Refactoring Class Hierarchies with KABA

Gregor Snelting, Mirko Streckenbach

Universität Passau
Fakultät für Informatik
Germany

Refactoring Proposals for Class Hierarchies

Problem:

- Good design of a class hierarchy is hard
- Long maintenance increases entropy

⇒ Refactoring: Patterns to enhance code [Fowler ’99]

but:

- Most tools only help rewriting the code,
  but can’t find good refactorings automatically
- Programmer has to care about preserving semantics

Introduction

The Snelting/Tip-Analysis [TOPLAS'00]

- Automatic generation of refactoring proposal
- Guaranteed preservation of behavior
- Refactoring with respect to a given set of clients

Introduction

The Snelting/Tip-Analysis [TOPLAS’00]

- Automatic generation of refactoring proposal
- Guaranteed preservation of behavior
- Refactoring with respect to a given set of clients

- Refactoring reacts to object’s member access patterns
- All objects contain only members they need
- Fine grained insight into program behavior

Introduction

The Snelting/Tip-Analysis [TOPLAS'00]

- Automatic generation of refactoring proposal
- Guaranteed preservation of behavior
- Refactoring with respect to a given set of clients

- Refactoring reacts to object's member access patterns
- All objects contain only members they need
- Fine grained insight into program behavior

KABA: Implementation for Java

Related Work

- Opdyke [ACM ’93], Casais [OOS ’94],
  Moore [OOPSLA ’96] :
  No semantic guarantees

Related Work

- Opdyke [ACM ’93], Casais [OOS ’94], Moore [OOPSLA ’96] :
  No semantic guarantees
- Bowdidge and Griswold [TOSEM ’98] :
  Not object-oriented

Related Work

- Opdyke [ACM ’93], Casais [OOS ’94], Moore [OOPSLA ’96] : No semantic guarantees
- Bowdidge and Griswold [TOSEM ’98] : Not object-oriented
- Kataoka et al. [ICSM’01] : Local, not global refactorings

Related Work

- Opdyke [ACM ’93], Casais [OOS ’94], Moore [OOPSLA ’96] : No semantic guarantees
- Bowdidge and Griswold [TOSEM ’98] : Not object-oriented
- Kataoka et al. [ICSM’01] : Local, not global refactorings
- Tip et al. [OOPSLA’03] : Semantic preserving, but less fine grained

Technical Base

- Collection of member accesses
  - Static: Points-to analysis
  - Dynamic: Instrumented virtual machine
- Type constraints
- Concept lattices

Algorithm explained later;
full details see OOPSLA'04 paper, TOPLAS'00 paper,
and Mirko's PhD thesis

Features

KABA can handle full Java:

- Support for full Java bytecode
- Stubs for `native` methods needed
  Currently 180 stubs provided

Features

KABA can handle full Java:

- Support for full Java bytecode
- Stubs for `native` methods needed
  Currently 180 stubs provided
- Transforms type-casts, `instanceof` and exception-handlers
- Support for object creation with reflection

Features

KABA can handle full Java:

- Support for full Java bytecode
- Stubs for `native` methods needed
  Currently 180 stubs provided
- Transforms type-casts, `instanceof` and exception-handlers
- Support for object creation with reflection
- Max program size:
  20kLOC static variant, ∞ dynamic variant

Features

KABA can handle full Java:

- Support for full Java bytecode
- Stubs for `native` methods needed
  Currently 180 stubs provided
- Transforms type-casts, `instanceof` and exception-handlers
- Support for object creation with reflection
- Max program size:
  20kLOC static variant, ∞ dynamic variant
- Practically validated by running testsuite with refactored `jlex` source code

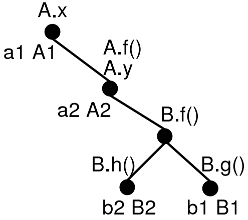
Example

Example source code and its KABA refactoring:

```
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();
  }
}
}
```

⇒



Example (2)

KABA refactors according to member access patterns

Example (2)

KABA refactors according to member access patterns

- B objects have different behaviour:
  one calls g, one calls h
  $\Longrightarrow$ original class B is split into two unrelated classes

Example (2)

KABA refactors according to member access patterns

- B objects have different behaviour:
  one calls g, one calls h
  $\Longrightarrow$ original class B is split into two unrelated classes
- A objects have related behaviour:
  A2 calls `A.f()` in addition
  $\Longrightarrow$ original class A is split into two subclasses

Example (2)

KABA refactors according to member access patterns

- B objects have different behaviour:
  one calls g, one calls h
  $\Longrightarrow$ original class B is split into two unrelated classes
- A objects have related behaviour:
  A2 calls `A.f()` in addition
  $\Longrightarrow$ original class A is split into two subclasses
- A1 does not use `A.y`; `A.z` is dead

Example (2)

KABA refactors according to member access patterns

- B objects have different behaviour:
  one calls g, one calls h
  $\Longrightarrow$ original class B is split into two unrelated classes
- A objects have related behaviour:
  A2 calls `A.f()` in addition
  $\Longrightarrow$ original class A is split into two subclasses
- A1 does not use `A.y`; `A.z` is dead

KABA determines most fine-grained refactoring which preserves behaviour

- Option: merge classes, eg two topmost new classes
  $\Longrightarrow$ refactoring less fine grained, but A1 bigger than necessary

Example (3)

refactored program:
statements are unchanged, only types change

```
class Aa {
  int x;
}

class Ab {
  int y;
  void f() {
    y = x;
  }
}
```

```
class B extends Ab {
  void f() {
    y++;
  }
}

class Ba extends B {
  void g() {
    x++;
    f();
  }
}

class Bb extends B {
  void h() {
    f();
    x--;
  }
}
```

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

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

Another Example: Professors and Students

```
class Person {
  String name;
  String address;
  int socialSecurityNumber;
}
```

```
class Student extends Person {
  int studentId;
  Professor advisor;

  Student(String sn, String sa,
          int si)
  {
    name = sn;
    address = sa;
    studentId = si;
  }

  void setAdvisor(Professor p)
  {
    advisor = p;
  }
}
```

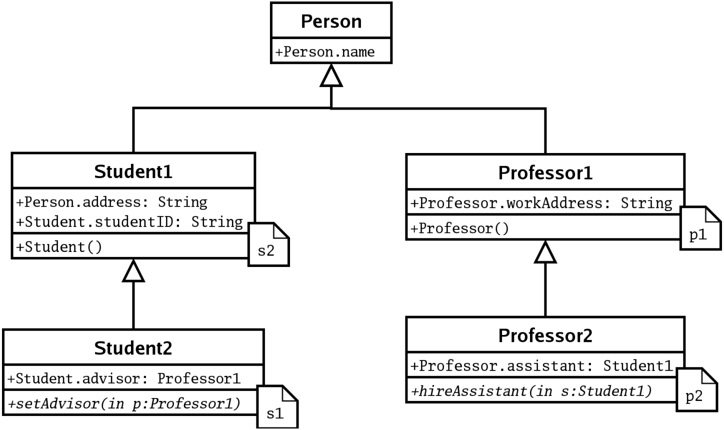
```
class Professor extends Person {
  String workAddress;
  Student assistant;

  Professor(String n, String wa)
  {
    name = n;
    workAddress = wa;
  }

  void hireAssistant(Student s)
  {
    assistant = s;
  }
}
```

Professors and Students (cont.)

Client code:

```
class Sample1 {
  static public void main(String[] args) {
    Student s1 = new Student("Carl", "here", 12345678);
    Professor p1 = new Professor("X", "there");
    s1.setAdvisor(p1);
  }
}

class Sample2 {
  static public void main(String[] args) {
    Student s2 = new Student("Susan", "also here", 87654321);
    Professor p2 = new Professor("Y", "not there");
    p2.hireAssistant(s2);
  }
}
```

KABA's refactoring



- Two kinds of students, two kinds of professors
- Method bodies are unchanged; but
  all variables/members obtain new type

⇒ Class cohesion and information hiding is improved

Reason for KABA's refactoring

```
class Sample1 {
  static public void main(String[] args) {
    Student s1 = new Student("Carl", "here", 12345678);
    Professor p1 = new Professor("X", "there");
    s1.setAdvisor(p1);
  }
}

class Sample2 {
  static public void main(String[] args) {
    Student s2 = new Student("Susan", "also here", 87654321);
    Professor p2 = new Professor("Y", "not there");
    p2.hireAssistant(s2);
  }
}
```

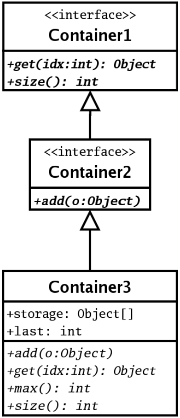
Refactored classes/objects contain only members they need!

Example: Interface Extraction

```
class Container {
  Object[] storage=...;
  int last=0;

  void add(Object o) {
    if(last<max())
      storage[last++]=o;
  }

  Object get(int idx) {
    return storage[idx];
  }

  int size() {
    return last;
  }

  int max() {
    return storage.length;
  }
}
```

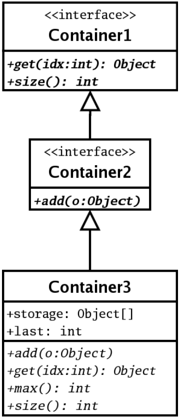
```
class Client {
  static void print(Container c) {
    for(int i=0;i!=c.size();++i)
      System.err.println(c.get(i));
  }

  static void main(String[] args) {
    Container c1=new Container();

    c1.add("hello");
    c1.add("world");

    print(c1);
  }
}
```

KABA's refactoring



Two different interfaces separated from implementation

KABA's refactoring



```
class Client {
  static void print(Container c) {
    for(int i=0;i!=c.size();++i)
      System.err.println(c.get(i));
  }

  static void main(String[] args) {
    Container c1=new Container();

    c1.add("hello");
    c1.add("world");

    print(c1);
  }
}
```

Two different interfaces separated from implementation

Case Studies

Today, KABA offers:

- Fine grained analysis of object behavior
- Semi-automatic simplification

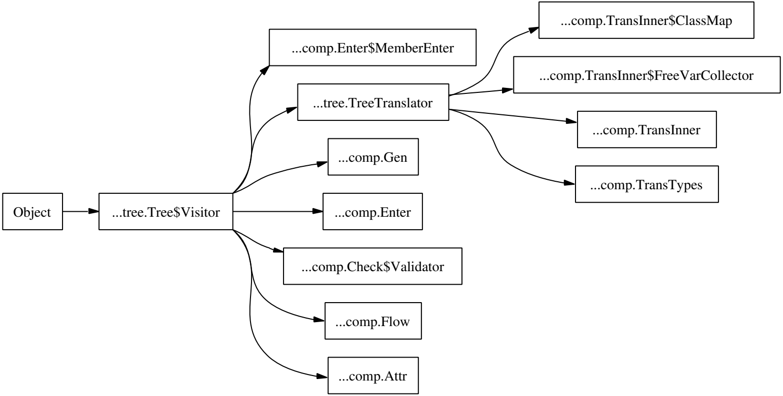
⇒ Practical refactorings with respect to object behavior

⇒ Evaluation of existing designs

Case Study: javac

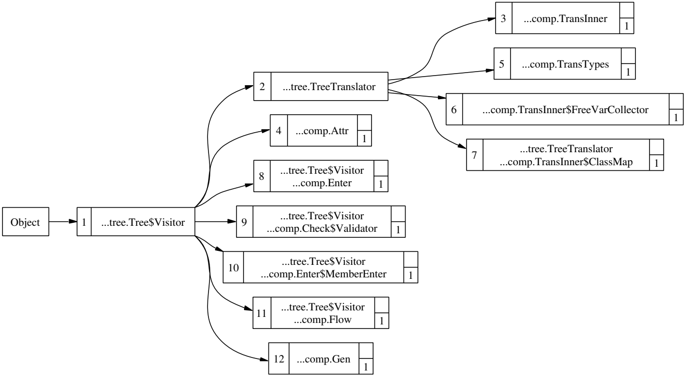
Tree visitor from Java compiler
(JDK 1.3.1: 129 classes, 27211 LOC, 1878 test runs)

Original hierarchy:



Case Study: javac

Refactoring:



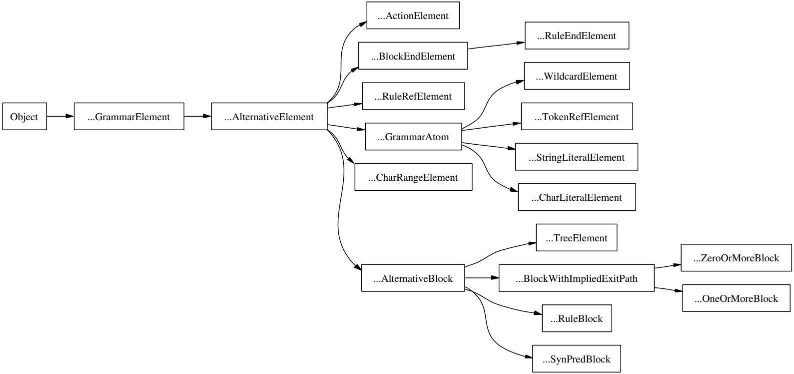
- Class structure unchanged, but members moved
- Improved cohesion with respect to client behavior

⇒ Overall design was good!

Case Study: ANTLR

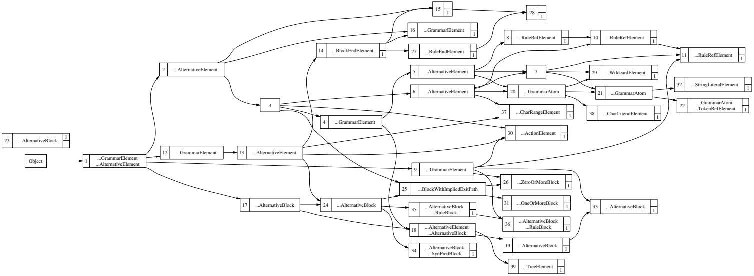
Syntax tree from ANTLR parser generator
(2.7.2: 108 classes, 38916 LOC, 84 test runs)

Original hierarchy:



Case Study: ANTLR

Fine-grained refactoring:

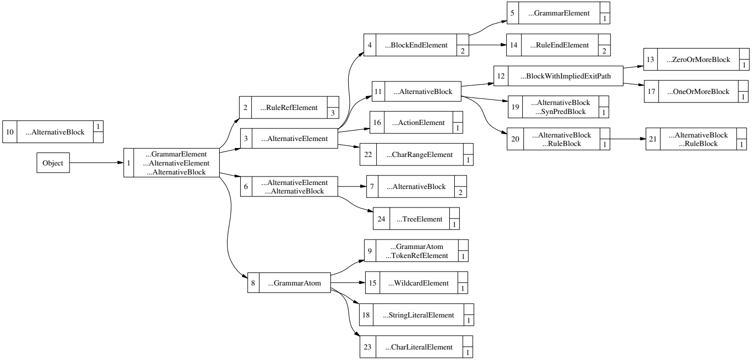


Complex object access patterns

⇒ Low functional cohesion of original design

Case Study: ANTLR

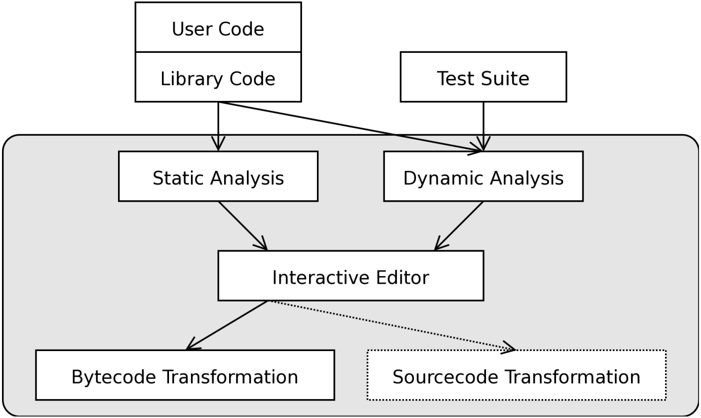
After more aggressive simplification:



Again improved functional cohesion

⇒ Original design questionable compared to javac

An Overview of KABA



User Code

Library Code

Test Suite

Static Analysis

Dynamic Analysis

Interactive Editor

Bytecode Transformation

Sourcecode Transformation

The Algorithm (Snelting/Tip, TOPLAS '00)

Step 1: Extract member accesses from source code $\mathcal{P}$ and construct member access table $\mathcal{T}$

The Algorithm (Snelting/Tip, TOPLAS '00)

Step 1: Extract member accesses from source code $\mathcal{P}$ and construct member access table $\mathcal{T}$

- dynamic variant: extract all runtime accesses by objects $O.m()$ using instrumented JVM; add entry $(O, C.m)$ to $\mathcal{T}$ where $C = \mathit{staticLookup}(\mathit{type}(O), m)$

The Algorithm (Snelting/Tip, TOPLAS '00)

Step 1: Extract member accesses from source code $\mathcal{P}$ and construct member access table $\mathcal{T}$

- dynamic variant: extract all runtime accesses by objects $O.m()$ using instrumented JVM; add entry $(O, C.m)$ to $\mathcal{T}$ where $C = \mathit{staticLookup}(\mathit{type}(O), m)$
- static variant: use points-to to approximate dynamic dispatch:
  if $o.m() \in \mathcal{P}$ and $O \in pt(o)$, add entry $(O, C.m)$ to $\mathcal{T}$

The Algorithm (Snelting/Tip, TOPLAS '00)

Step 1: Extract member accesses from source code $\mathcal{P}$ and construct member access table $\mathcal{T}$

- dynamic variant: extract all runtime accesses by objects $O.m()$ using instrumented JVM; add entry $(O, C.m)$ to $\mathcal{T}$ where $C = \mathit{staticLookup}(\mathit{type}(O), m)$
- static variant: use points-to to approximate dynamic dispatch:
  if $o.m() \in \mathcal{P}$ and $O \in pt(o)$, add entry $(O, C.m)$ to $\mathcal{T}$

Details for `this`-pointers, `instanceof` etc. see paper

The Algorithm: Example

Source code and its initial table:

```
class A {
  int x, y, z;
  void f() {      class Client {
    y = x;           public static void
  }                  main(String[] args) {
}                      A a1 = new A();  // A1
                       A a2 = new A();  // A2
class B extends A {    B b1 = new B();  // B1
  void f() {           B b2 = new B();  // B2
    y++;
  }                    a1.x = 17;
  void g() {           a2.x = 42;
    x++;               if (...) { a2 = b2; }
    f();               a2.f();
  }                    b1.g();
  void h() {           b2.h();
    f();             }
    x--;           }
  }
}
```

⇒

	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	×					×					×

For methods, distinction between $def(m)$ and $dcl(m)$ increases precision

$(C.m.this, def(C.m)) \in \mathcal{T}$ “glue” together method and its this-pointer

The Algorithm (2)

Step 2: incorporate type constraints for semantics preservation

The Algorithm (2)

Step 2: incorporate type constraints for semantics preservation

- assignment constraints: `x = y;`
  implies $type(x) \geq type(y)$ in refactored hierarchy
  requires "row implication" $x \rightarrow y$ in table:
  all members of x must also be members of y
  ⇒ copy entries from row x to row y

The Algorithm (2)

Step 2: incorporate type constraints for semantics preservation

- assignment constraints: `x = y;`
  implies $type(x) \geq type(y)$ in refactored hierarchy
  requires "row implication" $x \rightarrow y$ in table:
  all members of x must also be members of y
  $\Rightarrow$ copy entries from row x to row y
- dominance constraints: if $B \leq A$ both have member m,
  and $\exists o : (o, A.m) \in \mathcal{T}, (o, B.m) \in \mathcal{T}$,
  $newClass(B.m) \leq newClass(A.m)$ must hold to avoid ambiguities
  requires "column implication" $B.m \rightarrow A.m$ in table

The Algorithm (2)

Step 2: incorporate type constraints for semantics preservation

- assignment constraints: `x = y;`
  implies $type(x) \geq type(y)$ in refactored hierarchy
  requires “row implication” $x \rightarrow y$ in table:
  all members of x must also be members of y
  $\Rightarrow$ copy entries from row x to row y
- dominance constraints: if $B \leq A$ both have member m,
  and $\exists o : (o, A.m) \in \mathcal{T}, (o, B.m) \in \mathcal{T}$,
  $newClass(B.m) \leq newClass(A.m)$ must hold to avoid ambiguities
  requires “column implication” $B.m \rightarrow A.m$ in table
- implications are applied to $\mathcal{T}$ until no more entries are added

The Algorithm (2)

Step 2: incorporate type constraints for semantics preservation

- assignment constraints: `x = y;`
  implies $type(x) \geq type(y)$ in refactored hierarchy
  requires "row implication" $x \to y$ in table:
  all members of x must also be members of y
  $\Rightarrow$ copy entries from row x to row y
- dominance constraints: if $B \leq A$ both have member m, and $\exists o : (o, A.m) \in \mathcal{T}, (o, B.m) \in \mathcal{T}$, $newClass(B.m) \leq newClass(A.m)$ must hold to avoid ambiguities
  requires "column implication" $B.m \to A.m$ in table
- implications are applied to $\mathcal{T}$ until no more entries are added

Final table respects all type constraints; this guarantees semantics preservation [Tip Acta Inf. '00]

The Algorithm: Example (cont‘d)

incorporate assignment constraints $a1 \to A1$, $a2 \to b2$, ...
incorporate dominance constraints $dcl(B.f) \to dcl(A.f)$, ...

	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	×					×					×

⇒

	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	×	×		×		×	×			×	×

assignment/dominance constraints can interfere
⇒ fixpoint iteration

The Algorithm: Example (cont‘d)

incorporate assignment constraints $a1 \rightarrow A1$, $a2 \rightarrow b2, \ldots$
incorporate dominance constraints $dcl(B.f) \rightarrow dcl(A.f), \ldots$

	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	×					×					×

⇒

	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	×	×		×		×	×			×	×

assignment/dominance constraints can interfere
⇒ fixpoint iteration

The Algorithm: Example (cont'd)

incorporate assignment constraints $a1 \rightarrow A1$, $a2 \rightarrow b2$, ...
incorporate dominance constraints $dcl(B.f) \rightarrow dcl(A.f)$, ...

	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	×					×					×

$\Rightarrow$

	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	×	×		×		×	×			×	×

assignment/dominance constraints can interfere
⇒ fixpoint iteration

The Algorithm: Example (cont'd)

incorporate assignment constraints $a1 \to A1$, $a2 \to b2$, ...

incorporate dominance constraints $dcl(B.f) \to dcl(A.f)$, ...

	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	×			∘	←	×					×

⇒

	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	×	×		×		×	×			×	×

assignment/dominance constraints can interfere

⇒ fixpoint iteration

The Algorithm (3)

Step 3: compute concept lattice [Ganter & Wille 99] from final table

- concept lattices are natural inheritance structures
- each lattice element represents a new class
- lattice displays class members above elements
- lattice displays all variables having new class as its new type below element

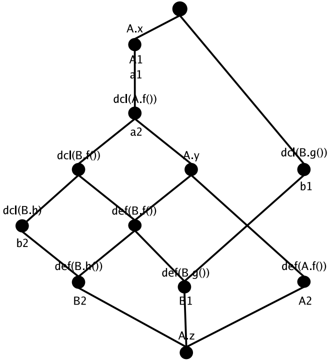
Beautiful theory and algorithms for concept lattices!

The Algorithm: Example (cont‘d)

Concept lattice generated from final table:

	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	×	×		×		×	×			×	×

⇒



$$(o, m) \in \mathcal{T} \iff \gamma(o) \le \mu(m)$$

fine-grained insight into object behaviour!

The Algorithm (4)

Step 4: simplify concept lattice

- remove “empty” elements
- merge elements
- move members up
- remove multiple inheritance (always possible!)
- ...

semi-automatic
semantics preserving!

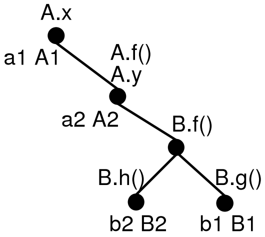
The Algorithm (4)

Step 4: simplify concept lattice

- remove “empty” elements
- merge elements
- move members up
- remove multiple inheritance (always possible!)
- ...

semi-automatic
semantics preserving!

Final refactoring for example:



can be simplified further

Analysis Challenges

Refactorings for large programs too fine-grained

- Semi-automatic simplification of the class hierarchy

Analysis Challenges

Refactorings for large programs too fine-grained

- Semi-automatic simplification of the class hierarchy

New class hierarchy contains multiple inheritance

- Removed by moving members “towards” the original hierarchy

Analysis Challenges

Refactorings for large programs too fine-grained

- Semi-automatic simplification of the class hierarchy

New class hierarchy contains multiple inheritance

- Removed by moving members “towards” the original hierarchy

Static analysis does not scale beyond 10 kLOC

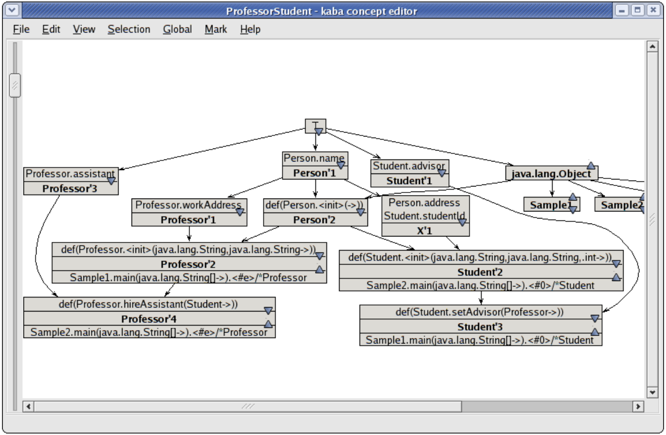
- Dynamic analysis
    - Omits pointers and creates simpler hierarchies
    - Preserves only behavior for test suite

The KABA Editor

- Browsing of the refactored class hierarchy
- Manual application of basic refactorings
  - Move member
  - Create/Delete inheritance
  - Add/Merge classes
- More complex algorithms
  - Simplification
  - Removal of multiple inheritance
- Detailed error messages if transformation changes program semantics

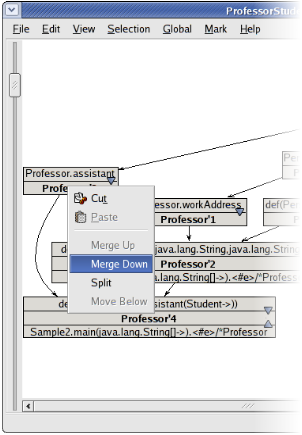
The KABA Editor

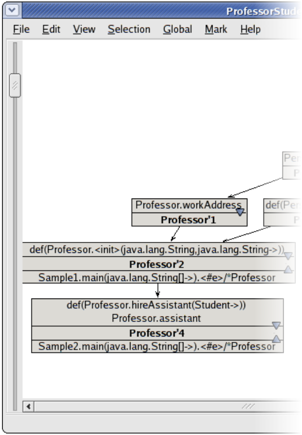
“Raw” class hierarchy as generated by KABA



The KABA Editor

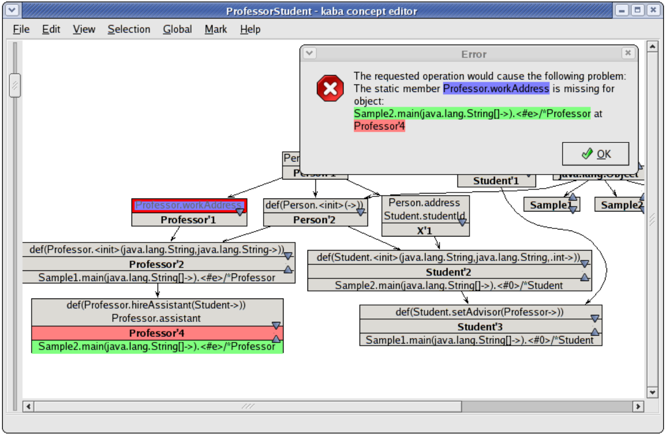
Interactive refactoring: Merging two classes





The KABA Editor

Interactive refactoring: Violation of semantics



The requested operation would cause the following problem:
The static member Professor.workAddress is missing for object:
Sample2.main(java.lang.String[]->).<#e>/*Professor at Professor'4

KABA: Conclusion

KABA's analysis:

- Semantics preserving refactorings
- Client specific
- Based on fine grained program analysis

KABA: Conclusion

KABA's analysis:

- Semantics preserving refactorings
- Client specific
- Based on fine grained program analysis

KABA's features:

- Semantics preserving refactoring editor
- Automated code transformation

KABA: Conclusion

KABA's analysis:

- Semantics preserving refactorings
- Client specific
- Based on fine grained program analysis

KABA's features:

- Semantics preserving refactoring editor
- Automated code transformation

KABA's results:

- Practical refactorings automatically
- Usable as a design evaluation tool