Refactoring Class Hierarchies with KABA

Mirko Streckenbach, Gregor Snelting
Universität Passau
Fakultät für Informatik, Innstr. 33, 94032 Passau, Germany
strecken@infosun.fmi.uni-passau.de

ABSTRACT

KABA is an innovative system for refactoring Java class hierarchies. It uses the Snelting/Tip algorithm [13] in order to determine a behavior-preserving refactoring which is optimal with respect to a given set of client programs. KABA can be based on dynamic as well as static program analysis. The static variant will preserve program behavior for all possible input values; the dynamic version guarantees preservation of behavior for all runs in a given test suite. KABA offers automatic refactoring as well as manual refactoring using a dedicated editor.

In this contribution, we recapitulate the Snelting/Tip algorithm, present the new dynamic version, and explain new extensions which allow to handle full Java. We then present five case studies which discuss the KABA refactoring proposals for programs such as `javac` and `antlr`. KABA proved that `javac` does not need refactoring, but generated semantics-based refactoring proposals for `antlr`.

1. INTRODUCTION

Refactoring transforms a given class hierarchy in order to improve its structure and evolution. Refactoring and, more generally, program transformation has been a popular research topic for some time, and has recently gained much interest due to the emergence of light-weight design methodologies such as Extreme Programming [2] that advocate continuous refactorings. The book by Fowler [5] presents a comprehensive list of refactoring transformations which has been implemented in some refactoring tools.

But while refactorings are well-understood, automated generation of refactoring proposals is still in its infancy. One of the key limiting factors is the fact that verifying the preconditions for many refactorings (in order to ensure that program behavior is preserved) may involve non-trivial program analysis. Another limiting factor is the fact that refactoring is a manual process even with tool support: the refactorer must know which refactorings to apply and why.

In this paper, we present a different approach to refactoring. We assume that a hierarchy is given together with a set of *client programs* using this hierarchy. We generate a refactoring proposal *automatically*, and this proposal is based on the *usage* of the hierarchy by the client programs. The refactoring is guaranteed to be semantically equivalent to the original program (with respect to client behavior).

The transformed hierarchy can then be subject to further manual refactorings, while preservation of semantics is still guaranteed. Eventually, code can be generated. The new code contains the same statements as the original code, except that the hierarchy has changed and for all variables a new type (i.e. class) has been computed.

Preservation of semantics is achieved by a combination of program analysis, type constraints and concept lattices. But the true value of the technique lies in the possibility to automatically refactor with respect to a given *purpose* – represented by a given set of client programs. For the given clients, the refactored hierarchy is *optimal* in the sense that every object contains only those methods or fields it really accesses. In fact, we determine the most fine-grained refactoring which still preserves behavior of all clients.

The refactoring editor KABA[1] offers two variants of this approach. The static approach requires static program analysis and guarantees behavior preservation for all analyzed client programs. The dynamic approach requires dynamic program analysis and guarantees behavior preservation for all client runs of a given test suite. Of course, the static analysis is a conservative approximation of any dynamic analysis. KABA also offers semantic support for manual refactoring, intended not only for post-processing an automatic refactoring, but for refactoring any class hierarchy.

Organization of this paper

The primary purpose of this contribution is the presentation of the innovative KABA refactoring system, as well as the discussion of case studies for its application. But in order to make KABA work for full Java, the original refactoring algorithm had to be considerably expanded. Thus we had to decide how the presentation should reconcile the technical and the practical aspects of our work.

We have chosen a non-standard organization: the main paper presents KABA from an application-oriented, software engineering view. All technical and algorithmic details and innovations are presented in a series of appendices.

2. THE REFACTORING ALGORITHM

In this section, we describe the Snelting/Tip algorithm from a user's viewpoint. The algorithm has been described in detail in [13]. Appendix 1 presents a more technical recapitulation of the most important steps of the algorithm, as well as technical references.

[1]KABA stands for "KlassenAnalyse mit BegriffsAnalyse" (class analysis by concept analysis). KABA is also a popular chocolate drink in Germany.

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

2.1 Basic steps of the algorithm

2.1.1 Collection of member accesses

The first analysis step collects all field and method accesses in the given source hierarchy and its client set. The algorithm sets up a table, where the rows are labeled with variable names (including parameters, this-pointers etc), and the columns are labeled with fields and methods from the hierarchy. An access $o.m()$ from an object will lead to a table entry for $(o, C::m)$ (where C is the static class for m).

There are two variants of the collection process, one based on dynamic program analysis and one based on static program analysis. In the dynamic variant, the JVM is instrumented such that every call $o.m()$ at runtime leads to a corresponding table entry. In the static variant, *points-to analysis* is used to approximate the effects of dynamic dispatch. For an object reference x, its points-to set $pt(x) = \{o_1, \ldots, o_n\}$ is used to collect possible method calls by $x.m()$: a table entry is generated for any $(o_i, C::m)$.

2.1.2 Incorporation of type constraints

In order to guarantee preservation of behavior, a set of type constraints is extracted from the source which must also be respected in the refactored hierarchy. For example, every assignment `a = b;` in the original source requires that $type(a) \geq type(b)$, and this must hold in the refactored hierarchy as well. Furthermore, if $A \geq B$ are classes which both define a method m, the constraint $type(A::m) \geq type(B::m)$ must sometimes be retained in order to avoid ambiguous accesses in the refactored hierarchy.

Once all type constraints have been extracted, they are incorporated into the table from phase 1: more entries are added, until a minimal table is obtained which respects all constraints. It is not obvious that this is always possible, and even less obvious that these constraints are enough to guarantee preservation of behavior. The details of constraint generation and incorporation are quite complex (see appendix 1).

2.1.3 Generation of concept lattice

Tables are nice, but the true reason for extracting a table from the source is that a class hierarchy can be generated automatically from the table using the well-known method of *mathematical concept analysis*. Concept analysis (see appendix 1) generates a lattice from the table, which represents exactly the same information as the table, but organized into a completely different, hierarchical view. Concept lattices are natural inheritance structures: every lattice element represents a class, and common fields or methods are factored out into super-classes. The refactored hierarchy is thus obtained as a concept lattice generated from the final table.

The statements in this refactored hierarchy are the same as in the original hierarchy. But every variable or object obtains a new type, and this type (i.e. refactored class) will contain all fields and methods needed by the variable. More precisely: every object that might be generated at runtime has a new type containing all methods and fields it may access at runtime; while methods or fields not accessed by an object resp. variable are not members of its new type.

Thus the original lattice provides fine-grained insight into the member access patterns of all objects and variables – a feature very valuable for program analysis and understanding.

2.1.4 Simplification of concept lattice

For practical refactoring purposes, the lattice must be simplified in order to be useful. Typically the lattice size can be condensed by 80% without affecting preservation of behaviour. The user may contribute background knowledge during lattice simplification in order to control the structure of the final refactoring.

Even the simplified lattice may contain multiple inheritance.[2] As an option, automatic elimination of multiple inheritance can be applied (see appendix 1). From a software engineering viewpoint, the resulting final refactorings are the most useful ones.

2.2 Properties of the algorithm

The algorithm has several outstanding properties, which can be summarized as follows:

- it generates a refactoring proposal automatically;
- it guarantees preservation of behavior (for all possible runs in the static variant, for the given test suite in the dynamic variant);
- it refactors with respect to a given purpose, that is, with respect to a given set of clients;
- it thus implements a certain form of program specialization – but as the number or size of clients grows, the refactorings will be more general in nature;
- it is optimal with respect to member distribution: all objects can access only members they really need (thus they usually become smaller);
- it identifies dead methods or fields as a by-product;
- it provides two levels of granularity: the raw lattice, acting like a "spectral analysis" tool allows remarkable insight into program behavior; the simplified lattice provides practical refactoring proposals.

We will illustrate the algorithm by a small example in the next subsection, and will discuss various case studies later in the paper. Right here, we would like to point out an important observation: The Snelting/Tip algorithm is not a software engineer. It is an analysis tool which shows *what can be done without destroying behavior*. KABA refactorings are proposals, and the software engineer decides about their application.

2.3 An example

As a simple example, consider the program in figure 1. B, being a subclass of A, redefines $f()$ and accesses the inherited fields x, y. The main program creates two objects of type A and two objects of type B, and performs some field accesses and method calls.

Figure 2 presents the KABA refactoring proposal, that is, the simplified concept lattice (the intermediate tables as well as the raw lattice are shown in appendix 1; in fact the simplified lattice is only a partial order). Lattice elements are the classes of the new hierarchy. They are marked with class members above, and with variables or objects below. The members above an element (i.e. a new class) define the new class' members; variables below an element will obtain this element as their new type.

The refactoring basically reacts to the different member access patterns of the objects in the program. Typically, a class is split into new unrelated classes if there are objects which access one subset of its members, and other objects which access another, disjoint subset of its members. New subclasses and inheritance relations are introduced if there are objects accessing only a subset of a member set accessed by other objects. In the example, we observe the following:

[2] The Snelting/Tip algorithm was originally designed with C++ in mind, where occasional multiple inheritance does not pose a practical problem.

```
class A {
  int x, y, z;
  void f() {
    y = x;
  }
}

class B extends A {
  void f() {
    y++;
  }
  void g() {
    x++;
    f();
  }
  void h() {
    f();
    x--;
  }
}
```

```
class Client {
   public static void
   main(String[] args) {
     A a1 = new A();  // A1
     A a2 = new A();  // A2
     B b1 = new B();  // B1
     B b2 = new B();  // B2

     a1.x = 17;
     a2.x = 42;
     if (...) { a2 = b2; }
     a2.f();
     b1.g();
     b2.h();
   }
}
```

Figure 1: A small example program

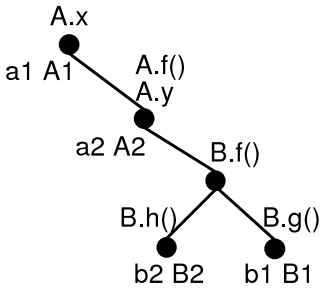

Figure 2: KABA refactoring for figure 1

- The two objects of original type B have different behavior, as one calls g and the other calls h. Therefore, the original B class is split into two unrelated classes.
- The two objects of original type A have related behavior, as $A2$ accesses everything accessed by $A1$, plus $A.f()$. Therefore, the original A class is split into a class and a subclass.
- $A1$ does only contain $A.x$ and not $A.y$. $A.z$ is dead anyway, as it does not appear in the refactored hierarchy. Thus objects become smaller in general, as unused members are physically absent in objects of the new hierarchy.

One might think of simplifying even further by merging the two topmost elements in figure 2, but that would make $A1$ bigger than necessary by including $A.y$ as a member. It is the refactorer's decision whether this disadvantage is out-weighted by a simpler lattice structure. If so, the refactoring editor must guarantee that behavior of all clients is still preserved after simplification.

3. THE KABA SYSTEM

KABA is an implementation of the approach described so far. KABA currently consists of four components: the static analysis, the dynamic analysis, the class hierarchy editor and the byte-code transformation tool KRS.

3.1 Program analysis

The static analysis can handle full Java byte-code and includes a full points-to analysis. Intraprocedural points-to analysis is flow-sensitive and can be parametrized to be context- and object-sensitive.[3]

Stubs are necessary to simulate the behavior of *native* methods. Stubs are provided for the most commonly used native functions of JDK 1.3, additional stubs can be added easily. The analysis can handle reflection features like the `.class` operator precisely, and uses a heuristic if a program loads classes with `Class.forName`. The techniques used to deal with this aspects of reflection could be extended to handle all of reflection.

Although mainly tested with class files generated by *javac* and *jikes*, KABA should be able to handle other Java and non-Java compilers as well. Besides the class files, the analysis needs one or more starting methods (`main`). All code reachable from these methods is included in the analysis. The main limitation of the analysis is its memory requirement. With 2GB of memory programs like *javac* (as of JDK 1.3, 28000 LOC) can be analyzed. Unfortunately 2GB is a technical barrier not easily overcome on most systems.

The dynamic analysis consists of the JVM *Kaffe*[4], whose byte-code interpreter has been modified to track all member accesses during program execution. It supports the same amount of reflection as the static analysis. No stubs for native programs have been provided because experience with the static analysis demonstrated that stubs are important for correct control flow, but not for member accesses. For the instrumented JVM, control flow is correct without stubs.

The output from different program runs are merged into the table format used by the static analysis. We have not observed any limitations in terms of program size, however general limitations of automated testing apply.

3.2 The refactoring editor

The KABA editor computes the (raw or simplified) concept lattice and displays it graphically. Figure 3 presents the lattice for figure 1 in form of a KABA screen shot. Every box represents one class, its name is printed in bold font in the center (nodes containing only members from the same original class C are named $C'n$; users have the option to manually rename classes).

Members are displayed above the class name, variables below it. To reduce the screen space requirements, attributes and objects are not displayed by default; little arrows next to the class name allow to expand them if necessary.[5]

In contrast to the simplified lattice in figure 2 this screen shot features all the ugly details necessary in practice: full method signatures, constructors, and unique object identifiers. Also noticeable are two different displays of methods, as method names are prefixed by either *dcl* or *def*. The *dcl* represents an abstract declaration of a method, whereas a *def* represents the implementation (see Appendix 1). In the example, a method's *dcl* and *def* are always at the same node, but this need not be the case if interfaces or abstract methods are present.

Additional views help users to browse and refactor the hierarchy. The first view (figure 4 upper) shows what "happened" to an original class. It shows all members and their new "home classes", making it easy to see how a class was restructured. Additional util-

[3]By default, context-sensitive analysis is used only for object constructors, while object-sensivity is used for the Java collection classes. These parameters can be tuned individually to achieve best performance.

[4]`http://www.kaffe.org/`

[5]We plan to use UML notation in the future, but right now there is no UML layouter satisfying KABA's needs.

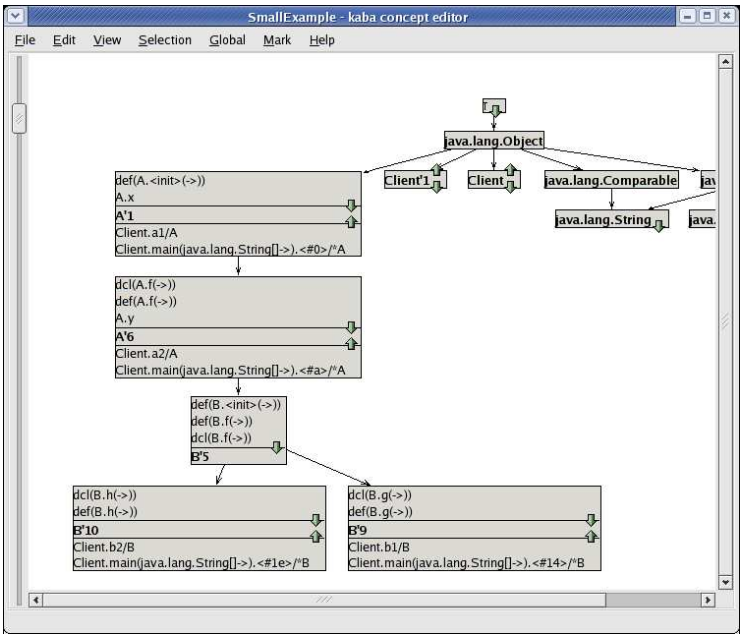

Figure 3: KABA screen-shot for figure 1

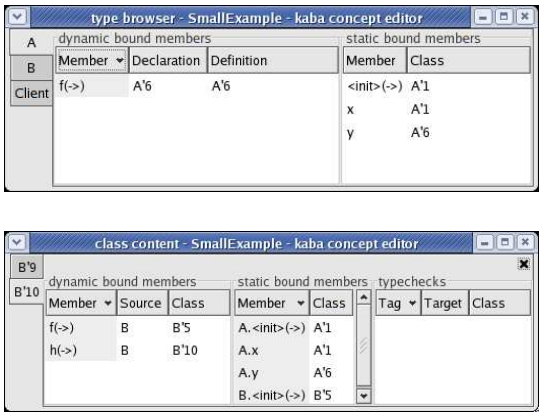

Figure 4: Browser for original types (upper); Browser for class content (lower).

ity functions to mark certain members are also provided.

The second view (figure 4 lower) gives an overview of a new class. In addition to members located in the class itself it includes members from parent classes. KABA displays all members of a class; for inherited members, their new "home class" is given. For every object shown in the class hierarchy, the source code of its creation site can be seen on mouse click.

Besides browsing, the user may modify the class hierarchy. The following basic operations are provided:

- Attributes and objects can be moved in a cut-and-paste fashion. Any number of them can be selected (cut) and moved to any class (paste).
- A class can be split into two. Incoming edges are attached to the first new class, outgoing edges are attached to the second, and the second inherits from the first. All members are moved to the first, all variables to the second class; they may be redistributed later.
- Two classes can be merged into one class, and a class can be made a subclass of another class.

These operations should be sufficient to refactor the class hierarchy in every possible way. Additional convenience functions are provided to make common tasks more easy (e.g. application of the lattice simplification algorithm).

Modification of the class hierarchy is only allowed if it will not affect client behavior. The user is given detailed feedback, if a certain refactoring is not allowed. Figure 5 shows an example: in order to merge classes `A'1` and `A'6` from figure 3, the user tried to move the pointer `a1` from class `A'1` to the subclass. KABA refused as `A'1` also contains an object (`A2` in terms of figure 2) and there is an assignment between the pointer and the object in the program: the proposed pointer move would make this assignment type-incorrect. It would however be valid to move pointer and object to the subclass in one step or just to merge the two classes.

As refactored hierarchies can contain multiple inheritance, the KABA editor will (on demand) mark classes which really inherit

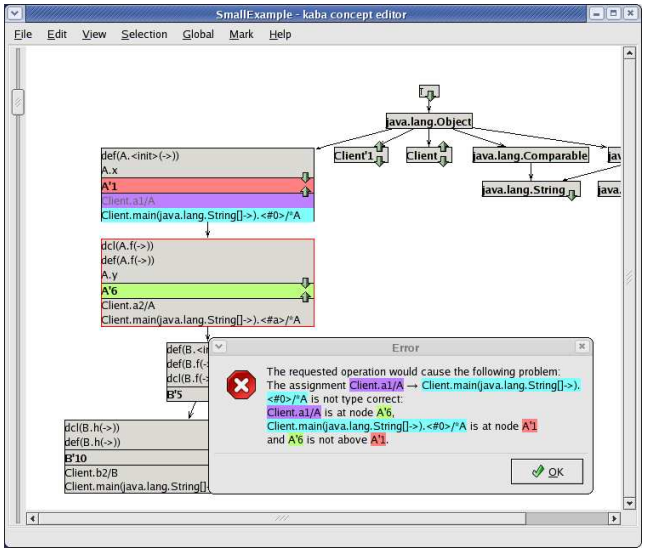

Figure 5: KABA reaction to an illegal refactoring

method definitions from different superclasses[6]. It offers to move method definitions from one of these superclasses further up in the hierarchy, thereby eliminating multiple inheritance.

3.3 Code generation

The final part of KABA is the byte-code transformation tool KRS [12]. Using transformation instructions generated by the editor, it transforms the original byte-code into a new set of classes matching the new hierarchy. The byte-code can be run directly or fed into one of the various byte-code decompilers to regain Java source code.

Code generation consists of two major tasks: first, all fields and

[6]Note that the type constraints prevent that the *same* method is inherited from different superclasses

methods must be reordered according to the new class hierarchy. The second task is more subtle: Every class name in the program must be replaced by a new one. This affects `new` expressions as well as types of local variables, method parameters, fields, `instanceof` operators, type casts and exception handlers. For reflection calls, parameters passed to certain methods are modified to match the new class names. In addition to the type changes, all dead code must be removed as it becomes non-typeable in general.

Currently only the static analysis provides all the information required for code generation, as the dynamic analysis omits information about pointers.

3.4 Application of KABA

KABA is most useful when a library is analyzed together with all clients using this library. It guarantees preservation of behavior for the clients (static analysis) or all test runs (dynamic analysis): for any input, visible program output remains the same.[7] Client code which was not analyzed by KABA may still work with the refactored version, but there is no guarantee. It is however possible to automatically check for preservation of behavior.

For both variants of the analysis, the full byte code must be available. If some methods are implemented as *native* methods, stubs are necessary which emulate member access in the same way the native methods would. The static analysis already contains stubs for the most commonly used native methods of JDK 1.3.

The user must decide which code is subject to refactoring (*user code*) and which part must remain unchanged (*library code*). Usually the analyzed program is user code and the classes from the JDK are library code. If a program requires third party libraries, they are most likely to be included in the library code. Library code must be self-contained, it may not contain references to the user code.

Once the analysis is complete, the KABA editor can be used to browse and modify the transformed class hierarchy. If the static analysis was used, the original byte code can be automatically transformed into code conforming to the refactored classes hierarchy.

4. CASE STUDIES

We have applied KABA to several real world programs. We use a condensed view for the resulting hierarchies in the figures below. Nodes will only contain the names of the original classes from which it contains members. Numbers at the left hand side of a node are used to reference a specific class. The right hand side contains two numbers: the number of pointers (upper half) and the number of objects (lower half) who have that specific type. Common package names are abbreviated "...". The presented hierarchies are not the raw lattices, but have been processed with the algorithm from appendix 1 to reduce their complexity without changing behavior. Interfaces consisting only of constants are omitted.

4.1 The `javac` compiler

Our first example is the Java compiler from Sun's JDK 1.3.1. It was analyzed using the dynamic analysis. The Java library itself was used as test suite, creating 1878 individual compiler runs. All refactorings shown below are guaranteed to preserve at least the behavior of these 1878 test cases.

4.1.1 The symbol table

Figure 6 presents a specific sub-hierarchy from `javac`, namely the symbol table. The topmost symbol table class is `Symbol` with 4 subclasses, 3 of them have further subclasses.[8] For this case study, the rest of the compiler is considered to be client code.

Figure 7 shows the KABA refactoring of the symbol table. The overall structure of the hierarchy is not affected by the transformation: there is still one top class with four subclasses. In two cases even their subclasses remain unchanged (2 , 3 , 15). The class 4 has some of is members moved into new subclasses 11 and 17 . Class 11 also contains members that were previously in the top class. Hence these examples illustrate automatic "extract class" and "move member" / "move method" refactorings.

Something similar happens to the two subclasses of 6 . They are split into a hierarchy containing members of the same original class. Additionally members of the former top class are moved down. For these members multiple inheritance is introduced, as they are used by different subclass branches, but not in their common ancestor class. Only multiple inheritance can visualize this phenomenon in the hierarchy.

But for a practical Java refactoring, multiple inheritance can be easily removed by merging class 8 and 13 with their superclass 6 . The Result can be seen in figure 8.

In general, refactored objects become smaller. The original class `Symbol` had 27 members, only 14 are left in class 1 . Thus the number of members in the top class was reduced by about 50 percent. The number could be further reduced if the constructor would not initialize all data members.

Let us now discuss the KABA refactoring proposal from a software engineering viewpoint. Even after removal of multiple inheritance, new subclasses are proposed in the KABA refactoring. Introducing new subclasses will in general improve information hiding, as certain "secrets" are moved into less-visible subclasses. Considering class cohesion, KABA will introduce new classes only if their methods are executed together and thus improves functional cohesion. But the source code may provide other arguments against splitting classes. Remember: KABA shows what can be done without affecting behavior, but is not a software engineer. The KABA editor is used to modify a refactoring proposal according to software engineering criteria.

In the example, the new subclasses are certainly justified from a software engineering viewpoint, as functional cohesion is much improved. But the more fundamental insight from this case study is that KABA basically reproduced the overall hierarchy structure. We therefore conclude that the original `javac` design was quite good. This example demonstrates that KABA can also be used as a design metrics: it is good if the original hierarchy is more or less reproduced, it is bad if the original hierarchy is completely refactored.

4.1.2 The tree visitor

The second example from `javac` is shown in figure 9; it is the tree visitor sub-hierarchy. The refactored KABA version of the sub-hierarchy (figure 10) is completely unchanged. However the "move member" / "move method" refactoring has been applied: several members have been moved into subclasses. The new class 1 consists of only 3 methods, 6 other methods from the original class are distributed to classes 8 ,9 ,10 and 11 , while 28 methods are presumed dead. As this is a very high number, we manually inspected the code and found that the original class is an abstract visitor with one method overloaded for 36 different syntactical units. But only 8 of them are actually called, the used versions are usually overwritten in a subclass.

If the dead (!) code is removed, the program produces a run-time

[7] As objects become smaller, KABA may act as a space optimizer; as the number of classes usually increases, some runtime overhead may be introduced. Both effects have not yet been evaluated.

[8] `Symbol$VarSymbol` indicates this class is an inner class of `Symbol`. An inner class may be a subclass of its outer class.

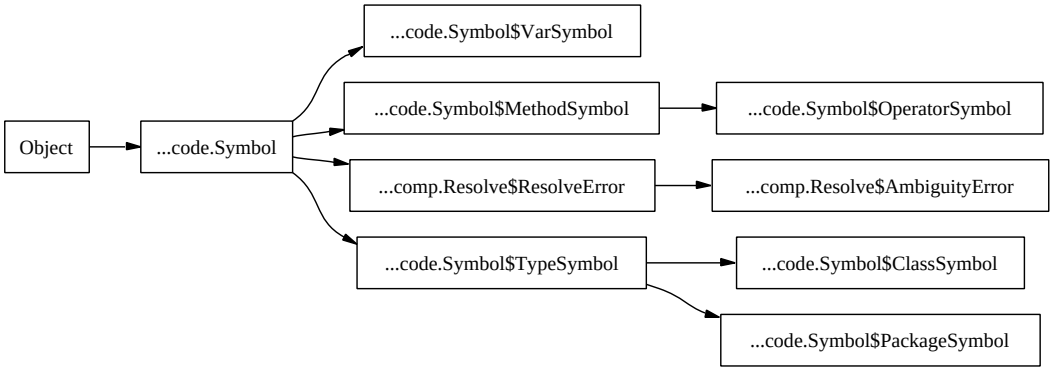

Figure 6: Original class hierarchy for `javac` symbol table

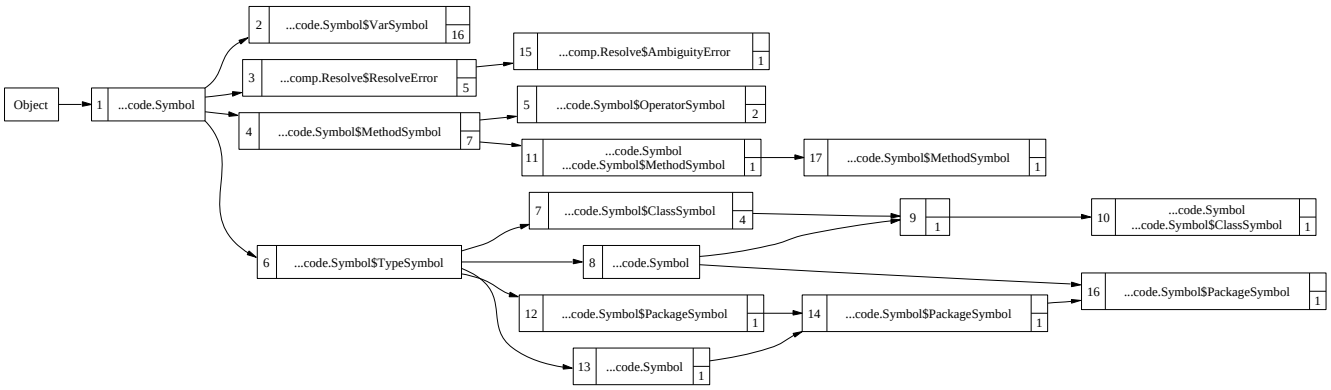

Figure 7: KABA refactoring for figure 6

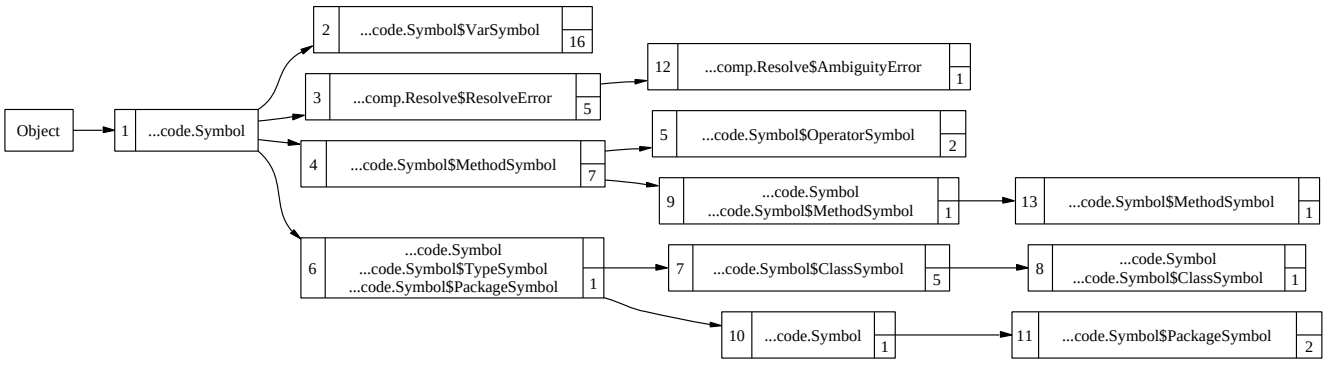

Figure 8: Final refactoring for figure 6 after automatic elimination of multiple inheritance

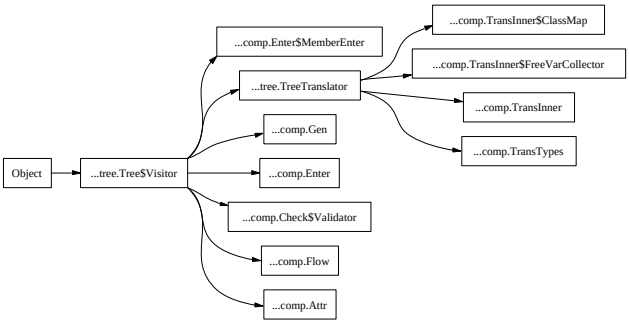

Figure 9: Original class hierarchy for `javac` tree visitor

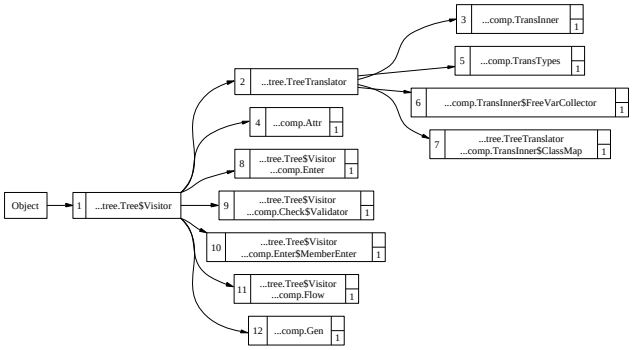

Figure 10: KABA refactoring for figure 9

error! This design seems questionable. Having dead code for future extensibility may be ok, but having dead code which breaks the program if removed, is a pitfall for programmers not fully familiar with the code. The visitor pattern should be implemented in such a way that removing dead code produces a compile-time error instead of throwing an exception.

In general, it must be decided individually weather the inclusion of dead code for future extensibility is a good idea. In some cases it will cause maintenance problems, as unused code is especially hard to maintain. In other cases omitting it may prevent all future extensions (e.g. removing the only public or protected constructor).

Thus KABA replicated the given hierarchy, assuring that the original design was basically good. But KABA still uncovered a minor design problem, as KABA identifies dead members as a by-product.

4.1.3 The syntax tree

Figure 11 shows the abstract syntax tree sub-hierarchy. It is exactly reproduced by KABA, even without automatic elimination of multiple inheritance. The top class contains just data members and access methods for these members. The subclasses represent the 36 syntactical units mentioned in the previous example.

The exact hierarchy reproduction shows that the members are used in the same way by all subclasses. Hence cohesion for this particular class is high. This indicates a good design which should be left unchanged. KABA confirms that `javac` was designed by experienced software engineers.

4.2 The `antlr` parser generator

Our next case study is `antlr`, a popular parser generator. For the dynamic analysis, all example grammars from the `antlr` distribution were used, creating 84 test runs. The `antlr` Java runtime classes were not analyzed.

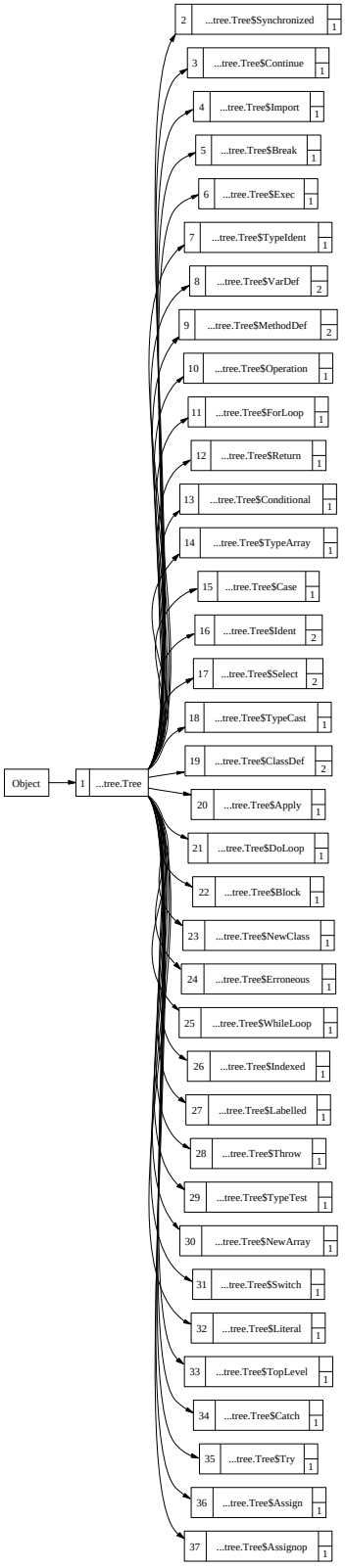

Figure 11: Original class hierarchy and KABA refactoring for `javac` syntax trees

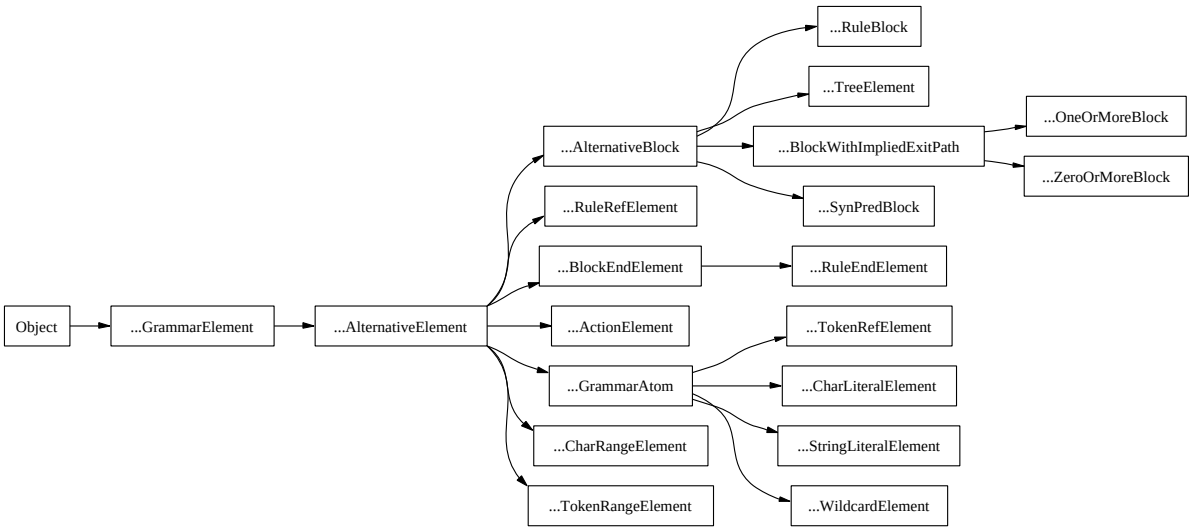

Figure 12: Original hierarchy for a part of `antlr`

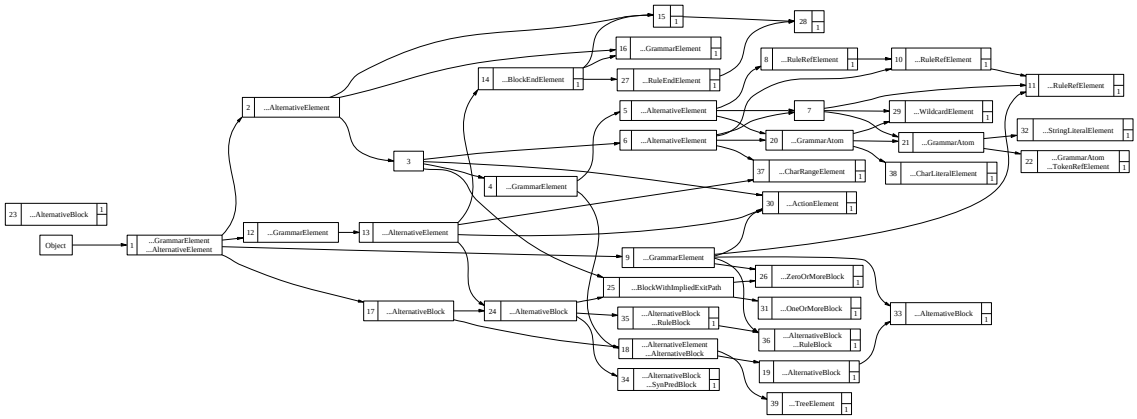

Figure 13: Fine-grained KABA refactoring proposal for figure 12 using dynamic analysis

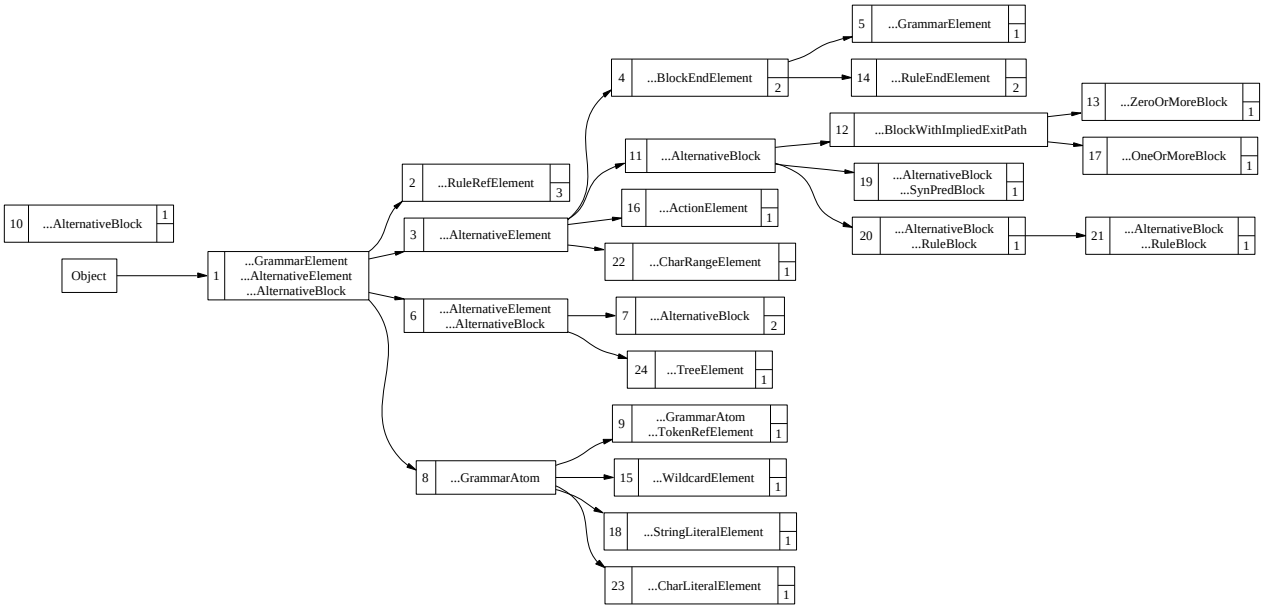

Figure 14: Final KABA refactoring proposal for figure 12 after removing multiple inheritance from figure 13

Figure 15: Original hierarchy for another part of `antlr`

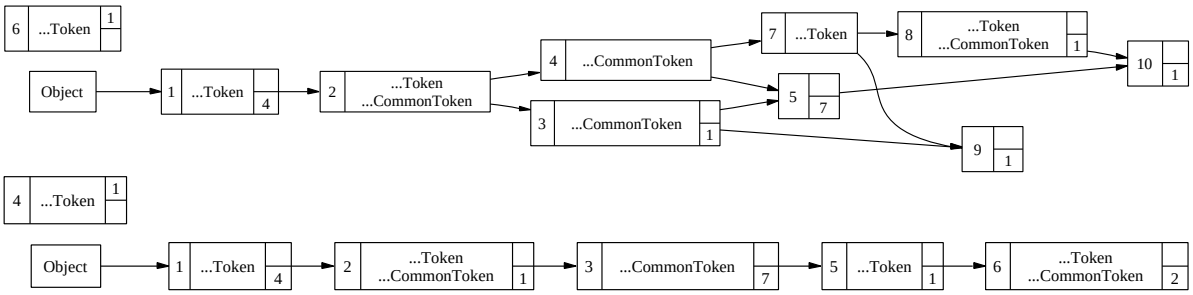

Figure 16: Fine-grained refactoring for figure 15 (upper); final refactoring after removal of multiple inheritance (lower).

4.2.1 First Example

Figure 12 shows part of the Antlr original hierarchy and figure 13 the fine-grained KABA refactoring. It is obvious that this hierarchy is much more changed than any of the `javac` examples:

1. 6 out of 20 classes have their members distributed to more than one class; most noticeable `AlternateBlock`, which was split into 9 classes, `AlternativeElement` (split into 6) and `GrammarElement` (split into 5).

2. The original hierarchy has `GrammarElement` and `AlternativeElement` as super classes of all other classes. In the new hierarchy a topmost class still exists (1), but it contains members from both classes. This indicates that the distinction between the two classes is redundant.

3. Originally `AlternativeBlock` was a subclass of `AlternativeElement`. This relation is weaker in the new hierarchy, as a lot of members from `AlternativeElement` are no longer contained in the classes which have members from `AlternativeBlock`. This indicates that the inheritance between these two classes is a candidate for further inspection.

4. The isolated class 23 contains only a static member. As static members are not influenced by member accesses, they appear as individual nodes in the new hierarchy and may be manually moved to any other class.

The fine-grained refactoring proposal looks pretty complex and is not realistic anyway, as it contains a lot of multiple inheritance. Still, KABA already demonstrated that the "natural" design from figure 12 does not stick to the principle of functional cohesion, and that refactoring as sketched in items 1.-4. can be done without affecting behavior.

Figure 14 shows the result of automatically removing multiple inheritance as described in appendix 1. Now the refactoring looks much more convincing! Note that preservation of behaviour is still guaranteed; items 1.-4. above still apply, but the overall structure is much more similar to the original hierarchy. But classes have been merged or splitted, new subclasses have been introduced, and members have been moved. The result is a dramatic increase in functional cohesion.

4.2.2 Second Example, Dynamic Variant

For our second example, the original hierarchy is shown in figure 15, the refactored version can be seen in figure 16. The original hierarchy consists of 3 classes, but the fine-grained refactoring triples this number:

1. No members of the original class `CommonHiddenStreamToken` are in the refactored version, revealing the original class to be dead code.

2. Some members of the former top class `Token` were moved into subclasses 7 and 8 . Manual inspection reveals that these members are methods called `getType` and `setType`, which are accessor functions for a data member `type`. This indicates some functionality of `Token` concerning `type` may be moved to a subclass. `type` itself however is contained in class 1, as it is accessed by the default constructor.

The multiple inheritance introduced by KABA can be removed by collapsing the diamond of classes 4 , 3 , 2 and 5 . As a result, the refactored hierarchy will be a chain just like the original hierarchy, but more fine grained (see figure 16). Thus the KABA refactoring improves locality and cohesion.

4.2.3 Second Example, Static Variant

The fine-grained class hierarchy created by the static analysis can be seen in figure 17. The hierarchy has more than thrice as many classes as the dynamic variant. It is not useful as a practical refactoring, but presents a detailed "spectral analysis" of true object behaviour. This feature is valuable for various program understanding tasks. In particular, the static analysis is useful in areas where automated testing is difficult or the transformed code is needed. The static variant also generates refactoring proposals for interfaces which is not possible with the dynamic analysis.

For example, 10 classes are not below `Object`. While in Java everything is a subclass of `Object`, in KABA this need not always be the case. Classes not below `Object` may contain data members, but are never instantiated, as an instance would require access to the constructor of `Object`. Non-instantiated classes are good candidates for the "create interface" refactoring.

The static variant, being a conservative approximation, contains some additional lattice elements due to imprecision of the underlying points-to analysis. But the main reason for the higher number of classes in the static refactoring is the inclusion of pointers (which

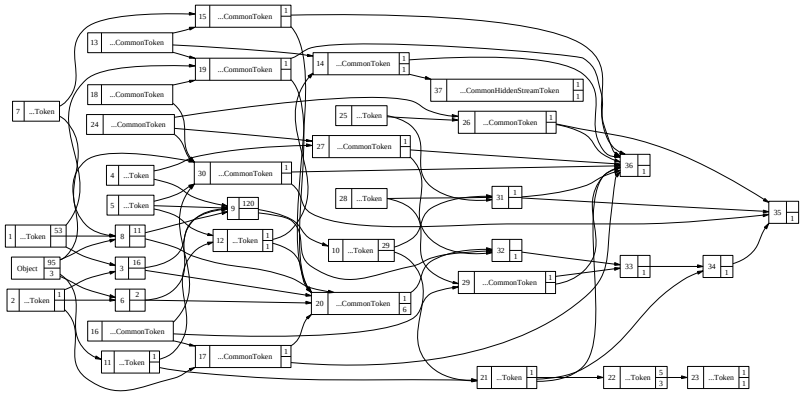

Figure 17: Fine-Grained KABA refactoring proposal for figure 15 using static analysis

Figure 18: Final KABA refactoring for figure 15 using static analysis

objects	3	1	1	5	1	1	1	1	1	1
dynamic	1	1	3	5	5	5	6	8	9	10
static	22	23	20	20	32	33	11	36	34	35

Figure 19: Position of objects in the dynamic and static analysis

are left out by the dynamic analysis). Often, lattice elements are created just because pointers of the same original type access different member subsets, even though this does not hold for the objects pointed to. This is the reason that including pointers is interesting for program understanding, but not for practical refactoring.

In order to compare precision of the static and dynamic analysis, Figure 19 correlates individual objects for the static and dynamic analysis. For example the first column means that 3 objects which are located in class 1 in the dynamic analysis are in class 22 in the static analysis. Remarkably, there is an example where the dynamic analysis is more detailed (columns 3 and 4; these objects are in the same class in the static analysis and in different classes in the dynamic analysis) as well as examples where the static analysis is more detailed (columns 4 to 6). This means that the level of detail of the refactored class hierarchy does not always stem from static vs. dynamic analysis, but may come from the analyzed code itself.

In rare cases, the conservative approximation generates artifacts, in particular if reflection is used. For example, the static refactoring contains members of `CommonHiddenStreamToken` in class 37 , which were presumed dead by the dynamic analysis. It seems that the static analysis includes code which was not covered by the test suite of the dynamic analysis. But this is not the case! Class 37 (as well as 14 and 12) are results of the approximation of Java's reflection capabilities. `antlr` uses reflection, but not within classes considered in this example. Thus the inclusion of members from class 37 is an artifact of statically approximating reflection.

The final static refactoring, based on agressive simplification of figure 17, can be seen in figure 18. It is even more coarse-grained than the dynamic version. In fact, the static refactoring can be obtained from the dynamic variant by merging classes 1/2 resp. 5/6, which preserves behaviour. It is the engineers task to decide which refactoring is more appropriate. In any case, the original distinction between `Token` and `CommonToken` was not designed properly, and KABA shows how to improve the hierarchy.

4.3 Discussion

We have seen that KABA can distinguish designs which respect actual member access patterns (and thus have high cohesion and good locality) from designs where this is not the case; in the latter situation, KABA provides practically useful proposals for refactoring. For practical refactorings, aggresive lattice simplification and automatic elimination of multiple inheritance must be used. The case studies have shown that the final refactorings definitely improve the quality of the design.

Still, the fine-grained lattice offers another important KABA feature: the possibility to obtain fine-grained insight into member access patters. KABA can act as a "spectral analysis" for a hierarchy, telling the engineer what the objects really do. Fine-grained analysis is also useful if the KABA lattice is used as a quality metrics for old or new programs: simple lattices are better than complex ones; lattices replicating the original design are better than lattices introducing many new classes and inheritance relationships. Whenever KABA proposes a refactoring which substantially differs from the manual design, classes and objects do not stick to the principle of functional cohesion. During development of new code, designers can react to this lack of software quality by applying some or all KABA refactorings.

Let us finally repeat a fact which was mentioned before: a KABA refactoring is just a *proposal*; it presents changes which *can be applied without changing behavior*. There may be good reasons not to apply some or all of these changes, like future extensibility or cohesive grouping of members into classes. Similarly, subsequent manual refactorings should obey software engineering criteria: "what can be done" is not always the same as "what should be done". Naturally, the decision to apply a refactoring requires a certain familiarity with the code.

5. RELATED WORK

In the recent Dagstuhl seminar "Program Analysis for Object-Oriented Evolution", analysis researchers met with refactoring researchers in order to explore the potential for program analysis in refactoring [16]. One insight of this workshop was that in a refactoring context, 100% preservation of behaviour is not always an objective. KABA reacts to this insight by offering a static variant which preserves the behavior of all clients, and a dynamic variant which preserves only the behavior of a given test suite.

Other authors proposed methods for automated refactoring. None of these are client-specific, which means they are valid for all clients, but also prevents client-specific refactorings. Opdyke [10] originally introduced the concept of refactoring a class hierarchy. Casais [4] was among the first authors to investigate automated refactoring. Unfortunately, his algorithm does not provide semantic guarantees. Moore [9] not only refactors classes, but also statements e.g. by extracting common subexpressions. His algorithm is for the dynamically-typed language *self*. Naturally, behavior guarantees are not provided, and realistic applications have not been reported. Bowdidge and Griswold [3] present semantics-preserving restructuring transformations for procedural programs based on so-called star diagrams; as it happens, star diagrams have similarities to concept lattices. Kataoka et al. [7] automatically extract simple invariants from Java source code and use these to derive code-level refactorings such as *Encapsulate Downcast* or *Remove Parameter*.

Tip et al. [15] also uses type constraints in order to support the refactorings *Extract Interface*, and *Pull up / Push down member*. Tip *does* guarantee behavior preservation, and offers an interactive tool similar to the KABA editor as part of Eclipse. Automatic generation of refactorings is however not supported. Earlier work by Tip described an algorithm to specialize a class hierarchy with respect to one client [17]; this method also influenced KABA, but to our knowledge was never implemented.

Identification of dead fields and methods is usually based on static analysis such as RTA [1], and often used in practical tools such as JAX [18]. KABA is a more general analysis which includes dead members as a by-product.

6. CONCLUSION

The original version of the Snelting/Tip algorithm was already published in 1998, but it took us several years to develop KABA on the basis of this algorithm, refine the foundations in order to handle full Java, add the dynamic analysis variant, find good lattice simplifications, and make KABA work for realistic applications. Compared to [13] our achievements can be summarized as follows.

- Today, the static KABA variant can handle up to 30000 LOC, while the new dynamic variant has no program size limitation.
- The innovative KABA refactoring editor guarantees behavior preservation for client-specific as well as general refactorings.
- Case studies have shown that KABA is helpful for analyzing class hierarchies, and generates practically useful refactorings.

It is interesting to note that KABA based on dynamic analysis works fine in practice, hence the much higher cost of static analysis and full behavior preservation seems questionable. Future work must show whether this observation represents a new trend in program analysis for software tools.

Acknowledgments. This work was funded by Deutsche Forschungsgemeinschaft, grants DFG Sn11/7-1 and Sn11/7-2.

7. REFERENCES

[1] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proc. 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, pages 324–341, 1996.

[2] K. Beck. *Extreme Programming Explained*. Longman Higher Education, 2000.

[3] R. Bowdidge and W. Griswold. Supporting the restructuring of data abstractions through manipulation of a program visualization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7:109–157, 1998.

[4] E. Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115, 1994.

[5] M. Fowler. *Refactoring*. Addison-Wesley, 1999.

[6] B. Ganter and R. Wille. *Formal Concept Analysis - Mathematical Foundations*. Springer Verlag, 1999.

[7] Y. Kataoka, M. Ernst, W. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. International Conference of Software Maintenance (ICSM'01)*, 2001.

[8] O. Lhotak and L. Hendren. Scaling Java points-to using Sparc. In *Compiler Construction, 12th International Conference*, LNCS, pages 153–169, 2003.

[9] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proc. 11th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'96)*, pages 235–250, 1996.

[10] W. Opdyke and R. Johnson. Creating abstract superclasses by refactoring. In *ACM 1993 Computer Science Conference*, 1993.

[11] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *Proc. 16th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, pages 43–55, 2001.

[12] P. Schneider. Umsetzung von Transformationen an Klassenhierarchien in der Sprache JAVA. Master's thesis, Universität Passau, 2003.

[13] G. Snelting and F. Tip. Understanding class hierarchies using concept analysis. *ACM Transactions on Programming Languages and Systems*, pages 540–582, May 2000.

[14] M. Streckenbach. Points-to-Analyse für Java. Technical Report MIP-0011, Fakultät für Mathematik und Informatik, Universität Passau, 2000.

[15] F. Tip, A. Kiezun, and D. Baeumer. Refactoring for generalization using type constraints. In *Proc. 18th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, pages 13–26, 2003.

[16] F. Tip, G. Snelting, and R. Johnson. Program analysis for object-oriented evolution. Technical report, Dagstuhl Seminar Report 03091, 2003.

[17] F. Tip and P. F. Sweeney. Class hierarchy specialization. *Acta Informatica*, 36:927–982, 2000.

[18] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter. Practical extraction techniques for java. *ACM Trans. Prog. Lang. Syst.*, 24(6):625–666, November 2002.

Appendix 1: The Snelting/Tip Algorithm

In order to keep this contribution self-contained, this appendix explains the main steps of the refactoring algorithms in a more technical way. Full details can be found in [13].

Collecting member accesses

The algorithm is based on a fine-grained analysis of object accesses. For all objects or object references o, it determines whether member m from class C is required in o. This information is extracted from a given hierarchy and its clients by (static or dynamic) program analysis. The result is a binary relation, coded in form of a table T.

For the example in figure 1, table T contains rows for object references $a1$, $a2$, $b1$, $b2$, $A.f.this$, $B.f.this$, $B.g.this$, $B.h.this$, as well as for object creation sites $A1$, $A2$, $B1$, $B2$.[9] Columns are labeled with fields and methods $A.x$, $A.y$, $A.z$, $A.f()$, $B.f()$, $B.g()$, $B.h()$. For methods, there is a distinction between declarations and definitions (i.e. implementations), that is, between $dcl(C.f())$ and $def(C.f())$, which makes the analysis more precise [13].

Dynamic Variant

KABA offers two variants of table construction, a static and a dynamic one. The dynamic variant analyzes member accesses for a given test suite. The JVM is instrumented such that every member access $O.x$ from a true object O (resp. its creation site) gives rise to a table entry

$$(O, C.m)$$

where $O = (C, m, i)$ is the object creation site at instruction i in method m of class C. Method calls $O.f()$ give rise to table entry

$$(O, def(C.f()))$$

For references, no entries are generated.

Static Variant

In the static variant, points-to analysis is used to determine for an object reference o to which object creation sites it might point to at runtime; this set is denoted

$$pt(o) = \{O_1, O_2, \ldots\}$$

$pt(o)$ may be too big (i.e. imprecise), but never too small (i.e. pt is a conservative approximation). Today, reasonable efficient and precise points-to analysis exists for Java, e.g. [14, 11, 8].

Now let $Type(o) = C$ be the static type of o, and let member accesses $o.m$ resp. $o.f()$ be given. Table T will contain entries

$$(o, C.m)$$

resp.

$$(o, dcl(C.f())$$

Furthermore, entries

$$(O, def(C.f()))$$

are added for all $O \in pt(o)$ where $C = StaticLookup(Type(O), f)$.

For the above example, the resulting table is shown in figure 20. Note that it contains some additional entries for *this*-pointers which are explained in [13]. In this simple example, static and dynamic table are identical, but in general this is of course not the case: due to the principle of conservative approximation, the entries of the static table are a superset of the entries of any dynamic table.

[9] Program analysis usually does not distinguish runtime objects created by the same `new` statement, so it is a standard technique to identify objects with the same creation site. Thus in the following "object" O in fact stands for O's creation site, coded as a triple $O = (C, m, i)$ of class C, method m and instruction address i.

	A.x	A.y	A.z	dcl(A.f)	def(A.f)	dcl(B.f)	def(B.f)	dcl(B.g)	def(B.g)	dcl(B.h)	def(B.h)
a1	×										
a2	×			×							
b1								×			
b2						×				×	
A1											
A2					×						
B1							×		×		
B2							×				×
A.f.this	×	×			×						
B.f.this		×					×				
B.g.this	×					×			×		
B.h.this	×					×					×

Figure 20: Member access table for figure 1

	A.x	A.y	A.z	dcl(A.f)	def(A.f)	dcl(B.f)	def(B.f)	dcl(B.g)	def(B.g)	dcl(B.h)	def(B.h)
a1	×										
a2	×			×							
b1								×			
b2	×			×		×				×	
A1	×										
A2	×	×		×	×						
B1	×	×		×		×	×	×	×		
B2	×	×		×		×	×			×	×
A.f.this	×	×		×	×						
B.f.this		×		×		×	×				
B.g.this	×	×		×		×	×	×	×		
B.h.this	×	×		×		×	×			×	×

Figure 21: Table after incorporating type constraints

Type constraints

In a second step, a set of type constraints is extracted from the program, which are necessary and sufficient for preservation of behavior. The refactoring algorithm computes a new type (i.e. class) for every variable or class-typed member field, and a new "home" class for every member. Therefore, constraints for a variable or field x are expressed over the (to be determined) new type of x in the refactored hierarchy, $type(x)$; constraints for a member or method $C.m$ are expressed over its (to be determined) new "home class", $def(C.m)$.

There are basically two kinds of type constraints. First, any assignment `x = y;` gives rise to a type constraint $type(y) \leq type(x)$. Such a constraint will be generated not only for explicit assignments, but also for implicit assignments due to parameter and return values, and even for implicit assignments to this-pointers. Another simple type constraint requires that for all methods $C.f$, $def(C.f) \leq dcl(C.f)$.

The second set of type constraints is more difficult to understand. These *dominance constraints* are necessary whenever a member m is defined in a class A and a subclass $B \leq A$. In order to avoid ambiguous member access in the refactored hierarchy, sometimes $B \leq A$ must be retained. More precisely, if subclass B of A redefines a member or method m, and some object x accesses both $A.m$ and $B.m$ (that is, $\exists x : (x, def(A.m)) \in T \wedge (x, def(B.m)) \in T$), then $def(B.m) < def(A.m)$ must be retained in order to avoid ambiguous access to m from x [13].

Once all type constraints have been extracted, they are incorporated into table T. To achieve this, we exploit the fact that a constraint can be seen as an *implication* between table rows resp. columns, and that there is an algorithm to incorporate any given set of implications into a table [6]. First we observe that even in the refactored hierarchy, a subtype inherits all members from its super-type. Therefore $type(y) \leq type(x)$ enforces that any table entry for x must also be present for y; that is $\forall m : (x, m) \in T \Rightarrow (y, m) \in T$, or $x \rightarrow y$ for short. Second, $def(B.m) < def(A.m)$ enforces that any table entry for $def(B.m)$ must also be present for $def(A.m)$, which is written as $def(B.m) \rightarrow def(A.m)$.[10]

Reconsidering figure 1, the following assignment constraints are collected in form of implications:

$$A.y \rightarrow A.x, A.f.this \rightarrow a2, B.f.this \rightarrow a2,$$
$$B.g.this \rightarrow b1, B.h.this \rightarrow b2, a1 \rightarrow A1,$$
$$a2 \rightarrow A2, b1 \rightarrow B1, b2 \rightarrow B2, a2 \rightarrow b2$$

Furthermore, the following dominance constraints are collected:

$$def(B.f) \rightarrow def(A.f), dcl(B.f) \rightarrow dcl(A.f)$$
$$def(A.f) \rightarrow dcl(A.f), def(B.f) \rightarrow dcl(B.f),$$
$$def(B.g) \rightarrow dcl(B.g), def(B.h) \rightarrow dcl(B.h),$$

These implications are easily incorporated into table 20 by copying row entries from row y to row x resp. column entries from column $def(A.f)$ to column $def(B.f)$ etc. Note that in general there may be cyclic and mutual dependences between row and/or column implications, thus a fix-point iteration is required to incorporate all constraints into the table. The final table for figure 20 is presented in figure 21.

Concept lattices

In a final step, concept analysis [6] is used to construct the refactored hierarchy from table T. Concept analysis can always be applied whenever hidden hierarchical structures have to be extracted from a given binary relation. The standard example is shown in figure 22. The table encodes a binary relation between "objects" O (in our case the planets) and "attributes" A, thus $T \subseteq O \times A$. From the table, the corresponding concept lattice is computed by some smart algorithm [6]. The elements of this lattice are labeled with "objects" and "attributes"; $\gamma(o)$ is the lattice element labeled with $o \in O$, $\mu(a)$ is the element labeled with $a \in A$. The lattice has the following characteristic properties:

1. $(o, a) \in T$ iff $\gamma(o) \leq \mu(a)$, that is object o has attribute a iff o appears below a in the lattice;
2. The supremum $\gamma(o_1) \sqcap \gamma(o_2)$ represents all attributes common to both o_1 and o_2;
3. The infimum $\mu(a_1) \sqcup \mu(a_2)$ represents all objects having both attributes a_1 and a_2;
4. $\mu(a_1) \leq \mu(a_2)$ iff $a_1 \rightarrow a_2$ (a_1 implies a_2), that is, if a_1 appears below a_2 in the lattice, all objects having attribute a_1 also have attribute a_2.
5. $\gamma(o_1) \leq \gamma(o_2)$ iff $o_2 \rightarrow o_1$, that is, if o_1 appears below o_2, all attributes fitting to o_2 will also fit to o_1.

Note that in big tables, common attributes (suprema), common objects (infima), and implications are not at all obvious from the table alone. One reason is that the concept lattice for a table is invariant against row or column permutations.

[10] Note that $x \rightarrow y$ is an implication between row labels, while $def(B.m) \rightarrow def(A.m)$ is an implication between column labels. Therefore the direction of the second implication is "reversed". But the effect is the same: if $x \rightarrow y$ holds in T, then y appears below x in the refactored hierarchy; if $def(B.m) \rightarrow def(A.m)$ holds in T, then $def(B.m)$ appears below $def(A.m)$ in the refactored hierarchy [6].

	small	medium	large	near	far	moon	no moon
Mercury	×			×			×
Venus	×			×			×
Earth	×			×		×	
Mars	×			×		×	
Jupiter			×		×	×	
Saturn			×		×	×	
Uranus		×			×	×	
Neptune		×			×	×	
Pluto	×				×	×	

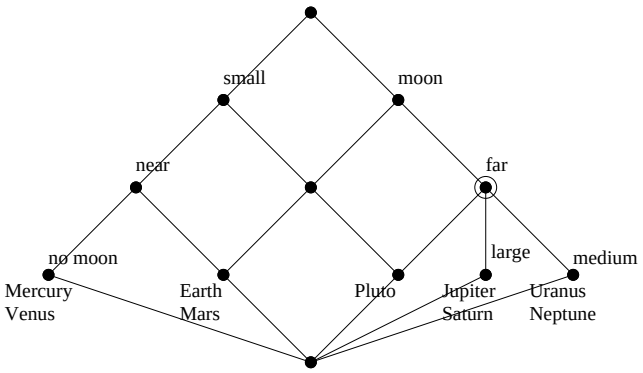

Figure 22: Example table and its concept lattice

The concept lattice for figure 1, as constructed from table 21, is given in figure 23. Concept lattices can naturally be interpreted as inheritance hierarchies as follows. Every lattice *element* represents a *class* in the refactored hierarchy. Method or field names *above* an element represent the *members* of this class. Objects or references *below* an element will have that element (i.e. class) as its new *type*. In particular, all objects now have a new type which contains only the members the object really accesses.

Typically, original classes are split and new subclasses are introduced. This is particularly true for figure 23, where the raw lattice introduces 12 refactored classes instead of the original two. These new classes represent object behavior patterns: $a1$ and $A1$ use $A.x$ but nothing else, which is clearly visible in the lattice. $a2$ additionally calls $a.f()$ and thus needs the declaration of this method. $b2$ calls $B.h(), B.f()$ plus anything called by $a2$. The "real objects" $A2, B2, B1$ are located far down in the lattice and use various subsets of the original members. $B2$ in particular not only accesses everything accessed by $b2$, but also calls $B.f()$ and thus needs $def(B.f())$ (references need "dcl", objects need "def"), which causes one of the multiple inheritances in the raw lattice.

Note that the raw lattice clearly distinguishes between a class and its interface: several new classes (e.g. the one labeled $dcl(B.g())$ in figure 23) contain only $dcl(...)$ entries, but no (inherited) $def(...)$ entries or fields, meaning that they are interfaces.

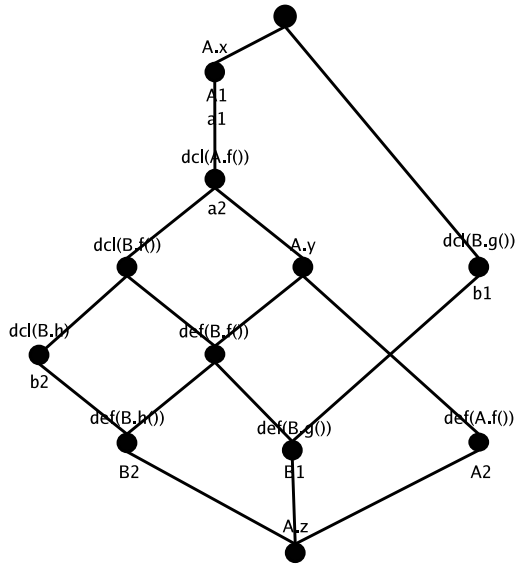

Figure 23: Concept lattice for figure 1, generated from figure 21

The lattice guarantees preservation of behavior for all clients [13]. It is rather fine-grained, and in its raw form represents the most fine-grained refactoring which respects client behavior.

Lattice simplification and elimination of multiple inheritance

From a software engineering viewpoint, the lattice must be simplified in order to be useful. For example, "empty" elements (i.e. new classes without own members) such as the top element in figure 23 can be removed; multiple inheritance can often be eliminated, and lattice elements can be merged according to certain (behavior-preserving) rules. In particular, the distinction between a class and its interface can be removed by merging lattice elements.

For simplification of the class hierarchy, we apply several transformations:

- If q is the only subclass of p and there are no instances of q, merge p and q.
- If p is the only superclass of q and q does not contain any members, merge p and q.
- If p is the only superclass of q and both classes contain only members of the same original class, merge p and q.

These transformations are repeated until a fixpoint is reached. They simplify the structure, but never affect the semantics of the hierarchy. Repeated application of the transformations to figure 23 resulted in figure 2. For example, the first transformation can be applied to the nodes labelled $a2$ and $A.y$, as the latter has no instances (no variables appear below the node).

The new hierarchy may exhibit multiple inheritance between the new classes. Many of these classes can be made interfaces (and the lattice be simplified as above), but cases remain in which a class inherits a non-abstract method from more than one superclass. These can always be removed manually without changing behavior.

In order to eliminate multiple inheritance automatically, more aggressive transformations are needed. It must be checked explicitely whether their application affects the semantics, that is, whether all type constraints are still valid after application.

- If q inherits members from p and p', and the only superclass of p' is $\top$, make p' a subclass of p.
- If q inherits members from p and p' with $p \not\leq p'$ and $p' \not\leq p$: Let r be a superclass of p with $r \leq p'$. If r is the only superclass of p, merge r and p, else move all members of p to r.
- If q inherits members from p and p' with $p \not\leq p'$ and $p' \not\leq p$, and p is a superclass of q and there are no instances of p, merge p and q

Repeated application of these transformations to figures 7, 13, 17 resulted in figures 8, 14, 18 respectively. In these examples, the application of the transformations had to be checked excessively for preservation of semantics, making removal of multiple inheritance expensive.

Dead variables, fields and methods

In the refactored hierarchy the top element is called $\top$, which is different from the node for `java.lang.Object`. One might argue they should be identical as in Java everything is derived from `Object`. The difference is that objects directly below $\top$ are dead. Similarly, fields and methods appearing directly above $\bot$ are dead as well. In figure 23, field $A.z$ is dead.

Appendix 2: Language Details

Several Java features require additional treatment [13], which will be sketched in the following. We would like to point out that full Java can be handled.

Libraries

We distinguish objects whose type is defined in user code, and objects whose type is defined in library or API code. Library code is never refactored. Nevertheless, all objects created (even those created inside library code) must be taken into account for the static analysis (in particular points-to analysis), as they impact the control flow of the analyzed program and may influence which members of the relevant objects are accessed.

The Java API also contains native code. These methods can access members too and do so in practice. For each of these methods a stub must be provided, which must be equivalent in terms of member access or dynamic type checks.

The effects of library code should not be underestimated. Even small Java programs load a huge amount of library code[11], providing big problems for the scalability of the static analysis. But analysis of this code and careful handling of native code is absolutely necessary when it comes to preservation of behavior. [12]

[11] As of JDK 1.4.2, a "hello world" example loads 248 library classes.

[12] Here is a small example:

```
class Main {
    String toString() { return "Hello,
World"; }
    public static void main(String args[]) {
    System.out.println(new Main());
    }
}
```

Without handling the effects of library code and native methods, the method `toString` will be declared dead, obviously breaking the behavior of the program. This is no esoteric example, code like this can be found in many Java programs.

Treatment of `instanceof`

Like object creation sites, different uses of the `instanceof` operator in the program are distinguished by their byte-code address. For an occurrence x `instanceof` T at site C in the program, two additional attributes (table columns) are generated: $C = true$ and $C = false$. For every $o \in pt(x)$ a table entry $(o, C = true)$ is generated if $type(x) \leq T$, a table entry $(o, C = false)$ otherwise.

In the class hierarchy, all objects returning true for the expression in the program will appear below $\mu(C = true)$. When code is regenerated, $\mu(C = true)$ is the new type for T in the original expression. The attribute $\mu(C = false)$ only becomes relevant for editing the class hierarchy. Variables below $\mu(C = false)$ may never be below $\mu(C = true)$ as well, because the transformed `instanceof` operator will match every object of classes below $\mu(C = true)$.

The result of the `instanceof` may be always true (indicated by $\mu(C = false) = \bot$) or never true ($\mu(C = true) = \bot$). In the latter case, the whole operator could be replaced by `false` (more aggressive dead code elimination is also possible). Unfortunately this is not possible for the "always true" case, as x may be `null`, causing the operator to return `false`, so the operator could be replaced by `x!=null`, also enabling further optimizations.

Treatment of Type Casts

Type casts are handled in a similar fashion. If `(T)x` is in the program at site C, two attributes, $C = true$ and $C = false$ are generated, objects $o \in pt(x)$ passing the type cast will create a table entry $(o, C = true)$, if the cast is not possible $(o, C = false)$ is generated. The new cast then can be rewritten to $(\mu(C = true))x$.

But for type casts the situation that the cast is never successful is more complicated. A little example illustrates this:

```
class A {
   void f() { }
}
class B extends A {
}
```

```
A a=new A();
a.f();
B b=(B)a;
b.f();
```

In this example no object gets a table entry at column $C = true$ because the cast always fails, so $\mu(C = true) = \bot$. But $\bot$ is not a type and cannot be used in the transformed program. To handle this for every cast, an additional pointer x/T, representing the result of cast, is created and the objects successfully casted are assigned to it ($o \in pt(x) \wedge Type(o) \leq T \Rightarrow x/T = o$). This pointer is further used to collect the member accesses from the casted value (e.g. a table entry $(a/B, dcl(B.f))$ would be generated for the example program). Because of the assignment, $\mu(C = true) \leq \gamma(a/B)$ is always valid. If $\mu(C = true) = \bot$, $\gamma(a/B)$ can be used as type for the result, but not for the typecast as there is no guarantee that no object is below $\gamma(a/B)$. For this special case the recreated code would be:

```
B b=null;
if(a!=null)
  throw new ClassCastException();
b.f();
```

Without the call `a.f();`, the new class hierarchy would have a class containing only a declaration of `f`, without subclasses or instances.

Treatment of Exceptions

To preserve the behavior of exceptions, the analysis must guarantee that every object thrown as exception shows the same behavior against every exception handler testing it while thrown. Exception handlers are listed as a table in byte-code, so they can be identified by a method name and a number referring to a table entry. The handlers a thrown object is tested against can be inferred from control flow information intraprocedural and from the call graph interprocedural. Again, attributes $H = false$ and $H = true$ are created for every exception handler H in the program. An object o tested against H raises a table entry $(o, H = true)$ if the exception is caught by that handler and $(o, H = false)$ else.

The new type for an exception handler H is the class $\mu(H = true)$. In case $\mu(H = true) = \bot$, the handler is never used and can be removed from the code.

The necessary table entries for handlers are currently not created by the dynamic analysis, making it impossible to refactor exception hierarchies. This does not affect the analysis of objects not used as exceptions.

Signatures of Overloaded Methods

As the analysis reduces every type in the program to its minimum, this may cause unwanted results in the context of overloaded methods. For the following example

```
class A {}
class B {}
...
   void f(A a) { System.out.println("an A"); }
   void f(B b) { System.out.println("a B"); }
...
   f(new A());
   f(new B());
```

Both parameters will be reduced to type `Object`, giving the overloaded methods equal signatures, which is not allowed in Java. This can be detected automatically and one or both variants of the method can be renamed.

Signatures of Overwritten Methods

Exactly the opposite will happen to the parameters of methods overwritten in a subclass. Here is a small example:

```
class A {
   void f(C c) { c.g(); }
}
class B extends A {
   void f(C c) { c.h(); }
}
...
```

```
A a=new A();
if(...)
   a=new B();
a.f(null);
```

For the parameter of f in A, a type containing only $dcl(C.g)$ will be calculated, for f in B a type containing only $dcl(C.h)$. But with different signatures Java would treat these as overloaded methods and no longer apply dynamic binding. It seems possible to take the infimum $\mu(dcl(C.g)) \sqcup \mu(dcl(C.h))$ as type for the parameter, but this may be $\bot$ (like in this example) and in general is not type correct. Instead assignments between all those parameters are added, forcing them to have the same type in the new class hierarchy (and giving all actual parameters the necessary type).

The same process is applied to the return type of overwritten methods.

Appendix 3: Preserving behavior

This appendix describes how the KABA editor guarantees preservation of behavior when manually modifying a refactoring proposal.

The initial refactoring proposal preserves behavior. Subsequent refactoring steps must only guarantee that the behavior of the new class hierarchy is identical to the behavior of the initially generated concept lattice. Behavior preservation is guaranteed by two groups of constraints. The first group consists of global constraints which must be fulfilled in order to interpret the graph as a class hierarchy. Constraints in the second group concern individual objects. The first group comprises:

- The graph must not contain cycles, and a class may not contain two method definitions with equal signature.
- All assignments in the program must still be type-correct. If `p = q;` was in the analyzed program, then $\gamma(q) \leq \gamma(p)$ must be valid.
- Dominance constraints must not be violated. If $A \rightarrow B$ is a dominance constraint, then $\mu(A) \leq \mu(B)$ must be valid.
- The additional constrains for type checks and exception handling must be respected. If C is the site of a type check or represents an exception handler, $C, C = false \not\leq C = true$ must be valid.

The second group comprises individual constraints for a true object O:

- If $Type(O)$ initially contained a statically bound member m, $\gamma(O)$ must also contain m.
- If $Type(O)$ initially contained a dynamically bound method m, the lookup must yield the same implementation: $StaticLookup(Type(O), m) = StaticLookup(\gamma(O), m)$.
- The class of O, $\gamma(O)$, must not become an abstract class.
- If the table for the initial graph had an entry $(o, C = true)$, $\gamma(o) \leq C = true$ must be valid.
- If the table for the initial graph had an entry $(o, C = false)$, $\gamma(o) \not\leq C = false$ must be valid.

Thus these constraints are the "interactive" version of the dominance constraints from appendix 1.