some situations in which a class wants to hide those universal properties.

As an example, when one wants to promote as much as possible aliasing [15], one may decide not to export the possibly slow `equals` comparison method. As stated in section 3.6 a class can now have ANY as an NC-ancestor and thus safely hide the inherited `equals` method. By doing so it is no longer possible to call the possibly slow `equals` method from its clients. This is a way to promote the usage of the basic fast comparison operator (the built-in `==` in Java). Similarly, the universal `clone` inherited method can be hidden in some situations in order to localize or to forbid object duplication (see section 5.1 about the Singleton design pattern).

4.4 Qualifying exportation

As seen in section 3.7, when a class has an NC-ancestor, then it is allowed to access all methods and attributes the NC-ancestor is allowed to access. We will now present an exportation issue that arises in the class STRING. The problem we present is typical of problems that can arise in many other classes. The well known Eiffel class STRING describes resizable strings. As one may guess, the class STRING is performance-critical, and need to hold a subtle balance between safety and performance. Indeed, STRING objects are used in many classes not to say in most classes. Most clients of the class STRING are satisfied with its ordinary and safe set of methods. Some other clients need to have some extra privileges in order to reach the desired efficiency. As an example, input- and output-related classes (for example the class TEXT_FILE_READ) do need to have a direct access to the internal storage area of the STRING which is used as a memory cache buffer.

Because the direct access to the storage area of STRINGs must not be the default, the `storage` instance variable must not be exported to all classes. Because we cannot predict in advance all the names of the classes which may, one day, need the privilege of direct `storage` access, it is not possible to enumerate the list of allowed classes. In order to achieve our goal the internal `storage` area of the class STRING is only exported for the class STRING_HANDLER. The class STRING_HANDLER which is also part of our standard library is just an empty class (no method and no attribute). All the classes that need to have access to the internal `storage` area of STRING just need to NC-inherit from the class STRING_HANDLER. Again, the use of NC-inheritance makes things clearer. The goal of the NC-inherited class STRING_HANDLER is just to have access to the `storage` area of STRINGs and not to define a STRING_HANDLER subtype or to use polymorphism.

4.5 Improved compiling efficiency

We have seen that NC-inheritance allows for various improvements in program design. But NC-inheritance can also have a small positive impact on compile-time and run-time performance. Since NC-inheritance does not entail a subtyping relationship as C-inheritance does, there are less edges in the subtyping graph when NC-inheritance is used. Compilers that do type prediction can predict smaller sets of possible dynamic subtypes for a given static type. Similarly, in Java it would be possible to mark a method as `final` if it is not redefined in conforming heirs, but non-conforming heirs would not need to be taken into account. This increased knowledge of static types can result in smaller dynamic dispatch functions or smaller virtual function tables.

5. NC-INHERITANCE AND DESIGN PATTERNS

The NC-inheritance mechanism allows us to better transcribe some design patterns [7] in Eiffel ([10] is the Eiffellish version of the GoF book).

We have not yet explored exhaustively the whole design pattern catalog, but the NC-inheritance mechanism impacts at least the following patterns: Singleton, Flyweight, Class Adapter and Template Method.

5.1 Singleton

The intent of the Singleton design pattern [7] is to ensure that a class has only one instance, and to provide a global access point to it. In Eiffel the universal class ANY contains among others, a duplication method analogous to Java's `clone`. In order to avoid the duplication of the Singleton object itself, the Singleton class has to hide the duplication methods which are by default exported by the ancestor class ANY. As stated in section 3 a class can now have ANY as an NC-ancestor and thus safely hide the duplication methods from its clients. Thus, it is better for the Singleton class to use NC-inheritance than traditional conforming inheritance.

5.2 Flyweight

The intent of the Flyweight design pattern [7] is to use sharing to support large numbers of fine-grained objects efficiently. In such a situation a maximal aliasing is required and the developer does not want duplication methods to be called (except, maybe, for the Flyweight factory itself). As for the Singleton design pattern, it is safer to limit the exportation of the duplication methods inherited from the ANY universal class. By using NC-inheritance for the Flyweight class definition, it is safe to completely hide all duplication methods or to limit the use of the duplication methods for the Flyweight factory only.

5.3 Class Adapter

The Class Adapter design pattern [7], not to be confused with the *Object* Adapter, converts the interface of a class into another interface the clients expects.

Adapters let classes with otherwise incompatible interfaces work together. As shown on figure 5, an ADAPTER has multiple parents: typically, the class to be adapted, the ADAPTEE, and the interface we want to implement, the TARGET. Still, we *don't* want the adapter to appear as having an "*is a*"

relationship with the adaptee—that's why we are using the Adapter pattern in the first place. The adapter naturally NC-inherits the adaptee.

As a side note, the UML diagram of the Class Adapter design pattern in [10] is annotated with an *implementation* tag for the inheritance link between the class ADAPTER and the class ADAPTEE, precisely the link for which we are using NC-inheritance.

5.4 Template Method

The Template Method pattern [7] defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine some steps of an algorithm without changing the algorithm structure.

Actually, when a common conforming ancestor for all concrete classes that need to define the Template Method exists, it is straightforward that the Template Method has to be stored in this common conforming ancestor and that the NC-inheritance mechanism is not useful. However, when we are in a situation similar to the one of figure 4, that is to say when the Template Method must be shared by classes which are not related by a common conforming ancestor, the NC-inheritance does help. Thus, the class CREATE_SUPPORT can contain the Template Methods to be shared by CREATE_EXPRESSION and CREATE_INSTRUCTION. Again, the usage of NC-inheritance emphasizes the fact that the purpose of CREATE_SUPPORT is to share code.

6. RELATED WORK

The *nested inheritance* mechanism introduced in [14] is designed to fit in a single inheritance model. It does not try to achieve the same issues as our NC-inheritance mechanism. It is interesting to notice that, like NC-inheritance, the nested inheritance mechanism does not introduce soundness problems in the type system.

6.1 Sather's INCLUDE mechanism

Sather has a mechanism for code reuse that is very similar to NC-inheritance [8]. We were not aware of the existence at the time of implementing NC-inheritance in the SmartEiffel compiler and libraries and, to the best of our knowledge, Sather did not influence the ECMA Eiffel normalization group when designing NC-inheritance.

Actually, Sather has a class/interface dichotomy just like Java. As their names suggest, classes can hold code while interfaces cannnot. Just like Java interfaces, Sather interfaces can inherit other Sather interfaces, and Sather classes can implement Sather interfaces. However, there is no such thing as C-inheritance between classes in Sather. Instead, a Sather class can `include` another class. Just like multiple NC-inheritance, multiple inclusion is handled seamlessly by Sather. Sather also allows the developper to restrict the availability of included methods or attributes, just like the mechanism discussed in section 3.7.

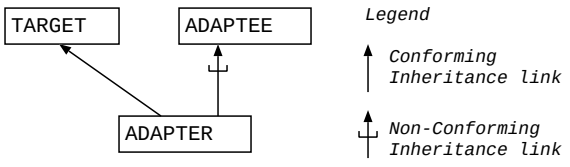

Figure 5: The class Adapter design pattern.

The main differrence between our mechanism and Sather's is the fact that that C-inheritance and NC-inheritance are handled uniformly by our mechanism, while Sather relies on a class/interface dichotomy. On another point, the designers of Sather did not reap the byproduct of safe covariant redefinitions through NC-inheritance. Actually, in Sather, only contravariant redefinitions are allowed.

6.2 C++ private/protected inheritance

The C++ mechanisms known as `private` and `protected` inheritance bear a superficial resemblance with our NC-inheritance mechanism. However, they differ in one substantial way: the C++ mechanisms do not really prevent polymorphism. When a class B privately inherits from a class A, the class B *is allowed* to perform polymorphic assignments of B objects to A variables (see C++ code of figure 6). Other classes are not allowed to perform such polymorphic *assignments*, but B can easily let pointers escape. Then, polymorphic *calls* can still happen outside of B.

```
#include <iostream>

class A {                          class B: private A {
public:                            public:
  A* as_A(void) {                    using A::as_A;
    return this;
  }                                private:
  virtual void f(void) {             virtual void f(void) {
    std::cout << "Hello";              std::cout << " world!\n";
  }                                  }
};                                 };

int main(void) {
  B* b = new B;
  A* a = new A;
  a->f();          // Polymorphic call prints "Hello"
  a = b->as_A();   // Polymorphic assignment
  a->f();          // Polymorphic call prints " world!\n"
}
```

Figure 6: The `private` inheritance mechanism of C++ does not prevent polymorphism.

As a corollary it is not possible to safely restrict the exportation of methods even when they are inherited the `private` way in C++. The example on figure 6 demonstrates that the method `f` can still be triggered on an object of type B.

As a side note, the C++ mechanism ties conformance issues to the issues of method and attribute availability: all public methods and attributes acquired through `protected` inheritance become protected, and all those acquired through `private` inheritance become private. This is not really an issue since `using` declarations can make hidden methods and attributes visible again, but we would still rather handle the separate concerns of inheritance and exportation with two separate mechanisms.

7. CONCLUSION

During our work on the Eiffel language [6], we have experimented the NC-inheritance mechanism on a large scale project and libraries. This simple mechanism appears to be valuable to capture more design information in the source code of applications. Some design patterns can be implemented in a cleaner way with NC-inheritance. We can now use this inheritance variation for implementation at no risk. With NC-inheritance, covariant redefinition of arguments is no longer dangerous and does not incur type soundness problems. Exportation can also be tightened in subclasses without any problems.

We demonstrated the mechanism in Eiffel, could be used as well in other statically typed object-oriented language such as Java.

8. ACKNOWLEDGMENTS

To all SmartEiffel users for their feedback and all their helpful comments during the experimentation of the NC-inheritance mechanism, to Bertrand Meyer as well the other members of the ECMA TC39-TG4 group normalizing committee [1].

9. REFERENCES

[1] ECMA normalisation group for the new eiffel language definition tc39-tg4 (to appear in the middle of 2005.).

[2] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, Massachusetts, USA, 1996.

[3] K. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3), 1996.

[4] G. Castagna. Covariance and Contravariance: Conflict Without a Cause. *Theory and Practice of Object Systems*, 17(3):431–447, 1995.

[5] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1996.

[6] D. Colnet, P. Ribet, C. Adrian, V. Croizier, and F. Merizen. Web site of smarteiffel, the GNU eiffel compiler tools and libraries. `http://SmartEiffel.loria.fr`.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addisson-Wesley, Reading, Massachusetts, 1995. ISBN 0201633612.

[8] B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. Sather 1.1: A language manual. `http://www.icsi.berkeley.edu/~sather/Documentation/LanguageDescription/%webmaker/DescriptionX2Erem-chapters-1.html`.

[9] A. Holub. Why extends is evil. `http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox_p.html`, 2003.

[10] J. M. Jézéquel, M. Train, and C. Mingins. *Design Patterns and Contracts*. Addisson-Wesley, 1999. ISBN 0-201-30959-9.

[11] B. Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, 1988.

[12] B. Meyer. Beware of polymorphic catcalls. `http://archive.eiffel.com/doc/manuals/technology/typing/cat.html`.

[13] B. Meyer. *Eiffel, The Language*. Prentice Hall, 1994.

[14] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 99–115. ACM Press, 2004.

[15] O. Zendra and D. Colnet. Coping with aliasing in the GNU Eiffel Compiler implementation. *Software Pratice and Experience (SP&E)*, 31(6):601–613, 2001. *J. Wiley & Sons*.