A Disciplined Approach to the Maintenance of the Class Hierarchy

Anne Eerola[1]

April 1, 1999

Abstract

Encapsulation and the definition of objects in the respective classes facilitates the modification of object-oriented systems, but the class hierarchy may also be a source of problems. First, before changing the definition of a class, the analyst must verify that the change is coherent with regard to subclasses of the class recursively. Second, the reusability of the properties and the uniformity and functionality of the system can be increased by defining the properties as high as possible in the class hierarchy. Consequently, the maintainer has to navigate up and down while investigating and implementing the changes in the class hierarchy. In this work it is shown that the maintenance is a disciplined process that can be well supported with object-oriented algorithms.

Index Terms - object-oriented, maintenance, object, class, class hierarchy, inheritance, reuse.

I Introduction

1.1 Maintenance of an object-oriented system

Maintenance is the stage in the life of a software system, during which corrective, adaptive, preventive and perfective changes are made to the software product [1, 2]. It is the most expensive stage in the software life cycle; about 50%-70% of the software cost is incurred during this phase [3, 4]. Changes in the system cannot, however, be avoided, because the environment of the system is changing. Furthermore, each system has errors and these must be corrected.

The *object-oriented* approach [5] supports the maintenance process: The objects discovered in the analysis stage survive in the design, implementation and maintenance, thus the number of transformations between different stages decreases. For this reason the maintainer (the person who makes changes in the system) knows why an object exists in the code, and is better informed when it should be removed or modified [1]. Due to *encapsulation* [6, 7], necessary changes can be better localized and thus maintainer can more easily understand the effect of the change.

In an object-oriented approach, objects are defined in classes, which form the *class hierar-*

[1] E-mail Anne.Eerola@uku.fi

chy [8]. The class hierarchy allows subclasses to inherit [9] *properties*, i.e. attributes and methods, from superclasses. This increases the reusability of the definitions of the properties and thus the amount of code to maintain reduces.

However, the changes can not be readily implemented. Before the class is changed, the designer must check that the change is acceptable to the clients of the class. Because of *polymorphism,* the name of the property does not uniquely identify the property. Thus the clients of the class are hard to find. In fact, an object-oriented system has more dependencies between its entities than a conventional one [10]: class to class, class to attribute, class to method, class to message, method to attribute, method to message, and method to method. All these dependencies complicate the maintenance process. On the other hand, polymorphism helps in the maintenance, because properties with the same meaning have the same name. This contributes to comprehensibility.

Furthermore, the object-oriented approach poses problems to the maintenance process. First, the best choice for modelling a real-world entity is not always readily apparent [11, 12]. Second, an object-oriented system is usually made up of a large number of small modules. Understanding a single line of code may require tracing a chain of method invocations through several different object classes, as well as moving up and down the class hierarchy, to find where the work has really been done [10]. Third, *dynamic binding* [3] means that it is not possible to know by reading the program code to which object the message will be really sent when the method is executed. The *yoyo problem* [13] is related to objects sending themselves messages which may cause the execution of methods up and down the class hierarchy. This leads to greater difficulty in modifying the implementation of classes in the middle of the class hierarchy [13].

1.2 Goals of this study

This research emphasizes the maintenance of the class hierarchy, when either single or multiple inheritance is used. The goal is to support the maintainer's work in fulfilling the changes defined in user requirements (perfective maintenance). It is assumed that several changes are accomplished during the same maintenance process.

The maintenance process is divided into four steps:

- The analyst *collects* the change requests of users.
- The analyst *translates* change requests of users into changes of the objects.
- The automatized tool *browses* through the class hierarchy and collects information about how wide in the class hierarchy the changes of the objects can be utilized. During this navigation the tool prompts the analyst for decisions presenting details to help him/her make the decision.
- The analyst selects the best way to *implement* the changes by using her or his understanding of the domain and the information collected with the tool.

We will present object-oriented heuristic algorithms and procedures for the maintenance process. The algorithms have been implemented in a CASE tool that helps the analyst to navigate in the class hierarchy [14].

The approach presented in this research differs from the previous following ways: Casais has presented algorithms for restructuring the class hierarchy [11, 15, 16], but his work emphasizes rather the reorganization of the class hierarchy of a particular object-oriented language than the maintenance of the class hierarchy according to change requests of the users. Bergstein [17] has presented a list of class transformations for improving class hierarchies but the order of the transformation operations is not considered, which is the case in this work. Lieberherr et al. [18] have presented algorithms for abstracting class definitions from a representative sample of *object examples*. Thus they consider derivation of the class hierarchy, not the maintenance of an existent class hierarchy. Ossher and Harrison [19] have presented an approach in which extensions and changes are clearly separated from the *base class hierarchy* upon which they are built. Chen et al. have presented an inheritance flow model [34].

1.3 Concepts

The object-oriented approach is characterized by the use of *objects*, each of which possesses a *state* (consisting of the values of its *attributes*), a *behaviour* (*methods*), and an *identity.* In the following, the attributes and methods are called *properties*. The attribute of the object can be an object. The objects are defined in a *class*, so that objects sharing a common definition belong to the same class, while objects resembling each other belong to the same *class hierarchy.* An object 1) responds to service requests sent from other objects, 2) controls its own state by changing the values of its attributes, 3) executes its methods, for example, by *sending messages* to other objects or by performing the task by itself.

The class hierarchy allows *subclasses* to inherit properties, i.e. definitions of attributes and methods, from *superclasses*. In *single inheritance,* a class may have only a single superclass, whereas in *multiple inheritance* a class can have more than one superclass.

In some object-oriented languages it is assumed that the class inherits all the properties of its superclass; and in some other languages, usually in those with multiple inheritance, the inherited properties and the ancestor classes are explicitly defined in order to avoid name conflicts. If the inherited properties are not defined explicitly, the analyst must search the program code for the properties that are inherited. For this reason, it is assumed here that the properties of the class are explicitly defined in one of the following places:

- The property is defined in an ancestor class of the object and need not be changed in the class of the object *(strictly inherited)*.
- The property is initially defined in an ancestor class of the object but is subsequently changed in the class of the object (*overridden*) [9].
- The property is defined in the class of the object (*self-defined*).

Generally speaking, a class may have objects and subclasses. For clarity and simplicity of algorithms, we assume that each class is either *abstract* (has subclasses) or *concrete* (has objects).

1.4 Outline

The rest of this work is organized as follows: Section 2 starts with introductory examples. Section 3 considers the general principles of maintenance. The phases preceding the maintenance process and the characteristics of high quality class hierarchy are described. Similarly, several metrics for evaluating the class hierarchy are briefly reviewed. In Section 4 the maintenance process is described. First, the investigations of concrete and abstract classes are considered separately. Second, the principle of the navigation in the class hierarchy is presented. Third, the implementation of applicable changes is discussed**.** Section 5 compares these results with related work. Section 6 is a summary of the results.

2 Introductory example

Before a detailed presentation of the maintenance process, the general principle of the algorithms is illustrated with simplified examples. In the class hierarchy diagrams, the classes are drawn as labelled rounded rectangles. The superclass is drawn above its subclasses and the generalization/specialization relationship is presented by a line that joins a subclass to its superclasses. For the purpose of illustration, the OBJECTS are written in capitals, the methods of objects in boldface and the attributes of objects in italics.

Example 1: The class hierarchy of Figure 1 describes the staff of the university. For reasons of simplicity all the attributes and methods of the classes are not presented. In this example, we suppose exceptionally that the subclass inherits all of the properties of its superclasses and that class STUDENT is both abstract and concrete.

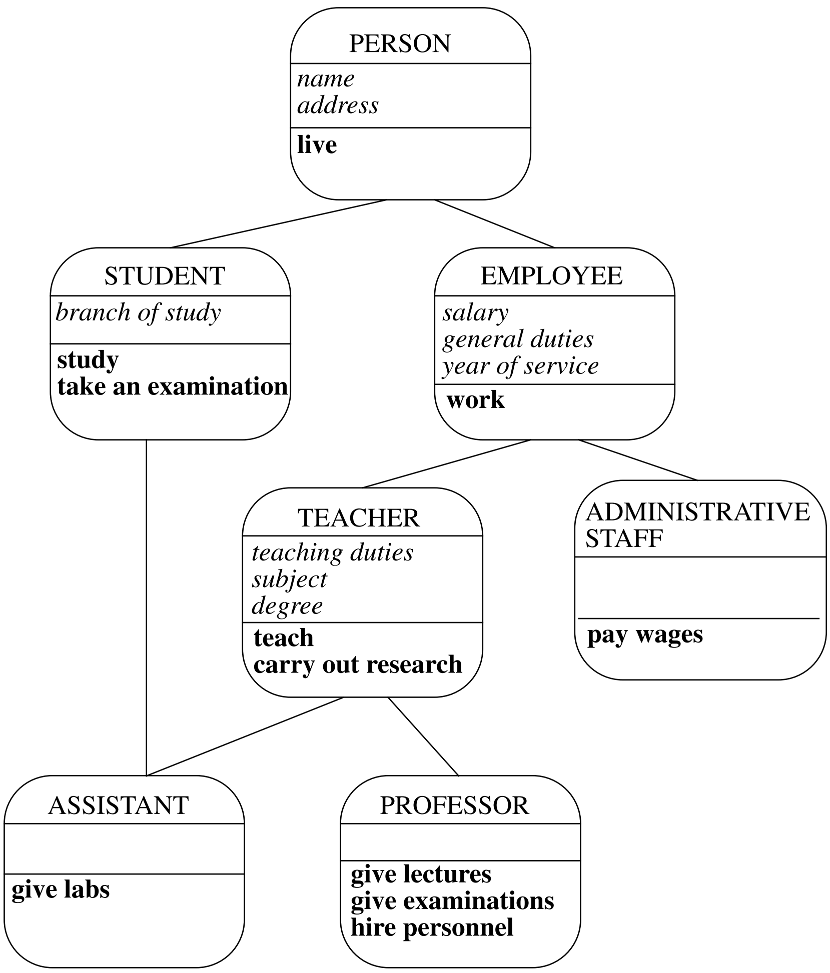

Figure 1: Class hierarchy of an information system called UNIVERSITY STAFF

Suppose now that the user of the system needs several changes and consider the observations that will be made:

a) The first change: assistants give examinations not only professors:
- The property **give examinations** is applicable to PROFESSOR and ASSISTANT. Thus it can be moved from class PROFESSOR to superclass TEACHER and deleted from class PROFESSOR that can inherit the property.

b) The second change: insert a new method **take a holiday** in class STUDENT:
- The method is applicable to subclass ASSISTANT, thus it can be defined in class STUDENT and inherited in class ASSISTANT.
- For increasing reusability it is wise to consider if the property can be defined in superclass PERSON.
- Class PERSON has subclasses, thus we must go down the class hierarchy until a concrete class is reached.
- The property is found to be applicable to class PROFESSOR and because we already know that it is applicable to class ASSISTANT the method can be defined in class TEACHER.
- The property is also found to be applicable to class ADMINISTRATIVE STAFF. Thus the method **take a holiday** can be defined in class EMPLOYEE and finally in class PERSON.
- The last problem is that we should make sure that class ASSISTANT inherits this property only from one class - from class STUDENT or class TEACHER.

c) The third change: insert a new method **obtain a grant** in class STUDENT:
- This method is not applicable to class ASSISTANT (assistants have a salary), thus we must prevent class ASSISTANT from inheriting the method, when we define it in class STUDENT.

3 General principles

3.1 Properties of the high quality class hierarchy

While designing and maintaining the class hierarchy, the analyst should increase the uniformity, functionality, reusability, and comprehensibility of the system. At the same time, the amount of the code should be decreased leading to decreased implementation and maintenance costs. Consequently, the maintenance of the class hierarchy is a multioptimum task where the criteria may be in conflict with each other. For example, the use of multiple inheritance increases the reusability but at the same time it may decrease the comprehensibility. It is not possible to collect all of the above quality factors into the same function. Instead of doing this, we present heuristic algorithms which emphasize the above optimization criteria. The idea here is that the class hierarchy is investigated, information about it is collected, and metrics which provide information about its quality are calculated. After that the maintainer evaluates different solutions and selects the best way to implement the changes needed.

The comprehensibility of the system strengthens if the class hierarchy models the domain in a natural way: First, there should be a "kind of" or "is a" relationship between a class and its subclasses, i.e. a superclass is a generalization of its subclasses. If the inheritance is used only to avoid code duplication, this relationship will be lost and the class hierarchy becomes confusing and very difficult to maintain. Second, the class hierarchy should not be too deeply nested by inserting too many abstract classes [20]. Third, if several properties are overridden or changed in the subclass level, there will be a decrease in understandability.

The uniformity of the system means that similar objects behave similarly. This increases the comprehensibility and maintainability of the system. Functionality of the system means that the customer gets more properties. Both of these criteria increase if the properties are defined as high and as generally as possible. The reusability of the system increases if the properties are defined at a higher level, but an excessively high level may restrict the reusability [21].

3.2 Local metrics

The quality of the class hierarchy can be evaluated by using several metrics. In this study we use the metrics of Chidamber and Kemerer [22, 23] as a starting point. These metrics are designed especially for object-oriented systems and their theoretical basis is well stated. Furthermore, the metrics have been empirically investigated and they appear to be useful to predict class fault-proneness during the early phases of the software life-cycle [24]. The metrics describe the quality of the class hierarchy. However, the most important guideline is, whether the hierarchy models the domain in a natural way.

For each class several metrics are calculated:

The *weighted attributes per class* and the *weighted methods per class* are indicators of the size of the class. When calculating these metrics, the maintainer should use different weights for the strictly inherited, overridden and self-defined properties. The weights of the self-defined properties are greatest, whereas those of the inherited properties are at a minimum. A large value of the metric the *number of overridden properties* indicates design problems and decreased maintainability. In contrast, the number of inherited properties indicates the strength of the subclassing [25]. Furthermore, it is possible to consider the *number of public methods* (the methods that are visible to all other classes) that tells about the responsibility of the class. To decrease the coupling, *the number of class methods and class attributes,* used for example in Smalltalk, should be minimized.

All the subclasses of a class can inherit the properties defined in the class. If the class has many descendants, the properties defined in the class can be widely reused, but the properties must also be designed more carefully. The *number of subclasses* describes this aspect. The *depth of the inheritance tree* (the average depth of different paths from the considered class to the root classes) informs about the complexity of the class, thus it should be minimized.

The object can be used as a property in several objects. The *number of objects in which the object is used as a property* tells about the usability of the object. However, if the object is used in several objects it will require more careful design.

Finally, the maintainer should consider the *cohesion of the objects* from the program code. The cohesion is best if all the methods of the object use all of the attributes of the object and worst if all the methods have their own attributes [22, 26, 27]. In this latter case, it may be profitable to split the object and the class that defines the object. Splitting the object results in multiple identities, that might be wrong from a modelling point of view.

3.3 Global metrics

Maintenance decisions cannot be made according to local metrics alone. Instead, the maintainer needs global metrics that describe the whole class hierarchy. For each of the local metrics presented above, it is possible to calculate the global one automatically by going through the whole class hierarcy and summarizing the metrics of each class. This calculation leads, for example, to the *weighted methods per class hierarchy,* the *number of overridden properties per class hierarchy*, and the *average number of subclasses.* The *average number of methods and attributes* is utilized while comparing the classes with each other. It is recommended that objects be designed equally intelligent [28] so that each object has on an average same number of methods. This reduces pluggability. The *average depth of the class hierarchy* is the average depth of inheritance trees of concrete classes. Of course, the *number of abstract classes,* and *the number of concrete* *classes* describe the total class hierarchy.

The *tree impurity* [29] describes the extent to which a graph deviates from being a tree. System designs should ideally strive for a value near to zero but not at the expense of unnecessary duplication of code. The other possibility to measure the complexity of the class hierarchy is the method presented in [30].

4 The maintenance process

4.1 Analysis of the new requirements

In this section we briefly consider those analysis steps that the maintainer accomplishes before she or he starts the maintenance of the class hierarchy. These steps depend on the analysis methodology used (for example [8, 20, 28, 31, 32]. In this research, we are not restricted to any particular methodology. The aim is to clarify what properties of the objects should be fixed before the maintenance of the class hierarchy starts.

It is wise to maintain an information system by versions, because this provides a clear understanding of the changes needed. Another advantage is that the necessity of changes is more carefully evaluated, and the analyst can make a set of changes in the same walk-through of the class hierarchies (see [15] for a discussion of version management).

The change requests by customers concentrate on enhanced or changed functionality and features of the system. Using this information the analyst chooses how the changes are carried out. The changes may cause:

- changes in the aggregation and in the association relationships between objects,
- changes in the message sending between objects,
- modification, insertion and deletion of properties of the objects,
- insertion and removal of objects.

Here it is assumed that the maintainer starts the maintenance by investigating the system specifications. From these documents she or he obtains information about the structure and properties of the system and can choose the objects to be changed.

The association relationship between objects is implemented as an attribute [20] which contains an object reference. The aggregation relationship between objects is defined by collecting the parts as an object (whole). In fulfilling a method, the object has two possibilities: it can fulfil the task by itself *(self-fulfilled)* or it can delegate the task to other objects *(delegated),* which can be *external* or *internal*. Those attributes of the object that are objects are called internal objects. An external object is limited outside the object.

While designing the message sending between objects, the maintainer should consider several quality aspects: First, the message structure should correspond to the analyst's view of the real world. Second, it is useful to rationalize the message sending. Third, the *coupling* should be decreased, i.e. if any two objects communicate, they should exchange as little information as possible [2, 26]. Fourth, long chains of messages should be avoided [21]. Thus errors are easier to find and correct when the system is executed. Fifth, the Law of Demeter restricts the number of objects to which an object can send a message [23, 33].

Methods of the object are derived on the basis of the messages that the object receives.

The term *cohesion* is used to describe how well individual tasks are unified within a method. Higher cohesion leads to the easier maintenance of the system and to the greater reusability of the objects and their methods. If a particular message leads to a method that is too “wide”, i.e. it does too many functions that have an unacceptable level of cohesion between them, the method and the message must be split [21, 26].

As result of the above analysis the maintainer 1) designs the changes of the objects, 2) designs the properties of new objects, and 3) selects the objects that can be removed. The objects are defined in concrete classes. Thus the analysis of the new requirements leeds to the definition of changes needed in concrete classes.

4.2 Investigation of the applicability of the changes

During analysis of the new requirements the analyst specifies the desired changes needed to be done in concrete classes, but she or he does not know how the implementation of changes effects on the whole class hierarchy.

Each object is defined in the class. The changes in the properties of a class affect the relationships between classes. Thus the maintenance of objects may cause:

- modification of properties, i.e. attributes and methods, defined in a class,
- insertion and deletion of classes in the class hierarchy,
- joining or splitting classes,
- moving properties up and down in the class hierarchy,
- changing the superclasses of a class and
- changing the subclasses of a class.

A maintainer may think that changes can be readily implemented by inserting a new subclass into the class hierarchy and defining the change in that subclass. This approach, however, is only acceptable in the case of minor changes, because 1) the class hierarchy may become confusing, 2) the same properties may be defined in many places, 3) the system may become overloaded with special cases [15], and 4) the maintainability of the system may decrease. Thus before the analyst change the properties of a class, she or he must consider what effects this process will have on subclasses and on superclasses.

For supporting the maintainer’s work we have implemented a CASE tool [14] which helps the maintainer to collect information about the applicability of the changes in the classes, for example about how wide in the class hierarchy a certain property is used.

First, the analyst gives as input to the tool the desired changes of the concrete classes. Second, the tool browses through the whole class hierarchy and collects information about how wide in the class hierarchy the changes can be utilized. At each stage the tool tries to conclude the properties applicable for classes and prompts the analyst for decisions when required. The tool proposes for each class the changes to be implemented, the changes that can possibly be implemented in the superclass level, and the properties that can be inherited after changes. Third, the tool calculates the local and global metrics that describe the quality of classes and the class hierarchy.

After this information has been collected, the maintainer decides utilizing domain knowledge how the changes will be implemented in order to increase functionality, reusability, maintainability and comprehensibility and to decrease the amount of code. In doing this, the local and global metrics of the class hierarchy are utilized.

4.2.1 Abstract and concrete classes

In this section we consider information collected from each class. The process starts from concrete classes:

- The *applicable changes in a concrete class* are the changes needed in the objects of the class.

The applicability of the change may differ between object instances of the class and thus the analyst can distinguish the *instance dependent changes*. If the objects of the class differ from each other after changes, the maintainer must insert a new concrete class (or possibly new concrete classes) as a sibling (siblings) of the previous class.

A class can inherit properties that have been defined in the superclass level. Thus the *changes that must be implemented in the class,* both abstract and concrete, have the following characteristics:

- the changes belong to the applicable changes in the class, and
- the changes are not defined in the superclass level.

In order to increase the functionality, uniformity and maintainability, the properties should be defined as high in the class hierarchy as possible, without restricting the reusability. However, the changes in abstract classes must be coherent with the properties applicable in the subclasses and in the subclasses of the subclasses recursively. Those changes that are applicable to the most subclasses are tried to be defined in the superclass level. If it results in violations the tool prompts the maintainer to select the solution:

- The definition of the unsuitable property will be deferred at the subclass level.
- The superclass will have to be split (Section 4.2).
- The overridden or self-defined properties are inserted in the subclass so that the subclass adapts to the changes.

In the following, the changes that can possibly be defined in the superclass level are called accepted changes in the class and they are defined as follows:

- The *accepted changes in the clas*s, both abstract and concrete, are the set of properties with the following characteristics:
- the changes belong to the applicable changes in a class, or
- changes are not used in the class, or
- the class can adapt to the changes.

To avoid code duplication, the properties that are applicable in at least two subclasses can be defined in the superclass, if all of the subclasses accept the change. Furthermore, the class can have properties that the subclasses do not inherit. From the above it follows that:

- The *applicable changes in an abstract class* are properties with the following characteristics:
- the change belongs to the accepted changes in all subclasses of the class, and
- the change belongs to the applicable changes of at least two[2] subclasses of the class, or the change belongs to the applicable changes of the class itself.

For further details of the information collected from the classes see [21].

Example 2: Consider Example 1. The property **hire personnel** is accepted, applicable and implemented in class PROFESSOR. The property **give lectures** is not applicable to class ASSISTANT. It is, however, accepted, because class ASSISTANT does not use it.

[2] This is a parameter that can be chosen.

4.2.2 Navigation in the class hierarchy

In this section, navigation in the class hierarchy is considered. The principle is that the tool does the navigation, prompts the analyst for decisions when needed, presenting details to help him/her to make that decision, and concludes the applicable, accepted and implemented changes of the abstract and concrete classes as presented in section 4.2.1. The process starts at the nearest common superclass[3] of all the concrete classes that need to be changed. Thus the input of the process is the union of the changes needed in concrete classes. In the following this set is called the set of *investigated changes*. This set is given as a list of attributes and methods that should be removed, inserted or changed in concrete classes. The process investigates the applicability of changes to a class as well as to its subclasses and superclasses.

Figure 2 describes the navigation in the class hierarchy: The desired changes concern to classes A, B, C, D. Thus the process starts from the common superclass E. The steps of the navigation are considered in the following:

First, the subclasses of the target class (in Figure 2 class E) are examined by a depth-first search. The algorithm proceeds recursively deeper in the levels of the subclass hierarchy until a concrete class is found. When the lowest class in a path is found, the tool prompts the analyst for decisions about the applicability of investigated changes to this class. After all the concrete classes have been considered the algorithm goes upwards in the class hierarchy. The accepted and applicable properties for each superclass are defined from the properties that are accepted and applicable to its subclasses and the class itself. When the tool is not able to make decisions, it prompts the analyst for help. The numbers in Figure 2 describe the order of investigation.

Second, the analyst tries to define the properties of a target class in the superclass level (Step 2 in Figure 2). This approach is profitable if the properties are applicable to the siblings of the target class. To achieve this, the analyst divides the changes into disjoint parts; an attempt is made to define these parts at different superclasses. The tool browses to the superclasses of the class. Each superclass and its subclasses will be considered in turn. Because a superclass may have further superclasses, the process is recursive. When going upwards in the class hierarchy, however, it is typical that the number of investigated properties decreases, because the siblings restrict the possibilities to implement the properties at a higher level.

Multiple inheritance may cause some problems in the navigation process:

The first problem is that the analyst must choose the ancestor class for each inherited property. The changes in strictly inherited and overridden properties are usually defined in the previous ancestor classes. However, for previously self-defined and new properties, the definition of the possible ancestor class is more difficult. To do this, the analyst may possibly have to guess the ancestor. If the ancestor classes have relationships between each other, the classes should be considered in order from the most general to the least general; and if the definition is not possible in one ancestor class, the analyst can try the next ancestor class.

The second problem is that name conflicts may occur in the case of multiple inheritance. The properties inherited from the different superclasses should be exclusive. It is, however, profitable to delay these considerations and avoid name conflicts by restricting the properties that the class inherits only in the implementation phase. In this approach, the classes can inherit more properties, and the properties are defined at a higher level than in the case where name conflicts are considered immediately [21].

[3] If the common superclass is not found several class hierarchies are considered separately.

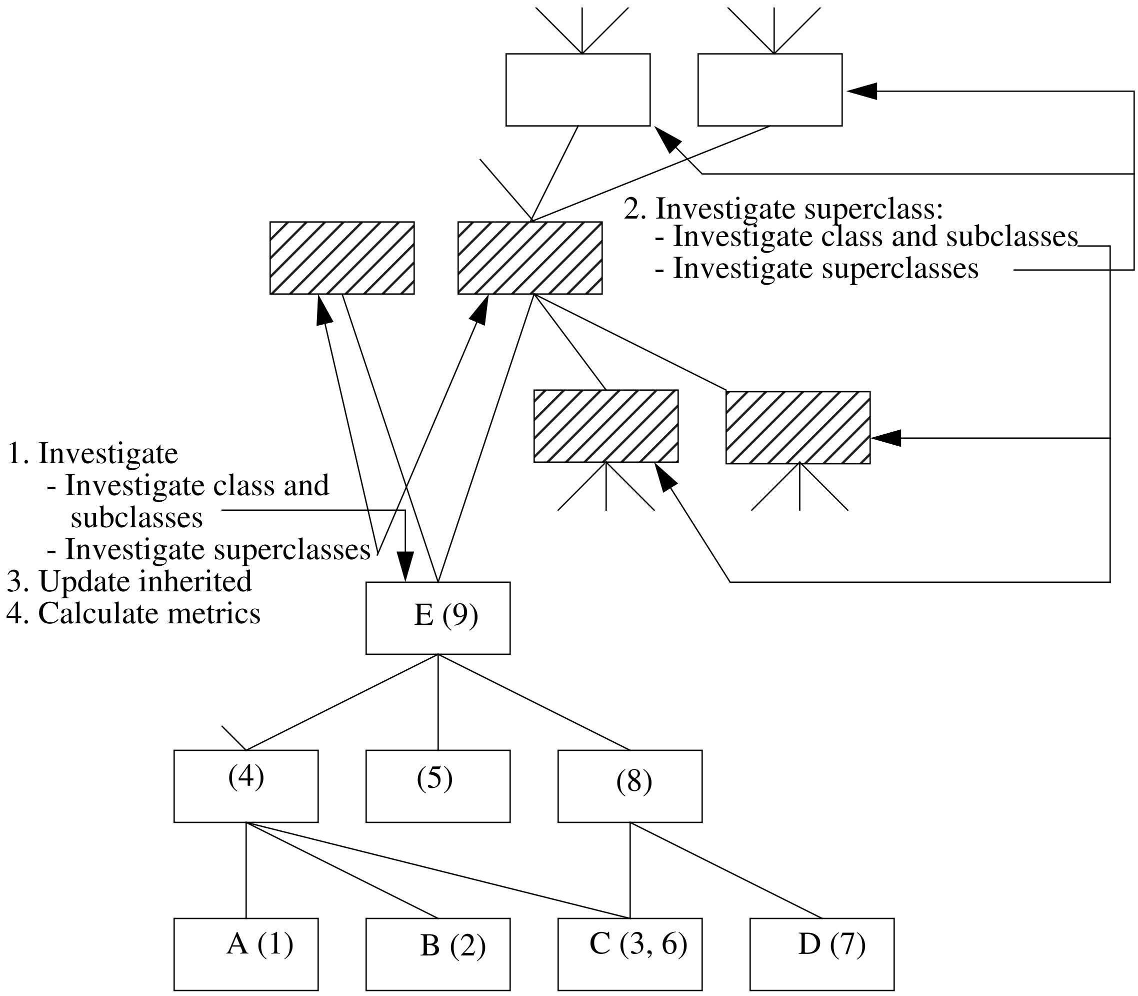

Figure 2: Navigation in the class hierarchy

The third problem in multiple inheritance is that the same class may be investigated many times. To avoid this unnecessary work, we need to obtain for each class a list of the properties that have been investigated thus far.

4.2.3 Algorithms

In this section we present shortly the CLASS MANAGEMENT routine that is built in order to support the maintenance and management of the class hierarchies. The system CLASS MANAGEMENT is itself object-oriented. Each object in CLASS MANAGEMENT contains information about one class of the class hierarchy that should be maintained. The class hierarchy of CLASS MANAGEMENT is presented in Figure 3. It contains a superclass CLASS and two subclasses: class ABSTRACT, whose objects are abstract classes, and class CONCRETE, whose objects are concrete classes.

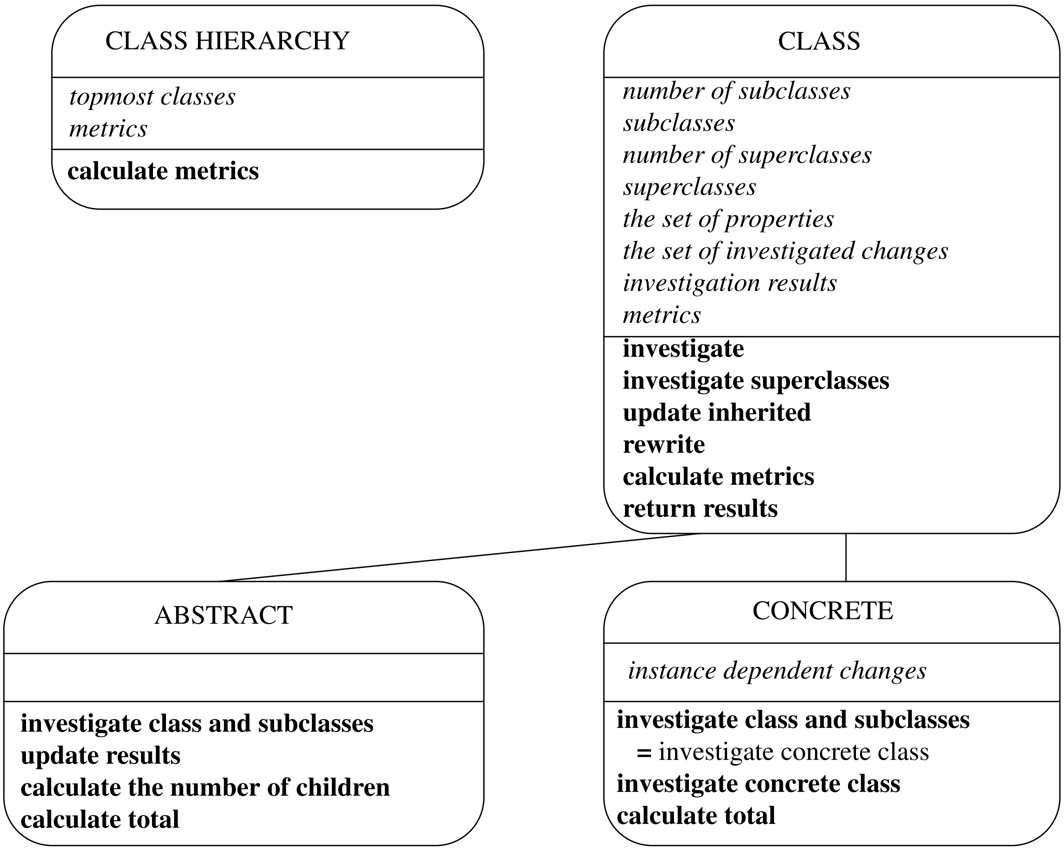

Figure 3: The class hierarchy of CLASS MANAGEMENT

The analyst starts investigation with the following command:
Target **investigate** (*CR*), where

- *Target* is the class, either concrete or abstract, from which the investigation starts, i.e. the common superclass of the concrete classes that need changes.
- *CR* is the set of desired changes. The elements of *CR* are attributes (given by the name and specification) and methods (given by name, specification and signature).

First we will consider the attributes of the classes. Thereafter, a short description of the most important methods of the classes is presented.

Attributes of the classes

The class CLASS has following attributes:

- n is the number of direct subclasses of the class.
- Sub_j, j=1,...,n are the direct subclasses of the class.
- m is the number of direct superclasses of the class.
- $Super_j$, j=1,...,m are the direct superclasses of the class[4].
- P is the list of properties, i.e. attributes and methods defined in the class. The set P is divided into *Strictly-inherited, Overridden* and *Self-defined* properties[5].

[4] If the superclasses have some relationship with each other, then organize them in order from the most general to the least general.

- *Investigated* (initially empty) is the set of changes that have been investigated.
- *Results* (initially empty) contains the following information from the investigation of the class:
  Accepted is the set of accepted changes,
  Applicable is the set of applicable changes and
  Implement is the set of changes to be implemented.
- *Metrics* are the calculated metrics telling about the quality of the class (see 3.2 and 3.3)

Subclasses ABSTRACT and CONCRETE inherit all of these attributes. In addition, class CONCRETE has the following attribute:
- *Instance dependent* is the set of changes that are different for the different object instances of the class.

Class CLASS HIERARCHY has the following attributes:
- l is the number of topmost classes of the class hierarchy.
- Top_j, $j=1,...,l$ are the topmost classes of the class hierarchy.
- *Metrics* are the calculated metrics informing about the quality of the class hierarchy.

Methods of the classes

Notations:

The messages used in the methods are described (resembling Smalltalk) with the following syntax:

Receiver **selector** (*arguments*, *answer*),

where

receiver is the object to which the message is sent,
selector defines the needed method,
arguments are the sent parameters and
answer is the result of the execution of the method.

Temporary variables are written in braces ({ }) in the beginning of the method description. The caret (^) indicates the value to be returned as the result of the method. The term *Self* refers to the object oneself.

As an example we give the most important methods:

Method: Investigate (in: *CR*, the set of changes to be investigated)
Comment: The method checks the possibilities to apply the set of changes *CR* in the class hierarchy. The class, its subclasses and its superclasses check the applicability of the changes to them and update their attribute *Results*.

begin
{j, (App_j, R_j, for $j=1, m$)}
Self **investigate class and subclasses** (*CR*).
if $Applicable \neq \emptyset$ and $m > 0$ **then**

[5] It is assumed that the attributes are identified by the name, and the methods by the signature that specifies the name of the method and the arguments used. It follows from the polymorphism that before making the decision that two properties are the same, the analyst must also consider their specifications. Thus two properties are the same if they have the same identification and the same specification.

begin

Self **Prompt** analyst for dividing *Applicable* into exclusive parts App_j, j=1,...,m that are tried to be defined in different superclasses.

for *j=1,...,m*:

$Super_j$ ***investigate superclass*** *(App_j, R_j).*

Comment: R_j is the investigation result of superclass.

Self **update inherited** (R_j, j=1,...,m).

end

Class hierarchy **calculate metrics**.

end

Method: Investigate superclass (in: *CR,* the set of changes to be investigated
out: *R,* the investigation result)

Comment: The method tries to define properties *CR* in class or in its superclasses.

begin

{*Z*, *j*, (App_j, R_j, for j=1, m)}

$Z = CR \backslash Investigated$, Comment: Z is the set of changes that have not been investigated.

If $Z \neq \emptyset$ **then**

begin

for j=1,...,n:

Sub_j **investigate class and subclasses** (*Z*).

Self **update results** (*Z*).

end

If $Applicable \neq \emptyset$ **and** $m > 0$ **then**

begin

Self **Prompt** analyst for dividing *Applicable* into exclusive parts App_j, j=1,...,m that are tried to be defined in different superclasses.

for *j=1,...,m:*

$Super_j$ **investigate superclass** (App_j, R_j).

Self **update inherited** (R_j, j=1,...,m)

end

^ *Results*

end

Method: Investigate class and subclasses (in: *CR,* the set of changes to be investigated)

Comment: The method ensures that class and its direct and indirect subclasses update their attribute *Results*.

begin

{Z, j}

$Z = CR \backslash Investigated$. Comment: Z is the set of changes that have not been investigated.

If $Z \neq \emptyset$ **then**

begin

for j=1,...,n:

Sub_j **investigate class and subclasses** (*Z*).

Self **update results** (*Z*).

end

end

4.3 Selecting the parts to be changed from the class hierarchy

The previous section described how information can be gathered about the applicability and usability of changes in the class hierarchy. The following sections will consider how the changes should be implemented by using the collected information. The question of the best implementation is a multioptimum task and remains partly open. However, some policies that can be used in the implementation are considered here. The most important thing while making the decisions is the analyst's knowledge of the domain. The other principle here is that the decisions are made as late as possibly. Thus the decisions concerning superclasses are made after the decisions of its subclasses.

First the analyst chooses how wide an effect the changes should have in the class hierarchy. If the changes have to be implemented in the ancestor classes and their subclasses, the whole class hierarchy must be considered. However, if some subclasses are to be left unchanged, the class hierarchy is divided into two parts: part one will be changed and the other will be left unchanged. Denote by *C* the class where the partitioning occurs. The analyst inserts a sibling *C*' for class *C*. The descendants that are to be changed are moved to the descendants of *C*'. Class *C* and its sibling *C*' may have common properties; thus it is possible to define for these classes a new abstract superclass. The situation is illustrated in Figure 4.

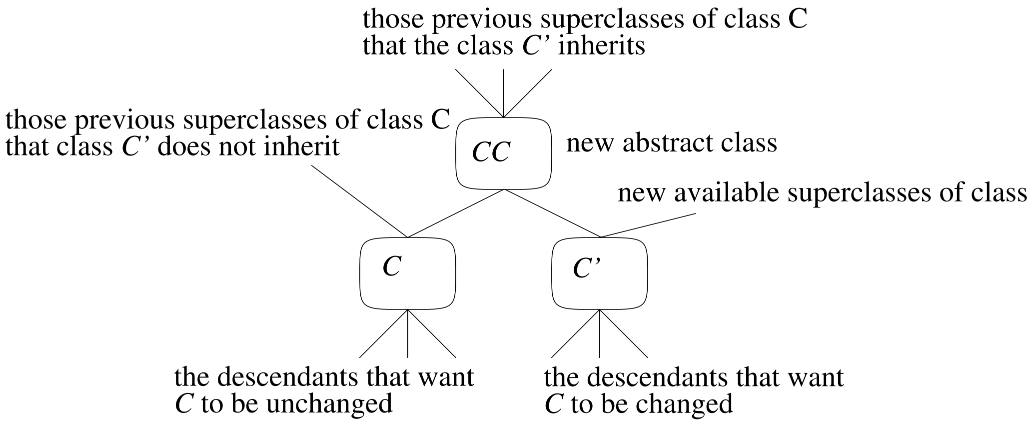

Figure 4: Some descendants are to remain unchanged

The common properties that are defined in the new abstract class are:

- those superclasses of class *C* that the sibling class *C*' needs and
- those properties of class *C* that need not be changed.

After the above considerations, the analyst has separated from the class hierarchy those parts that need some changes. Before implementing the changes in the classes, some general changes can be performed in the class hierarchy. These are considered in the next section.

4.4 General changes in the class hierarchy

The maintenance may cause changes in the relationships between classes, removal of classes, and changes in the properties of classes. When the implementation is considered, all the properties of classes must be considered. It is not sufficient to consider only the changed properties.

4.4.1 Concrete classes

The consideration of changes performed in the class hierarchy is started from concrete classes, because the properties of abstract classes are derived from the properties necessary for the concrete classes. The following changes can be made to concrete classes:

1) If the objects of a particular concrete class are unnecessary, the class can be removed.
2) Concrete classes with similar objects can be joined together.
3) If the cohesion of the objects of the concrete class is unacceptable the objects and thus the class should be split.
4) If some objects of a concrete class are changed and some are unchanged, a sibling is inserted for the concrete class (see Figure 4). This approach is particularly applicable when changing persistent objects.
5) If the objects of the concrete class differ from each other, the class is changed as an abstract class: First, the objects are divided into groups so that in each group the objects are similar. Second, a concrete class is inserted in the class hierarchy for each group and the objects are placed under the applicable new concrete class.
6) The "is kind of" relationship of the concrete class may change:

a) If the "is kind of" relationship between the target class and its ancestor does not hold and thus the target does not inherit any properties directly from its immediate superclass, the class can be moved up. The new superclasses of the target are those superclasses of its previous superclasses from which it can inherit properties. This process is recursive. The superclasses and the superclasses of superclasses are considered until the superclasses from which the target can inherit properties are found. In the case of multiple inheritance, the number of superclasses of the target may increase. After this process, the analyst may also notice that the class does not have any superclasses. In the following such classes are called *free classes*. For free classes, the analyst can try to find new superclasses from other class hierarchies.
b) If the target class is in an "is kind of" relation with its sibling which is an abstract class, the class can be moved down. The available sibling is inserted into the superclasses of the target class. The relationships to the previous superclasses of the target class can be removed, if the target class can inherit all of the necessary properties from the new superclass.

It is possible that some concrete class is first moved up (step 6a) and after that moved down (step 6b).

Example 3: The moving of concrete classes up and down is illustrated in Figure 5. In this example, the classes are numbered and the properties of the classes are presented by using letters. Self-defined properties are written in the normal way and the inherited properties are underlined. The abstract example is considered, because real examples would need too much space for illustrating all the steps of the maintenance process.

4.4.2 Abstract classes

Abstract classes are changed as follows:

1) The changes in the situation of concrete classes mean that the properties of classes must be updated. It is possible that some implemented properties of a class can be moved down and some moved up:

a) Special properties should be defined in a lower level. Thus if a property of an abstract class is applicable in only one subclass, its definition can be referred to the available subclass.
b) On the contrary, if a property is applicable in at least two subclasses and accepted to all subclasses, it can be defined at the superclass.

This step can be automated [23]. For example, in Figure 5 the abstract classes are considered in the order: 6, 2, 3, 1.

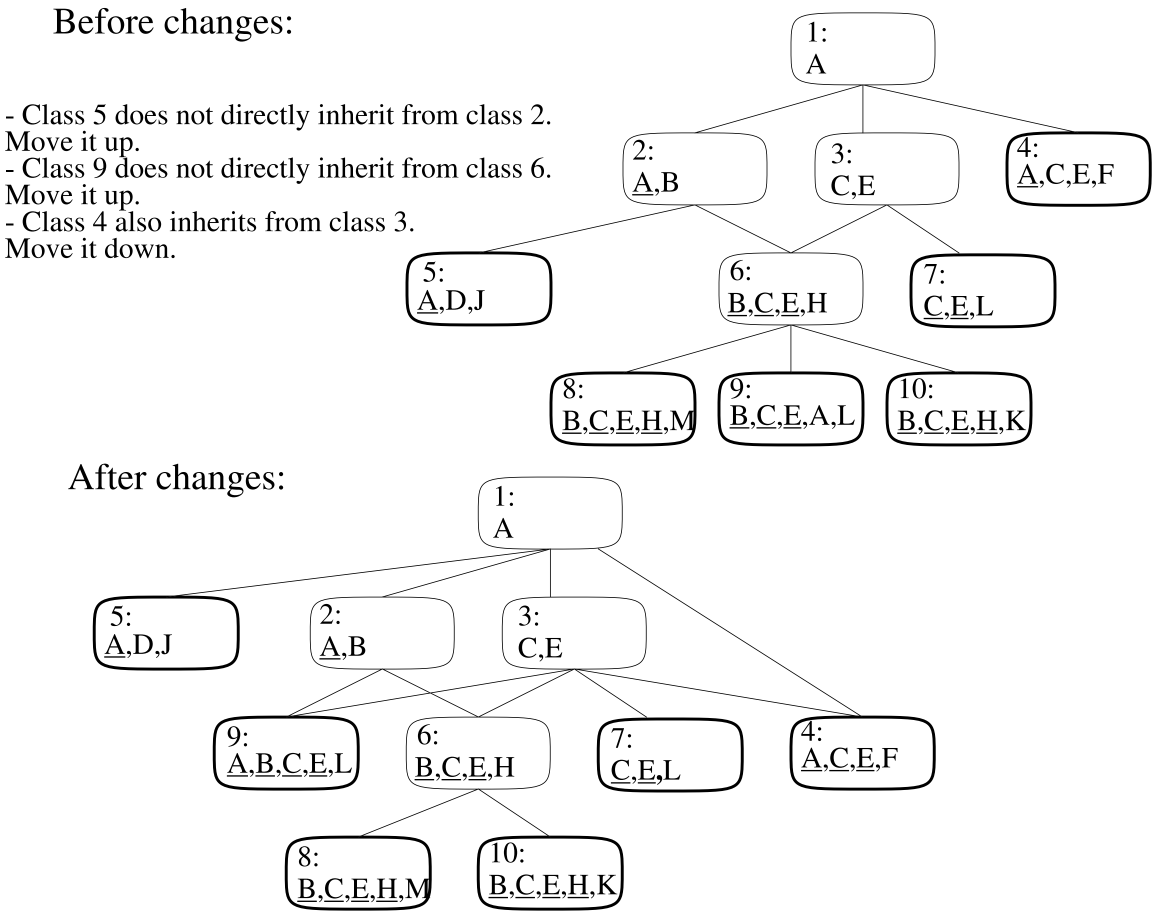

Figure 5: Moving concrete classes up or down

2) In Step 1, the properties of the classes were changed. Consequently, it is possible that a relationship between classes becomes unnecessary:

a) The relationship of the class and its superclass can be removed if
the class does not inherit (directly or indirectly) any properties of the superclass.

b) The classes, both abstract and concrete, can be moved up and down in the same way as in steps 6a and 6b of Section 4.4.1.

3) At step 2, the subclasses of the abstract class are moved away if they do not inherit any properties directly from the abstract class. Consequently, the abstract class can be removed:
• if it has no subclasses, or
• if it has no implemented properties.

4) Two abstract classes can be joined together if they have the same set of subclasses. It is important that unnecessary subclasses be removed at step 2 before this step.

5) If an abstract class *C* has only one subclass *C*' and *C*' is abstract, *C*' can be removed by moving its properties to *C* and its subclasses under *C*.

6) If the cohesion of the class is unacceptable or if some of the descendants are to be changed and some not, it is possible to split the class.

7) It follows from Step 2 that the properties of classes may require some changes. Consequently, the process returns to step 1.

It may be necessary to repeat the process a few times. The process, however, terminates because the classes that are moved up are never moved down below the same superclass and vice versa. The class that is moved up is a generalization of its superclass; whereas the class that is moved down is a specialization of its sibling.

4.4.3 Inserting abstract classes between a class and its subclasses

After the changes presented in the previous section have been made, the analyst may be able to insert new abstract classes between an abstract class and its subclasses. This is profitable if subclasses can be divided into groups that belong in some meaningful abstraction of the domain that is suggested to be reused. There should be more than one class in each group. The subclasses are moved below the most applicable new abstract classes. This process is recursive: the analyst may insert more new abstract classes below the new abstract classes and its subclasses. However, the subclasses should not be nested too deeply by inserting too many abstract classes [20].

If the abstraction is made according to the properties that are applicable to classes, the process can be collected as an heuristic algorithm [21].

4.4.4 Inserting a new abstract superclass

It is possible that the classes can be divided into groups so that the classes in a group resemble each other from some new viewpoint. The analyst can define for the group a new abstract superclass, from which the classes can inherit the common properties. As before, further abstract classes can be inserted between the new abstract class and its subclasses, as in Section 4.4.3.

4.5 Implementation of changes in the classes

The previous sections described changes being made to the total class hierarchy. After these changes, the changes in the individual classes can be implemented. The implementation of the changes in the classes is started from the topmost superclasses. The changes of each class are implemented after the implementation of the properties of its superclasses. Thus the properties that can be inherited are known when the changes are implemented in the subclasses.

The implementation of the changes of a class consists of removals, modifications and insertions of properties. In the previous considerations, the ancestor classes of each class have been updated. In the case of the multiple inheritance, however, it is possible that a certain property can be inherited from several different superclasses. Consequently, to avoid name conflicts, the analyst must choose the ancestor class for each property.

For example the modification of the property has following steps:

- Check whether the modified property can be strictly inherited. Then the only thing that must be done is to define the ancestor and insert the property into the strictly inherited properties. Otherwise, consider the possibilities of non-strict inheritance. Choose, if possible, the ancestor class, insert the property into the overridden properties and implement the overridden property. The last possibility is to insert the property into the self-defined properties and implement it.

Finally we provide a short example of the steps of the maintenance process:

Example 4: Figure 6 describes the class hierarchy in the beginning. The necessary changes in the objects of concrete classes are written by listing the new properties of the objects below the concrete classes.

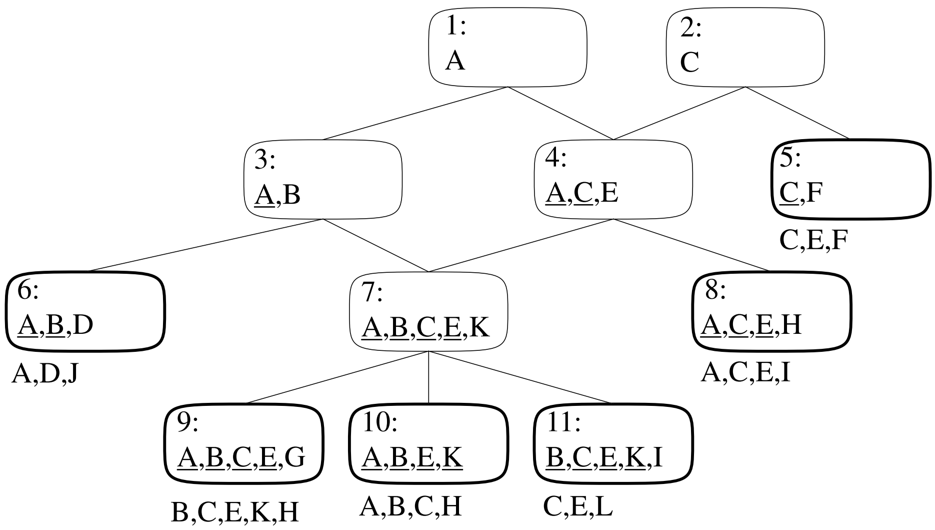

Figure 6: Class hierarchy in the beginning

The applicability of the changes to the class hierarchy is now investigated. Figure 7 shows the applicable properties of the classes and the changes that can be made to the class hierarchy.

Figure 8 shows one possible implementation of the class hierarchy. In this solution each property is defined only once. Thus the amount of code is minimized. It should be noticed that in evaluating the class hierarchy the analyst's understanding of the domain is of crucial importance. The class hierarchy should reflect the reality. Because this was an abstract example, it is not possible to give any good arguments to justify this solution.

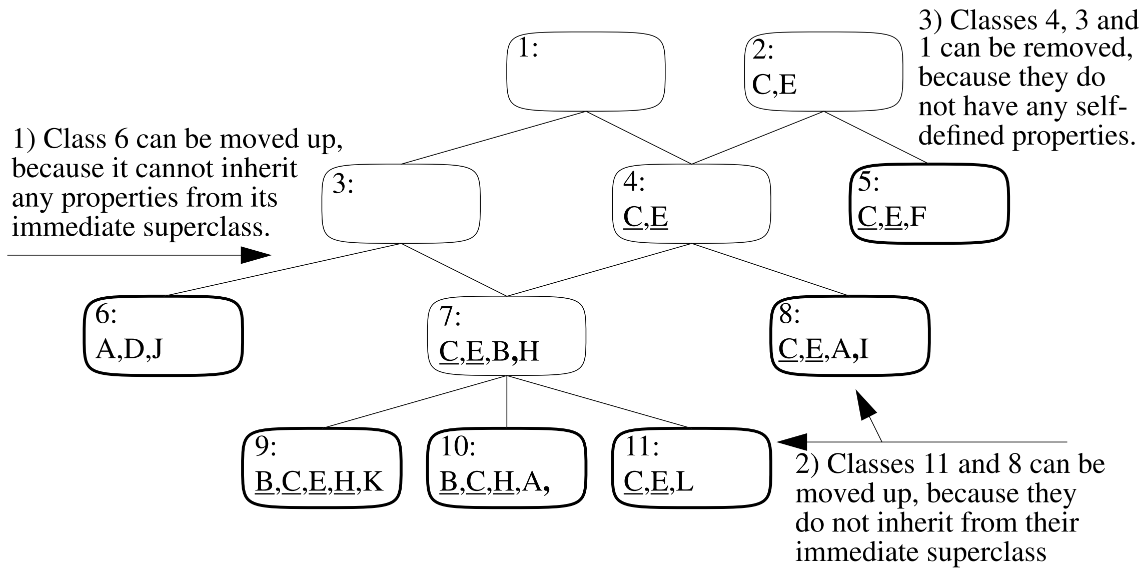

Figure 7: Applicable properties of the classes

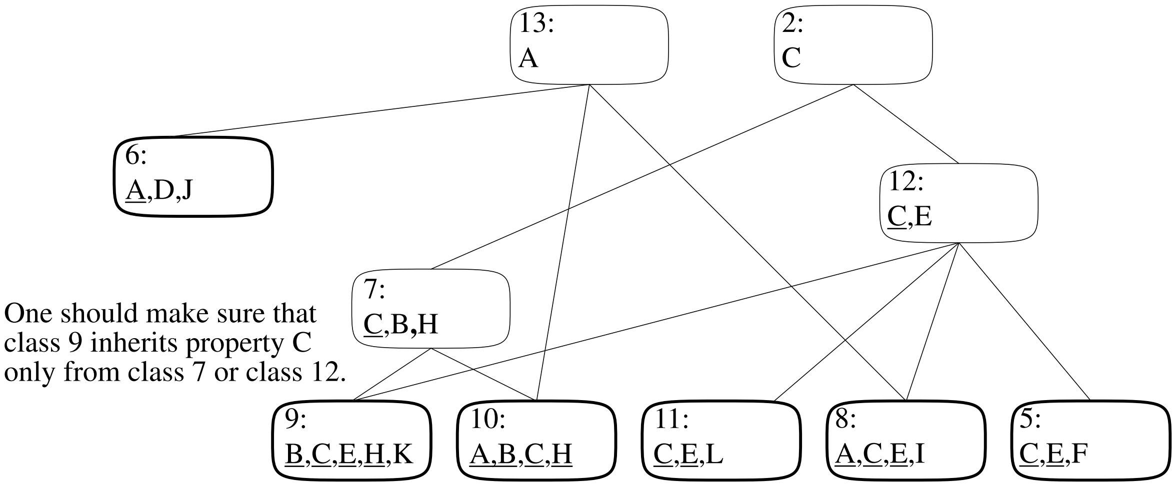

Figure 8: Updated class hierarchy

5 Related work

Lieberherr et al. [18] did not primarily consider maintenance, but they have presented algorithms for abstracting class definitions from a representative sample of *object examples*. An object example is described by giving its class name, followed by the named parts and attributes. The methods of the objects are not considered. The general principle is that those objects which have similar parts belong to the same class. In the first step, the non-optimal *class dictionary graph* is abstracted from the object examples. In the second step, the class dictionary graph is minimized in two phases by minimizing the number of edges. The approach presented in this research differs from the work of Lieberherr et al. because in the present research an attempt has been made to improve the existent class hierarchy. Furthermore, in this work we consider both attributes and methods.

Casais has presented algorithms for restructuring the class hierarchy [11, 16]. The general principle is to avoid explicit rejection of inherited properties. It is assumed that subclasses inherit all the properties of their superclasses. First, the algorithm proceeds by tracing unwanted properties up in the class hierarchy until reaching the nodes where they should be introduced. Second, control passes back to the lower level and the necessary new classes and inheritance relationships are inserted. The differences between the present study and the work of Casais are as follows: We suppose that the class need not inherit all the properties defined in the superclass and that non-strict inheritance is also possible, we emphasize the investigation of the applicability of the changes necessary for the users, and several changes are investigated in the same walk-through.

Bergstein [17] has presented a list of class transformations with which one can improve class hierarchies. The transformations are made according to the "part-of" and alternation relationships between classes. Methods of classes are not considered. The transformations are object-preserving, thus the objects remain unchanged. The correctness, completeness and minimality of the transformations are proved. The changes considered in Section 4.4 resemble the transformations defined by Bergstein but we give also the order of transformations.

6 Conclusion

Heuristic algorithms have been presented for investigating the effect of maintenance on the class hierarchy. When the analyst has decided by using these algorithms how much change can be accomplished with the old class hierarchy, she or he can design the required changes of the class hierarchy. Subsequently classes are inserted and removed, and the properties of the classes and the relationships between classes are changed if necessary. In the final step, changes are implemented in the classes.

The process presented is not fully automated for the following reasons: First, an abstract superclass should be generated only if it is a significant abstraction. Avoiding the explicit rejection of a property [16] or the fact that some classes have a common property [20] are not sufficient reasons for the generation. Second, there are many different way in which one can implement the inheritance relationships between classes [28]. Third, the analyst usually generalizes the properties while they are defined in the superclass level. Fourth, the name of the property does not always indicate the task of the property.

The above analysis shows that maintenance of the class hierarchy is rather a difficult task. If the class hierarchy has been designed carefully and the “is kind of” relationship between the subclasses and superclasses holds, large changes are fairly uncommon in practice. This is because 1) the relationships between classes do not change very much, 2) the properties applicable to the lower level classes are not applicable to the topmost classes, 3) the properties defined in the topmost classes are also applicable in the lower level classes after the changes. In maintenance, it is more common that 1) properties are inserted, removed or modified in the classes, 2) properties are moved a little up or down in the class hierarchy, and 3) a subclass or a superclass is inserted or removed. In the maintenance process presented, an attempt was made to take into consideration all changes possible in theory. Consequently, the class hierarchy may change considerably.

The algorithms have been implemented as a CASE tool [14]. The experiences of the use of the tool have been very encouraging. It seems that the process presented here is particularly valuable in cases where the needs defined by different customers are evaluated and merged. It may be possible to automate the maintenance process even further; but before attempting this, the criteria for selecting the best way of implementation should be specified.

Acknowledgement

The author wishes to thank Martti Penttonen and Raimo Rask for their very constructive criticism and comments in the course of this study.

References

[1] A. Goldberg, K. S. Rubin, *Succeeding with objects Decision Frameworks for Project Management,* Addison-Wesley Publishing Company, 1995

[2] K. H. Bennet: *Automated Support of Software Maintenance*, Information and Software Technology, Vol. 33, No.1, Jan/Feb. 1991 pp. 74-95

[3] B. Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[4] E. Yourdon, *Modern Structured Analysis*, Yourdon Press, 1989.

[5] P. Wegner, *Concepts and paradigms of object-oriented programming*, ACM OOPS Messenger, vol.1, no. 1, Aug. 1990, pp. 7-87.

[6] B. J. Cox, *Object-Oriented Programming, An Evolutionary Approach*, Addison-Wesley Publishing Company, Massachusetts, 1986.

[7] A. Snyder, *Encapsulation and inheritance in Object-Oriented Programming Languages*, in *Proc. OOPSLA'86 Conference Proceedings*, ACM Sigplan Notices, vol. 21, no. 11, Nov. 1986, pp. 38-45.

[8] G. Booch, *Object Oriented Design with applications*, The Benjamin/Cummings Publishing Company, 1991.

[9] T. Korson and J. D. McGregor, *Understanding object-oriented: a unifying paradigm*, Communications of the ACM, vol. 33, no. 9, Sep. 1990, pp. 40-60.

[10] N. Wilde and R. Huitt, *Maintenance Support for Object-Oriented Programs*, IEEE Transactions on Software Engineering, vol. 18, no. 12, Dec. 1992, pp. 1038-1044.

[11] E. Casais, *Automatic reorganization of object-oriented hierarchies: a case study*, Object Oriented Systems 1 (1994), pp. 95-115.

[12] C. P. Willis: *Analysis of inheritance and multiple inheritance*, Software Engineering Journal, Jul. 1996, pp. 215-224

[13] D. Taenzer, M. Ganti and S. Podar, *Object-Oriented Software Reuse: The Yoyo Problem*, Journal of Object-Oriented Programming, Sep./Oct. 1989, pp. 30-35.

[14] A. Putkonen, M. Kiekara, *A Case-Tool for Supporting Navigation in the Class Hierarchy*, Software Engineering Notes, Vol. 22, No. 1, 1997

[15] E. Casais, *Managing Class Evolution in Object-Oriented Systems*, in D. Tsichritzis (Ed.): Object management, Centre Universitaire D'Informatique, Geneve, 1990, pp. 133-195.

[16] E. Casais, *Reorganizing an Object System*, in D. Tsichritzis (Ed.): Object oriented development, Centre Universitaire D'Informatique, Geneve, 1989, pp. 161-189.

[17] P. L. Bergstein, *Object-Preserving Class Transformations*, in *Proc. OOPSLA'91 Conference Proceedings*, Addison Wesley 1991, pp. 299-313.

[18] K. J. Lieberherr, P. Bergstein and L. Silva-Lepe, *From objects to classes: algorithms for optimal object-oriented design,* Software Engineering Journal, vol. 6, no. 4, Jul. 1991, pp. 205-228.

[19] H. Ossher and W. Harrison, *Combination of Inheritance Hierarchies*, in *Proc. OOPSLA'92 Conference Proceedings*, ACM Sigplan Notices, vol. 27, no. 10, Oct. 1992, pp. 25-40.

[20] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[21] A. Putkonen, *A Methodology for Supporting Analysis, Design and Maintenance of Object-Oriented Systems*, Academic dissertation, University of Kuopio, 1994.

[22] S. R. Chidamber, C. F. Kemerer, *Towards a metrics suite for object-oriented design*, OPSLA'91 Conference Proceedings, Object-Oriented Programming Systems, Languages, and Applications, ACM Sigplan Notices, Addison Wesley 1991.

[23] M. Hitz, B. Montazeri, *Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective*, IEEE Transactions on Software Engineering, Vol. 22, No. 4, Apr. 1996, pp. 267-271

[24] V. R. Basili, W. L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Software Engineering, Vol. 22, No. 10, Oct. 1996

[25] M. Lorenz, J. Kidd, *Object-Oriented Software Metrics A Practical Guide*, Prentice Hall, 1994

[26] A. Kendall, *Introduction to Systems Analysis and Design, A Structured Approach*, Allyn and Bacon, 1987

[27] L. Etzkorn, C. Davis, W. Li, *A Practical Look at the Lack of Cohesion in Methods Metric*, Journal of Object-Oriented Programming, Vol. 11, No. 5, Sep. 1998, pp. 27-34

[28] R. Wirfs-Brock, B.Wilkerson, L.Wiener, *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, New Jersey, 1990

[29] N. E. Fenton, *Software Metrics A rigorous approach,* Chapman & Hall, 1991

[30] Chi-Ming Chung, Chun-Chia Wang, Class Hierarchy Based Metric for Object-Oriented Design, Proceedings of 1994 IEEE region 10's ninth annual international conference, theme: frontiers of computer technology, 22-26 Aug. 1994

[31] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes: *Object-Oriented Development: The Fusion Method*, Prentice-Hall, Englewood Cliffs, New Jersey, 1994

[32] M. Fowler, K. Scott, *UML Distilled Applying the Standard Object Modeling Language*, Addison-Wesley, 1997

[33] K. Lieberherr, I. Holland, A. Riel, *Object-Oriented Programming: An Objective Sense of Style*, OOPSLA'88 Conference Proceedings, ACM Sigplan Notices, Vol. 23, No. 11, Nov 1988.

[34] J.-L. Chen, F.-J. Wang, *An inheritance flow model for class hierarchy analysis*, Information Processing Letters 66 (1998), pp. 309-315.