Doc No: SC22/WG21/N1492
J16/03-0075
Date: 05-Apr-2003
Project: JTC1.22.32
Reply to: Daniel Gutson
danielgutson@hotmail.com

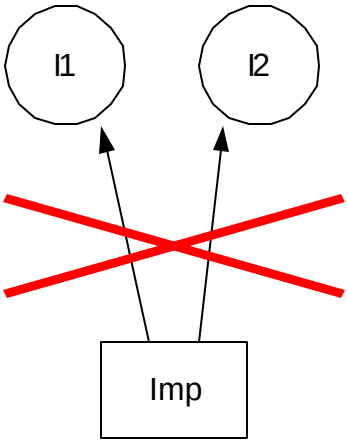
EXCLUSIVE INHERITANCE

1. The problem

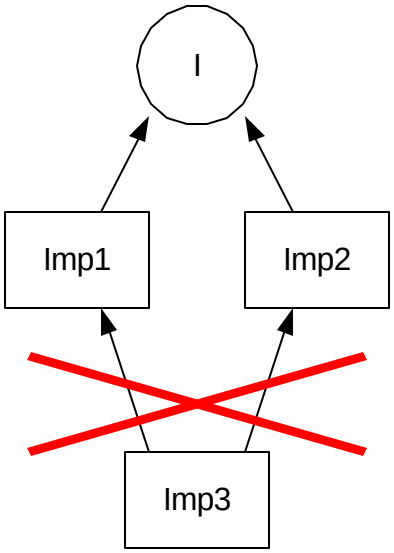
There is no way of prohibiting multiple inheritance from a given set of classes.
If two classes are –from design- 'mutually exclusive' in terms of inheritane –that is, no class can inherit multiply from both of them- additional documentation is required, and no static checking is performed –as far as there is no way of expliciting this design intention from the language-.

- Why is the problem important?

The following design situations may occur:
Situation a): Exclusiveness at the interface level.
A set of interfaces shall be implemented "exclusively", that is, no implementing class may implement both of them. (this is useful when visitors, etc)
Situation b): Exclusiveness at the implementation level:
A given interface shall be fully implemented by each implementor's branch.



Situation a): I1, I2: mutually-exclusive interfaces. Imp: Implementing class.



Situation b): I: interface; Imp1, Imp2: mutually-exclusive implementations; Imp3: implementations/subclass.

Situation a) may occur, for example, in a library API providing interfaces to be implemented by the user (such as listeners or visitors). In this kind of exclusiveness, an 'XOR' implementation relationship might be required. When a library component receives one of this mutually exclusive interfaces, it says "give me an implementation of this interface XOR this other interface".

Situation b) may occur when the implementation occurs 'sequentially' or 'linearly', that is, by a linear branch of subclasses. In other words, it forbids virtual inheritance.

There is no way of forcing such restrictions from the language.
If such restrictions come from design, the problem appears at the coding level.
This proposal focuses on the coding problem, rather than the design behind.

Semantic examples:

Let's consider that a being cannot be a human and an angel at the same time. Let's name this exclusiveness group 'beingEssence'.

Let's also suppose that an entity cannot be a C++ programmer and a Java programmer at the same time. Let's name this exclusiveness group 'programmerEssence'.

Let's denote

```
BeingEssence = { Human, Angel };
ProgrammeEssence = { CppPrg, JavaPrg };
```

Then,

```
Being1 : Human, CppPrg {}; // valid
Being2 : Human, Angel {};   // invalid
Being3 : Human, JavaPrg {}; // valid
```

(':' denotes 'inherits from')

-How are people working around or addressing this problem?
This problem can be solved by run-time checking, providing additional code and assertions.

-What are the consequences of not addressing this?
Additional documentation (and understanding) is required, and, optionally, additional code is also required (as mentioned above).

-Whom does it affect?
Library builders, legacy maintainers.

This feature fits in the following subset of categories:

* improve support for systems programming: provides a new attribute to inheritance relationships, making the code (as interface) closer to the design in self-documentation form. When one of the above situations occurs, additional code can be avoided as far as the static checking can be performed.
* improve support for library building: self-documentation of library interfaces, avoiding misuse and misunderstandings.
* remove embarrassments: there's no need of additional code and special mechanisms

2. The proposal

Allow an *exclusive* keyword at the class-level attribute, in order to define an 'exclusion group'. Classes deriving from such 'exclusive' class are mutually excluded of being subclassed.
The *exclusive* keyword proposed in this document is the 'exclusive' word itself.

2.1. Basic Example

```
exclusve struct BeingEssence {};
exclusive struct ProgrammerEssence {};

struct Angel : BeingEssence {};
struct Human : BeingEssence {};

struct CppPrg : ProgrammerEssence {};
struct JavaPrg : : ProgrammerEssence {};


class Entity1 : public CppPrg, protected Human
{ … };

class Faker : public Angel, private Human    //invalid
{ … };
```

2.2 Advanced Cases

(consider the same class and structure definitions of previous examples, 2.1)

```
class Person1 : Human
{ … };

class Person2 : Human
{ … };

class ComplexPerson : Person1, Person2 // invalid
{ … }:


class VPerson1 : virtual Human
{ … };

class VPerson2 : virtual Human
{ … };

class VComplexPerson : Person1, Person2 // still invalid
{ … }:
```

3. Interactions and Implementability

3.1. Interactions

- The ‘exclusive’ class attribute is transitive in terms of inheritance. If a class A1 inherits from an exclusive class A0, and a third A2 inherits from A1, A2 is still exclusive in terms of A0, [see discussion *] despite of the inheritance accessibility modifier (i.e. privately).
- Access modifiers do not affect exclusiveness (as seen in the example)

3.2. Implementability

- Exclusiveness should be handled at the compiler’s internal tables level, being transparent at run time.
- The compatibility break relies in the *exclusive* keyword selection only. As mentioned above, this proposal recommends the ‘exclusive’ word.

Discussion (*): why if A1 inherits privately from A0, A2 is still ‘A0’ exclusive, despite it is not a ‘A0’: the reason considered in this document relies in ‘security’: the exclusiveness mechanism could be ‘tricked’ through private inheritance, allowing a ‘far-descendent’ of the original classes to oversee the design purpose.