SMART COMPILING FOR EIFFEL

Miguel Katrib Mora, Miguel Mateo Carmona, Tomás Couso Alvarez

Department of Computer Science, University of Havana
email: mkm@matcom.uh.cu

Published in *Journal of Object Oriented Programming*, May 1998.

ABSTRACT

Contemporary programming environments for C++ introduce additional, extra-linguistic features, such as projects or make files, to organize the several interrelated program (compilation) units comprising an application. A conventional C++ compilation unit consists of two files: a header file and an implementation (source) file. However, current C++ compilers don't detect which modifications to one unit might require changes in other units; nor they try to avoid unnecessary recompilations. The pure OO language Eiffel uses a single file to hold the entire text of a class (i.e., no .h file). Moreover, Eiffel does not require any extra-linguistic file to define a project (i.e., no "make" file is mandated by the language). This paper analyzes and explains an optimization strategy to avoid unnecessary recompilations of classes comprising an Eiffel application.

1 INTRODUCTION

Two goals of dividing an application into different units or modules are to:

- facilitate localization of modifications; and,
- optimize application compiling time by avoiding recompiling a unit unnecessarily.

Some current Pascal compilers introduce linguistic features to express discrete compilation units (as **units** in Borland Pascal). Other languages, as C++, use extra-linguistic resources based on the division in header files (.H), source files (.CPP) and object files (.OBJ). In general, traditional compilers (mainly C++ and Pascal compilers), decide to compile (or not) a unit based on the *timestamp* of each source file (.PAS in Pascal, .CPP in C++) and its corresponding object file (.OBJ). For example, suppose an application (a project in C++ terminology) has units A and B compiled in a previous version, and B depends on A (in some way). To compile this application we will require the recompilation of unit B if unit A has a timestamp later than the timestamp of unit B (this means A was changed after the last compilation of B). One of the constraints of this approach is that it does not take into account the nature of the modification to A and whether a modification of such a nature really requires recompiling unit B.
The strategy discussed below tries to avoid unnecessary recompilations. It is being implemented in the Eiffel/UH[1] system, but can be applied to other Eiffel implementations. Section 2 explains the general strategy and section 3 analyzes various kinds of modifications and their implications for recompilation. In section 4 we discuss the general implementation. We compare our Eiffel strategy with C++ in section 5. Conclusions in section 6 contemplate the explained strategy with ISE Eiffel *melting ice*.

This strategy for Eiffel might be adapted to C++. However, two aspects of C++ make such an adaptation very difficult, viz: C++'s division of a compilation unit between header and implementation files (which C++ "inherited" from C) and, C++'s hybrid nature which does not guarantee that a C++ compilation unit will always correspond to a single class. We suspect that these two facts preclude a practical adaptation of a strategy such as ours to C++; a proof of this opinion goes beyond the scope of this paper.

2 A "SMART COMPILING" STRATEGY

In the Eiffel/UH system a class may evolve through 3 distinct states:

[1]An Eiffel system developed at the University of Havana

- An Eiffel class source code (in a file with ".E" extension).

An intermediate version called an Eiffel Unit (a file with ".EU" extension)

- A target file with native code (a Microsoft Windows ".OBJ" file)

An application comprised of one or more of these EUs can be executed by an interpreter. Alternatively, executable native code can be generated from these precompiled units. In this later case, the compiler generates a C file, then a C compiler generates the OBJ file. C files are only temporary; when the OBJ file is generated its corresponding C file is deleted. Hereafter, we will ignore .C files and only mention .E, .EU and .OBJ files.

The Eiffel/UH environment organizes these files in a "class data base". Programmers do not have to handle these EU and OBJ files. These files are only used by the compiler and the integrated environment.

In many C++ and Pascal systems, a benign change to our class A, such as inserting a new line or a space (to improve readability), usually precipitates the recompilation of class A and all of its dependents (clients or heirs). The goal of *smart compiling* is avoiding unnecessary recompilations. In the next section we will analyze which modifications imply a potential impact upon their dependent classes requiring recompilation of such dependents whenever a dependent is used.

In *Table 1* below, each row indicates a possible modification in the source text of class A. The columns show the side effects that these modifications can produce in the source text of the direct heirs and clients; and in the respective EU and OBJ files. An aster in the *i,j* cell indicates which that modification in row *i* has a side effect on the column *j*.
Most of these modifications can precipitate potential errors in the source text of its heirs and clients; for brevity, we say that the dependent's source or OBJ *"becomes wrong"*. To determine if a modification actually causes a dependent class to "become wrong", we have to analyze the source text of the dependent class; however, such an analysis may be unnecessary or premature if the dependent class is not a component of the application we are compiling at the present moment. In our strategy, when a modification is detected, we merely "*mark*" the potentially affected clients or heirs. Then, only a further reuse of these *marked* classes precipitates their recompilation.
For any modification to class A, we determine the kind of modification (i.e. which row in Table 1 of enumerated kinds of modifications) by comparing the new EU with the former EU. Subsequently, any dependent class B may be "marked", forcing its recompilation when any application uses it (often, though not necessarily, the same application that made the change in class A).[2]

If a change in A doesn't match any of the rows in the Table (i.e., it is a benign modification), or if the corresponding B (client or heir of the class A) column cell is empty, then it will not be necessary to recompile B; notwithstanding B's dependency on A and its involvement in a compilation of an application later than class A's (its .E file) timestamp. So: *smart compiling*

3 MODIFICATIONS OF A CLASS AFFECTING OTHER CLASSES

This section explains possible cases of modifications to a class that should precipitate modifications or require recompilations of other classes. First let's begin with some general notations and conventions:

Notation

> Every class beginning with a given letter is in the same inheritance hierarchy. Thus, classes A1 and A2 inherit from class A, classes A1_1 and A1_2 inherit from class A1 and so on.
>
> Ellipsis (…) indicate any segment of Eiffel code that is not important to understand the example.

1: The source text of a direct heir "becomes wrong"
An asterisk (*) in this column indicates that, as a consequence of a modification (indicated in the row) to class A, any A1, direct heir of A, may become wrong, therefore its EU will be marked. Thus, when the class A1 is used in any application, it should be recompiled.

[2]Of course, the same programmer could make the necessary changes in the source text of class B and then the B *timestamp* will also change, forcing its further recompilation.

It's important to note that indirect heirs that might become wrong will not be marked directly; rather, these will be marked by *propagation.* After a recompilation of A1, its new EU will embed the original modification done in A. Then, the new EU of A1 will differ from the previous one, which in turn, will be detected as a modification to A1, propagating the effect to its clients and direct heirs.

An EU embeds the information of each direct ancestor (which at the same time embeds the information of its ancestors). This policy could result in space overload, but simplifies the implementation of the propagation algorithm; moreover, it speeds up the compilation because the compiler will not need to traverse the inheritance hierarchy.

2 The source text of a direct client is wrong

An asterisk (*) in the second column means that after a modification in a class A, a class B, direct client of A, may become wrong, and accordingly, the EU of B will be marked. Thus, when class B is used further in an application, it will be recompiled.

Note: A class B is a direct client of class A if it has or uses some features of type A. Since this can depend not only on the export rules of A, but also on the code of B, then every time that a class B is compiled, the classes of which B is a client have to be updated.

3, 4 The OBJ of the heir or client becomes wrong.

Obviously, if it was necessary to make changes in the source text of class B, then we have to generate a new EU and OBJ files.[3]

An asterisk (*) in the OBJ cell of a heir or client means that the compiler must generate the OBJ again, even if the source code did not suffer modifications. This is because the *class object model* may change.

Two important features are in the object model:

1. the size of an instance of the class
2. the quantity, type and distribution of the features in the *virtual method table* (VMT)

A change in a class A may generate changes in the above features of a heir or client OBJ, even if its source text did not have modifications.

For example, if an attribute is added to a class A, then the source of a client class B doesn't suffer any modification. However, a client object can change in size if B has an expanded attribute of type A. The object's size of an heir class A1 will also change. Otherwise, in the OBJ of a client B a reference to an A's feature can change, because these features can change their offsets in the VMT of A. Thus, the heir's and client's OBJ will be marked to be regenerated.

A detailed object model discussion goes beyond the scope of this work [5]. For brevity, in most cases we have omitted the explanation of cells in columns 3 and 4 because they are similar to the explanation in the above paragraph.

5 Modification applicable to C++

This column indicates which modification has an equivalence in C++.

	1 Wrong Heir Source	2 Wrong Client Source	3 Wrong Heir OBJ	4 Wrong Client OBJ	5 Modification applicable to C++
1-Change a class from normal to **deferred**	*	*			
2-Change a class from normal to **expanded**		*		*	
3-Change a class from **deferred** to normal	*				

[3]Remember that OBJ is necessary only when we will generate an standalone application, i.e. running out the control of the interpreter.

4-Change a class from **deferred** to **expanded**	*	*		*	
5-Change a class from **expanded** to normal		see 5.2		*	
6-Change a class from **expanded** to **deferred**	*			*	
7-Make a feature **deferred**	*				**C++**
8-Remove the **deferred** specification of a feature	*				**C++**
9-Change the number of generic parameters	*	*			**C++**
10-Change the constraint type of a generic parameter	*	*			
11-Add a parent to the **inherit** clause	*	*	*	*	**C++**
12-Remove a parent from the **inherit** clause	*	*	*	*	**C++**
13-Add, remove or change a **rename** clause	*	*			
14-Redefine a feature	*		*		**C++**
15-Delete a feature redefinition	*		*		**C++**
16-Select a feature	*		*		
17-Remove the selection of a feature	*		*		
18-Remove a routine from the creation list		*			**C++**
19-Add a routine to the creation list		*			**C++**
20-Add a feature	*		*	*	**C++**
21-Delete a feature	*	*	*	*	**C++**
22-Change the export clause of a feature	*	*			Only if ANY to NONE or NONE to ANY
23-Change the number of routine parameters	*	*			**C++**
24-Change a parameter type	*	*		*	**C++**
25-Change an attribute type	*	*	*	*	**C++**
26-Change a function type	*	*		*	**C++**
27-Change a procedure to function or vice versa	*	*			**C++**
28-Insert a **require** precondition	*		*		
29-Insert an **ensure** postcondition	*		*		
30-Change function into constant or into attribute	*		*	*	**C++**
31-Change attribute into function or constant	*		*	*	**C++**
32-Change constant into **unique** constant		*	*	*	
33-Change constant to function	*	*	*	*	**C++**
34-Change constant value	*	*	*	*	**C++**
35-Switch between normal, **once** or **external** specification					
36-Change an **iterator** routine to normal, **once** or **external** or vice versa	*	*			
37-Define an existing feature as frozen	*				

Table 1 Modifications applied to a class that can affect other classes

In the following example, cells on the left-hand show the original class texts, cells on the right-hand show the same classes after modification. Middle cells explain the changes and the outcomes.

1 Change a class from normal to **deferred**

1.1 For a class A to become **deferred**, one or more of its feature must be declared **deferred**. When a class becomes **deferred** a heir becomes wrong if it redefines the feature that is now **deferred**. In this case, the **redefine** clause becomes wrong. Now the former redefinition feature must be an effecting feature.

class A ... f **is do** ... **end**; ... **end**	The class is made **deferred** because some feature was made **deferred**	**deferred class** A ... f **is deferred**; ... **end**

class A1 **inherit** A **redefine** f ... f **is do ... end**; ... **end**	f becomes a deferred feature in the parent class, then it could be effected in a heir class (but not redefined)	

1.2 The source text of a client *B* becomes wrong because it would try to create an instance of an abstract class.

class A ... g **is do** ... **end**; ... **end**	The class is now **deferred** (some feature was **deferred**)	**deferred class** A ... g **is deferred;** ... **end**
class B ... a:A; f **is do** !!a; ... **end**; ... **end**	The client can't create an instance of a **deferred** class	

2. Change a class from normal to **expanded**

2.2 A client becomes wrong if it explicitly creates an instance of the class that is now expanded, or if it assigned an object of an heir type to an entity of the class type.

class A ... **end;**	The class changes to **expanded**	**expanded class** A ... **end**
class B ... a:A; a1:A1; ... !!a; ... a:=a1; ... **end**	An object with a heir type cannot be assigned to an **expanded** attribute.	

3. Change a class from **deferred** to normal

3.1 A heir becomes wrong if it makes effective a **deferred** feature. In such a case, it must remove its own definition or put the feature's name in a **redefine** clause.

deferred class A ... g **is deferred**; ... **end**	class A is now normal and feature g is made effective.	**class** A ... g **is do ... end**; ... **end**
class A1 **inherit** A ... g **is do end**; ... **end**	The feature that was effective in A1 must be deleted or be included in a redefine clause.	**class** A1 **inherit** A **redefine** g ... g **is do end**; ... **end**

4. Change a class from **deferred** to **expanded.**

4.1 The source of a heir can become wrong for the same reasons cited in 3.1

4.2 Same reason cited in the second choice of 2.2.

5. Change a class from **expanded** to normal

5.2 A client would be affected if the original class has a creation routine without parameters. In that case a client of the expanded class can implicitly apply this creation routine when it creates an

instance. Then, if the routine remains in the creation list, the client must explicitly call some creation routine when trying to create that instance.
5.4 Only the object model in the OBJ of the client becomes wrong because it could have an attribute of the modified expanded type.

6. Change a class from **expanded** to **deferred**
6.1 Similar to 1.1

7. Make a feature **deferred**
This modification can be associated with a modification like (1) or (6). It can also occur independently, in the case of a class which was already **deferred**.
7.1 This example is similar to 1.1

8. Remove the **deferred** specification of a feature
This modification can be accomplished with (3) or (4), but it is not necessary if the class has other features declared as **deferred**.
8.1 A heir can become wrong for the same reasons cited in 3.1.

9. Change the number of formal generic parameters
9.1, 9.2 The source text of a heir or client becomes wrong.

10. Change the constraint type for a formal generic parameter
10.1, 10.2 A client or heir might initialize a formal parameter with a type that does not conform to the new constraint type parameter.

class A[T->C1] ... **end**	the constrain***t*** type C1 is changed by a new type B	**class** A[T->B] ... **end**
class D ... a :A[C1_1]; ... **end**	becomes wrong because C1_1 does not conform to B	

11. Add a parent to the **inherit** clause
11.1 A heir becomes wrong because cycles of repeated inheritance might appear. The new conflict has to be solved using **select** clauses and renaming.
11.2 The class added as a new parent might have an expanded attribute with the type of the class that added it as an ancestor, then an infinite expansion arises.

class A1 **inherit** A ... **end**	Adding B in the inheritance list	**class** A1 **inherit** A; B ... **end**
class B ... a1:**expanded** A1; f **is do** ... **end**; ... **end**	The class B as a client of A1 make***s*** an infinite cycle of expansion of attribute a1	
class A1_1 **inherit** A1; B **redefine** f ... **end**	The change in A1 left class A1_1 inheriting repetitively from B. A conflict arises with f and must be solved using a **select**	

12. Remove a parent from the **inherit** clause
12.1 A heir might be redefining a feature that has been introduced in the removed parent
12.2 The clients are affected if they were using a feature belonging to the removed class.

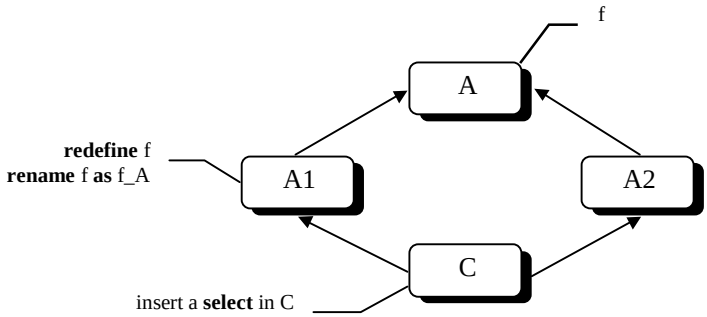
13. Add or eliminate a renaming clause; or, insert a modification in an existing one

13.1 A heir might be redefining a feature that now has changed its name
13.2 The clients could be using a feature that now has another name.

class A1 **inherit** A ... **end**	The name of feature f inherited from A is changed	**class** A1 **inherit** A **rename** f **as** h ... **end**
class B ... a1:A1; ... a1.f; ... **end**	This reference to f becomes wrong because it does not exists with this name for the clients of A1. The new name of f is h	

14. Redefine a feature

14.1 In a heir the redefined feature might be involved in a graph of repeated inheritance. Then a new conflict may arise and must be solved (possibly using a select clause).

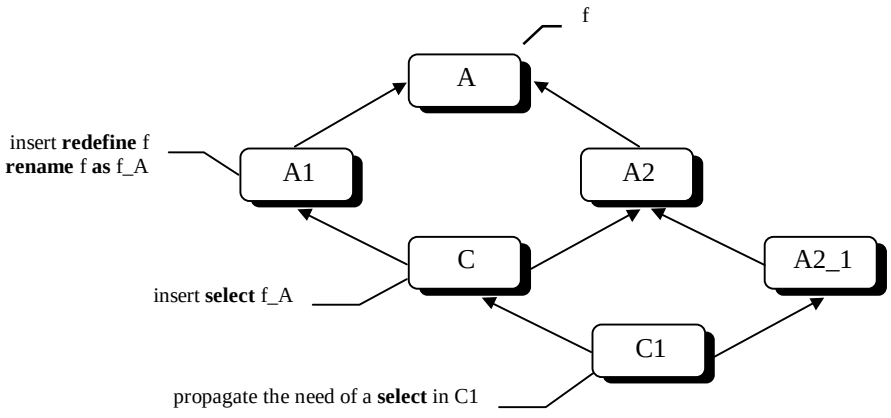


15. Delete a feature redefinition

15.1 The redefinition might be involved in a graph of repeated inheritance. Then, for reciprocal reasons cited in 14.1, an idle **select** clause could arise and must be removed.

16. Introduce the selection of a feature (for example, as a result of its having been introduced in a redefinition in a parent)

16.1 The need for a selection in a heir could be propagated to grandchild. Suppose the same example in section 14.1, the inclusion of a **select** clause in *C* could propagate the need for a **select** clause in a***n*** heir of *C*.



17. Remove the selection of a feature

17.1 Could propagate the need to remove a feature from a heir's **select** clause. Suppose the same repeated inheritance graph 16.1, if removing the **select** f of class *C* (as a consequence of removing the **redefine** *f* in the class *A1* parent of *C*) will imply removing the selection in *C1.*

18. Remove a routine from the **creation** clause (removing the whole selection clause is equivalent to eliminating all the routines in that clause).

18.2 A client could be creating an object using a routine that now is not a creation routine

class A **creation** make, make1 ... **end**	Routine make is eliminated from the creation clause	**class** A **creation** make1 ... **end**
class B ... a :A; ... !!a.make; ... **end**	The instruction !!a.make is wrong because make is not a creation method	

19. Add a creation routine

19.2 If the class didn't have a creation routine, a client class could create an instance without using a creation routine. This creation becomes wrong.

If the class had only a single creation routine without parameters, some **expanded** instances of a class might be implicitly created. Those creations would be incorrect.

class A ... **end**	A **creation** is introduced	**class** A **creation** create ... **end**
class B ... a :A; ... !!a; ... **end**	Now the instruction !!a is wrong because it need***s*** to use the create routine.	

20. Add a feature (routine, attribute or constant)

20.1 An heir could be affected if it's using the same name of this new feature

class A ... **end**	The feature f is added to the class	**class** A ... f **is** ... **end**
class A1 **inherit** A ... f **is** ... **end**	The definition of f inside A1 becomes wrong. It must be deleted or a redefinition must be inserted.	

21. Delete a feature in a class

21.1 An heir might be redefining the deleted feature

21.2 A client might be using the deleted feature.

22. Change the export clause of a feature (changes the visibility of the feature)

22.1 An heir might be using the feature in a routine precondition, and this routine maybe exported to a class that can't use the feature. Thus, the client class of the routine can't do an *a priori* check of the precondition.

Class A ... **feature** x :INTEGER; ... **end**	Feature x is now exported only to class C	**class** A ... **feature** {C} x :INTEGER; ... **end**
class A1 **inherit** A ... f **require** x>1 ... **end**	Becomes wrong because routine f might be used for a class other than C. Now this class can't check the precondition in an a priori approach	

22.2 Could have been effected in old clients of the previously exported feature. Note that the feature is not removed from the class (unlike the case 21.4), accordingly, the OBJ remains unchanged.

Class A ... **feature** f ... **end**	Now f is exported just to B and C	**class** A ... **feature** {B, C} f ... **end**
class D ... a :A; ...a.f; ... **end**	D was a client of the feature f of A but after the change the feature f is not exported to A	

23. Change the number of formal parameters in a routine signature.
    23.1 An heir might be redefining the routine with a wrong number of actual parameters.
    23.2 A client might be using the routine.

24. Change the type of the formal parameter in the definition of a routine to a type not a super type
    24.1 An heir might be redefining the routine with no conforming parameters
    24.2 A client might be using the routine with a no conforming parameter.
    24.4 If we change a reference type to another reference type then the client's OBJ doesn't change. In the other cases (an expanded type is involved) the client's OBJ must be generated again.

25. Change an attribute type
    25.1 An heir might be redefining the attribute with a no conforming type
    25.2 A client would become wrong if it is using (through this attribute) a feature that no longer exists or is not exported. Also, a heir might make an assignment that becomes wrong because of non-conformance.
    Note: If the attribute is private (exported to NONE) then clients are not affected.
    25.4 As 24.4

26. Change a function type
    26.1 An heir might be redefining the function with a no conforming type
    26.2 A client or heir would become wrong if it is using (through the result of the function) a feature that no longer exists or which has other export rules in the new type. Also the function result might be assigned to an entity *x*, but now the new function type does not conform to *x*.
    26.4 As 24.4

27 Change a procedure to a function or vice versa
    27.1 An heir might be redefining the routine in a wrong way
    27.2 A client might use the routine where such use is no longer valid.

28. Insert or remove a **require** precondition clause in a routine

28.1 Upon inserting a require precondition, heir becomes wrong if it also has a **require** precondition in the same routine. The heir precondition clause should be deleted, or replaced by a **require else**. Conversely, if a routine's **require** precondition is removed, then any heir that has a **require else** precondition must delete it or transform it into a **require** precondition.

class A ... f **is do** ... **end**; ... **end**	A precondition to f is introduced	**class** A ... f **is** **require** ... **do**...**end** ... **end**
class A1 **inherit** A **redefine** f ... f **is** **require** **do** ... **end** ... **end**	This **require** should be deleted or changed into a **require else**	**class** A1 **inherit** A **redefine** f ... f **is** **require else** **do** ... **end** ... **end**

29. Insert or remove an **ensure** postcondition clause in a routine.
    29.1 Analogous to 28.1

30. Change a function into a constant or an attribute
    30.1 An heir might redefine the function into another function, but now cannot redefine a constant or an attribute

class A ... f :INTEGER **is do** ... **end**; ... **end**	is changed to a constant	**class** A ... f :INTEGER **is** 10; ... **end**
class A1 **inherit** A **redefine** f ... **end**	becomes wrong because the constant f can't be redefined	

31. Change an attribute into a function or constant
    31.1 A heir object might be assigning a value to the attribute. The client source is OK because it could not be assigning any value to the attribute (due to the *read only* Eiffel rule).

32. Change a constant into a **unique** constant or vice versa.
    32.2 A client could be affected if the constant was used in a multiple selection. Eiffel requires that if a **unique** constant is used in a multiple selection then all the others constants in the selection should also be **unique**.

class A ... cte :INTEGER **is** 3; ... **end**	is changed by	**class** A ... cte :INTEGER **is** **unique**; ... **end**
class A1 **inherit** A ... **inspect** exp **when** 1 **then** ... **when** cte **then** ... **end** ... **end**	Wrong because cte is now **unique** and constant 1 is not **unique**	

33. Change a constant into a function

33.1, 33.2 An heir or a client becomes wrong if it is using the constant in a multiple selection. A function cannot be used in a multiple selection clause.

34. Change the constant value
    34.1, 34.2 The use of the constant in a multiple selection in a heir or in a client may become ambiguous.

35. Any change in a routine among normal, **once** and **external**
    The generated code for the routine changes, whereas any client or heir is still using the routine in the old way.
    35.3 The EU of a heir is changed only for documentation.

36. Change a normal, **once** or **external** routine into an *iterator* routine (**iterator**)[4] or vice versa
    36.1, 36.2 The use of the iterator in a **for all** statement becomes wrong

37. Define an existing feature as frozen
    37.1 A heir might redefine the feature which is not permitted for a frozen feature. To remain as an heir the heir class must remove the redefinition.

A client class doesn't change. Nevertheless, the compiler might generate static binding when a user calls a frozen feature, if such is the case, the *Wrong Client OBJ* cell must be marked to force generation of a more efficient class.

4 THE IMPLEMENTATION

Most of the above cases are only potential reasons for further modifications. The need for a modification depends not only on the kind of effected change itself, but also on the client or heir source. This fact suggests that we analyze the text of a client or heir source to determine if it really is affected by a modification to a class upon which it depends; however, such an analysis is usually more expensive than assuming the worst case and recompiling the dependent class. That is why our algorithm only "*marks*" the classes that are "*potentially wrong*". This forces a further recompilation only when the compiler needs the class.

Change detection is implemented by comparing the EU before compilation with the EU after compilation. In the current Eiffel/UH implementation an EU is a *persistent object*.[5] All the information about a compiled class is included in the EU, therefore there is no need to inspect additional files to perform this smart compilation. Changes are detected solely by inspecting these EU objects.

Table 1 serves to detail the meaning of each modification. Inside the actual implementation, an algorithm can do other sophisticated optimizations. For example, the lines 11 (*add a parent*) and 12 (*remove a parent*) each generate potential problems (albeit for different reasons) for clients or heirs. From the algorithm's viewpoint this difference is not important. Therefore, it's sufficient to detect any change in the ancestors list in each EU.

Another optimization would be to sequence the search for each kind of modification to best advantage. By searching first for modifications with a high probability of incidence we may be able to avoid searching for modifications with lower probabilities of incidence. By searching first for modifications requiring little time to detect, we may avoid searching for modifications requiring relatively more time to detect, etc.[6]

The discussion to this point implies that the marking process flows solely from *A* to *B* (supplier to client). It's important to understand that there is also a flow of marking from *B* to *A* (client to supplier). Class *B* is identified as a client of class *A* when *B* is compiled (if *B* really uses *A*). Then, after *B*'s compilation, the relationship *B is-a-client-of A* is stored in *B*'s EU; and, it's also stored in *A*'s EU. So, when we say that the algorithm "marks" *A*'s client class *B*, the client class *B*'s identity is also stored in *A*'s EU.

[4]The feature *iterator* is an extension of Eiffel/UH to the Eiffel language. The reader should refer to [3] for more information.
[5] For the present an EU is implemented as a C++ persistent object, it will be translated to an Eiffel/UH persistent object

[6]From the programming environment the order of the Table rows could be customized, tuning smart compilation to our convenience.

The client-supplier relation ship between a supplier class *A* and a client class *B* could be more specific if we identify which features of *A* are used in *B*. If a class *B* is a client of a class *A* because it uses feature *f* of *A* but does not use *A*'s feature *g*, then a change in *A* involving only *g* will not affect the class *B*. For simplicity, we didn't consider these cases in Table 1 but they are detected in the implementation being developed.

5 COMPARATION WITH C++

In a disciplined OOP in C++ for each class we must have a header text file (.H) as well as a source text file (.CPP) containing the implementation of the member functions of the class. An extra-linguistic mechanism, called a "make" file identifies the .H and .CPP files comprising a "project" (application).

In Eiffel there is no analogue to C++'s .H file. The source text file (.E) contains the sole definition of each class. Eiffel's "ace" file is also an extra-linguistic mechanism analogous to C++ make file in that it identifies the classes which comprise an application; however, the programmer does not have to identify any inter-class dependencies in the Eiffel ace file.

In Eiffel, the definition of *dependent classes* is a compiler responsibility. It's the compiler which looks for classes that are needed to compile a given class. In C++ it's the programmer who is obligated to identify these dependencies (through **#include** file sequences). This is one of the nightmares of C++ novice programmers: redundancies, unnecessary dependencies and cycles in the links arise.

The following examples show classes *A, B* and *C* in Eiffel and C++

class A ... **end**	**class** B ...a:A; --*B is client of A* ... **end**	**class** C ...b:B; --*C is client of B* ...—`nothing` *in C makes* *--reference to b.a, so* *-- C is not a client of A* **end**	**class** A ...**redefine** f ... **end**

`class A {`
`...`
`}`
	`#include "A.H"`
	`class B {`
	`A a;`
	`...`
	`}`
		`#include "B.H"`
		`class C {`
		`B b;`
		`...`
		`/* A.H will always be compiled */`
		`}`
			`class A {`
			`...f();`
			`...`
			`}`

In Eiffel, class *B* is direct client of *A, C* is direct client of *B*, but *C* is not a direct client of *A*. However, in C++ the source of class *C* is *dependent* on the header source of class *A* because it has the line `#include "B.H"` which in turn, has a `#include "A.H"` line.
In Eiffel a modification in *A* may cause a modification in *B* (according to Table 1 above). Thus, certain kinds of modifications to class *A* may call for recompiling class *B*; however, these kinds of modifications do *not necessarily* propagate to recompile class *C*. In contrast, any change to class *A*'s C++ header file always precipitates recompilation of *B* and *C.*
As another example, consider row 14 of Table 1. Redefining class *A*'s feature *f* does not require recompilation of client class *B* (nor of class *C* of course). Unfortunately, such a modification to class *A* in C++ requires a change in *A:H* which precipitates the recompilation of *B* and *C*.

We found a similar situation with modifications that can affect the heirs of a class. As shown in Table 1, in Eiffel/UH only direct heirs are recompiled and, if actual changes are not, in fact, needed, then recompilations are not propagated to indirect heirs. In contrast, in C++, an indirect heir is also dependent on .H header file of the indirect class' ancestor base class; consequently, any indirect heir must be recompiled whenever a grandparent's header file changes.

In Eiffel/UH a modification such as line 18 of Table 1 does not generate recompilation of direct heirs. (A similar analysis could be considered for line 19). Adding the name of a routine (that already exists) to a creation clause, does not require recompilation of the heirs. However, an equivalent change in C++ demands, at least, changing the name of the routine into the name of the class (to make it a constructor). This implies a

change in the .H and then compilation of *all* the heirs (direct or not). Such inefficiencies don't have to exist in a smart compilation for Eiffel.

6 CONCLUSIONS

Smart compilation in Eiffel avoids recompilation of classes in cases where most C++ environments force unnecessary recompilation. (Eiffel's richer semantics require that we address additional cases which are not applicable in C++; however, the cost of potential recompilation in these additional cases must be considered in light of the benefits provided by the richer semantics.)

We are implementing smart compilation in our academic version of Eiffel/UH. Three commercial Eiffel implementations are available: ISE Eiffel; Tower Eiffel; and, SIG's Eiffel/S. So far as we know, neither the Tower nor the SiG implementations attempt to avoid unnecessary recompilation. ISE's novel *melting ice* [4] scheme tries to address unnecessary recompilation.

Regrettably, the limited published documentation of ISE's *compilation degree* notion precluded a more formal comparison with our smart compiling strategy. Preliminary empirical comparisons with Personal ISE Eiffel for Windows are shown in Table 2. In *Smart Compiling*, an * means to recompile the source, or to regenerate the OBJ from the EU, whereas, in ISE Eiffel, an * means that the compiler repeats compilation degrees for that class. In rows appearing in Table 1 but omitted in Table 2, both approaches seem to have similar behavior.

	Smart Compiling			*ISE Eiffel Melting Ice*	
Nature of the modification to an original class	Heir Source	Client Source	Client OBJ	Heir Source	Client Source
2-Change a class from normal to **expanded**		*		*	*
3-Change a class from **deferred** to normal	*			*	*
5-Change a class from **expanded** to normal		*		*	*
6-Change a class from **expanded** to **deferred**	*		*	*	*
18-Remove a routine from the creation list		*		Recompiles direct heirs	*
19-Add a routine to the creation list		*			
29-Insert an **ensure** postcondition	*		*	*	*
30-Change function into constant or attribute	*		*	*	*

Table 2 Comparison between *Smart Compiling* and *ISE Eiffel Melting Ice*

To avoid unnecessary recompilations Personal ISE Eiffel for Windows *melting ice* saves all the internal information about the compiling process of a project [7] in a web of files referred as *eifgen*. Unfortunately, the eifgen file seems to keep the track of valuable status information only from the last compilation of a project. To illustrate, suppose you compile a project including a client class *Z* and its suppliers *X* and *Y*.. Now, in a second compilation of the project suppose you do some changes in class *Z* excluding class *X* as *Z*'s supplier; in this second compilation, neither class *X* (not being used) nor class *Y* (compiled in the previous compilation) will be recompiled. Finally, suppose we recompile a third time the project to accommodate another change to *Z* which resumes using *X*. This third recompilation will recompile *X* unnecessarily in the ISE melting ice scheme.

So far we are aware, all current Eiffel compilers preclude the independent compilation of individual classes. Any compilation of a class must be invoked through a project and its root class.

We believe that the foregoing limitations undermine the *component culture* that Object Oriented Programming tries to encourage.

[7] Remember that an Eiffel project is defined from the root class in the Ace file.

The rich Eiffel syntactic and semantic model doesn't allow a compiler to work in a single one-pass mode. Consequently, an intermediate representation of classes (such as our EU) is a valuable aid to optimizing compilation. The kind of information encapsulated in a single EU is persistent and it doesn't depend on projects; rather, it is dependent only on relations between classes.

The modifications analyzed in the present paper can be encapsulated in the EUs where they can support our smart compiling strategy. Furthermore, the EU provide a good platform for developing other algorithms such as detecting system-level type failures ([6], [7]). Also, using the EU information may be incorporated into documentation and browsing tools (such as Eiffel's short or flat class interface representations or others). Unlike other Eiffel environments, this can be done without the presence of the source code. Therefore, true encapsulation and information hiding might be achieved.

The present strategy would be more complex if we include the notion of class version. This an open problem to be researched.

Acknowledgments
This paper was prepared with the support of DGICYT, Education and Science Ministry of Spain, SAB 95-0038. We thank: Yania Crespo (University of Havana) for her detailed revision of Table 1; Ernesto Barreras (University of Havana) for his analysis of several cases in ISE's Personal Eiffel for Windows; Ray Fernández (Eiffel Software S.L., San Sebastián, Spain), and Ernesto Pimentel (University of Málaga, Spain) for their comments and suggestions. The Alma Mater grant of the University of Havana supported the practical proofs of this work.

REFERENCES

[1] Meyer Bertrand *Object Oriented Software Construction*, Prentice Hall 1988
[2] Meyer Bertrand *Eiffel: The Language*, Prentice Hall 1992
[3] Katrib M, Martínez I, *Collections and Iterators in Eiffel*, Journal of Object Oriented Programming, Nov-Dec 1993.
[4] Meyer Bertrand, *Eiffel the Environment*, Prentice Hall 1994
[5] Couso T, Katrib M, *Object Model for Eiffel/UH*, Internal Report. Department of Computer Science, University of Havana, 1993
[6] Meyer Bertrand, *Static typing for Eiffel*, An Eiffel Collection ISE 1989
[7] Crespo Yania, Katrib Miguel *New system-level type errors in Eiffel*, (to be published)