# METRICS BASED REFACTORING

Frank Simon, Frank Steinbrückner, Claus Lewerentz

Software Systems Engineering Research Group
Technical University Cottbus, Germany
E-mail: (simon|fsteinbr|cl)@informatik.tu-cottbus.de

**Abstract:** Refactoring is one key issue to increase internal software quality during the whole software lifecycle. Since identifying structures where refactorings should be applied often is explained with subjective perceptions like "bad taste" or "bad smells" an automatic refactoring location finder seems difficult. We show that a special kind of metrics can support these subjective perceptions and thus can be used as effective and efficient way to get support for the decision where to apply which refactoring. Due to the fact that the software developer is the last authority we provide powerful and metrics based software visualisation to support the developers judging their products. In this paper we demonstrate this approach for four typical refactorings and present both a tool supporting the identification and case studies of its application.

## 1 Introduction

The longer object oriented systems are in use, the more probable it is that these systems have to be maintained [Pres97], i.e. they have to be optimised to a given goal (*perfective maintenance*), they have to be corrected with respect to identified defects (*corrective maintenance*) and they have to be adjusted to a changing environment (*adaptive maintenance*). Whereas many of these activities can be subsumed under the reengineering area, there are additional changing activities that are much less difficult to apply than typical reengineering activities and which do not change the external behaviour. The main goal of these "mini-reengineering-activities" is to improve the understandability and to simplify reengineering activities. Fowler calls these activities Refactorings, which he defines as "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour*" ([Fowl99], p. 53). Refactoring in this sense is an activity that accompanies all phases of the software lifecycle (cf. [LuNe00]); even attempts just to understand foreign code can motivate refactorings, e.g. by adding comments.

While Fowler presents many different kinds of refactorings, one of the main problems when applying this technique to large systems is the question where to apply which refactoring. This question becomes even more difficult by Fowler's statements that refactorings are based on "*human intuition*" ([Fowl99], p. 75) and on subjective perceptions like *smells* and *stinks* (p. 76), because this gives the impression that refactorings never can be made automatically. Fowler explicitly mentions metrics: "*In our experience no set of metrics rivals informed human intuition*" (p. 75).

Our work shows that metrics can help to identify special anomalies for certain refactorings. Like Fowler we believe that the developer should be the last authority for the decision where to apply which refactoring. Nevertheless, tool support is necessary to assist the human intuition in a very efficient and effective way. We believe that software visualisation based on static structure analysis and metrics is a key issue for this task.

To demonstrate our support of intuition we concentrate on some typical refactorings that focus on the members of a class, i.e. methods and attributes. We briefly present both these refactorings and their corresponding "bad smell" which should help to identify parts of the system where to apply the corresponding refactoring (cf. [Fowl99]):

- *Move Method*, i.e. move a method $m$ of a class $\mathcal{A}$ to another class $\mathcal{B}$ that uses the method most; the method $m$ of class $\mathcal{A}$ should be turned into a simple delegation, or it should be removed completely.
  The bad smell motivating this refactoring is that a method uses or is used by more features of another class than the class in which it is defined.

- *Move Attribute*[1], i.e. move an attribute $a$ of a class $\mathcal{A}$ to another class $\mathcal{B}$ that uses the attribute most; all users of the attribute $a$ have to be changed.
  The bad smell motivating this refactoring is that an attribute is used by another class more than the class in which it is defined.
- *Extract Class*, i.e. create a new class and move some cohesive attributes and methods from the old class into the new class.
  The bad smell motivating this refactoring is that one class offers too much functionality that should be provided by at least two classes.
- *Inline Class*, i.e. move all members of a class into another class and delete the old class.
  The bad smell motivating this refactoring is that a class is not doing very much.

With respect to the special character of refactorings the following constraints should be satisfied:

- The support should be available without long delays. Because the application of refactorings does not take many hours (Fowler suggests that the longest refactoring should take one hour) the creation of the support should not take more than a few seconds.
- The support should give immediate feedback after refactoring's application. This feedback is necessary to demonstrate the impact of the applied refactoring on the understandability.
- The support should be integrated into a case environment because refactorings always are applied to source code.
- The support should be easily understandable and should be adjustable to individual goals. Because internal quality is a subjective impression, which has to be made explicit for an analysis, the tool should be adaptable to this subjectivity.
- The support should be available for different object oriented languages. The advantages of the *refactoring browser* – a tool within the SmallTalk environment that supports the execution of a refactoring (cf. [Fowl99], chapter 14) – to be completely embedded within a special case tool is on the other hand the largest disadvantage: the reduction to one language, i.e. SmallTalk.

Our solution to automatically identify the explained "bad smells" is to quickly generate adjustable and powerful software visualisations for different object oriented languages. The creation of support is started within a case tool and the displayed objects within the visualisation are linked to the corresponding source-code parts within the case tool.
Section 2 of this paper briefly explains our generic visualisation framework that is based on a generic similarity measure. Section 3 applies this framework to the presented refactorings and gives some examples. Section 4 presents the tools we created for the automatic visualisation. Section 5 gives some ideas how to apply our visualisation for large systems with several 1000 classes. The paper closes with a summary and some outlooks.

## 2 Distance based cohesion

Many refactorings, including those explained in Section 1, are based on violations of the principle "*Put together what belongs together*". To measure the degree to which some parts belong together there exist many cohesion measures (cf. [ChKe94], [BiKa98], [BiOt93]).
One generic cohesion measure that was firstly introduced by [SiLöLe99] is strongly connected with the theory of similarity and dissimilarity. In correspondence to Bunge [Bung77] we define that the similarity between two entities relates to the collection of their shared properties. Suppose $\mathbb{B}$ to be the set of considered properties for a special similarity viewpoint. Based on this property set it is possible to define a metrical distance measure as follows (cf. [SiLöLe99], p. 71):

$$dist_{\mathbb{B}}(x,y) := 1 - \frac{|p(x) \cap p(y)|}{|p(x) \cup p(y)|}$$

with x and y are two considered entities,
p(x):= {properties $p_i \in \mathbb{B}$ | x possesses $p_i$}.

[1] In opposite to Fowler we call *fields* as *attributes*. Therefore, the refactoring *Move Attribute* is equal to the refactoring *Move Field* ([Fowl99], p. 146ff).

The so-defined distance measure supports the measurement of cohesion. Bieman and Kang define cohesion as "*Cohesion refers to the degree to which module components belong together*" ([BiKa98], p. 111). With respect to our distance concept this means that parts with low distances are cohesive whereas parts with higher distances are less cohesive.
Very important is the special role of the definition of the property set $\mathbb{B}$, since similarity depends a lot on the point of view. Calculating the degree of similarity for two distinct property sets can lead from "identical" to "opposite" for a given entity pair.
The instantiation of this generic measurement framework consists of the following two steps:

1. Decide which kinds of entities should be measured. For example in [SiLöLe99] and [SiLö00] classes are measured and the resulting distances are used to motivate restructurings like move a class from one subsystem into another subsystem.
2. Decide which kind of properties should be considered for the particular similarity concept. For example the properties might be given by the set of superclasses and subclasses one class has, by the names of its methods or by used features (cf. [SiLöLe99]).

With respect to the goal of this paper for the first step only class members are considered, i.e. the distances have to be calculated for methods and attributes.
The second step depends on the special view on cohesion. The four explained refactorings of the last section depend all on usage relations: Two entities belong together if they use each other.
The following section explains the distance concept for the different refactorings in detail.

## 3 Distance motivated refactorings

All four refactorings are based on use relations: Features that heavily use each other should belong to the same class. With respect to our distance concept this can be translated to the following requirements:
The identification of a possible refactoring – *move method/attribute* and *extract/inline class* – can be reduced to the identification of high distances between class members and low distances to some members of a foreign class.
In the following subsection the application of the distance based cohesion concept is explained for all four refactorings in three steps:

1. Identify the corresponding properties used for the distance concept,
2. present a typical pathological case where the application of the refactoring might make sense,
3. calculate the distances for the case study identified in 2) and give an intuitive visualisation. This is possible because distance has a geometric equivalent, that is the Euclidean distance. With special techniques like multidimensional scaling or Spring Embedding (cf. [Chen99]) it is possible to present the calculated distances within three dimensions. We use a very fast spring embedder program [SiStLe00a] to produce 3D-models with the Virtual Reality Modelling Language (VRML) that can be displayed by any VRML-Client.

### 3.1 Identification of *Move Method* Situations

The considered entities for this refactoring are methods. Since methods can use methods and attributes it is helpful to add attributes to the set of considered entities. To apply the distance concept for the identification of the *move method* refactoring it makes sense to consider the following properties for a feature $f$: $\mathbb{B}_f$ should consist at least of

| for a method $f$: | for an attribute $f$: |
|---|---|
| the considered method itself $\cup$ | the considered attribute itself $\cup$ |
| all directly used methods $\cup$ | all methods using it. |
| all directly used attributes. | |

For the calculation of a distance between two entities the needed $\mathbb{B}$ is given by the union of both $\mathbb{B}_f$. This definition of $\mathbb{B}$ has the effect that a method using many attributes and/or methods of other classes has a low distance to those members while methods using only locally defined methods/attributes have a high distance to them.
Imagine the following code for two classes:

```
class class_A
{
public:
     static void methodA1()
     {    attributA1=0;
          methodA2();}
     static void methodA2()
     {    attributA2=0;
          attributA1=0;}
     static void methodA3()
     {    attributA1=0;
          attributA2=0;
          methodA1();
          methodA2();}
     static int attributA1;
     static int attributA2;
}
```

```
class class_B
{
public:
     static void methodB1()
     {    class_A::attributA1=0;
          class_A::attributA2=0;
          class_A::methodA1();}
     static void methodB2()
     {    attributB1=0;
          attributB2=0;}
     static void methodB3()
     {    attributB1=0;
          methodB1();
          methodB2();}
     static int attributB1;
     static int attributB2;
}
```

The location of the method `methodB1()` has a very "bad smell" because it uses features of `class_A` only. Thus, the refactoring *move method* should be applied to this method. To automate the identification of this problem the distances between each two entities (six methods and four attributes) are calculated. The resulting distances can be displayed in a so called *distance matrix* ([FaHa84], p. 374) in which a single element $d_{n,m}$ ($n \geq m$) represents the distance between entity n and entity m (we use the abbreviation `mA1()` for the method `methodA1()`, `aA1` for the attribute `attributA1`, etc.).

| | mA1() | mA2() | mA3() | mB1() | mB2() | mB3() | aA1 | aA2 | aB1 | aB2 |
|---|---|---|---|---|---|---|---|---|---|---|
| mA1() | 0 | | | | | | | | | |
| mA2() | 0.5 | 0 | | | | | | | | |
| mA3() | 0.4 | 0. | 0 | | | | | | | |
| mB1() | 0.6 | 0.6 | 0.5 | 0 | | | | | | |
| mB2() | 1 | 1 | 1 | 1 | 0 | | | | | |
| mB3() | 1 | 1 | 1 | 0.86 | 0.6 | 0 | | | | |
| aA1 | 0.5 | 0.67 | 0.33 | 0.5 | 1 | 0.88 | 0 | | | |
| aA2 | 0.83 | 0.6 | 0.5 | 0.67 | 1 | 0.86 | 0.5 | 0 | | |
| aB1 | 1 | 1 | 1 | 1 | 0.5 | 0.25 | 1 | 1 | 0 | |
| aB2 | 1 | 1 | 1 | 1 | 0.33 | 0.8 | 1 | 1 | 0.75 | 0 |

The above mentioned spring embedder arranges the 10 entities within the Euclidean space so that the calculated distances correspond to the displayed distances. These geometrical positions are used to produce a VRML-world as shown in Figure 1[2]:
The visualisation within the VRML-world uses the following parameters:

- All green (light) displayed figures are elements of `class_A`, all blue (dark) displayed figures are elements of `class_B`.
- All attributes are displayed as squares, all methods are displayed as spheres.

For a better understanding it is possible to click on every displayed figure to display the corresponding source code with the case tool from which the measurement and visualisation was started. Moving the mouse cursor upon a displayed element shows additional information like element-name, corresponding class, file and subsystem, and a list of some use- and used-by relations. The feature of displaying member names like shown in Figure 1 can be turned on and off for each object separately in an interactive way.

[2] The presentation of VRML-worlds within a paper is not very satisfactory because it is difficult to guess distances, colours and other properties. Additionally, many features for an easy understanding like space related navigation possibilities (fly-over, fly-through, etc.) can not be presented here (cf. [DäPa98]).

The visualisation in Figure 1 strongly recommends in a very simple and understandable way to move `methodB1` (indicated by the arrow) from `class_B` to `class_A`. Only in very few cases it would make sense to leave such a situation unchanged. But, because the developer is the last authority, this still would be possible. This visualisation only should support its analysis.

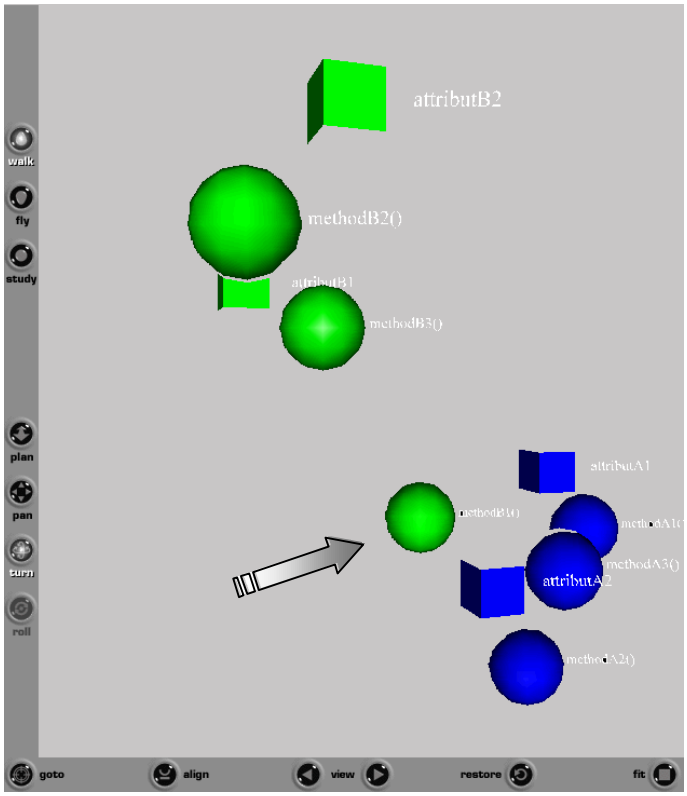


Fig. 1: VRML-World that motivates refactoring *move method*

## 3.2 Identification of *Move Attribute* Situations

In this case the considered entities are attributes. Because attributes only make sense if used by methods it is helpful to add methods to the set of considered entities. The set of considered properties is equivalent to the set of properties used for the refactoring *move method* because the underlying relationships are the same.

To give an example of a pathological case we modify the source code example from the last section:

```
class class_A
{
public:
     static void methodA1()
     {    attributA1=0;
          methodA2();
     }
     static void methodA2()
     {    attributA1=0;
          methodA1();
     }
     static void methodA3()
     {    attributA1=0;
          methodA1();
          methodA2();
     }
     static int attributA1;
     static int attributA2;
}
```

```
class class_B
{
public:
     static void methodB1()
     {    class_A::attributA2=0;
          methodB2();
          methodB3();
     }
     static void methodB2()
     {    class_A::attributA2=0;
          attributB1=0;
     }
     static void methodB3()
     {    class_A::attributA2=0;
          attributB1=0;
          methodB1();
     }
     static int attributB1;
     static int attributB2;
}
```

It is quite clear that the location of the attribute `attributA2` would be better within `class_B` than in `class_A`. After having calculated

a) the distances,
b) the positions of the graphical objects within the Euclidean Space and
c) the VRML-world

the diagram shown in Figure 2 can be analysed (the same visualisation parameters are taken as in the last section):

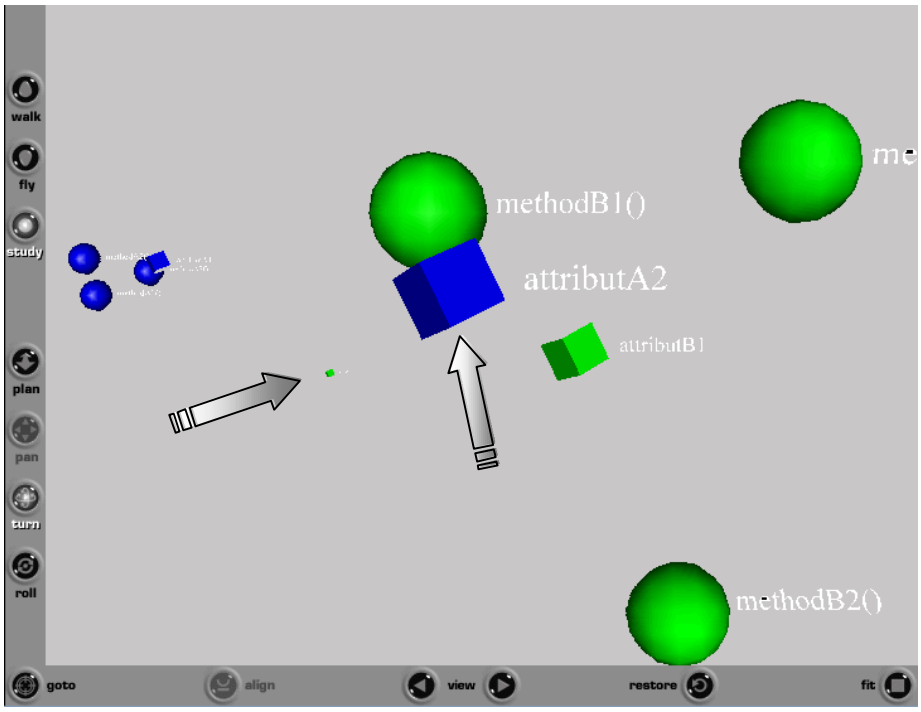


Fig. 2: VRML-world that motivates refactoring *move attribute*

Two things can be easily seen:

- The visualisation supports the decision to apply the refactoring *move attribute* to the attribute `attributA2` (blue/dark box within the cluster at the right, cf. right arrow).
- Additionally, it is obvious that the attribute `attributB2` is not used at all (in Figure 2 this attribute is displayed as little box between the two clusters of elements, cf. left arrow): its position outside represents high distances to all other members, i.e. it is not used by anyone. For this attribute it might make sense to delete it (for the special case where inheritance is used cf. Section 5.1).

### 3.3 Identification of *Extract Class / Inline Class* Situations

In this section the both refactorings *extract class* and *inline class* are considered together because each is the reverse of the other: If a class was extracted, i.e. some methods and attributes from one class were separated into another class, this extracting can be undone by inlining the extracted members into the original class. The opposite is also true. That means that the decision whether to apply the extract or inline class refactoring is discussible to a certain degree. To support this decision a common visualisation is helpful.
The goal of both refactorings is to get cohesive classes. While cohesiveness can be reached on several levels, from the weakest *coincidental cohesion* to the strongest *functional cohesion* ([YoCo79]), a static analysis technique is only able to consider indicators for cohesion. Most cohesion measures (e.g. LCOM [ChKe94], SFC or TCC [BiOt93]) consider use-relations between functions and attributes as indicators for a functional cohesion, because if all members work together, i.e. they use each other or a common member, they relate to each other. Therefore, for our distance measure we need the same properties for attributes and methods.
To demonstrate a pathological case for one of these refactorings a large example has to be created. Instead of showing the source code we explain a class where one refactoring might make sense. Imagine a class with 20 methods $m_i$ and four attributes $a_i$ with the following properties:

- For the first 10 methods hold:
  $m_1$ uses $m_2$ , $a_1$ and $a_2$ ; $m_2$ uses $m_3$ , $a_1$ and $a_2$ ; $m_3$ uses $m_4$ , $a_1$ and $a_2$ ; ... ; $m_{10}$ uses $m_1$ , $a_1$ and $a_2$;
- For the last 10 methods hold:
  $m_{11}$ uses $m_{12}$ , $a_3$ and $a_4$; $m_{12}$ uses $m_{13}$ , $a_3$ and $a_4$; $m_{13}$ uses $m_{14}$, $a_3$ and $a_4$; ... ; $m_{20}$ uses $m_{11}$ , $a_3$ and $a_4$;

For this construction it should be obvious that it would make sense to extract the last 10 methods with the last two attributes into a separate class.
After calculating the distances, the positions of the figures within the Euclidean Space and the VRML-world the diagram shown in Figure 3 can be analysed (the same visualisation parameters are taken as in the last two sections):

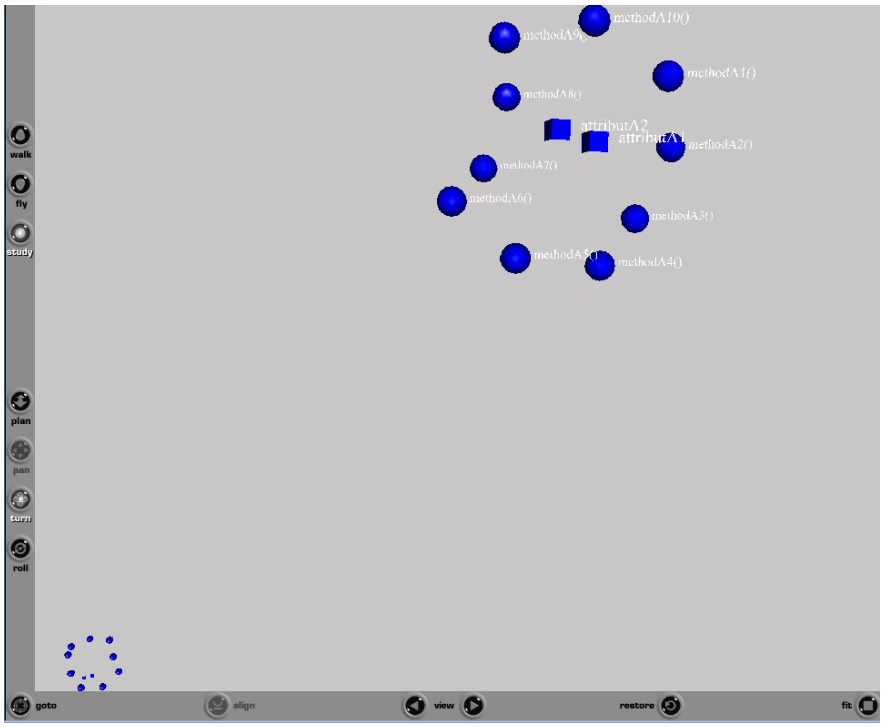


Fig. 3: VRML-world that motivates refactoring *extract class*

Since members of only one class are presented all displayed figures have the same colour. The presented layout shows a typical anti-pattern for the motivation of the refactoring *extract class*: The separation of the members presented at the right side from the members presented at the left side should yield two more cohesive classes.

The same kind of visualisation can recommend the restructuring *inline class*. Imagine that the methods $m_1$ to $m_5$ and attribute $a_1$ would be a separate class. Then the visualisation would be similar but with two different colours. The recommendation to merge both classes would be easily detectable.

## 4 Tool Support

The presented approach to support the identification of "bad smells" by a powerful software visualisation has practical relevance only, if it can be generated on the fly. A software engineer has to be enabled to just click on a button to select the members he wants to examine and to immediately analyse the visualisation. This all has to be done in a few seconds.

To fulfil these requirements we enhanced a metrics tool called *Crocodile* that was developed at our research group with our distance concept (cf. [LeSi98]). It is well suitable for this task because it is fully integrated into an industrial case tool, language independent and, especially for a small number of measured entities, very fast. The steps to create the distance matrix needed for the generation of the visualisation are the following:

1. The user has to create a repository for the whole project. This has to be done only once and is started by selecting a special menu item within the case tool. The goal of this action is to extract the relevant symbols like classes, methods, use-relations etc. into Crocodile's repository, which is implemented as a relational database. For large projects with about 1500 classes this step needs 15 minutes. But this extraction has to be done once only. After that the created database can be used several times to support refactorings of this project. However, to get the required feedback after each applied refactoring (cf. Section 1) the database has to be modified slightly, which is not supported yet. Therefore, currently the database has to be refilled completely in order to get feedback.
2. The user has to select the classes whose members have to be analysed. This task is GUI-supported and does not take much time.
3. Afterwards the distances can be calculated. For up to 100 members this does not take more than a few seconds.
4. The distances have to be re-arranged by the spring-embedder-program. For up to 100 members this does not take more than a few seconds.
5. Based on the calculated positions and the information from the repository the VRML-world can be created and displayed within a VRML-client that is available as plug-in for many browsers like *InternetExplorer*® or *Netscape*®. Also this task is very fast.

The steps 3 to 5 can be done automatically because no user input is necessary anymore.

All tools are running under Windows® and Solaris®. In the next section we show how the visualisation approach can be applied to real-world systems with thousands of classes and – especially – inheritance.

## 5 "Bad Smells" in large systems

After demonstrating the power of our visualisation approach for some artificial, pathological cases it has to be examined what kind of problems might occur when applying our approach to large systems. There are three main problems:

1. In large object oriented systems many classes can not be examined separately because they are included in an inheritance hierarchy. The problem of inheritance and how it increases the difficulty to apply refactorings is also discussed in [Fowl99]: Exploring a separated class without any inheritance context might suggest wrong refactorings because the inherited attributes and methods are not considered.
2. Which classes should be analysed to identify possible refactorings? Since in large systems there exist thousands of classes it is time consuming to analyse all classes in such a detail as explained in the last section.
3. Whereas the analysis of the artificial, pathological cases is very simply such obvious situations can be rarely found in practical systems. How difficult is the interpretation of the generated visualisations of real-world systems and the identification of possible refactorings?

The following subsections give some answers to these questions.

### 5.1 Impact of inheritance to "Bad Smells"

Ignoring the inheritance might have impacts on the identification and application of refactorings, including the ones we focus on in this paper. One problem covers the problem of refactoring identification and one the problem of its application:

- Ignoring inherited members can give a wrong impression: For example the impression that some methods are not cohesive can only be caused by the fact that the common used attributes are not considered because they are inherited. The same problem occurs with use-relations: Static analysis often reduces coupling to couplings to super classes because only this is directly detectable from code. The possibility to call the same methods within subclasses – especially if the super class is an abstract class or even an interface – is ignored.
- Ignoring possible subclasses can increase the difficulty to apply the suggested refactoring. If for example the visualisation suggests to move a method to another class this can be wrong if the method is used in some sub classes of the current class.

For the first problem we have introduced the concept of flattening, i.e. to represent a class with all members it has access to [SiBeLe00]. Additionally, this concept can extend use-relations to potential use-relations, i.e. coupling to superclasses is extended to possible couplings to all subclasses. The flattening itself can be done automatically, so it is possible to generate visualisations of only flattened elements (including inherited members) to prevent wrong interpretations. To distinguish between self defined and inherited members different visualisation parameters can be used (e.g. different layouts).

The second problem is partly solved: If the members of the subclasses are also selected for visualisation it can be easily detected a) if a member of the superclass is inherited into a subclass (by detecting a member with the same name but different visualisation attributes within the subclass) and b) if the method is used within the subclass (by the distance of the inherited member to other class members). If the subclass is not selected for visualisation the problems for the refactoring application is not visible and have to be tested manually be using cross-referencers of the case tool.

### 5.2 Selection of candidates for identification of "Bad Smells"

One of the main questions is which classes should be selected for visualisation. This question depends a lot on the special use case in which possible refactorings are identified:

- *Understanding*: Developers try to understand a foreign, given software system: They focus on some classes they are interested in (mostly some start classes, classes near main() or classes with a name like the whole system) and start the visualisation process because the presented approach also can simplify the understanding of software. Examining the visualisation the developers detect some "bad smells" and might apply some refactorings. If a class seems to be important they select new classes from the local context.
- *Modification*: Developers try to modify a given software system. Doing so many problems might occur that should be solved only after some particular refactorings (cf. [Fowl99], chapter 1). After selecting the

appropriate classes where the modification has to be applied the corresponding visualisation simplifies the process to detect "bad smells".

- *Quality improvement*: Because Crocodile also has a metrics engine for classical object oriented metrics the developer is able to get some quantitative feedback about his system. Its interpretation might show some anomalies for some classes that should be resolved by applying refactorings. These classes can be selected for a detailed visualisation. For example a high value of the *weighted method count* metric (WMC, [ChKe94]) can not only be analysed within the presented visualisation but it might be possible to easily detect some WMC improvements, e.g. by application of the refactoring *extract class*. Another example for suspicious metric values could be high values of the *lack of cohesion of methods* metric (LCOM, [ChKe94]): Again, the visualisation of this class does not only help to analyse it but suggests some solutions, e.g. by application of the refactorings *move method* or *move attribute*.
  In addition to the last strategy to start on some anomalous candidates it is possible to create such a distance view on the class level (cf. [SiStLe00b], [SiLö00]): the measured entities are classes and the corresponding distances can be calculated on the properties, which members of one class use which members of another class. A distance between two classes is the lower the more interactions between both exist. Also in this kind of visualisation some anomalies can be detected, which in turn can be used as set for a detailed visualisation like shown in this paper.

However, if no useful selection can be made it is also possible to select all classes for a visualisation. The only disadvantage of this technique is the confusing visualisation if more than 2000 elements are used. Another problem is the increasing time for the spring embedder: For 2000 elements it can take nearly one hour to calculate meaningful positions.

### 5.3 Interpretation of visualisations of real-world systems

Depending on the number of selected classes, the number of contained elements and their relationships it might be more or less difficult to identify some candidates for refactorings. In Figure 4 a typical VRML-world of a still maintained system for 4 classes, consisting of 40 methods and 18 attributes is presented.

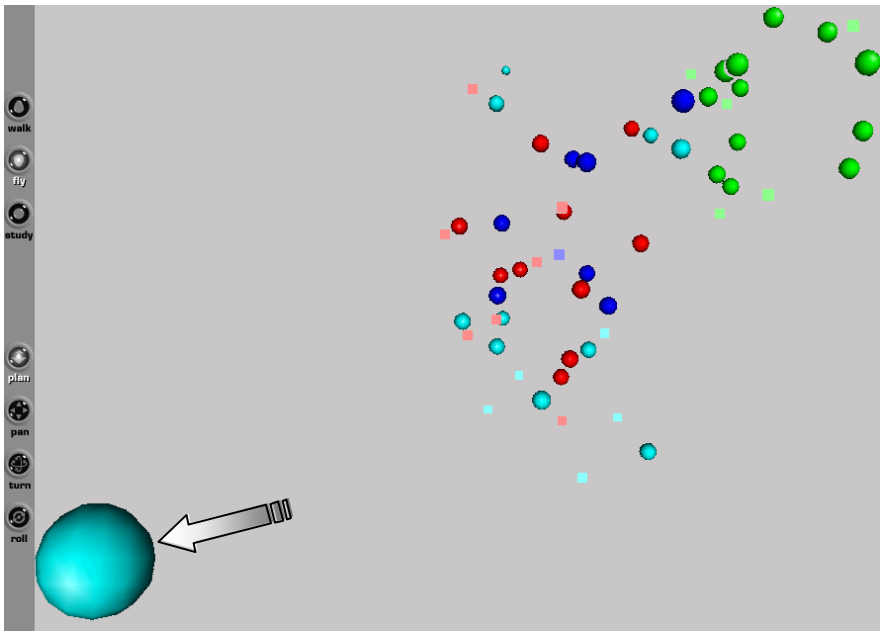


Fig. 4: VRML-world of 4 classes of a a real-practise system

These four classes are very cohesive (cf. cluster at the top right that contains only elements of one class). Interesting is the obviously separated method at bottom left (cf. arrow). Indeed, this method was used in previous releases but after applying the refactoring *move method* this method is not used anymore here (even by no other class)! Thus, the deletion of this method, which is another basic refactoring, can be done without problems.

Although the identification of such obvious anomalies like the one presented in Figure 4 is very simple, a more detailed interpretation of such visualisations has to be learned. With some experience it is possible to get a feeling how "bad smells" look like. Then this kind of visualisation is very helpful and is taken as support whenever possible.

## 6 Summary

We have presented a generic approach to generate visualisations supporting the developer to identify candidates for refactorings. Due to the premise that the developer has to be the last authority in identifying and applying refactorings our work focuses on providing decision support. The generic approach was instantiated in a way to analyse the source code for four refactorings: *move method*, *move attribute*, *extract class* and *inline class*. All four refactorings result in a common visualisation, i.e. one visualisation can be taken for the identification of different refactorings. We demonstrated how the approach can be applied to artificial, pathological cases and presented a tool that enables the developer to get these visualisations on the fly integrated to the used case tool. Afterwards we extended our concept to real-world systems and explained, how inheritance can be considered adequately, which classes should be analysed with respect to the underlying use case and how a typical interpretation of a visualisation of a real-life system looks like.
From our point of view this approach looks very promising: It combines the consensus that refactorings are necessary and positive to software quality with the necessity of being fast and effective. Our future research will focus on further refactorings – especially the ones that cover inheritance like *pull up / push down method /attribute* and that can be analysed because of our flattening concept – and how their identification can be supported by this visualisation.

## 7 References


[BiKa98] James M. Bieman, Byung-Kyoo Kang: "*Measuring Design-Level Cohesion*", IEEE Transactions on Software Engineering, Vol 24, Nr. 2, February 1998

[BiOt93] James M. Bieman, Linda M. Ott: "*Measuring functional cohesion*", Technical Report CS-93-109, Michigan Technological University, 1993

[Bung77] Mario Bunge: "*Treatise on Basic Philosophy*", Volume 3: Ontology I, The furniture of the world", D. Reidel Publishing Company, Dordrecht-Holland, 1977

[Chen99] Chaomei Chen: „*Information Visualisation and Virtual Environments*", Springer-Verlag, London, 1999

[ChKe94] S. Chidamber, C. Kemerer: "*A metrics suite for object oriented design*", IEEE Transactions on Softwareengineering, 20, Nr, 6, p. 476-493, 1994

[DäPa98] Rolf Däßler, Hartmut Palm: „*Virtuelle Informationsräume mit VRML: Informationen recherchieren und präsentieren in 3D*", dpunkt-Verlag, Heidelberg, 1998

[FaHa84] Ludwig Fahrmeier, Alfred Hamerle: "*Multivariate statistische Verfahren*", Walter de Gruyter, Berlin,1984

[Fowl99] Martin Fowler: "*Refactoring: improving the design of code*", Addison-Wesley, New York, 1999

[LeSi98] Claus Lewerentz, Frank Simon: "*A product metrics tool integrated into a software development environment*", in proceedings of the European Software Measurement Conference FESMA 98, Technologisch Instituut, Antwerpen, 1998

[LuNe00] Andreas Ludwig, Rainer Neumann: "*Refactorings - Engineering meets Reengineering*", in proceedings of 2nd workshop on Reengineering, Bad Honnef, to be published as TR at University Koblenz, 2000

[Pres97] Roger S. Pressman: „*Software Engineering: A practitioner's approach*", McGraw-Hill, New York, 1997

[SiBeLe00] Frank Simon, Dirk Beyer, Claus Lewerentz: " "*Considering Inheritance, Overriding, Overloading and Polymorphism for Measuring C++ Sources*", Technical Report 04/00, Computer Science Reports, Technical University Cottbus, May 2000 (to be published partly in the proceedings of the 10th international workshop on Software Measurement in Berlin, Oct. 4-6, 2000

[SiLö00] Frank Simon, Silvio Löffler: "*Semiautomatische, kohäsionsbasierte Subsystembildung*", in Reiner Dumke, Franz Lehner (Hrsg): "Software-Metriken: Entwicklungen, Werkzeuge und Anwendungsverfahren", Pages 153-170, Gabler Verlag, Wiesbaden, 2000

[SiLöLe99] Frank Simon, Silvio Löffler, Claus Lewerentz: "*Distance based cohesion measuring*", in proceedings of the 2nd European Software Measurement Conference (FESMA) 99, Technologisch Instituut Amsterdam, 1999

[SiStLe00a] Frank Simon, Frank Steinbrückner, Claus Lewerentz: "*3D-Spring Embedder for Complete Graphs* ", Technical Report 11/00, Computer Science Reports, Technical University Cottbus, September 2000

[SiStLe00b] Frank Simon, Frank Steinbrückner, Claus Lewerentz: "*Multidimensionale Mess- und Strukturbasierte Softwarevisualisierung*", to be published in proceedings of 2th workshop "Reengineering" in Bad Honnef, 2000

[YoCo79] E. Yourdon, L.L. Constantine: "*Structured Design*", Prentice Hall, Englewood Cliffs, 1979