JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 4 (April 2004)
Special issue: TOOLS USA 2003

How to get a Singleton in Eiffel?

Karine Arnout, Swiss Federal Institute of Technology, Zurich, Switzerland
Éric Bezault, Axa Rosenberg, U.S.A.

Abstract

Can the *Singleton* pattern [Gamma95] be turned into a reusable component? To help answer this question, we have reviewed existing implementations and tried to improve them. This article explains the difficulties of having a single-instance class in Eiffel and proposes language extensions, namely once creation procedures, which would be satisfactory in most cases, or frozen classes.

1 INTRODUCTION

Achieving reusability in software is one of the core goals of object technology and key aspect of software quality. "*Quality through components*" [Arnout02] should be the motto for all software programmers. However, much code is still written anew whenever a new project starts without benefiting from previous similar developments.

This is particularly true for design patterns [Gamma95] [Vliss98]. Most existing implementations are just possible ways to get the pattern in a particular context and a given example. But an example — or a template — is not a reusable library. One should pursue a higher degree of reuse and examine whether the pattern mechanism could be provided in a component that clients could reuse off-the-shelf and just focus on the implementation part that is specific to their applications.

The Eiffel event library [Arslan03] [Meyer03], which provides the subscription/notification mechanisms of the Observer design attern, shows that building reusable components from design patterns is not a pure utopia.

The rest of this presentation focuses on one "*creational design pattern*" [Gamma95]: the *Singleton*. First we examine existing attempts to implement the Singleton pattern, and discuss their limitations; then we propose an extension to the Eiffel language that would facilitate writing singletons in Eiffel:

Section 2 focuses on the *Singleton* pattern: it describes the difficulty (even impossibility) to obtain single-instance classes in current status of the Eiffel programming language with concrete examples.

Cite this article as follows: Karine Arnout, Eric Bezault: "How to get a Singleton in Eiffel?", in *Journal of Object Technology*, vol. 3, no. 4, April 2004, Special issue: TOOLS USA 2003, pp. 75-95, http://www.jot.fm/issues/issue_2004_04/article5

Section 3 proposes to loosen the existing creation clause rule of Eiffel to allow once creation procedures, supplies an appropriate semantics, and discusses the limitations of such a solution.

Section 4 explores a language extension that already exists in Eiffel for .NET, namely frozen classes.

Section 5 concludes with an assessment of the analysis performed and gives further research directions.

The analysis reported here is part of a broader research plan of trying to turn design patterns described in the *Design Patterns* book [Gamma95] (at least some of them) into reusable components.

2 SINGLETON

The Singleton pattern

The intent of the Singleton pattern is to "*ensure a class only has one instance, and provide a global point of access to it*" ([Gamma95], p 127).

The following diagram shows the classes involved in the *Singleton* pattern and the relationships between them ([Jézéq99], p 79):

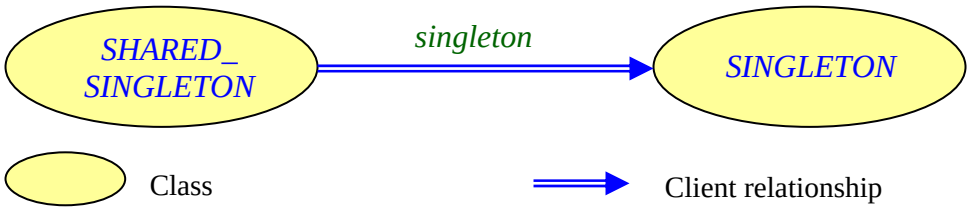


Fig. 1: Class diagram of the Singleton Pattern

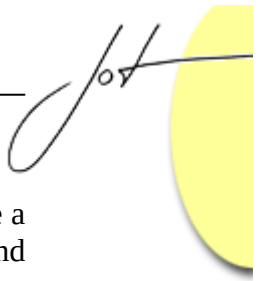
Note that class `SHARED_SINGLETON` *was called* `SINGLETON_ACCESSOR` *in [Jézéq99]. We changed its name to better comply with well accepted Eiffel naming conventions.*

How to get a "Singleton" in Eiffel

The *Design Patterns* book [Gamma95] explains with C++ examples how difficult it may be to ensure that a class has no more than one instance. C++ uses static functions for that purpose. Since Eiffel does not have static features, we need to explore another way: once routines.

Although the Eiffel programming language natively includes a keyword — **once** — which guarantees that a function is executed only once (subsequent calls return the same value as the one computed at first call), the implementation of the *Singleton* pattern is not trivial.

Note that once routines are executed once in the whole system not once per class.



The *Design Patterns and Contracts* book [Jézéql99] tries but fails [JézéqErr] to provide a solution. Let's examine the proposed scheme to identify what was wrong with it and attempt to correct it.

The *Design Patterns and Contracts* solution

Here is the approach suggested in [Jézéq99]: Make a class inherit from *SINGLETON* (**Fig. 2**) to specify that it can only have one instance thanks to its invariant and provide a global access point to it through a class *SHARED_SINGLETON* (**Fig. 3**).

```
class SINGLETON

feature {NONE} -- Implementation

   frozen the_singleton: SINGLETON is
                -- The unique instance of this class
          once
                Result := Current
          end

invariant

   only_one_instance: Current = the_singleton

end
```

Fig. 2: Class SINGLETON

```
deferred class SHARED_SINGLETON

feature {NONE} -- Implementation

   singleton: SINGLETON is
                -- Access to a unique instance.
                -- Should be redefined as a once function in
                -- concrete subclasses.
          deferred
          end

   is_real_singleton: BOOLEAN is
                -- Do multiple calls to singleton return the same
                -- result?
          do
                Result := singleton = singleton
          end

invariant
   singleton_is_real_singleton: is_real_singleton
end
```

Fig. 3: Class SHARED_SINGLETON (Access point to SINGLETON)

In fact, as explained in the errata of the book [JézéqErr], such an implementation does not work: it allows only one singleton per system. Indeed, if one inherits from class *SINGLETON* several times, the feature *the_singleton* (**Fig. 2**), because it is a once function inherited by all descendant classes, would keep the value of the first created instance, and then all these descendants would share the same value. This is not what we want because it would violate the invariant of *SINGLETON* in all its descendant classes except the one for which the singleton was created first.

One would need "*once per class semantics to create singletons as suggested by the book. Since the concept does not exist in Eiffel,* [one] then [has] *to copy all the code that is in SINGLETON to* [one's] *actual singletons*" [JézéqErr].

The last sentence by Jean-Marc Jézéquel suggests writing a "singleton skeleton" in Eiffel. We will now examine this approach.

Singleton "skeleton"

Fig. 4 shows a possible "skeleton" for the singleton pattern:

```
indexing

   description: "Template to transform a class into a Singleton"
   usage: "[
             Copy/paste this code into the class you want to
             transform into a singleton
             and change the class names SHARED_SINGLETON and
             SINGLETON if needed.
           ]"

class SHARED_SINGLETON

feature {NONE} -- Implementation

   singleton: SINGLETON is
               -- Access to a unique instance
         once
               create Result
         ensure
               singleton_not_void: Result /= Void
         end

   is_real_singleton: BOOLEAN is
               -- Do multiple calls to singleton return the same
               -- result?
         do
               Result := singleton = singleton
         end
```

```
invariant

   accessing_real_singleton: is_real_singleton

end
```

Fig. 4: Singleton skeleton

With:

```
deferred class SINGLETON

feature {NONE} -- Access

   singleton: SINGLETON is
              -- Effect this as a (frozen) once routine.
              -- It should return Current.
         deferred
         end

invariant

   remain_single: Current = singleton

end
```

Fig. 5: Class SINGLETON used by the Singleton skeleton (wrong solution)

What's wrong with this implementation?

In spite of the name *is_real_singleton*, this code does not provide a "real" singleton. Declaring *singleton* as a once function does ensure that any call to this function returns the same object, but nothing prevents the program from creating another instance of class *SINGLETON* somewhere else in the code, which breaks the whole idea of a singleton.

Having an invariant in class *SINGLETON* to detect attempts to create a singleton twice is not a proper solution either. The problem is that, in debugging mode, even though the invariant will catch errors at run-time when the singleton pattern is violated, clients of class *SINGLETON* have no means to ensure that this invariant will never be violated (they cannot test for it as they can do for a precondition before calling a routine), which reveals a bug in the implementation of the class. The Design by Contract™ method gives the following definition of class correctness ([Meyer92], p 128; [Meyer97], p 370):

A class is correct with respect to its assertions if and only if:

C1: For any valid set of arguments x_p to a creation procedure p:
$\{Default_c \text{ and } pre_p\ (x_p)\}\ Body_p\ \{post_p\ (x_p) \text{ and } INV\}$

C2: For every exported routine r and any set of valid arguments x_r:
$\{pre_r\ (x_r) \text{ and } INV\}\ Body_r\ \{post_r\ (x_r) \text{ and } INV\}$

Fig. 6: Definition of class correctness

A violation of $\{Default_c \text{ and } pre_p\ (x_p)\}$ or $\{pre_r\ (x_r) \text{ and } INV\}$ is the manifestation of a bug in the client.

A violation of $\{post_p\ (x_p) \text{ and } INV\}$ or $\{post_r\ (x_r) \text{ and } INV\}$ is the manifestation of a bug in the supplier.

How is class correctness related with this singleton implementation?

The definition of the singleton pattern given in [Gamma95] (p 127) states that the corresponding class should have at most one instance, which means that we want to prevent creating more than one such object. In other words, as a client of class *SINGLETON*, I want to know whether the instruction:

```
create s.make
```

with:

```
s: SINGLETON
```

is valid before calling it, hence I want to write code like:

```
if is_valid_to_create_a_new_instance then
   create s.make
else
   -- Either report an error or
   -- try to return a reference to the already created object.
end
```

The problem with class *SINGLETON* is that it provides no way to ensure the condition *is_valid_to_create_a_new_instance* before calling $Body_p$. Since we are dealing with creation routines, the relevant rule for assessing class correctness is C1. We will get a violation of INV (on the right hand side of the formula) if we create a second instance of the class. This indicates a bug in the class *SINGLETON* itself, not in the client of the class.

> *Note that restricting access of the creation procedure of SINGLETON to class SHARED_SINGLETON would still not ensure correctness since one can inherit from SHARED_SINGLETON — and this is the expected way to use SHARED_SINGLETON to get access to feature singleton — and then call a creation procedure on SINGLETON at will. A possible solution — although not perfect because it violates the Open-Closed principle, [Meyer97] p 57-61 — is to use frozen classes (classes from which one cannot inherit) as we describe later (see section 4), but the current version of Eiffel does not authorize them (it only allows frozen features).*

Besides, relying on the evaluation of invariants to guarantee the correctness of a class is not good design: a program should behave the same way regardless of the assertion monitoring level.

Tentative correction: Singleton with creation control

Let's try to correct the previous implementation and define a Boolean feature *may_create_singleton* in *SINGLETON* accessor (**Fig. 7** and **8**).

```
class MY_SINGLETON

inherit

   SINGLETON

create

   make

feature {NONE} -- Initialization

   make (an_accessor: MY_SHARED_SINGLETON) is
            -- Create a singleton from an_accessor.
         require
            an_accessor_not_void: an_accessor /= Void
            may_create: an_accessor.may_create_singleton
         do
         end

feature {NONE} -- Implementation

   singleton: SINGLETON is
            -- Access to unique instance
         once
            Result := Current
         end

end
```

Fig. 7: Singleton with creation control (wrong solution)

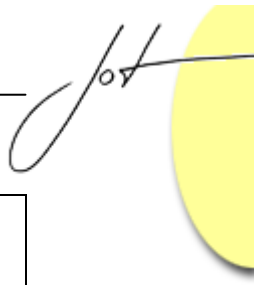
```
class MY_SHARED_SINGLETON

feature -- Status report

   may_create_singleton: BOOLEAN is
            -- May a new singleton be created? (i.e.
            -- is there no already created singleton?)
         do
            Result := not singleton_created.item
         end

feature -- Access
```

```
    singleton: MY_SINGLETON is
                -- Access to unique instance
          once
                create Result.make (Current)
                singleton_created.set_item (True)
          ensure
                singleton_not_void: Result /= Void
                may_not_create_singleton:
                      not may_create_singleton
          end

   feature {NONE} -- Implementation

    singleton_created: BOOLEAN_REF is
                -- Has singleton already been created?
          once
                create Result
          ensure
                result_not_void: Result /= Void
          end

   end
```

Fig. 8: Accessor to singleton with creation control (wrong solution)

However, the feature *may_create_singleton* does not solve the correctness problem detailed earlier: it does not prevent from calling two creation instructions as in the following example (**Fig. 9**) and breaking our “singleton”.

```
   class MY_TEST

   inherit

      MY_SHARED_SINGLETON

   create

      make

   feature {NONE} -- Initialization

      make is
                -- Create two instances of type MY_SINGLETON.
          local
                s1, s2: MY_SINGLETON
          do
```



```
                if may_create_singleton then
                        create s1.make (Current)
                end
                if may_create_singleton then
                        create s2.make (Current)
                end
        end

end
```

Fig. 9: Class breaking the singleton with creation control

Indeed, *MY_TEST* does not call the once function singleton of class *MY_SHARED_SINGLETON*, which means that *may_create_singleton* is never set to **False** and both *s1* and *s2* get instantiated.

The important point here is that we have broken the "singleton skeleton" by just looking at the interface forms of classes *MY_SINGLETON* and *MY_SHARED_SINGLETON* and writing code that does not violate the Design by Contract principles (although it would violate an invariant when executed!).

> *The interface form of a class retains only specification-related information of the publicly available features: the signature of features (of both immediate and inherited features), the comments, and contracts (involving exported features only), namely what a client of the class needs to know about.*

The Gobo Eiffel singleton example

Fig. 10 and **11** show a better approach to the "singleton pattern problem" in Eiffel. (It is a Gobo Eiffel Example [Gobo].)

```
class MY_SINGLETON

inherit

   MY_SHARED_SINGLETON

create

   make

feature {NONE} -- Initialization

   make is
                -- Create a singleton object.
        require
                singleton_not_created: not singleton_created
        do
                singleton_cell.put (Current)
        end
```

```
   invariant

      singleton_created: singleton_created
      singleton_pattern: Current = singleton

   end
```

Fig. 10: The Gobo Eiffel singleton example

```
   class MY_SHARED_SINGLETON

   feature -- Access

      singleton: MY_SINGLETON is
                  -- Singleton object
            do
                  Result := singleton_cell.item
                  if Result = Void then
                        create Result.make
                  end
            ensure
                  singleton_created: singleton_created
                  singleton_not_void: Result /= Void
            end

   feature -- Status report

      singleton_created: BOOLEAN is
                  -- Has singleton already been created?
            do
                  Result := singleton_cell.item /= Void
            end

   feature {NONE} -- Implementation

      singleton_cell: CELL [MY_SINGLETON] is
                  -- Cell containing the singleton if already created
            once
                  create Result.put (Void)
            ensure
                  cell_not_void: Result /= Void
            end

   end
```

Fig. 11: Accessor to the Gobo Eiffel singleton example

This implementation is still not perfect; one can still violate the invariant of class *MY_SINGLETON* by:

cloning a singleton — using feature `clone` or `deep_clone` from `ANY`;
using persistence — retrieving a "singleton" object that had been stored before (using the `STORABLE` mechanism of Eiffel or a database library);
inheriting from `MY_SHARED_SINGLETON` and "cheating" by putting back **`Void`** to the cell after the singleton has already been created. Note though that here one needs to access and modify non-exported features — in this case `singleton_cell` — to "break" the singleton implementation given above (**Fig. 10** and **11**), whereas one could "break" the code defined previously (**Fig. 7**, and **8**) easily by looking only at the interface of the classes.

Besides, the use of the invariant

```
Current = singleton
```

is not fully satisfactory because it means that descendants of this class may not have their own direct instances without breaking this invariant.

> *Eiffel distinguishes between direct instances and instances of a type* `T`*, the latter including the direct instances of type* `T` *and those of any type conforming to* `T` *(i.e. its descendants) [Meyer92]. We think it should be the duty of the users of the Singleton library to decide when implementing a singleton whether there should be only one instance or only one direct instance of that type; it shouldn't be up to the authors of the library to decide.*

Finally, this code is **not** a library component: it is just an example implementing (or trying to implement) the singleton pattern.

Other tentative implementations

In a discussion in the comp.lang.eiffel newsgroup [Cohen01], Paul Cohen gives an interesting but somewhat overweight solution. The idea is that the singletons in system can register their instance by name in a registry. The *Design Patterns* book [Gamma95] calls it the "*registry of singletons*" approach (see 2. *Subclassing the Singleton class*, p 130). **Fig. 12** gives the corresponding Eiffel implementation:

```
class SINGLETON

feature {NONE} -- Initialization

   frozen register_in_system is
               -- Register an instance of this singleton.
               --|Must be called by every creation procedure
               --|of every descendants of SINGLETON
               --|to fulfill the class invariant is_singleton.
         require
               no_singleton_in_system:
                     not singletons_in_system.has (generating_type)
         do
               singletons_in_system.put (Current, generating_type)
         ensure
```

```
                count_increased: singletons_in_system.count =
                      old singletons_in_system.count + 1
                singleton_registered:
                      singletons_in_system.has (generating_type)
          end

   feature {NONE} -- Implementation

      frozen singletons_in_system: HASH_TABLE [SINGLETON, STRING] is
                -- All singletons in system
                -- stored by name of generating type
          once
                create Result.make (1)
          ensure
                singletons_in_system_not_void: Result /= Void
          end

   end
```

Fig. 12: A “registry of singletons”

Feature *generating_type* is defined in class *ANY*; it returns a string corresponding to the name of current object’s generating type (namely the type of which it is a direct instance). It is a feature of the Eiffel Library Kernel Standard (ELKS, see [Gobosoft] and appendix A of ETL3 [Meyer0?a]). However, class *HASH_TABLE* is not a standard Eiffel class (for example, SmartEiffel [Inria] does not define it).

Let’s write a descendant of class *SINGLETON* to understand how this “registry of singletons” works. A particular singleton implementation should look like the one shown in **Fig. 13**:

```
   class MY_SINGLETON

   inherit

      SINGLETON

   create

      make

   feature {NONE} -- Initialization

      make is
            -- Initialize singleton and
            -- add it to the registry of singletons.
            require
                  singleton_not_created:
                   not singletons_in_system.has (generating_type)
            do
                  -- Something here
                  register_in_system
```

```
            ensure
                  singleton_created:
                        singletons_in_system.has (generating_type)
            end
      ...
end
```

Fig. 13:A particular singleton in the “registry”

Each time a singleton gets created, it adds itself to the registry of singleton. The problem with this approach is that a client of *MY_SINGLETON* cannot test for the precondition of *make* before calling the routine: first, it does not have access to *singletons_in_system*; second, it does not know about the value of *generating_type* because the corresponding object has not been created yet.

Doug Pardee gives another possible implementation of the Singleton pattern in the Eiffel Forum [Pardee01], but it seems too complex to be reused effectively.

Impossible?

The unfruitful attempts reviewed so far illustrate how difficult it is to implement the singleton pattern in Eiffel, especially as a reusable library. In fact, it is not possible at all without violating the Design by Contract principles, namely a non-checkable invariant, even when controlling the creation of the singleton object because it can get involved in some cloning (*clone*/*deep_clone*) or in some persistence mechanisms (*store*/*retrieve* from *STORABLE*, or a database library).

Assuming we do not take clone and *STORABLE* into account, one solution could be to allow once creation procedures in Eiffel with a special semantics ensuring class correctness. That’s what we will review now.

3 ONCE CREATION PROCEDURES

We propose an extension to the Eiffel programming language that would allow declaring a creation procedure as a once-procedure — which is currently forbidden by the sixth clause of the “Creation Instruction rule”, [Meyer92] p 286. (This idea first appeared in the newsgroup comp.lang.eiffel in 2001 [Silva01].)

Rationale

Let’s consider a creation instruction with target *x*, a creation reference type *TC* and a creation procedure *make*:

```
x: TC
create x.make
```

The semantics of the Creation instruction ([Meyer92], p 289) for a reference creation type *TC* is as follows:

1. Allocate memory.

2. Initialize the fields with their default values.
3. Call the creation procedure *make* (to ensure the invariant).
4. Attach the resulting object to the creation target entity *x*.

This semantics forbids the use of once-procedures as creation procedures. Indeed, with a once procedure, the first object created would satisfy the class invariant (assuming the creation procedure is correct), but subsequent creation instructions would not execute the call, and hence would limit themselves to the default initializations, which might not ensure the invariant.

But we could think of another semantics for the "Creation_instruction" when the creation procedure is a once-procedure (namely a procedure declared as **once**):

If the once creation procedure has not been called yet to create an object of the given type *TC* then create an object as indicated above (steps 1 to 4).
Otherwise attach to the creation target entity *x* the object which has been created by the first call to the once creation procedure for this type.

This new semantics would make it possible to write a Singleton pattern in Eiffel (**Fig. 14** and **15**) and would also simplify the implementation of shared objects.

```
class SINGLETON

create

   make

feature {NONE} -- Initialization

   make is
         once
               ...
         end
end
```

Fig. 14: Class SINGLETON with once creation procedures

```
use_singleton is
         -- Declare two variables of type SINGLETON and
         -- check they point to the same object.
   local
         s1, s2: SINGLETON
   do
         create s1.make
         create s2.make

         check
               singleton_pattern: s1 = s2
         end
   end
```

Fig. 15: Feature using a singleton (declared with a once creation procedure)

Another possible application would be in the field of graphical user interfaces to display error messages in the same window:

```
(create {ERROR_WINDOW}).display (error_message)
```

default_create being declared as a once creation procedure.

For these examples to be valid, one should remove the sixth clause from the "Creation Instruction rule" ([Meyer92], p 286).

Open issues

Expanded creation type: It is not clear yet what should be the semantics when the creation type is expanded. Here is a possible solution:

If the once creation procedure has not been called yet in a creation instruction/expression of creation type *TC* then apply steps 3 and 4 described above to the object attached to *x*.
Otherwise do nothing.

Creation instruction and "onceness" status: Loosening the creation validity constraints ([Meyer92], p 286) to allow once creation procedures and applying the semantics described above would mean that a once-procedure has several "onceness" statuses (i.e. is it the first or a subsequent call?):

when it is called as a regular procedure
when it is called as a creation procedure (for each type for which it is declared as creation procedure).

Let's consider an example to better understand the issue:

```
class A

create

   make

feature

   make is
         once
               ...
         end

end
```

Fig. 16: Class A declaring a once creation procedure

```
class B

inherit

   A
```

```
create
   make
end
```

Fig. 17: Descendant class B inheriting once creation procedure from A

```
use_once_creation_procedures is
         -- Use once creation procedure from classes A and B.
   local
         a1, a2: A
         b1, b2: B
   do
         create a1.make
         create a2.make
         create b1.make
         create b2.make
   end
```

Fig. 18: Feature using once creation procedure from classes A and B

If it is clear that `a1` and `a2` should be attached to the same object, and likewise for `b1` and `b2`, however it is not the case of `a1` and `b1`, which have two different creation types and thus cannot be attached to the same object. Therefore the "onceness" status of `make` should be per creation type. But `make` can also be called as a regular procedure:

```
a1.make
b2.make
```

Should we take into account whether `make` has already been called as a creation procedure or not in that case? In our opinion, the "onceness" of a procedure should be different when used as a creation procedure and when used as a regular procedure. Indeed, even if the once-procedure has already been called as a regular procedure, we still want the initialization of the object to be made properly when this procedure is called for the first time as part of a creation instruction:

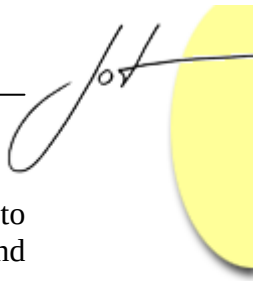
```
a1.make
create a2.make
```

Finally, we should probably combine all the above with the semantics of "once per thread", "once per process", etc. mentioned in section 8.6 of ETL3 [Meyer0?a].

Limitations

Coming back to our discussion about the Singleton pattern, once creation procedures in Eiffel still would not completely solve the issues described in section 2; in particular:

- We would still have the problem of not being able to forbid the duplication of the singleton object with `clone`/`deep_clone` or `STORABLE`/databases.
- We would not have the global access point to the singleton as demanded by the definition of a singleton in the *Design Patterns* book ([Gamma95], p 127) although we can provide a `SHARED_SINGLETON` access with a once function per `SINGLETON` class.



Another approach would be to extend the notion of **frozen** features ([Meyer92], p 63) to frozen classes as it already exists in Eiffel for .NET. We will now review the pros and cons of this solution.

4 FROZEN CLASSES

Eiffel: The Language ([Meyer92], p 63) defines the notion of **frozen** features, namely features that cannot be redefined in descendants (their declaration is final). By broadening the scope of final declarations from features to classes — as already done in the current implementation of Eiffel for .NET — it would become possible to implement a "real" singleton in Eiffel with a proper access point (as defined in [Gamma95], p 127) as a reusable component.

Rationale

Eiffel features whose declaration starts with the **frozen** keyword are final: they are not subject to redefinition in descendants. They are called "frozen features".

The idea is to extend this notion to classes. The semantics of "frozen classes" is that one may not inherit from these classes, which as a consequence cannot be deferred (because they cannot have any descendants and could never be effected).

The only syntactical change to the Eiffel language would be the introduction of the keyword **frozen** on classes. The *Header_mark* defined in section 4.8 of [Meyer92] (p 50) should be extended to:

```
Header_mark △= deferred | expanded | reference | separate | frozen
```

with the consequence that a class cannot be both frozen and deferred.

> *The keywords* **reference** *and* **separate** *do not appear in the first two versions of Eiffel; they are novelties of the third edition (see ETL3 [Meyer0?a], section 4.9, p 65).*

Singleton library using frozen classes

Having frozen classes would enable writing a "singleton library" relying on two classes:

- A frozen class *SHARED_SINGLETON* (**Fig. 19**) exposing a feature singleton, which is a once function returning an instance of type *SINGLETON*.
- A class *SINGLETON* (**Fig. 20**) whose creation procedure *make* is exported to class *SHARED_SINGLETON* and its descendants only.

```
frozen class SHARED_SINGLETON

feature -- Access

   singleton: SINGLETON is
                -- Global access point to singleton
```

```
            once
                  create Result
            ensure
                  singleton_not_void: Result /= Void
            end
end
```

Fig. 19: Frozen class SHARED_SINGLETON (global access point to SINGLETON)

```
class SINGLETON

create {SHARED_SINGLETON}
   default_create
end
```

Fig. 20: Class SINGLETON

Typical use of the "Singleton library" would be to create a *SHARED_SINGLETON* to get one's own unique instance, as in class *MY_SHARED_SINGLETON* written below (**Fig. 21**).

```
class MY_SHARED_SINGLETON

feature -- Access

   singleton: SINGLETON is
                  -- Unique instance
            once
                  Result := create {SHARED_SINGLETON}.singleton
            end
end
```

Fig. 21: Typical use of the "Singleton library"

Pros and cons of introducing frozen classes

Weak point:

The disadvantage of frozen classes is that it goes against the core principles of object-oriented development. Indeed, the *Open-Closed principle* ([Meyer97], p 57-61) states that a module should always be both *closed* (meaning usable by clients) and *open* (meaning it can be extended). Having frozen classes, which by definition cannot be redefined, violates this principle.

Strong points:

The main advantage of the last solution using frozen classes is that it provides a very straightforward way (introduction of just one keyword, **frozen**, with the appropriate semantics) to get a real singleton in Eiffel, including a global access point to it — which one could not have with the solution using once creation procedures.

Besides, there is no such problem as different once statuses depending on whether the same feature is called as a creation procedure or as a regular procedure.

On a lower level, having frozen classes would enable the compiler to perform code optimization, which it could not do for non-final classes.

5 CONCLUSION

This analysis has shown that implementing the *Singleton* pattern [Gamma95] as a reusable library in Eiffel is not feasible with the current definition of the language; an implementation like the *Gobo Eiffel* example [Gobo-Eiffel] is acceptable, but it is neither secure nor robust.

Among the two Eiffel language extensions suggested in this paper, the introduction of frozen classes is the most elegant and would lead to a straightforward way of writing "real" singletons in Eiffel (including a global access point). The main argument against authorizing frozen classes is that users may start using them excessively, which would violate the *Open-Closed principle* ([Meyer97], p 57-61); we believe it will not be the case. Indeed, Eiffel developers already have the possibility to declare features as "frozen" (meaning these features may not be redefined), but they use it only sparsely, in well-identified and justified cases. Besides, the utility of frozen classes is wider than just the implementation of the *Singleton* pattern; for example, it is already used in the .NET extension of the Eiffel language.

We think that extending Eiffel with frozen classes would provide an elegant way of writing real singletons in Eiffel. Nevertheless, it would not enable — at least we have not succeeded in — having reusable library components corresponding to the *Singleton* pattern.

REFERENCES

[Arnout02] Karine Arnout. "Contracts and Tests". *Ph.D. research plan*, ETH Zurich, December 2002. http://se.inf.ethz.ch/people/arnout/phd_research_plan.pdf.

[Arslan03] Volkan Arslan, Piotr Nienaltowski, and Karine Arnout. "Event library: an object-oriented library for event-driven design". *JMLC (Joint Modular Languages Conference)*, Klagenfurt, Austria, August 25-27, 2003. http://se.inf.ethz.ch/people/arslan/data/scoop/conferences/Event_Library_JMLC_2003_Arslan.pdf.

[Cohen01] Paul Cohen. "Re: Working singleton implementation". In newsgroup comp.lang.eiffel [online discussion group]. Cited 15 April 2001. http://groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&selm=3AD984B6.CCEC91AA%40enea.se&rnum=8.

[Eiffel] Eiffel Forum: *Singleton Pattern In Eiffel*. http://efsa.sourceforge.net/cgi-bin/view/Main/SingletonPatternInEiffel.

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides: *Design Patterns*. Addison-Wesley, 1995.

[Gobo] Gobo Eiffel Example: *gobo-eiffel/gobo/example/pattern/singleton*. http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/gobo-eiffel/gobo/example/pattern/singleton/.

[Gobosoft] Gobosoft: *The Eiffel Library Kernel Standard, 2001 Vintage*. http://www.gobosoft.com/eiffel/nice/elks01/index.html.

[Inria] INRIA, Loria: *Welcome to SmartEiffel! Home of the GNU Eiffel Compiler, Tools, and Libraries*. http://smarteiffel.loria.fr/.

[Jézéq99] Jean-Marc Jézéquel, Michel Train, and Christine Mingins: *Design Patterns and Contracts*. Addison-Wesley, 1999.

[JézéqErr] Jean-Marc Jézéquel: *Errata*. http://www.irisa.fr/prive/jezequel/DesignPatterns/#Errata.

[Meyer92] Bertrand Meyer: *Eiffel: The Language*. Prentice Hall, 1992.

[Meyer94] Bertrand Meyer: Reusable Software: The Base Object-Oriented Component Libraries. Prentice Hall, 1994.

[Meyer97] Bertrand Meyer: Object-Oriented Software Construction, second edition. Prentice Hall, 1997.

[Meyer0?a] Bertrand Meyer: *Eiffel: The Language*, Third edition (in preparation). http://www.inf.ethz.ch/personal/meyer/#Progress.

[Meyer03] Bertrand Meyer. "The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design". In *Festschrift in Honor of Ole-Johan Dahl*, Eds. Olaf Owe et al., Springer-Verlag, Lecture Notes in Computer Science 2635, 2003. Available from http://www.inf.ethz.ch/~meyer/publications/events/events.pdf. Accessed June 2003.

[Silva01] Miguel Oliveira e Silva. "Once creation procedures". In newsgroup comp.lang.eiffel [online discussion group]. Cited 7 September 2001. http://groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&threadm=GJnJzK.9v6%40ecf.utoronto.ca&prev=/groups%3Fdq%3D%26num%3D25%26hl%3Den%26lr%3D%26ie%3DUTF-8%26group%3Dcomp.lang.eiffel%26start%3D525.

[Pardee01] Doug Pardee. "Re: Once creation procedures". In newsgroup comp.lang.eiffel [online discussion group]. Cited 8 September 2001. http://groups.google.com/groups?dq=&hl=en&lr=&ie=UTF-8&threadm=GJnJzK.9v6%40ecf.utoronto.ca&prev=/groups%3Fdq%3D%26num%3D25%26hl%3Den%26lr%3D%26ie%3DUTF-8%26group%3Dcomp.lang.eiffel%26start%3D525.

[Vliss98] John Vlissides: *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, USA, 1998.

ACKNOWLEDGEMENTS

We gratefully acknowledge Bertrand Meyer (ETH Zurich & Eiffel Software Inc.) for his valuable comments and feedback on the paper. We are also thankful to Emmanuel Stapf (Eiffel Software Inc.), Mark Howard (Axa Rosenberg), and Dominique Colnet (LORIA), members of the standardization committee ECMA TC39-TG4 for fruitful discussions about once creation procedures and frozen classes.

About the authors

Karine Arnout is Ph.D. student in the Chair of Software Engineering held by Prof. Dr. Bertrand Meyer at ETH Zurich. Her Ph.D. topic deals with contracts and tests. She is a member of the ECMA working group standardizing Eiffel and worked at Eiffel Software on porting Eiffel to .NET. She can be reached at Karine.Arnout@inf.ethz.ch.

Éric Bezault is a senior engineer at Axa Rosenberg, research center, USA. He is the leader of the Gobo Eiffel Project, which provides free and portable Eiffel tools and libraries. He has several years experience on Eiffel projects in the financial industry. He is member of the ECMA group standardizing Eiffel. He can be reached at ericb@gobosoft.com.