Vrije Universiteit Brussel
Faculteit Wetenschappen

Normalising Class Components

Marc Van Limberghen

Techreport vub-prog-tr-95-05

Programming Technology Lab
PROG(WE)
VUB
Pleinlaan 2
1050 Brussel
BELGIUM

Fax: (+32) 2-629-3495
Tel: (+32) 2-629-3308
Anon. FTP: progftp.vub.ac.be
WWW: progwww.vub.ac.be

Normalising Class Components

Marc Van Limberghen
Department of Computer Science
Faculty of Sciences
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussels, BELGIUM
E-mail: mvlimber@vnet3.vub.ac.be
Phone: +32.2.6293492
Fax: +32.2.6293495

Abstract: *Class hierarchies are currently often subject to reorganisation even when the intended universe of discourse hasn't changed. These revisions unnecessarily burden the implementation process and are due to the fact that some design information is only implicitly present in the hierarchy. Whereas the relational database community has recognised the import of distributing data attributes over tables, the equivalent in OO class design, i.e. the distribution of attributes with behaviour over class components, has been totally neglected. This paper exactly determines the design information that is equivalent with this kind of attribute distribution and accordingly defines normal forms for class components.*

Keywords: *inheritance, normalisation, class components, class library*

1. Introduction

A class library is not merely a collection of classes to instantiate, but should also be seen as a set of reusable components. New classes can be built by combining the offered components without adding new code. This paper focuses on this form of class construction. We examine the relation between class components and the classes that can be constructed with them, i.e. the induced class space.

Class hierarchies restrict the ways to combine the class components of a library. OMT ([Rumbaugh&al91]), [Hamer&al92] and [McGregor&Korson93] enriched traditional class hierarchies with "generalisation properties", "classifiers" respectively "dimensions" as extra restricting capabilities. This way it becomes for example possible to prevent a class to inherit simultaneously from several class components meant to be exclusive (Male and Female for instance). Consequently such classes are excluded from the induced class space.

But before imposing restrictions on the class component combination, one should understand the restrictions enforced by the components themselves. The grouping of attributes into components already affects the induced class space. Grouping attribute labels (i.e. method selectors) into class interfaces has already been investigated ([Godin&Mili93], [Lammari&al94]). The presence of overriding in OMT however proves that this design methodology not only considers attribute labels, but also the associated method definitions. This paper goes a step further than previous work on attribute grouping by taking the definition of the class methods into account.

Given a distribution of attributes (labels and definitions) over class components, we will rigorously determine the corresponding information about the universe of discourse (UoD). This clearly indicates the intended class space boundaries for the user of the library. The other way around we will point out how to distribute attributes over components given the UoD. This knowledge permits or even forces the designer of the class library to motivate his grouping choice.

> *In the domain of relational databases the process of grouping attributes into tables is called normalisation [Hawryszkiewycz91]. We will clarify the philosophy of our approach through comparisons with this widely known database table normalisation. Grouping database attributes into database tables affects the possible data populations that can be entered1. Analogously grouping OO attributes into class components affects the possible classes that can be constructed. Restrictions on database populations are specified by the designer via so-called functional dependencies between sets of attributes. Similarly we will define a design relation between OO attributes that describes restrictions on the intended classes. In both media a grouping is said to be in normal form if it reflects the design restrictions in a satisfactory way.*

In order to be able to clearly express the influences of an attribute grouping on the induced class space, the inheritance medium should be relieved from any restricting capabilities on the class component combination. Mixins, as defined in ([Bracha&Cook90]), can be considered as class components detached from any class hierarchy restrictions. Consequently mixin-based inheritance establishes a virginal inheritance mechanism for our exploration.

We will show that in most inheritance systems (including mixin-based inheritance), grouping attributes can conflict with reusability which is considered to be one of the key features of OO systems. We will point out that this conflict can be avoided by restricting parent class references. Although this restriction theoretically restrains the expressivity, it is often accepted as good OO programming style. Moreover it

1 Splitting database tables (i.e. grouping attributes into tables) is known to avoid redundancy. This redundancy results from the absence of certain populations in the universe of discourse. Consequently splitting database tables can also be considered as excluding these populations.

allows us to better comprehend inheritance design. Libraries that comply with this restriction will be defined as being in first normal form.

In this disciplined inheritance medium we can identify the design information that corresponds to grouping attributes. Eventually a class library will be called to be in second normal form if it reflects this design information in a satisfactory way.

2. Class components and inheritance.

In this section we will be more specific about attributes, class components and inheritance.

Definition: A *label* is simply a string serving as attribute-selector. Let **Labels** be the set of all possible labels.

For the purpose of this paper it is not necessary to give a complete formal definition of method-bodies. It suffices to note that *bodies* are entities performing a computation with an object as result. To this extent they can perform *self* and *super* calls. Let **Bodies** be the set of these bodies.

Definition: An *attribute* $a = (a_{label}, a_{body})$ is an element of **Labels** $\times$ **Bodies**.
Let **Attributes** be the finite set of all attributes involved in a specific case.

Definition: A *component* c is a set of attributes with different labels[2]:

$$c \subseteq \mathbf{Attributes}: \forall\ a, b \in c: a_{label} = b_{label} \Rightarrow a = b$$

Composition of components c_1 and c_2 is noted as $c_1 + c_2$ and accords to common inheritance where c_1 is the parent and c_2 is the child. In [Bracha&Cook90] this is formally defined under the notation $c_2 * c_1$.

3. The conflict between inheritance and reusability.

This section shows that most inheritance mechanisms can force the programmer to duplicate program code, whereas inheritance is just intended to support code reuse.

2 The term *component* is used instead of *mixin* in order to indicate a slight difference with the mixins of [Bracha&Cook90] , where composition of two mixins yields a new mixin. Components on the contrary are only groups of primitive attributes. The composition of components that share a label results in a mixin that is no longer a component.

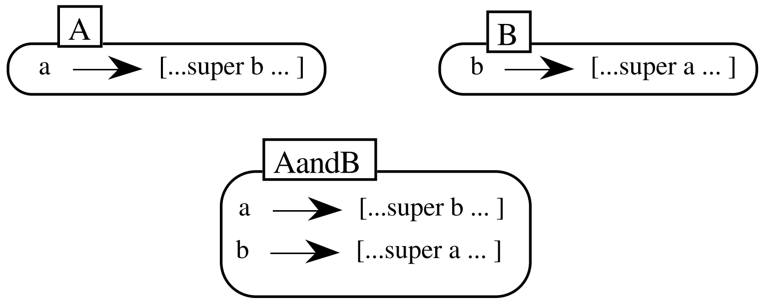


figure 1: The attribute with label a is duplicated in the components A and AandB. This attribute performs a super call of the label b. Analogously for the attribute with label b.

Consider the three components presented in figure 1. The component AandB turns out to be not equivalent to A+B, nor to B+A, as showed in figure 2. Consequently, duplicating and grouping attributes performing arbitrary super calls can *extend* the possible behaviours: the three components of figure 1 allow more behaviours than only A and B separately. The duplication of the two attributes is however against the spirit of reuse by inheritance.

If we restrain super calls of attributes to their proper label, the duplication problem is avoided: The behaviour associated with a component that simply unites other components can then be constructed by composing these components with inheritance. This can be expressed more formally as follows:

Property1: If c_1 and c_2 are two components
with different interface, i.e. $\forall\, a \in c_1,\ \forall\, b \in c_2$: $a_{label} \neq b_{label}$, and
if $\forall\, a \in c_1 \cup c_2$: a_{body} performs only super calls of a_{label}, then

$$c_1 + c_2 = c_1 \cup c_2$$

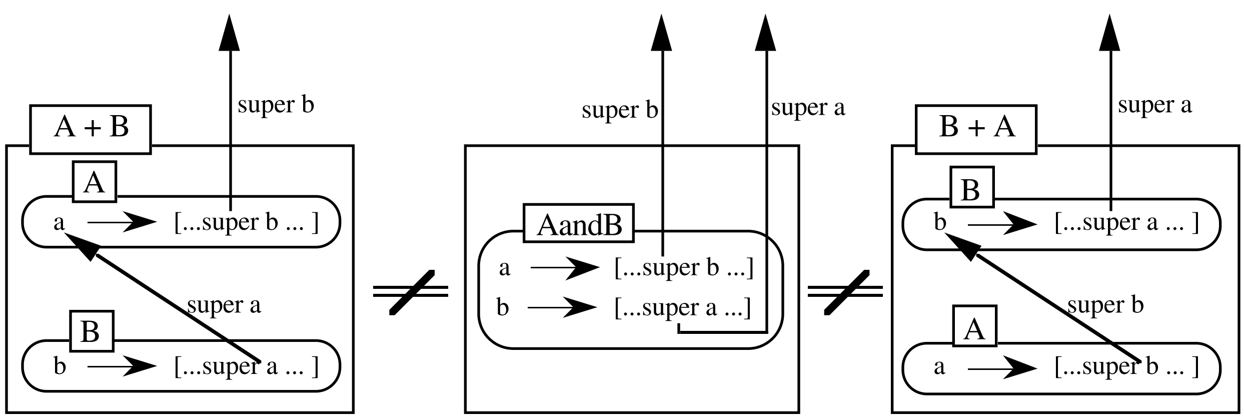


figure 2: AandB is not equivalent to A+B, nor to B+A

This restriction on super calls restrains somehow our expressivity. By calling super a, the attribute b of AandB in figure 2 performs a only partially. Our restriction on super calls can be interpreted as a guarantee for atomic attributes: except for the specialisation of an attribute it is impossible to invoke part of the behaviour of an attribute, even from within the class the attribute belongs to!

We consider atomic attributes not as a restriction, but rather as a good OO programming style. Behaviours according to free super calls should be obtained by a slightly different class hierarchy design. Invoking an attribute partially indicates that this part is worth an extra full attribute. The new attribute can then be called with a self call and can, if desired, be hidden to further users by a visibility mechanism such as the *hide* operator of [Bracha&Lindstrom92].

4. Class representation

Classes are constructed by composing class components. Thanks to the super restriction of the previous section we obtain a simple representation for classes. The implementation of a class attribute can now be represented as a sequence of bodies where the super-variant of a body is represented by its successor in the sequence (cf. the index column in figure 3). Graphically this means that no cross-references occur between bodies.

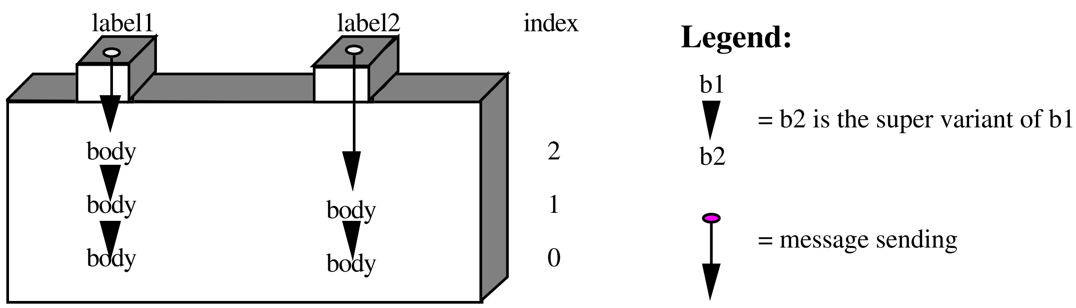


figure 3: a class with two labels and the associated bodies

This class representation is formally translated in the following definitions:

Definition: A *bodysequence* is a sequence of bodies, i.e. a discrete function f: $\{0,..,n\} \rightarrow$ **Bodies**. Let **Bodysequences** be the set of all bodysequences.

Definition: A *class* is a discrete function cl with Dom(cl) $\subseteq$ **Labels** and Im(cl) $\subseteq$ **Bodysequences** such that $\forall$ l $\in$ Dom(cl): $\forall$ b $\in$ Im(cl(l)): (l,b) $\in$ **Attributes**

Intuitively, a class can only contain a body b in the bodysequence associated with a label l, if (l,b) is an attribute of **Attributes**, our casual attribute set. Let **Classes** be the set of all classes.

This class representation is suited for our investigation because it is independent from the components the class is constructed with. This way we can compare classes constructed with different components. Note that we will distinguish two classes with another bodysequence structure as two different classes, even if they occasionally have the same computational behaviour. For example, if the bodies in figure 3 perform no super calls, then substituting their super variant yields a class with the same computational behaviour. Our formalism however will distinguish this class from the original. Future research on class designer customs is required to find out if such cases should be treated as yielding equal classes. If so, the integration of a second kind of attributes, namely attributes without super call, can catch this feature. For clarity reasons we omitted this extension that does not influence the main ideas in the rest of this paper.

5. Grouping attributes reduces the induced class space

In this section we examine how the grouping of our attributes into components influences the induced class space. As mentioned earlier, we consider a library as a collection of class components rather than a collection of classes.

Definition: A *library* L is a partition of **Attributes** into components:

1) $\forall\, c \in L: c \neq \emptyset$
2) $\bigcup_{c \in L} c = \textbf{Attributes}$
3) $\forall\, c_1, c_2 \in L$ with $c_1 \neq c_2$: $c_1 \cap c_2 = \emptyset$

The last requirement assures that an attribute is only present in one component. This excludes any attribute duplication. In this paper we take maximal code reuse as primary goal and neglect what influences attribute duplication might have on the induced class space. Remember that we are talking about entire attributes (i.e. labels and bodies) so that it is quite possible that components of L each contain an attribute with the same label.

We will only reason in the context of restricted super calls as proposed in section 3. Our first normal form implies this restriction:

Definition: A library L is in *1NF*

$$\Leftrightarrow$$

$\forall\, c \in L: \forall\, a \in c$: a_{body} performs only super calls of a_{label}

Although the super restriction is associated with single attributes rather than with a grouping of attributes, the 1NF imposes this restriction on an entire library. We copied this peculiarity from

relational databases where the 1NF that logically puts a restriction on the domain of the attributes, is defined on an entire table.
It will become clear later in this text why in our case normal forms are defined on a library instead of on a component, the analogical counterpart of a database table.

Thanks to this restriction, we can now use the class representation of the previous section. For instance the induced class space of a library, i.e. the classes that can be constructed by composing its components, is formally introduced as follows:

Definition: Let L be a library in 1NF, then

$$classSpace(L) = \{cl \in \textbf{Classes} \mid \exists\, n \in N, \exists\, c_1, c_2, .., c_n \in L:\ cl = c_1 + c_2 + .. + c_n\}$$

In section 3 we saw that, thanks to our restriction on super calls, uniting components (necessarily with different labels) in a new component can not give rise to new classes. Of course uniting components in a new component necessitates to disregard the old components in order to avoid duplication (as required by our library definition). Hence this attribute grouping operation can only *restrict* the induced class space. Formally this can be expressed as follows:

Definition: Let L_1 and L_2 be two libraries, then L_2 *further groups* L_1

$$\Leftrightarrow$$

$$\forall\, c \in L_2:\ \exists\, n \in N,\ \exists\, c_1, c_2, .., c_n \in L_1:\ c = c_1 \cup c_2 \cup .. \cup c_n$$

Property2: If L_1 and L_2 are two libraries in 1NF such that L_2 *further groups* L_1, then

$$classSpace(L_2) \subseteq classSpace(L_1)$$

Consequently the library consisting of components containing only one attribute induces the largest class space. Grouping attributes always reduces the induced class space. It is precisely this monotone character of attribute grouping that allows normalisation reasoning. Remember that this result stems from our super restrictions in section 3.

In relational databases normalisation means splitting*: the most general population can be stored in one table containing all attributes; by splitting up tables in subtables, some populations can no longer be stored. Splitting tables is also a monotone operation reducing the induced populations. In our case normalising is the reverse operation: we start from components with only one attribute and then* group *attributes so that some classes can no longer be constructed.*

6. Constraints implied by attribute grouping

By means of an example we will now examine the specific restrictions on the set of possible classes implied by grouping attributes. Suppose we want to build classes that represent the behaviour of some device with a regulatable intensity (e.g. the contrast of a monitor). Figure 4 shows four attributes with the label increase and four with label decrease. As a consequence of our restriction on super calls, we must no longer specify the label of a super call. We give each attribute-body an alias to be able to easily reference them later in this text. As will be apparent from the context, the alias will sometimes stand for the entire attribute instead of only the body.

Alias	Attribute bodies with label increase
basic-increase	[intensity:=intensity+delta; delta:=0]
increase-one	[delta:=delta+1; super]
monitorincrease	[super; intensity display]
boundedincrease	[intensity<10 ifTrue:[super] ifFalse:["error" display]]

Alias	Attribute bodies with label decrease
basic-decrease	[intensity:=intensity-delta; delta:=0]
decrease-one	[delta:=delta-1; super]
monitordecrease	[super; intensity display]
boundeddecrease	[intensity>0 ifTrue:[super] ifFalse:["error" display]]

figure 4: attributes for a regulatable intensity

6.1 Coexistence

Grouping attributes can fine-tune the regulating mechanism. For example by putting basic-increase and basic-decrease together in a component we guarantee that every intensity device will be both increasable and decreasable. In this case we will call basic-increase and basic-decrease to be coexistent.

Since components are detached from any class hierarchy restrictions, repeated inheritance is naturally present: We can use the same component multiple times in the construction of a class. For instance it makes quite sense to add the increase-one and decrease-one attributes several times to a class: a class that contains increase-one twice represents a machine that increases the intensity with two units for one push on the increase button (see figure 5).

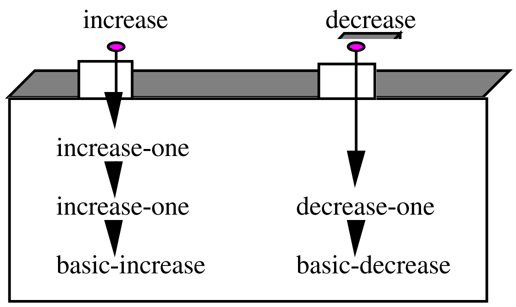


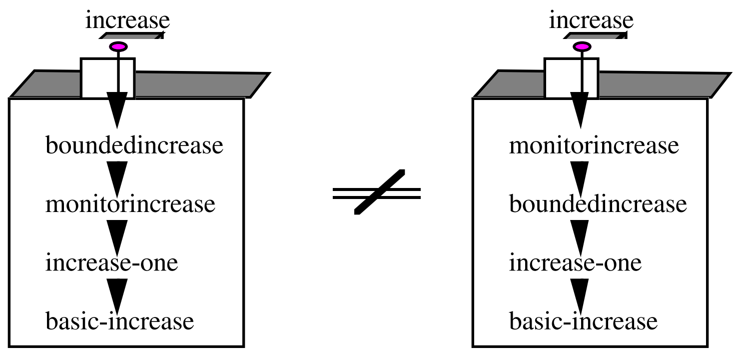
figure 5: a class where increasing goes twice as fast as decreasing

Repeated inheritance renders coexistence more significant. Putting increase-one and decrease-one together in one component enforces that, in a single device, pushing the increase or the decrease button alters the intensity with the same amount. Consequently the class in figure 5 can not be constructed with the two components {basic-increase, basic-decrease} and {increase-one, decrease-one}. We conclude that we can only group attributes in a component if each of these attributes shows up the same number of times in every class of our intended class space. Let's take this attribute property as more general meaning for coexistence.

6.2 Interleaving

Consider the other attributes in figure 4. monitorincrease and monitordecrease print the current intensity on a display. boundedincrease and boundeddecrease limit the intensity and print "error" on the same display when one tries to exceed the limit. As concluded above, we should not put boundedincrease and boundeddecrease in one component if for example our UoD contains devices with only a lower-bounded intensity.

A second import of grouping attributes arises because inheritance composition is not commutative: the order in which attributes are added to classes is significant for the resulting behaviour. Instances of the class on the left in figure 6 will permanently (or until the intensity is decreased) show "error" on the display when one tries to exceed its upper-limit. Devices corresponding to the right class will print in such case the current intensity on the display ("error" will only appear for a moment).



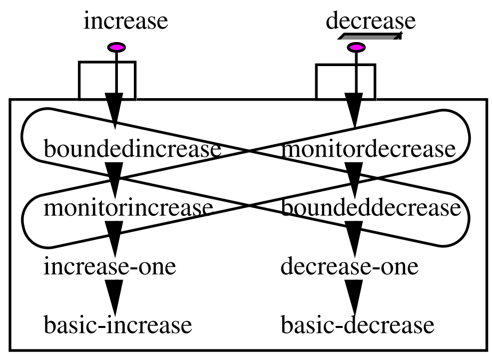


figure 7: crossed monitor and bound behaviour

Due to this non commutative character of inheritance, the class in figure 7 shows "error" on exceeding the upper-bound and "0" on exceeding the lower-bound. We can avoid our devices to have a different bound treatment for increasing and decreasing by putting the two bound-attributes in a single component and the two monitor-attributes in another one: {monitorincrease, monitordecrease} and {boundedincrease, boundeddecrease}. This way crossed monitor and bound behaviour can no longer be constructed. Therefor the monitor and bound attributes will be said to "interleave".

As a final illustration, figure 8 represents a possible library containing the attributes of figure 4. This library induces only classes where increase and decrease alter the intensity with the same amount and have the same limit and monitoring treatment.

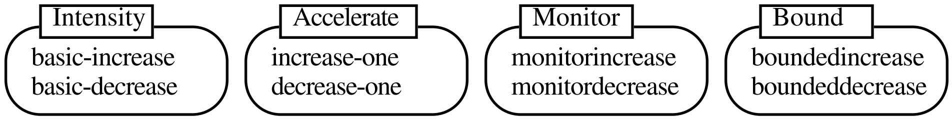


figure 8: components for classes with uniform increasing and decreasing

7. Design level dependencies

In this section we will formally introduce the design information encountered in the example of the previous section. Let's call **Classgoal** $\subseteq$ **Classes** the set of all intended classes, i.e. the class space according to our UoD.

Mathematically we will express attribute properties on sets of natural numbers. A set of numbers corresponds to the indices of an attribute in a class. The projection of a class c on an attribute a, noted π(c, a) is defined as follows:

Definition: $\pi(cl, a) = \{j \in N \mid (cl(a_{label}))(j) = a_{body}\}$

For example π(class figure 5, increase-one) = {1,2}.

Definition: Two attributes a and b are coexistent

$$\Leftrightarrow$$

$$\forall\ cl \in \textbf{Classgoal}:\ \#\pi(cl, a) = \#\pi(cl, b)^3$$

We will also need projection on numbers to formalise the notion of attribute interleaving. The property that in some class cl the bound and monitor attributes interleave, will be expressed on numbers:

$(\pi(cl,\text{boundedincrease}), \pi(cl,\text{boundeddecrease})) \approx (\pi(cl,\text{monitorincrease}), \pi(cl,\text{monitordecrease}))$

For example the fact that in figure 7 the bound and monitor attributes do <u>not</u> interleave, will be indicated by

$$\neg\ (\{3\},\{2\}) \approx (\{2\},\{3\})^4$$

In figure 7 this corresponds to the crossing of the grouping ovals. Interleaved behaviour can in general also occur with attributes destined for repeated inheritance. Therefor the ≈ relation should indeed be defined on pairs of sets of numbers, and not on pairs of numbers. Figure 9 illustrates the working of ≈ for repeatedly inherited attributes.

---

[3] #X denotes the number of elements in the set X.
[4] ¬ denotes not.

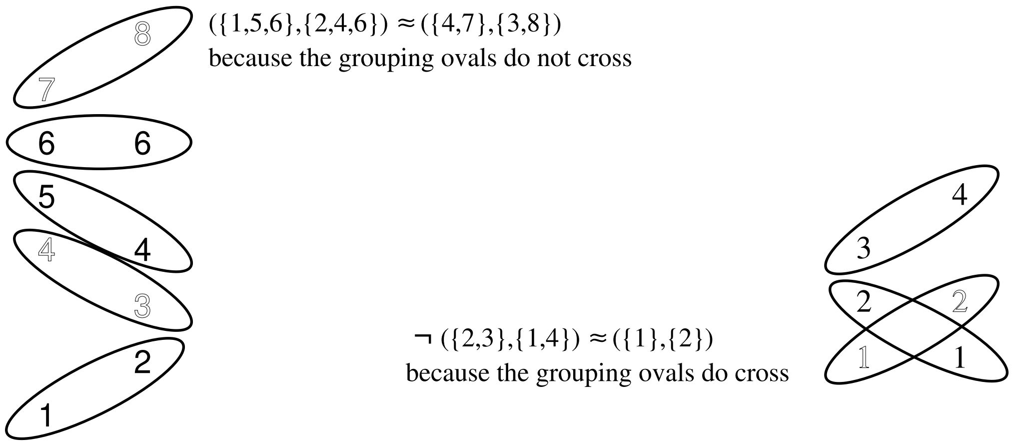


figure 9: two examples of the ≈ relation

Definition: The ≈ relation is defined by induction:

$(\emptyset, \emptyset) \approx (\emptyset,\emptyset)$

$\forall\ X, Y, X', Y' \subseteq N, \forall\ x, y \in N$:

$x < \min(X)$[5], $y < \min(Y),\ x \le \min(X'),\ y \le \min(Y')$ and $(X,Y) \approx (X',Y')$

$\Rightarrow (\{x\} \cup X\ ,\ \{y\} \cup Y) \approx (X',Y')$

$\forall\ X, Y, X', Y' \subseteq N$:

$(X,Y) \approx (X',Y') \Rightarrow (X',Y') \approx (X,Y)$

Later on we will need the following two properties of the ≈ relation.

Property3: $\forall\ X, Y, X', Y' \subseteq N$:

$(X,Y) \approx (X',Y') \Rightarrow (Y,X) \approx (Y',X')$

Property4: $\forall\ X, Y \subseteq N: \#X = \#Y \Leftrightarrow (X,Y) \approx (X,Y)$

Note that as a consequence of property4, the ≈ relation is not reflexive. It is neither transitive:

$(\{2\},\{3\}) \approx (\{1\},\{1\})$ and $(\{1\},\{1\}) \approx (\{3\},\{2\})$ but $\neg\ (\{2\}, \{3\}) \approx (\{3\},\{2\})$

Interleaving, until now only defined on sets of numbers, will be translated to attributes. We will only retain interesting cases with regard to attribute grouping as expressed in the following long condition:

5 min(X) denotes the minimum of X.

Definition: Let (a,b) and (a',b') $\in$ **Attributes** $\times$ **Attributes**, then

$$(a,b) \textit{ interacts with } (a',b')$$

$$\Leftrightarrow$$

$$a_{label} = a'_{label} \text{ and } b_{label} = b'_{label} \text{ and } a_{label} \neq b_{label} \text{ and } ((a \neq a' \text{ and } b \neq b') \text{ or } ((a,b) = (a',b')))$$

Definition: Let (a,b), (a',b') $\in$ **Attributes** $\times$ **Attributes** such that (a,b) interacts with (a',b'), then

$$(a,b) \textit{ interleaves } (a',b')$$

$$\Leftrightarrow$$

$$\forall \text{ cl} \in \textbf{Classgoal}: (\pi(cl, a), \pi(cl, b)) \approx (\pi(cl, a'), \pi(cl, b'))$$

Corollary1: direct consequence of property3: $\forall$ a, b, a', b' $\in$ **Attributes**:

$$(a,b) \text{ interleaves } (a',b') \Leftrightarrow (b,a) \text{ interleaves } (b',a')$$

Corollary2: direct consequence of property4: $\forall$ a, b $\in$ **Attributes** with $a_{label} \neq b_{label}$:

$$(a,b) \text{ interleaves } (a,b) \Leftrightarrow a \text{ and } b \text{ are coexistent}$$

Corollary2 shows that our design relation, initially intended to express interleaving, also catches coexistence. With other words, coexistence is only a special kind of interleaving, namely interleaving of a pair of attributes with itself.

> *The interleaves relation is similar to functional dependencies in relational databases in the sense that they both describe restrictions: on the intended classes respectively populations. In databases multivalued dependencies also represent restrictions, but are context dependent: they are a characteristic of a certain grouping. The interleaves relation on the contrary does not depend on the grouping, just like functional dependencies.*
> *Certain functional dependencies between the attributes of a table indicate that the table can be split "lossless" in subtables:* joining *the subtables yields the correct result for every intended population In our case certain interleave dependencies between attributes indicate that some subcomponents can be "lossless" united in one component : all intended classes can correctly be constructed by* composing *with the single component, as well as by composing with the subcomponents separately.*

8. Normal form

From the above we can recapitulate that there are three reasons not to group two attributes:

- the attributes have the same label: this is inherent to components
- the attributes show up in some class a different number of times (will be formally translated via corollary2)
- the two attributes appear in some class in crossed behaviour with two other attributes that are already grouped.

These three conditions are formally repeated in the following definition:

Definition: Two attributes a and b are *ungroupable for* a library L

$$\Leftrightarrow$$

$$a_{label} = b_{label} \text{ or}$$

$$\neg((a,b) \text{ interleaves } (a,b)) \text{ or}$$

$$\exists\, c \in L: \exists\, a', b' \in c \text{ such that } (a,b) \text{ interacts with } (a',b'): \neg((a,b) \text{ interleaves } (a',b'))$$

Thanks to corollary1 the relative order of a and b is unimportant in this definition of unconnectable.

Finally a library will be defined to be in 2NF if any further attribute grouping, i.e. uniting two components into a single one, is impossible because the two components contain unconnectable attributes:

Definition: A library L is in 2NF

$$\Leftrightarrow$$

$$L \text{ is in 1NF and}$$

$$\forall\, c_1, c_2 \in L \text{ with } c_1 \neq c_2: \exists\, a \in c_1, \exists\, b \in c_2: a \text{ and } b \text{ are ungroupable for } L$$

In relational databases a normal form is a property of a table: further splitting *the table does no longer avoid a certain type of redundancy. In our case the normal form indicates that further* grouping *can no longer avoid a certain category of unwanted classes. But further grouping depends on the other groups in the library, in contrast to splitting that is totally independent from the other tables in the database. Hence in our case a normal form is not a property of a component, the counterpart of a table, but of an entire library.*

The complexity of this normal form is comparable to that of the 3NF and fortunately not the 4NF in relational databases in the sense that it is only based on (grouping-)context free design information.

This normal form does not mathematically require that each class of our Classgoal can be constructed, although it is only intended to be used for libraries that do so. This approach is copied from the relational database world.

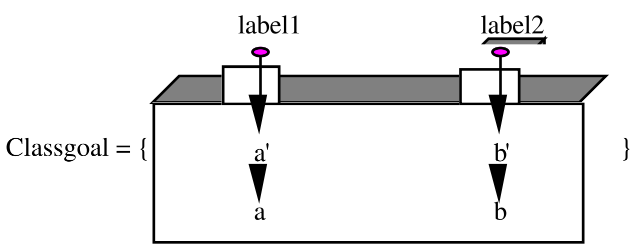
Two ungroupable attributes may not reside in the same component. If not, some class of our **Classgoal** can no longer be constructed. Formally:

Property5: If a library L_1 is in 2NF, then

$$\forall \text{ library } L_2 \neq L_1 : (L_2 \text{ further groups } L_1 \Rightarrow \textbf{Classgoal} \not\subset \text{classSpace}(L_2))$$

This property proves that our 2NF is the "last" normal form. In relational databases the last normal form is often called the 5NF: further splitting is impossible since some intended populations could no longer be stored. In contrast with functional dependencies that do not catch all possible data redundancies, our interleaves relation is complete.

As illustrated in figure 10, different libraries, each with an induced class space that covers our entire **Classgoal**, can be in 2NF. The one is not objectively better than the other; both express other restrictions on the possible classes. This holds even in general as stated formally in property6.



Both basismixinsets { {a,b}, {a',b'} } and { {a,b'}, {a'}, {b} } are in 2NF.

figure 10: a singleton **Classgoal** and two libraries in 2NF with an induced class space covering this **Classgoal**

Property6: If L_1 and L_2 are two different libraries both in 2NF and **Classgoal** $\subset$ classSpace(L_1) and **Classgoal** $\subset$ classSpace(L_2) then

$$\text{classSpace}(L_1) \not\subset \text{classSpace}(L_2) \text{ and } \text{classSpace}(L_2) \not\subset \text{classSpace}(L_1)$$

In relational databases on the contrary, two databases containing only tables in 3NF (or in 4NF) can quantitatively differ. The difference stems from the transitivity of functional dependencies (in section 7 interleaves is shown to be not transitive). This transitivity causes the order in which functional dependencies are used as motivation for splitting, to be important. Hence it is interesting to search a database normalising algorithm that finds the best *grouping. In our case however it is rather trivial to algorithmically construct all quality-equivalent 2NF grouping combinations.*

9. Practical use and future work

The exact significance of attribute grouping, as determined in this paper, is of course a very important knowledge for a class hierarchy designer. But besides this knowledge aspect, it is interesting to find out if automatic normalisation can be made practically useful. This highly depends on the comprehensibility of the associated design information, since it is this information that must be specified by the designer. Our interleaves relation though causes a mass of pairs of attributes to be tested on their interleaving, and can not directly be considered as an exemplar of comprehensibility.

> *In relational databases determining* all *the functional dependencies is also a tremendous task: a lot of them will only afterwards be discovered to be redundant. Normalisation algorithms directly starting from functional dependencies are not frequently used in practice.*

However, the properties of the interleaves relation can be used to reduce the number of pairs to be tested. For example first determining the "coexistent" attributes will discharge us to from testing a lot of pairs (thanks to Corollary2). Other properties allow to automatically complete a partially defined interleaving (Corollary1 for example).

On top of this we can add supplementary design level attribute properties that further eliminate attribute pairs to be considered. For example the attribute property "always precedes". basic-increase for instance always precedes an increase-one . Such properties interfere with class hierarchy restrictions. Future research is required to determine if some of these restrictions should be put on attribute level to guide component normalisation.

On one hand we intend to search the most opportune properties for a designer to specify. On the other hand we will try to develop tools that guide the designer in ordering his specifications, such that design dependencies can be automatically deduced as much as possible. This way the interleaves relation, on its own rather large and complicated, could be constructed in a simplified way.

10. Conclusions

Class hierarchies put restrictions on the way in which class components can be combined, prohibiting some unwanted classes to be constructed. But a more primitive way of influencing the induced class space, is the distribution of attributes over components. We stressed the significance of this distribution.

In order to clearly indicate the restrictions implied by attribute grouping, we chose an inheritance medium without any component combination restriction, namely mixin-based inheritance. By restricting super calls, argued to be a good programming style rather than a real restriction, we obtained that further uniting attributes always restrains the induced class space. This monotone character of attribute grouping allowed normalisation reasoning.

We exactly determined which constraints are enforced by attribute grouping, informally by means of an example, and formally under the form of design level dependencies. These dependencies can be used by a class designer as UoD information to be filled in. Or the other way around, a library user can read them as a description of the induced class space of the library.

Moreover we defined a normal form on class libraries. This normal form indicates that attribute grouping has been performed as far as needed, according to the given UoD information. We clarified our approach by a comparison with normalisation of relational database tables.

11. Acknowledgements

We want to express our gratitude to Serge Demeyer, Wolfgang De Meuter, Carine Lucas, Kim Mens and Tom Mens for their useful comments on earlier versions of this paper.

References

[Bracha&Lindstrom92] G. Bracha and G. Lindstrom: Modularity meets Inheritance. In Proc. of OOPSLA '92 Conference Proceedings, pp. 282-290, ACM Press.

[Bracha&Cook90] G. Bracha and W. Cook: Mixin-based Inheritance. In Proc. of OOPSLA/ECOOP'90, pp.303-311, ACM Press.

[Godin&Mili93] R. Godin and H. Mili: Building and Maintaining Analysis-Level Class hierarchies Using Galois Lattices. In Proc. of OOPSLA'93, pp. 394-410, ACM Press.

[Hamer&al92] Hamer, J, Hosking, J G and Mugridge W B: Static Subclass Constraints and Dynamic Class Membership Using Classifiers. Technical Report, University of Auckland, Computer Science Department.

[Hawryszkiewycz91] I.T. Hawryszkiewycz: Database Analysis and Design. Second edition. Macmillan Publishing Company, ISBN 0-02-35-1851-0

[Lammari&al94] N. Lammari, R. Laleau, M. Jouve and X. Castellani: An Algorithm for IS_A Hierarchy Derivation. In Proc. of OOIS'94, pp 469-479, Springer-Verlag.

[McGregor&Korson93] J.D. McGregor and T. Korson: Supporting dimensions of classification in object-oriented design. In Journal of Object Oriented Programming, pp. 25-30, SIGS Publications.

[Rumbaugh&al91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: Object-Orient Modeling and Design. Prentice-Hall International Editions, ISBN 0-13-630064-5