FLATTENING INHERITANCE STRUCTURES
- OR -
GETTING THE RIGHT PICTURE OF LARGE OO-SYSTEMS

Dirk Beyer, Claus Lewerentz and Frank Simon

Software Systems Engineering Research Group
Technical University Cottbus, Germany
Computer Science Reports I-12/2000, November 2000

E-mail: (db|cl|simon)@informatik.tu-cottbus.de

Abstract: More and more software systems are developed using the object oriented paradigm. Thus, large systems contain inheritance structures to provide a flexible and re-usable design and to allow for polymorphic method calls. This paper gives a detailed overview about the impact of using inheritance on measuring, understanding and using subclasses in such class systems. Usually, considering classes within an inheritance relation is reduced to the consideration of locally defined members of a class. This view might be incomplete or even misleading in some use cases. To provide an additional view on a given system we define a tool-supported *flattening process* which transforms an inheritance structure to a representation in which all the inherited members are explicit in each subclass. This representation provides additional insights for measuring, understanding, and developing large software systems.

1 Introduction

Analysing, understanding and using large object oriented systems are very difficult tasks although there are tools available that help to navigate through class systems for this purpose. Especially for systems with extensive use of inheritance structures it is not easy to determine e.g. which methods are provided by a particular class due to the inherited members. Usually, tools display only those members of a class which are declared in that class, not what the class has inherited from superclasses. Thus, what you see is *not* what you *really have* !
This one view on a system might cause the following problems:

- A measurement process can consider only the members declared within the class. Because inherited members are not considered the measurement values can lead to wrong interpretations.
- The analysis and understanding phase of a reverse engineering process for a large object oriented software system is very time-consuming for determining a class' functionality.
- For using a class of a library it is essential to get an overview about the set of usable members. In the design and the implementation phase of a forward engineering process the user of a class has no fast access to the really accessible members of the class.

Our work presents an approach to create a flat version of a class system, i.e. we eliminate the inheritance structure by copying inherited members down to the subclasses by some language-dependent *flattening functions*. This process considers the related concepts of overriding, overloading and polymorphism. We implemented such *flattening functions* in our tool *Crocodile* [LeSi98]. This *Flattening process* solves the problems mentioned above.
The language independent concept of *flattening* has to be adjusted to the special programming languages with respect to the inheritance-related concepts (overriding, overloading and polymorphism). This task is not easy, especially for C++. This paper tries to explain in detail how the *flattening process* has to work for C++ class systems.
The idea of representing a class as it really is, i.e. considering all inherited attributes and operations, was introduced by Meyer [Meye88]. He uses a function *flat*, which constructs the flat representation of a class. In the field of object oriented measurement exists a concept called *inheritance context* [ErLe96]: It allows the selection of superclasses which have to be copied into their subclasses. Thus, we can restrict the set of classes to be considered in the *flattening process.*
In Section 2 of this paper, the basic concepts of overloading, overriding and polymorphism are explained with respect to the flattening process. This part explains these concepts for the language C++ in detail and introduces some corresponding *flattening functions*. Section 3 explains how the *flattening process* has to work for large C++ class systems. The different *flattening functions* of the previous section are transferred for use with large inheritance structures, including multiple inheritance and inheritance chains. Section 4 shows some useful applications for the *flattening concept*.

2 Overloading, overriding and polymorphism in C++

For the explanation of some C++ concepts in detail, we want to give some examples. They consist of some UML class diagrams (cf. [Oest97]) and – if necessary – source code parts. For our purposes, the method implementations are less important than the question which methods and attributes are accessible in detail.
The problems of overloading, overriding and polymorphism only occur, if two method names (without parameters) are identical. Attributes can not be overloaden but with conflicting names they can be overridden in some cases (see below).
In the following, we discuss overloading, overriding and polymorphism separately for attributes and methods, and we distinguish the case where conflicting names occur from the case where the name is unique. To simplify the introduction of our concepts we start to consider single inheritance. How our concepts are applied to systems with multiple inheritance is explained in Section 3.
For each situation we define a special flattening function which copies some members, i.e. methods, attributes and use-relations, into a subclass with respect to some given rules. The flattening functions have three parameters: the superclass, the class where to copy members into (which has to be a direct subclass of the superclass) and the type of inheritance.

2.1 Attributes with unique names

Without name conflicts for attributes, the technique of flattening classes in an inheritance structure is simple: The attributes of a superclass are copied into all subclasses, if they are still visible there. Whether they are still visible depends on:

- the attribute's visibility type: Private attributes are always invisible within the subclasses. Protected attributes can remain protected at maximum (but they even can be private). Public attributes can remain public at maximum (but they even can be protected or private).
- the inheritance type: Inheritance itself has a visibility restriction in C++. The default inheritance type is public. In this case, the visibility of all attributes remains unchanged (except private attributes, which change to invisible). The protected inheritance changes public attributes' visibility to protected and protected attributes' visibility remains unchanged. Through private inheritance, the visibility of public and protected attributes changes to private.

The following table summarises the changes of visibility (cf. [Schi98], pp. 420). Note that we do not consider the possibilities to restore the original visibility by *access declaration* ([Schi98], pp. 436ff]) because we think that this strongly reduces the understandability of a given program and this technique is deprecated in the current C++ standard.

is changed by	**Public member**	**Protected member**	**Private member**
public inheritance to	Public	Protected	Not visible
protected inheritance to	Protected	Protected	Not visible
private inheritance to	Private	Private	Not visible

Table 1: Visibility changes through inheritance

This copy technique for attributes from the superclass into the subclass is only correct, if there does not exist a separate attribute declaration with the same name in the subclass.
Before defining the flattening function for attributes with unique names we have to think about ***wrapped attributes***: These are attributes that are not directly accessible, i.e. they have the visibility private, but are accessible through visible get and set methods. A strictly used encapsulation principle in an object oriented design forbids direct access to any attribute. The recommended construction is to access these attributes by special get and set methods. Our above consideration of such wrapped attributes is important, because ignoring these attributes within the subclass does not reflect the real situation: Imagine a class `A` with 10 private attributes and each attribute has one get method and one set method. If then a class `B` inherits from this class and does not add, modify or delete anything there would be no reason why class `A` and class `B` should be considered as different regarding their tasks. However, this would happen if no flattening function would be applied. These wrapped attributes, too, have to be considered for the flattening process. Therefore, we have to define a new visibility type that we call *invisible*:

> An **invisible member** of class X is a member of X which is not visible in class X because of private declaration in one of the superclasses of X, but there is an indirect way to access the member (through a visible method of the superclass which provides access to the member).

Nevertheless, attributes can exist that have a private declaration within a superclass but in contrast to being invisible, they have no indirect access method; these members are *inaccessible*:

> An **inaccessible member** of class X is a member of X which is not visible and also not accessible in class X. There exists no way (neither direct nor indirect) to use that member.

To copy the invisible attributes into the subclasses we have to add the class name to the names of the wrapped attributes to avoid name clashes, i.e. instead of attribute `i` we write `A::i`.
Now we are able to define the flattening function for attributes with unique names:

> **flatten_unique_attribute_names** (*superclass, subclass, inheritance type*): This flattening version copies all relevant attributes of the *superclass* into the *subclass* if no attribute with the same name exists in the *subclass*. It copies all wrapped attributes of the superclass, i.e. private attributes that are used by at least one method within the superclass which again is inherited by the subclass, into the subclass. The visibility of the wrapped attributes is set to *invisible* and the class name is added to the attribute name. Inaccessible attributes are not copied.

Figure 1 shows an example how *flatten_unique_attribute_names* works. This example assumes that the method `print_A1()` uses the attribute `A3`, i.e. `A3` is a wrapped attribute. Because of this, it is copied into the new version of class B (here, no visibility sign represents invisible). In this case the flattening function would copy three attributes (methods are dealt with later).

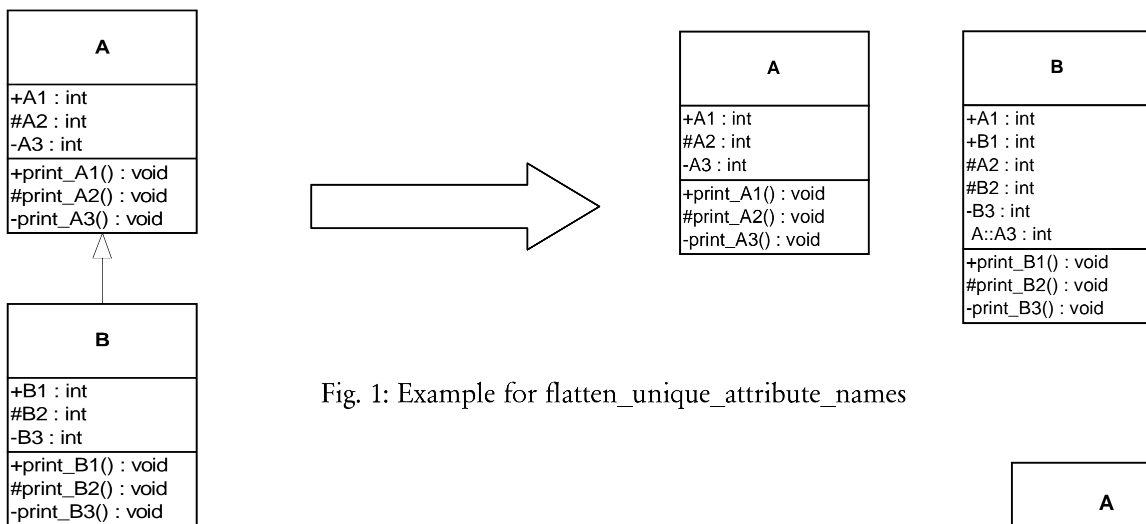

Fig. 1: Example for flatten_unique_attribute_names

2.2 Attributes with conflicting names

Concepts like overloading, overriding and polymorphism are only applied to members with same name. Because attributes have no signature, i.e. a return value or parameters, overloading is not possible. The data type is not considered, i.e. it is not possible to have two attributes in one scope with the same name but different data types.
Nevertheless overriding of attributes in some cases exists: If a subclass defines an attribute with the same name as a visible attribute of a superclass, the subclass attribute overrides the superclass attribute, i.e. that-one is no longer directly accessible.
Example (cf. Figure 2): Every object of type A has access to the attributes `A::A1, A::A2` and `A::A3` by only using `A1, A2` and `A3`. Each object of type B has access to `B::B1, B::A1, B::A2, B::B2, B::A3` and `B::B3` by only using `B1, A1, A2, B2, A3` and `B3`. In this case `B::A1` overrides `A::A1` and `B::A2` overrides `A::A2`.
However, it is still possible to access the overridden attributes by the use of the scope operator "::". E.g., an object of class B can use the public attribute A1 of class A by `<object>.A::A1`.

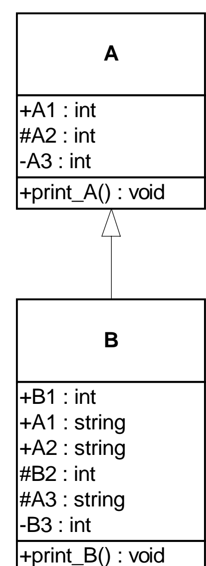

Fig. 2: Attribute overriding

Because attributes can not be virtual in C++ the type of the object is very important. If we have the following code:

```
A* test1;
B* test2;
test2=new B();
test1=test2
```

then the attribute `A1` of `*test1` is of type `int` but the attribute `A1` of `*test2` is of type `string`!
For the later effect, it has to be decided, whether also attributes that are accessible by explicitly using the scope operator have to be considered.
For the purpose of this paper, we do not consider the use of members via the scope operator due to the following reasons:

- For static members it is applicable to all classes independently of any inheritance relation between them.
- The re-definition of an attribute within a subclass is a deliberate decision. Using the scope operator bypasses this decision. We can not really imagine examples in which the use of both attributes, the inherited via scope operator and the re-defined one, might make much sense.
- In real-life applications we analysed (cf. [LeRuSi00]) the scope operator is rarely used.
- Using the scope operator improves the dependencies between particular classes. The re-use of a class is much more difficult because it is not sufficient to reproduce all use-relations and inheritance relations but it is necessary to adapt all code using the scope operator to the new environment.

Because of this, we do not need to define a further flattening function for attributes that have the same name because if two such attributes exist, the one of the superclass can be ignored.

2.3 Methods with unique names

The flattening function for methods with unique names is similar to the flattening function for attributes with unique names: the methods of a superclass are copied into the subclass. The conditions for their visibility are as explained in Section 2.1.
As for attributes we have to think about *wrapped methods*, which are not directly accessible methods, i.e. they have the visibility private, but they are accessible through other visible methods. This concept of private auxiliary methods is often used to extract some specific functionality. To demonstrate the concept of wrapped methods let us imagine a class `A` with 10 private methods and 1 public method that uses all 10 private methods. If a class `B` inherits from this class and does not add, modify or delete any member there would be no reason why class `A` and class `B` should be considered as different, because the provided functionality is the same. Nevertheless, this would happen if wrapped methods would not be considered. Because of the functional equality of both classes of our example, all private methods that are used by a non-private method have to be considered for the flattening process. Their visibility within the target class is changed to invisible. Furthermore, we have to add the class name to the names of the wrapped methods to avoid name clashes, i.e. instead of method `A()` we write `A::A()`.
Now we are able to define the flattening function for methods that do not cause name conflicts:

> **flatten_unique_method_names** (*superclass, subclass, inheritance type*): This flattening version copies all relevant methods of the *superclass* into the *subclass* if no method with the same name exists in the *subclass*. The target visibility is chosen in correspondence to Table 1. It copies all wrapped methods of the superclass, i.e. private methods that are used by at least one method within the superclass, into the subclass. The visibility of the wrapped methods is set to invisible and the class name is added to the method name. Inaccessible methods are not copied.

Applying *flatten_unique_method_names* to the original classes `A` and `B` from Figure 2 yield the new versions of class `A` and `B` that are displayed in Figure 3, if it is assumed that `print_A3()` within class `A` is used by `print_A1()`. In this case, the flattening function copies three methods.

A
+A1 : int #A2 : int -A3 : int
+print_A1() : void #print_A2() : void -print_A3() : void

B
+B1 : int #B2 : int -B3 : int
+print_B1() : void +print_A1()() : void #print_B2() : void #print_A2() : void -print_B3() : void A::print_A3() : void

Fig. 3: Example for flatten_unique_method_names (cf. Figure 2)

2.4 Methods with conflicting names

Methods in general have a return value and several parameters. The type of the return value and the types of the parameters together with their ordering are called *signature* [Meye97]. For our purpose we ignore the return value type because it is not considered for overloading and overriding in C++ (cf. [Schi98])[1].

2.4.1 Overriding

A method `A::M1` is overridden within B by a method `B::M1` if the following constraints are fulfilled:

- `A::M1` and `B::M1` are both visible within B; i.e. `A::M1` has at least the visibility 'protected'.
- `A::M1` has the same name as `B::M1`.
- `A::M1` and `B::M1` have the same number of arguments (also parameters with default values (e.g. `int x=7`) have to be counted) and every type t_i at position i of the parameter list `A::M1` has to be equal to the type t_i at position i of the parameter list of `B::M1`.

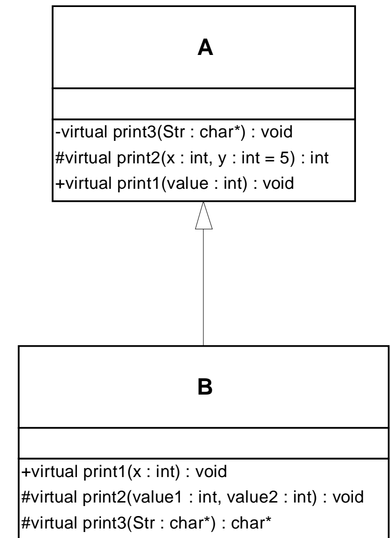

Fig. 4: Overriding methods

Assuming a public inheritance between class A and B in Figure 4 yields the following statements:

- `void B::print1(int)` overrides `void A::print1(int)`
- `void B::print2(int, int)` overrides `int A::print2(int, int=5)` because the return value is not considered for overriding and the parameter lists are equal (including default values).
- `char* B::print3(char*)` does not override `void A::print3(char*)` because the latter one is not visible within class B.

In the next section, we have to examine overloading because it influences the overriding mechanism.

2.4.2 Overloading

Since overriding is a kind of redefinition of a method, overloading allows the definition of similar operations with the same name but different types or numbers of parameters. The definition of overloading is done in a similar way as for overriding:

A method `A::M1` overloads a method `A::M2` if the following constraints are fulfilled:

- `A::M1` has the same name as `A::M2`.
- If `A::M1` and `A::M2` have the same number of arguments n (parameters with default values (e.g. `int x=7`) are not counted): At least one type t_i at position i of the parameter list of `A::M1` has to be different to the type t_i at position i of the parameter list of `A::M2` $(1 \leq i \leq n)$. Thus, the mismatching parameter has to occur before any default parameter.

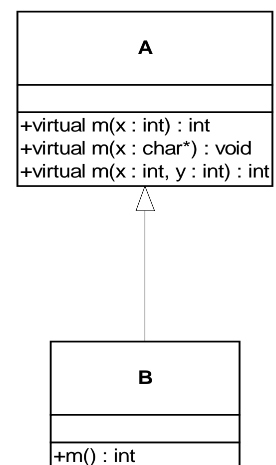

Fig. 5: Overloading methods

In Figure 5, the method m is overloaded twice within class A.

With respect to inheritance there is a very important point to remember if particularly dealing with overriding or overloading - especially in C++ -, because it contrasts the general overriding concept of Section 2.4.1: a method in a subclass will override all methods with the same name from the superclass, never overload them. As shown in Figure 5 class B inherits from class A and defines its own method `m(): int`. Within class B all other methods m from class A are overridden, i.e. they are not members of class B.

---

[1] In more detail: Changing only the return type does not cause overloading because the difference has to be within the parameter list. On the other hand it is not possible to override a method by changing the return type. Because this can cause problems for polymorphic structures the compiler rejects these cases. Since software measurement is done for compilable sources we can ignore this case.

So, whenever two methods from two classes in an inheritance relation have the same name, the flattening function must not copy the method from the superclass into the subclass and, therefore, there is no need for an additional flattening function.

2.5 Polymorphic use-relations

In this section we deal with *polymorphism*, in particular *run-time polymorphism* (in contrast to *compile-time polymorphism* that can be achieved by overloading and that can be solved by the compiler) (cf. [Schi98]). The technique to implement this kind of polymorphism is called *late binding*. This means that the determination which version of a function is called when a message with appropriate name occurs is made at run-time and is based on the object type. In C++ this type of late binding is only possible if an object is accessed via pointer (or via reference, which is an implicit pointer, [Schi98], pp. 341) and if the called method is declared virtual. This can be shown by an example as shown in Figure 6:
Let us take the following example code:

```
A  object_a;
A  *p_object_a;
B  object_b;
```

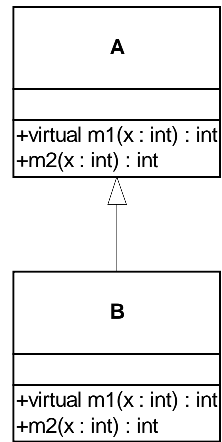

Fig. 6: Example for possible polymorphism

It should be clear, that if the message `m1(int)` is sent to `object_a`, the function `A::m1(int)` is executed. The same message sent to `object_b` executes the function `B::m1(int)`. It is important to notice, that this message-function pattern – that describes which function is executed after a message is received – does not change even after the assignment of `object_a=object_b`. The reason is the semantics of the '='-operator: The method `A& operator = (const A&)` of class `A` is called with object `object_b` as parameter and all attributes of `object_a` are set with the values of the corresponding attribute of `object_b`. In this case this is possible because `object_b` is an `A`. The reverse fails, i.e. the assignment `object_b=object_a` would not pass the compiler (because `object_a` is not a B).
After an assignment of the reference of `object_a` to `p_object_a` the function `A::m1(int)` is executed, if the message `m1(int)` is sent to `p_object_a`. Changing only the assignment to `p_object_a=&object_b` causes the execution of `B::m1(int)` if the message `m1(int)` is received (because `p_object_a` points to a B now). Additionally it is important to notice, that in both cases the function `A::m2(int)` is called after the message `m2(int)` is received, because the method `m2` is not declared as virtual in class `A`.
While considering polymorphic structures it is not always possible to decide at compile-time which function will be executed at run-time. Due to this, the static analysis of polymorphic use-relations between classes is often reduced to use-relations between superclasses [LeSi98]. The application of this technique for a method `m3` of class `C`, that has a pointer to class `A` of Figure 7 would show a use-coupling between class `C` and class `A` but not between class `C` and class `B`. This simplification might be wrong: If for example the pointer within `m3` of class `C` would point to an object of class `B` – which is possible because `B` is derived from `A` - the methods overridden (or implemented) in `B` can be used by the pointer to class `A`.
Because the static analysis of source code always considers only potential of use and not the actual frequency of use, there should be a function *flatten_polymorphic_use_relations* that adds the use-relations into all methods of the subclass that could be executed by method `C::m3`. This flattening function depends not only on the superclass and subclass but also on the class that uses methods of the superclass. The corresponding flattening function is defined as follows:

flatten_polymorphic_use_relations (*superclass, subclass, client_class, inheritance's type*): This flattening version adds these use-relations that cover calls of the client class to public methods of the subclass. This is done in the following way:
A use-relation between a method $m_{clientc}$ of the client class and a public method m_{subc} within the subclass is added if

- the inheritance's type is public,
- $m_{clientc}$ uses the method m_{superc}, declared in the superclass,
- m_{superc} is defined as virtual,
- m_{superc} is overridden or implemented by m_{subc} within the subclass.

Figure 7 shows an example how *flatten_polymorphic_use_relations* works:
In the original structure there exists only a use-relation between C and A (namely `C::m3()` uses `A::m1(int)` and `A::m2(int)`). After flattening a further use-relation between C and B is added (namely `C::m3()` uses `B::m1(int)`) because all conditions for possible polymorphic use of `m1(int)` are fulfilled. This is not the case for `B::m2(int)` because it is not defined as virtual within the superclass.

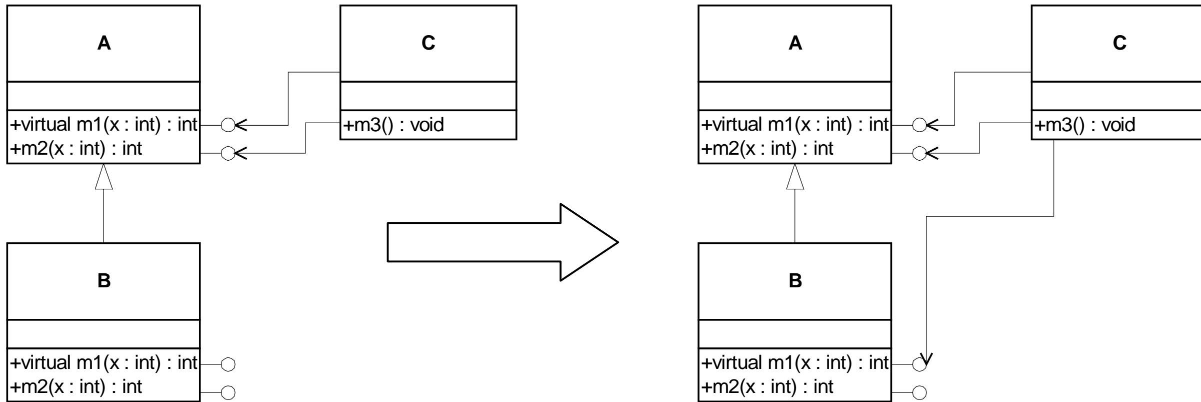

Fig. 7: Example for flattening of polymorphic use-relations

3 How to use the flattening process for large C++ systems

This section deals with the question how the different flattening versions are applied to large object oriented systems written in C++, which can be characterised by the following points:

- In large C++ systems multiple inheritance is often used, i.e. one subclass has more than one direct superclass. How should the flattening function be applied in such cases?
- Large C++ systems contain *inheritance chains*, i.e. a class inherits from another class which itself again inherits from another class. In which order should the flattening functions be applied?
- How to handle the different flattening versions if large class libraries are used or whole frameworks are instantiated? To which parts should the flattening function be applied ?

3.1 Flattening and multiple inheritance

It is possible for a derived class to inherit directly from two or more superclasses. This concept is called *multiple inheritance* ([Schi98], pp. 427). We distinguish two cases:

- Inheriting from all superclasses does not lead to any name conflict, i.e. neither there exists any public or protected method nor any public or protected attribute that has the same name within the set of superclasses.
- Inheriting from all superclasses leads to some name conflicts.

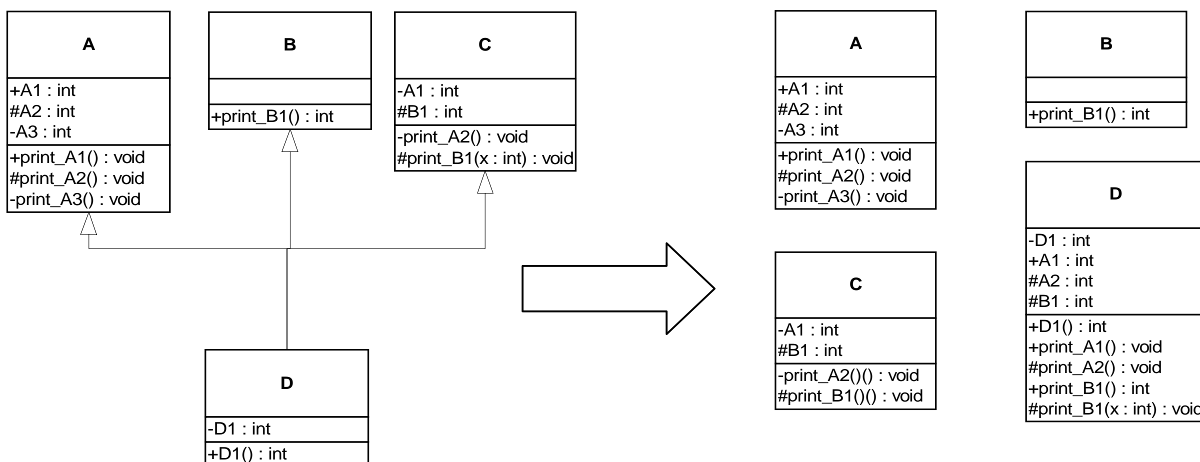

Fig. 8: Class structure with multiple inheritance and its flattened version

The former case is easier to handle, because all flattening functions can be applied sequentially in arbitrary order to every superclass. Figure 8 shows an example of a class structure using multiple inheritance and the same class structure after the flattening process.

In the case of name conflicts we distinguish two situations:

- At least two classes that are inherited directly by another class have one member with the same name. The use of this member within the inheriting class would cause name conflicts. Because of this, the members are still accessible only by using the scope operator which is not considered in our context (cf. Section 2). This case can be handled as multiple inheritance without occurring name conflicts. Thus, the flattening function does not copy members that cause conflicts within the subclass.
- One class is inherited several times. One example for this *rhombus-like inheritance* ([Stro98], pp.415) is presented in Figure 9. Since both classes (`B` and `C`) inherit `A1` and `A2`, class `D` inherits these attributes two times. Usually, we handle this occurring name conflict in the same way as occurring name conflicts by members with same name within different classes (see above). However, C++ offers a possibility to prevent conflicts if using rhombus-like inheritance: The definition of *virtual base classes*. To avoid the multiple occurrence of members of a class that is multiply indirectly inherited, all inheritances of this class have to be defined as `virtual`. This can be accomplished by preceding the base class name with the keyword *virtual*. Thus, if the definition of class B would be '`class B: virtual public A`' and the one of class `C` would be '`class C: virtual public A`', then class `D` would have two unique attributes `A1` and `A2` which have also to be considered for our purposes. Respecting this, the flattening function copies only members with conflicting names if class A is virtually inherited. Otherwise the flattening function does not copy the members with conflicting names.

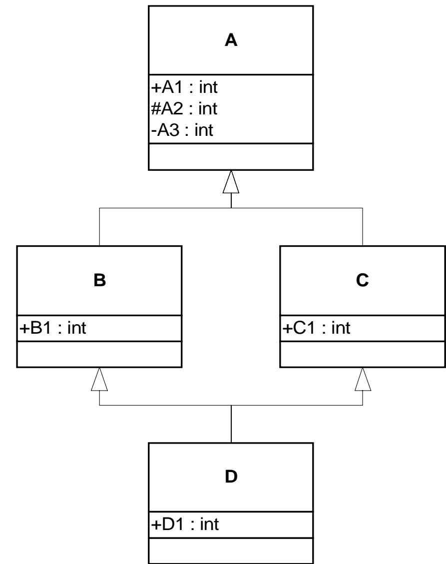

Fig. 9 : Rhombus-like inheritance

To include these special cases into the flattening process we need another flattening function for multiple inheritance. To deal with several superclasses we have to change the parameters of the flattening function to ***flatten_multiple_inheritance****(superclass_relations, subclass)*, with *superclass_relations* is a set of pairs (*superclass*, *inheritance_type*), where *inheritance_type* corresponds to table 1.

> **flatten_multiple_inheritance** (*superclass_relations, subclass*): This flattening function combines all the flattening tasks provided by the flattening functions *flatten_unique_attribute_names*, *flatten_unique_method_names* and *flatten_polymorphic_use_relations* and extends their functionality to multiple inheritance. This is done in the following way:
>
> - For every superclass of the *subclass*: Call the different flattening functions with the corresponding inheritance type.
> - Before a flattening function copies a member into the *subclass*, it has to be checked if a name conflict exists with potential inherited members of the other superclasses.
> - If no name conflict arises, the member is copied. Otherwise, it has to be checked if the name conflict is caused by virtual rhombus-like inheritance that also leads to copying the member.

3.2 Flattening of inheritance chains

In real-life applications, inheritance can be used very often and there can be used long inheritance chains, i.e. the class, of which another class inherits from, itself inherits from another class. Figure 9 from last section shows an example for two inheritance chains of the length 2.

To be able to consider also members that are not a direct but an indirect superclass we have to define a function *flatten_deep (X)* which has to be applied recursively to calculate the real flattened version of a class `X`:

flatten_deep (X): The following steps define the function flatten_deep (X):
1.) For each superclass Y_i of X, do:
    1.1) Call the function *flatten_deep(Y_i)*, if Y_i is not yet marked as flattened.
2.) If class X has at least one superclass:
    2.1) Call the function *flatten_multiple_inheritance(superclass_relations, X)*.
3.) Mark class X as flattened.

For our example in Figure 10, this means that for *flatten_deep(X)* at first we have to call *flatten_deep(C)*. In step 1 of this task we call *flatten_deep(B)* (which itself calls *flatten_deep*(*A*) but for this class *flatten_deep* does nothing but to mark it as flattened) and *flatten_deep (D)* (which also does nothing but to mark it as flattened because the recursion ends). After returning from these tasks the *flatten_multiple_inheritance* (step 2) can be called for class `C` and at end for class `X`.
In Figure 10 the new class names are used only to demonstrate the set of classes from which class members are already copied into the class, i.e. class `BA` has already members of class `A` in it.

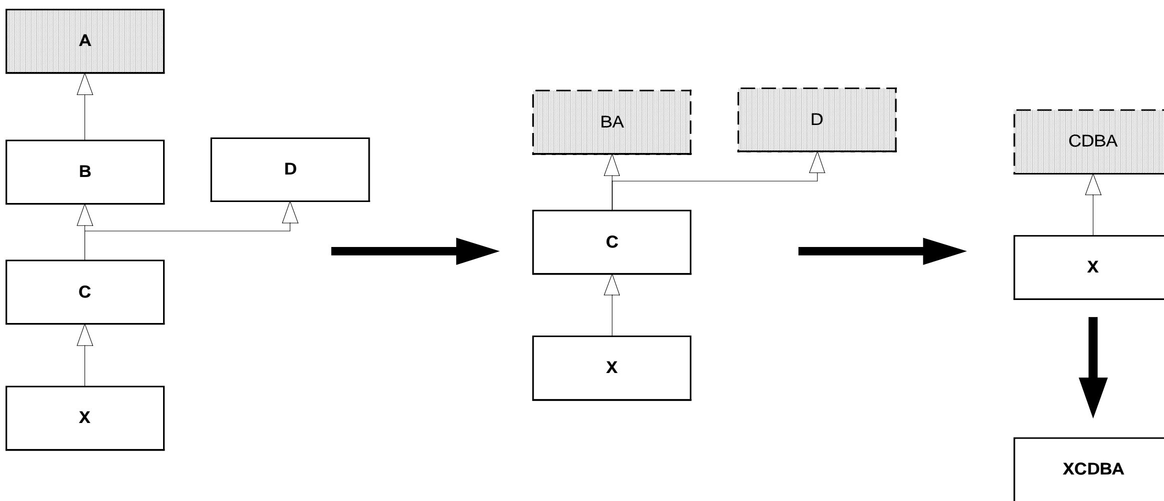

Fig. 10: The flattening function applied to a large system

3.3 Flattening of library- or framework-based systems

Usually, a large system is based on a lot of prefabricated libraries or frameworks. In many cases, the classes from libraries are re-used by inheritance. In such systems two problems can occur:

- The application of the flattening functions as explained in the previous section might distort the view on the flattened classes because too much information of the libraries and frameworks are flattened into a class.
- For understanding or using of a particular set of classes it is essential to have a flat view based only on the classes of the library or framework. Classes from other libraries (e.g. system libraries) used by the class set are not interesting for this purpose.
- If measurement should be applied to the system, it yields numbers which should be interpreted and which should point out some anomalies and some hints for restructuring (cf. [LeRuSi00]). Of course, such anomalies and hints make only sense for software parts for which the developer is responsible. Generally, this is not the case for libraries and frameworks supplied by someone else.

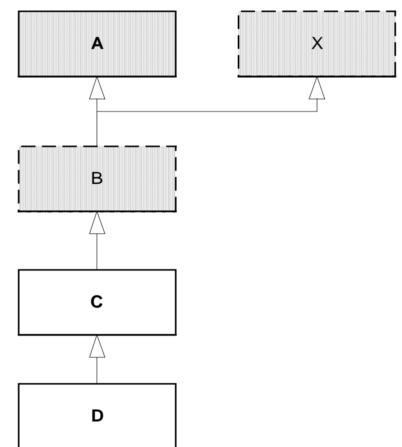

Fig. 11 System based on a class library

To use only parts of the system for which the developer is responsible or interested in for flattening, two mechanisms are necessary:

- The setting of a *focus set*: The interest is focused only to classes that are part of the focus set [ErLe96]. The example of Figure 11 shows a system consisting of a class library (grey classes) and self written application classes (white classes).

If only class C and D are set into the focus, then only both classes are flattened and e.g. only their measurement values are presented.

- The setting of an *inheritance context*: The flattening process for an inheritance chain is reduced to classes that are contained in the inheritance context [ErLe96]. Thus, if the inheritance context is empty the system is used without any flattening. If all classes are set into the inheritance context, all existing inheritance chains are fully flattened. To come back to the example displayed in Figure 11: By setting class C into the inheritance context only the members of class C are flattened into class D.
  Please note that because of later performance reasons the inheritance context and the focus set should not be set independently. After setting the focus set, the set of possible inheritance context's classes is reduced to these classes of which at least one class in the focus directly or indirectly inherits from (through non-focus classes that are already part of the inheritance context). If for example in Figure 11 the focus is set to classes B and C it has no effect to set class C or D into the inheritance context.

The focus set and the inheritance context should be usable in a very interactive manner because in some cases it might make sense to explore different settings. One such implementation is explained in the next section.

4 Applications of the flattening concept

This section discusses some useful applications of the flattening process. We recommend the use of flattening as an additional view on the system for software measurement, for code understanding and for development of object oriented systems.

To evaluate our concepts we used the tool *Crocodile* [LeSi98]. Crocodile works on a database containing relevant data about the object oriented system structure extracted from a given source code. The database contains all the information about classes, attributes and methods, and also all the relations between them (regarding inheritance and method calls). All flattening functions are applied on these data and insert attributes, methods, and use-relations into subclasses. The only restriction we have to make is the inability to check whether a method of a class is used through a pointer variable or a normal variable (cf. Section 2.5). Thus, we have to assume that all methods that can be used polymorphically in an inheritance tree are called through a pointer variables.

After the extraction of all necessary data the next task is to set the contexts as explained in Section 3.3. This is done in the following order:

1. All classes that are part of the class system for which the information is extracted can be set into the focus set. To simplify this task for large projects, it is possible in a first step to set whole subprojects, which typically can be identified by different subdirectories, into the focus set and afterwards to refine this selection on the class level. This is necessary to have only classes in the focus we are potentially interested in. E.g. in programs where code is implemented for different operating systems it is wanted to set only one version into the focus set.
2. Depending on the focus set the inheritance context can be set, i.e. classes of which at least one class in the focus set directly or indirectly inherits from. Because of the consideration of indirect inheritance through classes of the inheritance context the set of possible classes that can be set into the focus set can be extended iteratively, i.e. after setting a non-focus class into the inheritance context its superclasses again can be set into the inheritance context.

These settings influence the flattening process because the different flattening functions are only called for classes that are within the inheritance context. The flattening process itself is done by adding new members into Crocodile's database: For example a new attribute that a subclass gets from its superclass is inserted into the database by adding a new attribute entry with the same name, the same use-relations of methods, the same location of implementation, but a new class membership.

To get the possibility to try several context selections without the necessity to re-extract the data from source code, Crocodile has two databases: One contains the original data and the other one contains the flattened data.

After the flattening process one can use the flattened representation, e.g. to calculate the measurement values on the basis of the modified data, i.e. the added members and use-relations are considered by the measurement values.

4.1 Impact on software measurements

Measuring object oriented systems without considering the possibility to distribute functionality and data over several classes in an inheritance structure is only one view. In a case study (cf. [SiBe00]) we validated that the new "flattened view" on a system might be helpful with respect to the following three points:

1. In the non-flattened version we detected many classes whose interpretation of the measurement values would be misleading. If the view on a system that is presented by the measurement values might suggest wrong interpretations, a further view seems to be useful.
2. With the flattened version we detected many classes that showed some anomalies and that in fact would be good candidates for restructuring.
3. Considering the differences between the measurement values of the flattened and non-flattened version produces another interesting view on a system. This information gives new insights into the kind of inheritance, e.g. if inheritance is used only for source code sharing, for creating type hierarchies or specialisation.

4.2 Analysing and understanding class systems

To support analysing and understanding of a reengineering process for large software development projects our research team has developed a tool for navigation in such object oriented systems [LeSiSt00]. One part of the tool is to provide the syntactical structure of a system (CrocoBrowse). It is possible to click through the components on different levels like packages, classes and members and also to look on the compositional hierarchy.
The tool gets more expressive power by providing also the additional view of the flattened class system. Thus, a user of such a navigation tool has the advantage to switch between the two views on the system and he can choose the view that helps him most for his purpose.

4.3 Using class libraries and frameworks

There exists another interesting application of the flattening concept for providers as well as users of object oriented class libraries or frameworks. When using a particular class of a library or framework there always is the question: What methods and attributes are provided by the class which might be more than the locally defined ones? If the class system has long inheritance chains it is a very time-consuming task to find the relevant method name by only clicking through the inheritance structure. With the additional choice of navigating on the flattened version the use of the library or framework is more simple and more effective.
Another way to use the flattened version of a class system is to generate useful documentation about it because it is very time-consuming to determine which functionality a class really has.
The flattening concept can also be used by software engineering environments (SEE) to provide some nice features. One of them is the automatic method name extension (like presented in Visual C++ 6.0). Using the flattened representation of a class system makes the SEE able to provide all *accessible members* (regarding all the tricky concepts like overloading and overriding), which are directly usable by the user.

References

[ErLe96] Karin Erni, Claus Lewerentz: “*Applying design-metrics to object oriented frameworks*” in “Software metrics symposium”, pp. 64-74, IEEE Computer Society Press, 1996

[LeRuSi00] Claus Lewerentz, Heinrich Rust, Frank Simon: "*Quality - Metrics - Numbers - Consequences: Lessons learned*", in Reiner Dumke, Franz Lehner (Hrsg.): “Software-Metriken: Entwicklungen, Werkzeuge und Anwendungsverfahren”, pp. 51-70, Gabler Verlag, Wiesbaden, 2000

[LeSi98] Claus Lewerentz, Frank Simon: “*A product metrics tool integrated into a software development environment*”, in proceedings of object oriented product metrics for software quality assessment workshop (at 12th European conference on object oriented programming), CRIM Montreal, 1998

[LeSiSt00] Frank Simon, Claus Lewerentz, Frank Steinbrückner: "Multidimensionale Mess- und Strukturbasierte Softwarevisualisierung", to be published in proceedings of 2th workshop "Reengineering" in Bad Honnef, 2000

[Meye88] Bertrand Meyer: “*Object-oriented Software construction*”, Prentice Hall, London, 1988

[Meye97] Bertrand Meyer: “*Object oriented software construction*”, 2nd edition, Prentice Hall, London 1997

[Oest97] Bernd Oestereich: “*Objekt orientierte Softwareentwicklung mit der Unified Modeling Language*”, Oldenbourg Verlag, 3rd edition, München, 1997

[Schi98] Herbert Schildt: “*C++: The Complete Reference*”, 3rd edition, McGraw-Hill, Berkeley, 1998

[SiBe00] Frank Simon, Dirk Beyer: "Considering Inheritance, Overriding, Overloading and Polymorphism for Measuring C++ Sources", Computer Science Reports, 04/00, Technical University Cottbus, May 2000

[Stro98] Bjarne Stroustrup: “*Die C++ Programmiersprache*”, Addison-Wesley, 3rd extended version, Bonn, 1998