Reconciling Subtyping and Code Reuse in Object-Oriented Languages: Using *inherit* and *insert* in SmartEiffel, The GNU Eiffel Compiler.

Dominique Colnet[1] Guillem Marpons[2] and Frederic Merizen[1]

[1] LORIA (UMR 7503 CNRS-INPL-INRIA-Nancy2-UHP) - France,
`colnet@loria.fr`, `merizen@loria.fr`
[2] Universitat Politècnica de Catalunya (UPC) - Spain,
`gmarpons@lsi.upc.edu`

Abstract SmartEiffel has been enjoying two different mechanisms to express subtyping and implementation inheritance for one year. After large scale practical tests and thanks to user feedback, this paper finalises the new typing policy of SmartEiffel, which combines two forms of multiple inheritance with genericity in one statically-checked, object-oriented language.
Having two forms of inheritance allows the designer to capture more design decisions in the source code. It is now straightforward to share reusable code between otherwise unrelated types. The new mechanism allows to reuse code from an existing class without polluting the reuser's interface. It also enables an elegant implementation of some design patterns.
Furthermore this mechanism helps compilers to statically remove more dynamic dispatch code. It can also be used to add a no-penalty and no-risk multiple-inheritance-like construct to a single-inheritance language such as Java.

1 Introduction

The Eiffel language is intended to favour good software engineering practises. Its concern about building reusable software artifacts [10] should be stressed as the most salient trend of its design. The inheritance mechanism, in particular, was conceived as a powerful reuse mechanism with a panoply of adaptation facilities for inherited methods and attributes, and the possibility of multiple inheritance.

Still, subclassing and subtyping are strongly coupled in traditional Eiffel. Reusing code through inheritance is desirable for the great flexibility it provides and because of performance considerations. In spite of this, enforcing a subtyping relation when inheriting has a number of drawbacks. The problem is obviously not specific to the Eiffel world [3] and this coupling sometimes produces flawed designs (examples in section 4). The designers always have to balance these disadvantages with the hindrance of reusing code through other means than

inheritance, or even not reusing code at all—the worst being copy-pasting. In fact, the designers of Eiffel have recognised this conflict, slightly relaxing the subclassing/subtyping relation. This has been done by allowing argument types to be covariantly redefined and method visibility to be restricted in subclasses, leading to the well-known type-safety problems of the language [11].

The SmartEiffel team [3] has been working on Eiffel tools, compilers and libraries for more than ten years [1,2,12,16,17]. In the remainder of this paper, we will use the unqualified term "*Eiffel*" as a shorthand for "*the Eiffel dialect of the SmartEiffel team*" [4]. This paper discusses an implementation-only inheritance language construct that we introduced in February 2005 (release 2.1 of the SmartEiffel compiler). This mechanism is conceived as a complement to the traditional Eiffel inheritance and gives the designer full control about which subtyping relations are actually established. To avoid confusion we use the terms *inheritance* or *inheritance mechanism* for the traditional Eiffel inheritance, and *insertion* or *insertion mechanism* for the recently devised language construct [5].

Since we first introduced the insertion mechanism, we have had the opportunity to extensively put it to use, on large scale programs and libraries, and with the feed-back of many SmartEiffel users. Thus, this paper is not only an *academic research work* but also a practical report. Practising the mechanism has helped us to clarify ideas about what kind of type-system we need to best fit the new mechanism into the language. SmartEiffel currently implements the type-checking rules presented here, with one restriction regarding expanded classes that will be lifted in the next release (SmartEiffel 2.3). We refactored SmartEiffel's 450 standard library classes to exercise the insertion mechanism, and the classes now make use of 350 inherit links and 300 insert links. Our work on the library clearly indicates that the new language construct favours the reconciliation between subtyping, static checking and reusability.

The paper is organised as follows. Section 2 introduces the insertion mechanism and some necessary background about Eiffel. The new typing policy is presented in section 3. Examples of how the insertion and inheritance mechanisms can be used together to remove certain specific design flaws and improve reuse opportunities are presented in section 4. Section 5 compares our work with papers or existing languages with similar aims. Section 6 concludes.

2 Adding the insertion mechanism to Eiffel

Eiffel is an object-oriented language that features multiple inheritance, static typing, (constrained) genericity and design by contract. As in Smalltalk [6] even

[3] All authors of this paper are members of the current SmartEiffel team. This paper can be considered as the work of the whole SmartEiffel team.

[4] The reader might be aware that there are several dialects of the Eiffel language. The SmartEiffel dialect shares a common root with the ECMA standardisation attempt [4] but diverged from it in May 2005. Since then, SmartEiffel has made its own way.

[5] This mechanism has been known to Eiffel users as *non-conforming inheritance*.

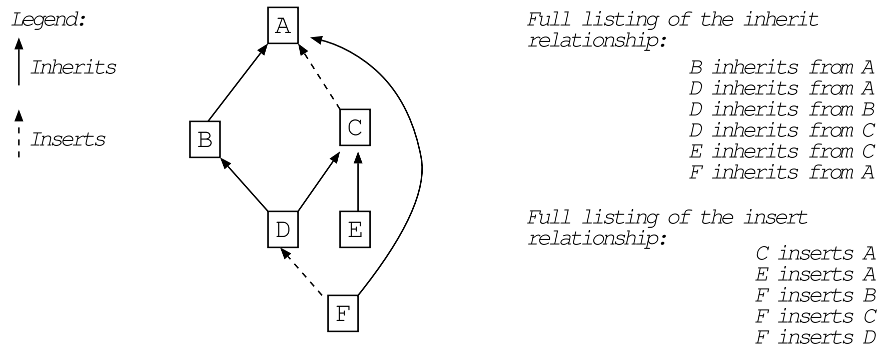

Figure 1. The inherit and the insert relationships on a non-trivial example.

basic entities such as small integers are true objects of some existing class. Uniformity in the type system helps to keep the language simple and to make all those facilities as broadly appliccable as possible. The language is well-suited for large software projects and teams.

2.1 Keeping `inherit` and adding `insert`

The traditional inheritance mechanism introduced from the very beginning of Eiffel is still present and unchanged. The inheritance relationship both induces a subtyping relationship and introduces code reuse. The classes that a class inherits from constitute a list of zero, one or more elements called the *inherit list* of that class. Syntactically, the inherit list is introduced by the `inherit` keyword if non-empty. In the remainder of this paper, "A inherits from B" means that a path using only inherit links exists from A to B. Saying "A inherits directly from B" means that B is syntactically a member of the `inherit` list of A.

In the new Eiffel syntax, a class can also have a second parent list introduced by the new `insert` keyword. This is the relationship to be used when code has to be inserted just for implementation purpose. The `insert` list can syntactically have zero, one or more parents. In the remainder of this paper, "A inserts B" means that a path using insert or inherit links exists from A to B, but A does not inherit from B (i.e. at least one insertion link exists from A to B in every path). The sum of the inheritance graph and the insertion graph must be an acyclic directed graph. Saying "A inserts directly B" means that B is syntactically a member of the `insert` list of A. Figure 1 summarises the inheritance and insertion relationships on a nontrivial example.

The idea behind the insertion mechanism is simple: to complement the traditional inheritance mechanism with a new one that keeps the code reuse aspects but discards the subtyping relationship. As we will see in the section 3, the fact that A inserts B does not allow us to assign an expression of type A into some

variable of type B. For such a polymorphic assignment to be legal, A must be a subtype of B which can also be expressed as "A inherits from B".

2.2 The special status of the ANY class

There is a special class in Eiffel, named ANY, somehow comparable with Java's `Object` class, that has been traditionally seen as the universal ancestor of the inheritance graph. A special rule still exists for class ANY, but now that we have added the new insert relationship, the status of ANY is slightly different. Class ANY is the only class that has no direct ancestor at all: both its inherit list and its insert list are empty. All other classes must have at least one direct ancestor either through the `inherit` list or through the `insert` list. Class ANY is now the final ancestor of the inheritance / insertion graph. As a consequence, if a class X is not the ANY class itself, then X either inherits from or inserts ANY.

Actually, the ANY class contains important methods that all classes must have at runtime, including code necessary for object comparison, cloning and introspection support. Thanks to the rules we have just presented, we are sure that this code is part of all Eiffel classes.

2.3 Runtime requirements

We do not only want a language dedicated to software engineering and specification, we also want Eiffel to keep the potential efficiency one can expect from a low-level language such as the C language[6]. A simple integer addition must be as efficient in Eiffel as in C. In Eiffel's runtime model, this goal is achieved by having two kinds of classes: normal classes and *expanded classes*. *Expanded objects* are the instances of expanded classes.

Most classes are normal (non-expanded). Normal objects are handled through a reference. When passed as an argument, a pointer to the object is copied onto the stack. Dynamic dispatch is enabled for such objects.

The machine's basic entities such as integers, floating-point numbers and booleans are defined by expanded classes. Users can create their own expanded classes using the `expanded` keyword. Expanded objects are directly represented as values and cannot be shared. Argument-passing is performed by copying the object's value onto the machine stack. There is no polymorphism for expanded objects.

We require expanded classes to have an empty inherit list and never appear in another class's inherit list. In that way, rule 1 (section 3) is enough to express within the type system the fact that there is no polymorphism involving expanded objects.

---

[6] The runtime efficiency of numerous languages is compared at the "Computer Language Shootout Benchmarks" (`http://shootout.alioth.debian.org`). Although we don't completely agree with the measuring scale, we want our Eiffel language to be able to sustain a good rank in this comparison.

2.4 Genericity

Eiffel has got generic classes from the very beginning. One of our design goals was to get the most constructive interaction possible between reuse through genericity and reuse through insertion.

A typical example of genericity at work is the class ARRAY[E_] where E_ is called the formal generic argument. The generic argument E_ could carry a constraint (see next paragraph), but this is not the case here so E_ can be replaced with any type (called an actual generic argument), including expanded types: ARRAY[STRING], ARRAY[INTEGER_8], ARRAY[ARRAY[INTEGER_16]] etc. are all valid types. Being able to directly use integers of the underlying machine as the element type of a generically defined class is an important benefit of having a very uniform typing policy.

Constrained genericity is the possibility to require a generic parameter to implement some existing class, generally an abstract class with just a few abstract methods. For example, the elements of a hash-based implementation of a set must implement the methods of class HASHABLE. This is denoted by giving the hashed set a constrained generic parameter: HASHED_SET[E_->HASHABLE]. In section 3 (rule 5), we claim that it is sufficient for the actual generic parameter to insert the constraint. An unconstrained generic parameter can be treated as a syntactic shortcut for a parameter constrained by the class ANY. For instance, the array class can be considered as defined as ARRAY[E_->ANY]. An important consequence is that expanded classes can be used as generic parameters although they cannot inherit from any class (including ANY).

3 Typing and checking policy

Because of space concerns, it is not possible to give the whole set of rules we apply to check Eiffel programs in this paper. We will focus on the main questions about our typing and checking policy in order to be able to answer the following crucial questions: What can be assigned into what? What type can be used when some method or attribute is overridden? What type can be used in case of a generic derivation? What about constrained genericity?

Rule 1 (Assignment) An expression of type A can be assigned into a variable of type B if and only if A and B are the same type or A inherits from B.

This first rule prevents an expression of one type from being assigned to a variable of another type if no subtyping relation exists between them. It directly follows that if A inserts B, the assignment is statically rejected and polymorphism is not possible. Since expanded classes cannot be at either end of an inheritance relationship (section 2.3), rule 1 is enough to forbid assignments from an expanded type to any other different type.

Rule 2 (Argument Passing) An expression of type A can be passed as an argument of formal type B if and only if A and B are the same type or A inherits from B.

This second rule simply states that the same policy is used both for assignments (rule 1) and for argument passing (rule 2).

Rule 3 (Redefinition under Inheritance) When overriding an inherited method or attribute, the types of any argument and/or of its result can be replaced covariantly with a type that inherits from the replaced type (i.e. a subtype).

Actually, for inheritance, the legacy covariant principle of Eiffel is kept unchanged: all types are treated the same way (both result type and argument types), the number of arguments cannot change; if the overridden signature has a result type, the new one must also have a result type. Thus, all legacy code can be reused as is.

Rule 4 (Redefinition under Insertion) When overriding an inserted method or attribute, the type of any argument and/or of its result can be replaced covariantly with a type that inherits from *or inserts* the replaced type.

As in the case of rule 3, we assume that the overridden signature keeps the same number of arguments and that a result type cannot be discarded or added. Rule 4 is more permissive than rule 3 about the new type one can use because the overridden feature comes through a direct member of its `insert` list. Polymorphism between this inserted ancestor and the current type is not possible and it is thus safe to do so. This increased freedom is in fact beneficial and gives us more flexibility to adapt inserted code.

In a similar way, the exportation status of the inserted methods and/or attributes (i.e. from which other classes they are visible or can be called) can be freely changed. For example, we can insert a public attribute and convert it to a private one, or any other of the fine-grained intermediate possibilities that Eiffel offers. In traditional Eiffel this was also allowed under inheritance, but this creates type safety holes in the same manner as covariant argument changing [11]. Under insertion, the mechanism is safe and in fact plays an important methodological role as shown in section 4.2.

Rule 5 (Generic Derivation) When deriving an actual type from a generic class, we can instantiate each parameter using any class that inserts or inherits from the type that constrains the corresponding placeholder. The resulting fully instantiated type is only valid if statically proved correct (recursively) with respect to rules 1-5.

This rule allows to instantiate a generic class using a parameter that is not a subtype of the constraining type. Both inheritance and insertion are permitted here. This can be surprising at the beginning, but is type safe, because the replacement of the formal parameter by the actual one is done at compile time and the generic derivation obtained is statically checked. In fact, polymorphism is often not needed between the generic type constraint and the type that replaces it in a generic derivation. If the actual parameter is a subtype of the constraining type, the derivation will be statically valid. On the contrary, if the actual parameter only inserts the constraining type, the validity of the derivation must be checked by the compiler.

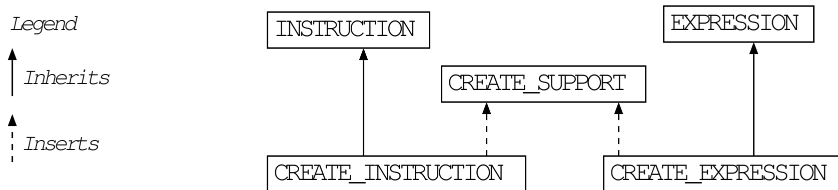

Figure 2. Sharing code between unrelated classes. A routine storage example.

Rule 5 is meant to accept expanded types as any other parameter in classes such as ARRAY[E_->ANY]. In this way the insertion mechanism helps us to fit non-polymorphic and efficient basic types (i.e expanded types) in a very uniform type system that maximises type combining opportunities.

The insertion mechanism defined by those rules provides useful program design options, but it can also improve runtime and compile-time performance. Since insertion does not entail a subtyping relationship as inheritance does, there are less edges in the subtyping graph when insertion is used. Compilers that do type prediction [16] can predict smaller sets of possible dynamic subtypes for a given static type. This increased knowledge of static types can result in smaller dynamic dispatch functions or smaller virtual function tables. It would also be possible to extend single-inheritance based languages like Java with this mechanism, without changing the dynamic dispatch mechanism.

4 Examples of code reuse

The next examples are intended to show how the new insertion mechanism can help to reconcile reuse and other software engineering principles. We start with code sharing (section 4.1) because it is the most simple use of the insertion mechanism, although it is not the most illustrative of the benefits of insertion when—as in Eiffel—multiple inheritance is available. Many of the examples come from the SmartEiffel libraries and tools [1].

4.1 Sharing methods and attributes with insert

Two or more classes will sometimes have a lot of commonalities in their *implementation*, while playing quite different *roles* in the design. The insertion mechanism allows those classes to share code without adding any spurious subtyping relations. For instance, in the SmartEiffel compiler the class CREATE_EXPRESSION is a specialisation of the abstract class EXPRESSION (a CREATE_EXPRESSION *is an* EXPRESSION), as shown in figure 2. Similarly, the class CREATE_INSTRUCTION is a specialisation of the abstract class INSTRUCTION. Because the implementation of the class CREATE_EXPRESSION is similar to the implementation of the class CREATE_INSTRUCTION, many common instance variables and methods are defined in CREATE_SUPPORT. The use of insertion (figure 2) emphasises the fact

that CREATE_SUPPORT is just a useful place to share common code which is necessary to implement both CREATE_EXPRESSION and CREATE_INSTRUCTION and that there is no possible substitution of CREATE_EXPRESSION with CREATE_INSTRUCTION (or conversely). If inheritance was used (this would probably be the solution adopted with traditional Eiffel) this design decision would not appear in the code, making the role of each class more obscure.

Interestingly, the methods defined in CREATE_SUPPORT can be triggered due to dynamic dispatch although they are inserted (not inherited). For example, a shared method of CREATE_SUPPORT can be 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 a method is inserted does not preclude using dynamic dispatch on that method.

While this example shows that the insert mechanism is a valuabale addition to the Eiffel language, it would be even more beneficial to languages such as Java that do not have multiple class inheritance. Note that all three of CODE_SUPPORT, EXPRESSION and INSTRUCTION contain code, and reducing any of them to mere interfaces would result in significant code duplication. Furthermore, all three of them also have state (instance variables). It would not be reasonable, for instance, to move CODE_SUPPORT up in the hierarchy and turn it into a common ancestor EXPRESSION and INSTRUCTION, as this would clutter numerous non-creation-related instructions and expressions with useless attributes. Adding the insert mechanism to the Java language would solve this problem with little or no changes required to the runtime system. Particularly, since insertion does not create dynamic dispatch possibilities, the single-inheritance dispatch mechanism of the JVM could be kept intact.

As a special case of code sharing, the insertion mechanism can be used to share a set of related constant definitions. This effectively emulates the primary use of Java's "static import" mechanism.

4.2 Hiding excess methods

When inheritance is the only code reuse mechanism available, the benefits of reusing code will sometimes be mitigated by the fact that the interface of the child gets polluted by some public methods from the parent. Using the insertion mechanism, the programmer can effectively make those superfluous methods private, eliminating their impact on the child's interface.

For instance, in Java the class `java.util.Stack` extends `java.util.Vector`, where the class `Vector` is a resizable ordered collection of elements. This technique makes `Stack` very easy to implement and avoids duplicating `Vector`'s code. Unfortunately, the spurious subtyping relationship between a Java `Stack` and a `Vector` makes it possible to use all methods of the class `Vector` on `Stack` objects as well. The class obtained is not a pure stack because one can directly peek or put elements at any position in the `Stack`.

In traditional Eiffel, the class STACK can hide methods and attributes inherited from the class VECTOR, but this protection is quite weak. If STACK inherited

from VECTOR then a STACK object could be polymorphically assigned to a VECTOR variable and any method of VECTOR we wanted applied to the STACK object could be called (even supposing that some methods like `put` had been removed from its public interface). The following code is accepted by the compiler in case of inheritance:

```
local
  s: STACK
  v: VECTOR
  ...
do
  create s.make
  ...
  v := s
  v.put(an_element, a_position)
  ...
```

It is dangerous for an external client of an object to call methods that have been eliminated from its public interface because they can violate the invariant of the class and the object would have an unpredictable behaviour as from that point. In fact, code like that must be considered incorrect, even if current Eiffel compilers are not capable of discovering this situation in the general case. SmartEiffel avoids this problem by forbidding narrowing the exportation status of an inherited method or attribute.

On the contrary, the protection is effective if the class STACK reuses code from the class VECTOR by inserting it. With insertion it is no longer possible to assign an expression of type STACK into a VECTOR variable, so the previous code is rejected. It is always safe to narrow the exportation status of inserted methods and attributes because they can not be secretly called via a parent interface. This way we can easily implement a pure and safe stack.

Changing covariantly the type of an argument to an inserted method is also safe. If the class VECTOR had a binary method `merge(other: VECTOR): VECTOR` for getting a new vector containing the elements of two vectors merged in a particular order, it would be possible to redefine the method in class STACK as to cope with stacks only. The STACK's version of the method would have signature `merge(other: STACK): STACK`, and as it could never be called through the VECTOR's interface, the compiler could guarantee that only stack objects were passed to the method. Covariant change of argument types is still useful also when inheriting, so the traditional rule, even if unsafe, has been kept.

As inheriting from ANY is no longer mandatory (see section 2.2), even its methods can be reliably hidden if the designer wants to. For instance, one may decide not to export the possibly slow `equals` comparison method. This is a way to promote the usage of the basic fast comparison operator (the built-in `==` in Java), and to favour the use of as much aliasing as possible [17].

Similarly, the universal `clone` inserted method can be hidden in some classes in order to make duplicating instances of that class a restricted operation. This

is useful when implementing the Singleton and Flyweight design patterns [5]. The intent of the Singleton design pattern is to ensure that a class has only one instance, and to provide a global access point to it. The intent of the Flyweight design pattern is to use sharing to support large numbers of fine-grained objects efficiently. A maximal aliasing of flyweight objects is required and the developer does not want duplication methods to be called (except, maybe, for the Flyweight factory itself). Thanks to the insertion mechanism, the accidental duplicating of a Singleton or Flyweight object can be avoided safely.

4.3 Other design pattern examples

Finally, we show how some common design patterns can be better implemented with insertion. The Class Adapter design pattern [5] (not to be confused with the *Object* Adapter) converts the interface of a class into another interface the clients expect.

Adapters let classes with otherwise incompatible interfaces work together. As shown on figure 3, 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 be a subtype of the adaptee—that's why we are using the Adapter pattern in the first place. It is natural for the adapter to insert the adaptee.

As a side note, the UML diagram of the Class Adapter design pattern in [8] 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 insertion.

The Template Method pattern [5] 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.

When all concrete classes that need to define the Template Method are subtypes of a common ancestor type, it is straightforward that the Template Method has to be stored in this common ancestor and that the insertion mechanism is not useful. However, in a situation similar to the one of figure 2, that is to say when the Template Method must be shared by classes that are by no other means related, the insertion does help. Thus, the class CREATE_SUPPORT can contain Template Methods to be shared by CREATE_EXPRESSION and CREATE_INSTRUCTION. Again, the use of insertion emphasises the fact that the purpose of CREATE_SUPPORT is to share code.

Figure 3. Using insertion for the Adapter design pattern.

5 Related work

There have been many attempts to provide separate mechanisms for code reuse and subtyping. Most of those solutions also separate the unit of code reuse (modules, implementations, traits. etc.) from the unit of subtyping (interfaces), while C++ and our solution rely on a single unit, the class.

5.1 ECMA Eiffel

Our proposal is rooted in the discussions for providing an insertion-like mechanism for Eiffel during the standarization efforts of the last years, discussions in which the SmartEiffel team has been involved. The ECMA final document [4] includes such a mechanism—with a different syntax—that at the time this is being written has not been implemented by any compiler. As, among other differences, the runtime model supported by ECMA Eiffel differs from the one of SmartEiffel (for example, expanded objects have a completely different meaning), the typing policy and the integration of the insertion mechanism with other features of the language have been specifically developed for SmartEiffel.

5.2 SCG traits

Schärli *et al.* of the Software Composition Group present a mechanism called traits and discuss its implementation in Smalltalk [13]. Unlike classes, traits cannot have attributes and they cannot be part of an inheritance relationship, neither as parents nor as children. However, traits and classes can be *composed* of one or more sub-traits. Like insertion, the trait composition mechanism allows to reuse code without implying a subtyping relationship.

The trait composition mechanism avoids the most difficult issues of diamond inheritance, so the authors claim, thanks to traits not having attributes. In Eiffel, it is possible to insert a class even if it has attributes without suffering adverse effects. In the case of diamond inheritance, Eiffel's standard feature renaming mechanism allows the program to choose for each attribute if it is to be shared or replicated.

Because they do not take part in inheritance relations, traits enjoy the *flattening property*, which means that trait composition can be interpreted as simple textual substitution (with a few simple restrictions to take naming conflicts into account). Most importantly, if a class obtains a method that contains the `super` keyword through trait composition, that keyword's meaning does not depend on the trait where the keyword is written, but only on the class that contains the method through composition. The flattening property makes it possible to write traits that work as generic extensions. For instance, the authors show how to write a trait that can be used to make a synchronising stream from any input stream. We cannot hope to duplicate this result in a language with multiple inheritance such as Eiffel because, in the case of an inheritance diamond, the `Precursor` keyword must be complemented with the name of a parent class to avoid ambiguity. Of course, it is not possible to write the name of the parent class in a trait, since this name is only known in the classes that use this trait.

5.3 Mixins

In the mixin inclusion mechanism as illustrated by Ruby's [15] implementation, the unit of code reuse, called a *module*, can hold methods and define state variables. Mixin inclusion is similar to class insertion in the sense that a class or a module can include one or more modules to reuse their implementation without entering a subtyping relationship. Mixins differ from classes in the sense that classes cannot inherit from modules, and modules cannot inherit from anything. Moreover, modules cannot have direct instances. Technically, mixin inclusion is handled by language processors through anonymous classes and an underlying single inheritance mechanism, which requires the module inclusion graph to be linearised.

5.4 Sather's class inclusion

Sather has a mechanism for code reuse that is very similar to insertion [7]. Unlike Eiffel, Sather has a class/interface dichotomy just like Java. As their names suggest, classes can hold code while interfaces cannot. In Sather, interfaces can inherit from interfaces, and classes can implement interfaces. However, there is no such thing as inheritance between classes in Sather. Instead, a class can `include` another class. Sather's inclusion mechanism resembles Eiffel's insertion mechanism in that it handles multiple inclusion seamlessly, and allows the developer to restrict the availability of included methods or attributes, just like the technique discussed in section 4.2.

The main difference between our mechanism and Sather's is the fact that inheritance and insertion 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 insertion. In Sather, only contravariant redefinitions of argument types are allowed.

5.5 Timor's reuse variables

The Timor language [9] separates interfaces from implementations in a way reminiscent of Sather.

Interfaces define types by listing the names and signatures of their methods, and they can take part in subtyping relationships by *extending* other interfaces. Interfaces can also *include* other interfaces, which allows them to inherit from interfaces without entering a subtyping relationship—a process that can be simulated in Eiffel by abstract classes inserting other abstract classes.

Implementations of a given interface supply actual code and attributes to support that interface's semantics. Implementations cannot play the role of a parent or a child in an inheritance relation. Instead, to achieve code reuse, implementations can have *reuse variables*, which behave like normal sub-objects with one exception. If an implementation lacks a method that is part of its interface, then that method is statically looked up in its reuse variables.

One interesting consequence of this model that completely decouples code reuse from inheritance is that the reused code does not pollute the reusing implementation. Its state variables are kept logically separate, and methods that are not needed to implement the reuser's interface are not included into it.

5.6 C++ private/protected inheritance

The C++ mechanisms known as `private` and `protected` inheritance [14] bear a superficial resemblance with our insertion 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. While other classes are not allowed to perform such polymorphic *assignments*, there is no mechanism to prevent B from letting pointers escape, so polymorphic *calls* can still happen even outside of B. As a corollary, when it comes to hiding inherited methods C++ private or protected inheritance has the same problem as Eiffel inheritance. If B lets a pointer to itself of type A escape, that pointer can be used to access even `private`ly inherited methods.

6 Conclusion

During our work on the Eiffel language, we have exercised the insertion mechanism on a large scale project (the whole SmartEiffel compiler and tools as well as its large and numerous libraries [1]). This simple mechanism turns out to be valuable to capture more design information in the source code of applications. Code and design reuse opportunities then arise that were previously in contradiction with other design goals as type-safety, information-hiding, performance, or small memory footprint. Some design patterns can be implemented in a cleaner way with insertion. With insertion, 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.

The slight increase in complexity that this new mechanism entails is more than offset by the better match of efficient non-polymorphic types (expanded types) with the insertion mechanism. The uniformity of the type-system has been kept.

Acknowledgements

The research of G. Marpons is funded by the Ministerio de Educación y Ciencia and by FEDER under Grant TIC2004-06320-C03-01.

We want to acknowledge the work of Cyril Adrian and Philippe Ribet, the main architects of the compiler's new core. Many thanks to Marko Van Dooren for his feedback during the preparation of this paper. We are also grateful to all SmartEiffel users for their feedback and all their helpful comments during the practice of the insertion mechanism.

References

1. 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`.
2. D. Colnet, P. Coucaud and O. Zendra. "Compiler Support to Customize the Mark and Sweep Algorithm." 1st ACM SIGPLAN International Symposium on Memory Management (ISMM'98), pages 154-165, 1998, Vancouver, BC, Canada.
3. W. Cook, W. Hill, and P. Canning. "Inheritance is Not Subtyping." 17th ACM Symposium on Principles of Programming Languages, pages 125–135, 1990.
4. ECMA Normalisation group for the Eiffel language definition TC39-TG4, `http://www.ecma-international.org/memento/TC39-TG4.html`.
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." Addisson-Wesley, Reading, Massachusetts, 1995. ISBN 0201633612.
6. A. Goldberg and D. Robson. "Smalltalk-80, the Language and its Implementation." Addison-Wesley, Reading, Massachusetts, 1983.
7. B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. "Sather 1.1: A language manual." `http://www.icsi.berkeley.edu/~sather`.
8. J. M. Jézéquel, M. Train, and C. Mingins. "Design Patterns and Contracts." Addisson-Wesley, 1999. ISBN 0201309599.
9. J. L. Keedy, C. Heinlein and G. Menger. "Reuse Variables: Reusing Code and State in Timor." International Conference on Software Reuse 2004, Vol. 3107 of LNCS, Springer Verlag, pages 205–214, 2004.
10. B. Meyer. "Reusable Software." Prentice Hall, 1994. ISBN 0132454998.
11. B. Meyer. "Beware of polymorphic catcalls." `http://archive.eiffel.com`.
12. P. Ribet, C. Adrian, O. Zendra and D. Colnet. "Conformance of agents in the Eiffel language." Journal of Object Technology (JOT), 2004, Volume 3. No 4., pages 125-143.
13. N. Schärli, S. Ducasse, O. Nierstrasz, A. Black. "Traits: Composable Units of Behavior." ECOOP 2003, European Conference on Object-Oriented Programming, vol. 2743 of LNCS, Springer Verlag, pages 248–274, July, 2003.
14. B. Stroustrup. "The C++ Programming Language." Addisson-Wesley, 2000. ISBN 0201889544.
15. D. Thomas and A. Hunt. "Programming Ruby: The Pragmatic Programmer's Guide." Addison-Wesley, Reading, Massachusetts, 2001. ISBN 0201710897.
16. O. Zendra, D. Colnet and S. Collin. "Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler." OOPSLA'97, 12th ACM Conference on Object-Oriented Programming Systems, Languages and Applications, pages 125–141, 1997.
17. O. Zendra and D. Colnet. "Coping with aliasing in the GNU Eiffel Compiler implementation." Software Practice and Experience (SP&E), Vol. 31 No 6, pages 601–613, 2001. J. Wiley & Sons.