Aktueller Stand des Java-TX Projekts

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

8. Mai 2017

Überblick

- Reenginering der Archtektur und des TI-Algorithmus
  Hiwi–Job
- Neuer Parser
  bisher: jay (yacc für Java)
  jetzt: antlr
  (Bachelorarbeit Uni Tübingen)
- Round trip engineering/Modell driven development
  Kombination Objectif und Eclipse
  (Studienarbeit DHBW Stuttgart)
- Implementierung *Structural Types* – Experiences
  (Studienarbeit DHBW Stuttgart, Campus Horb)

Überblick

- Reenginering der Archtektur und des TI-Algorithmus
  Hiwi–Job
- Neuer Parser
  bisher: jay (yacc für Java)
  jetzt: antlr
  (Bachelorarbeit Uni Tübingen)
- Round trip engineering/model driven development
  Kombination Objectif und Eclipse
  (Studienarbeit DHBW Stuttgart)
- Implementierung *Structural Types* – Experiences
  (Studienarbeit DHBW Stuttgart, Campus Horb)

Java–TX Projekt

Functional Programming Features in OO-Languages

	Java	Scala	C#
Parametric polymorphism	use-site variance	declaration-site and use-site variance	declaration-site variance
Lambda expressions	target types: functional interfaces	typed by function types	Delegates
Type inference	local type inference	local type inference	local type inference

There is no *full* type–inference in OO-languages

Motivating example

```
interface Fun1<R, T> { R apply(T arg); }

interface Fun2<R, T1, T2> { R apply(T1 arg1, T2 arg2); }

class Matrix extends Vector<Vector<Integer>> {

    Fun1<Fun1<Matrix, Fun2<Matrix, Matrix,Matrix>>, Matrix>
    op = (m) -> (f) -> f.apply(this, m);
}
```

Our goal

```
class Matrix extends Vector<Vector<Integer>> {

    op = (m) -> (f) -> f.apply(this, m);
}
```

Java-Type eXtended (Java-TX)

- *full* type–inference
- real function types in Java

Round trip engineering/model driven development

Entwicklung an der DHBW

- bis zu 30 Studienarbeiten von jeweils 300h workload
- 50% Hiwi–Stelle

Entwicklung an der DHBW

- bis zu 30 Studienarbeiten von jeweils 300h workload
- 50% Hiwi–Stelle

Management einer Vielzahl von Programmierern

- Einsatz von Software–Engineering Tools: Objectif
- professionelles Versionsmanagement: git

Objectif von microtool

Entwicklung in Java

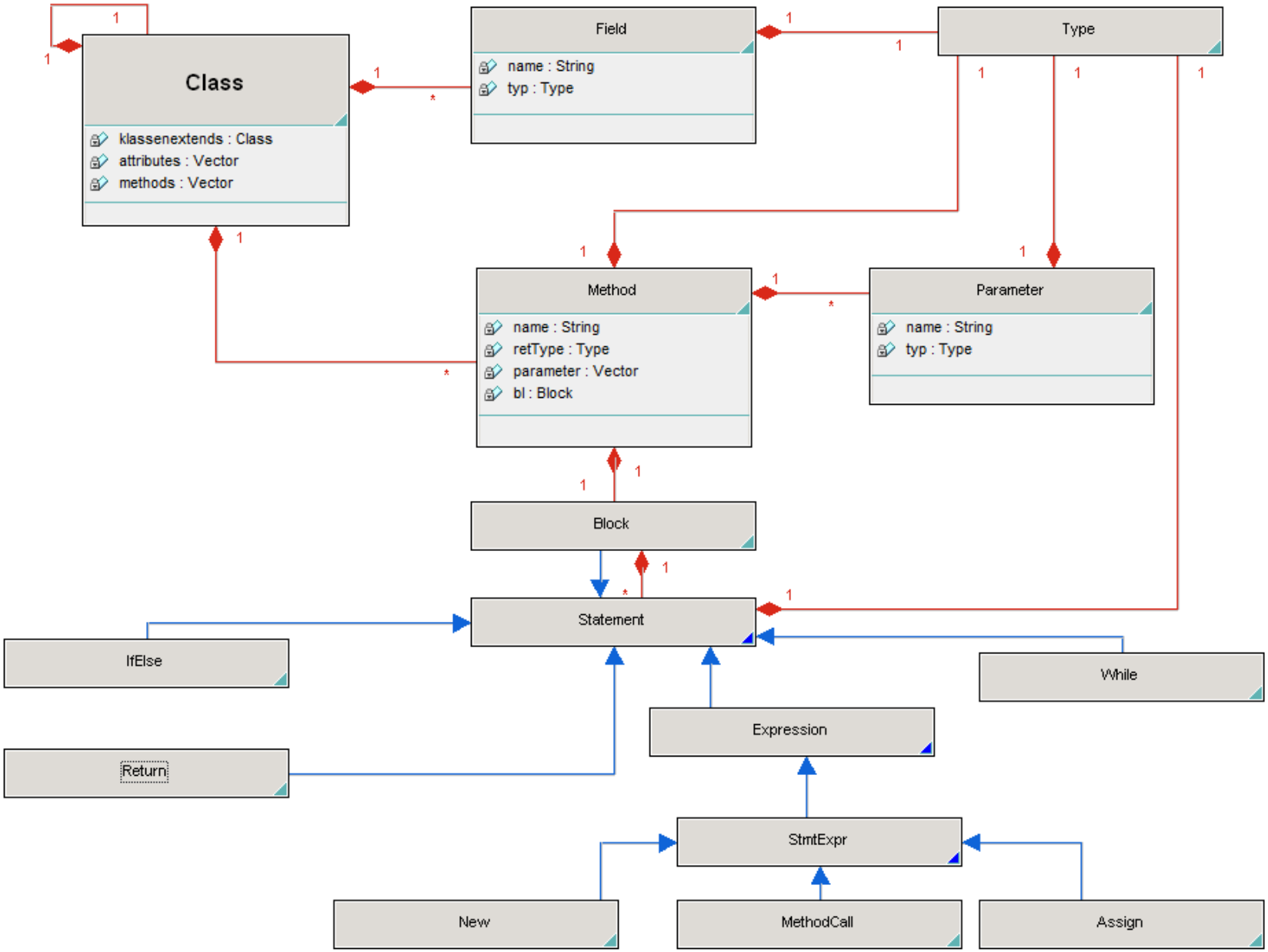
objectiF Eclipse Edition – modellgetriebene Entwicklung in Java

Wenn Sie in Java mit Eclipse entwickeln, ist die objectiF Eclipse Edition die richtige Wahl.

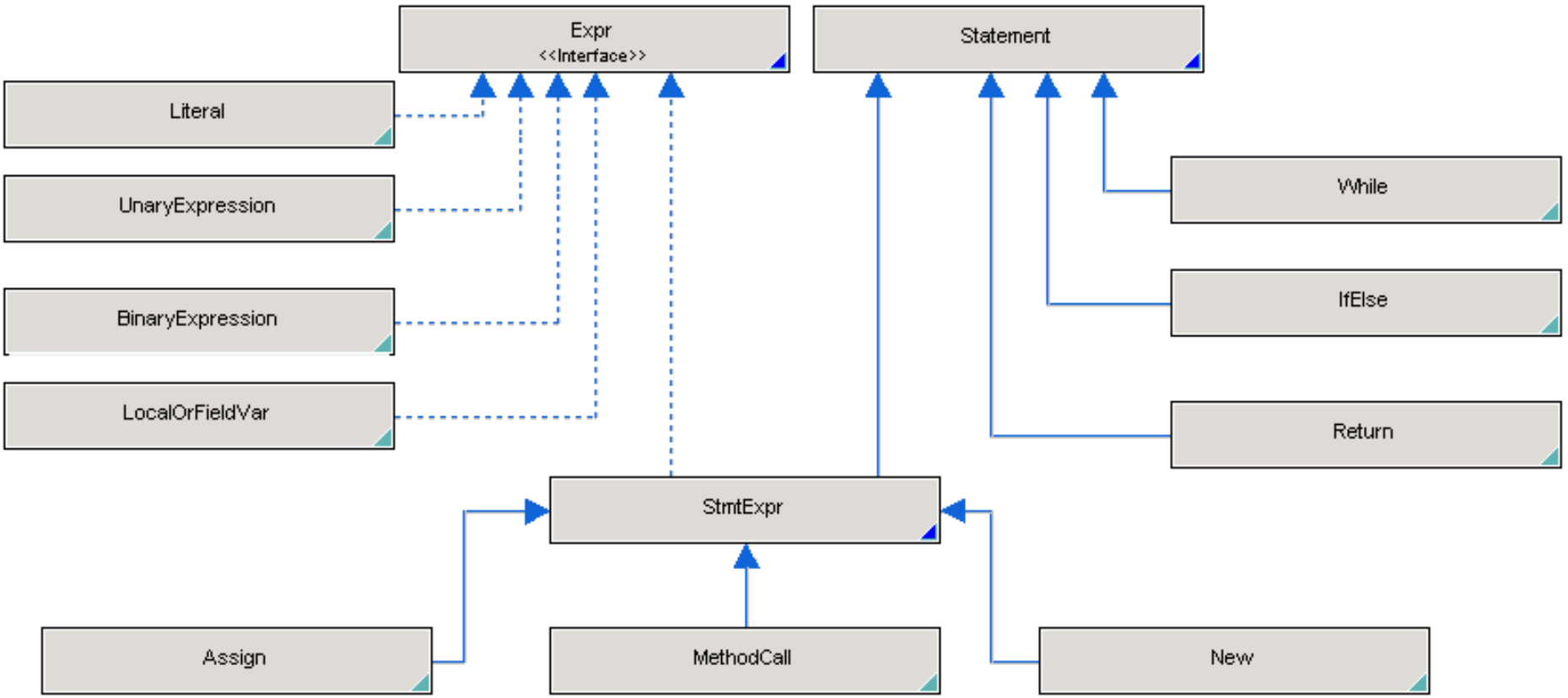
Schneller und agiler entwickeln mit der objectiF Eclipse Edition – so geht's:

- Sie beschreiben die Anforderungen an ein System mit der UML und generieren zielgruppengerechte Dokumentation mit objectiF.
- Sie modellieren die Fachlichkeit eines Systems und neue Geschäftsprozesse mit der UML und BPMN in objectiF.
- objectiF setzt per Modelltransformation Ihre fachlichen Modelle in die technischen Modelle des Softwareentwurfs um.
- objectiF generiert bei der Transformation große Mengen an Code
- Sie vervollständigen den Code im Round Trip mit Eclipse. objectiF und Eclipse sind nahtlos integriert.

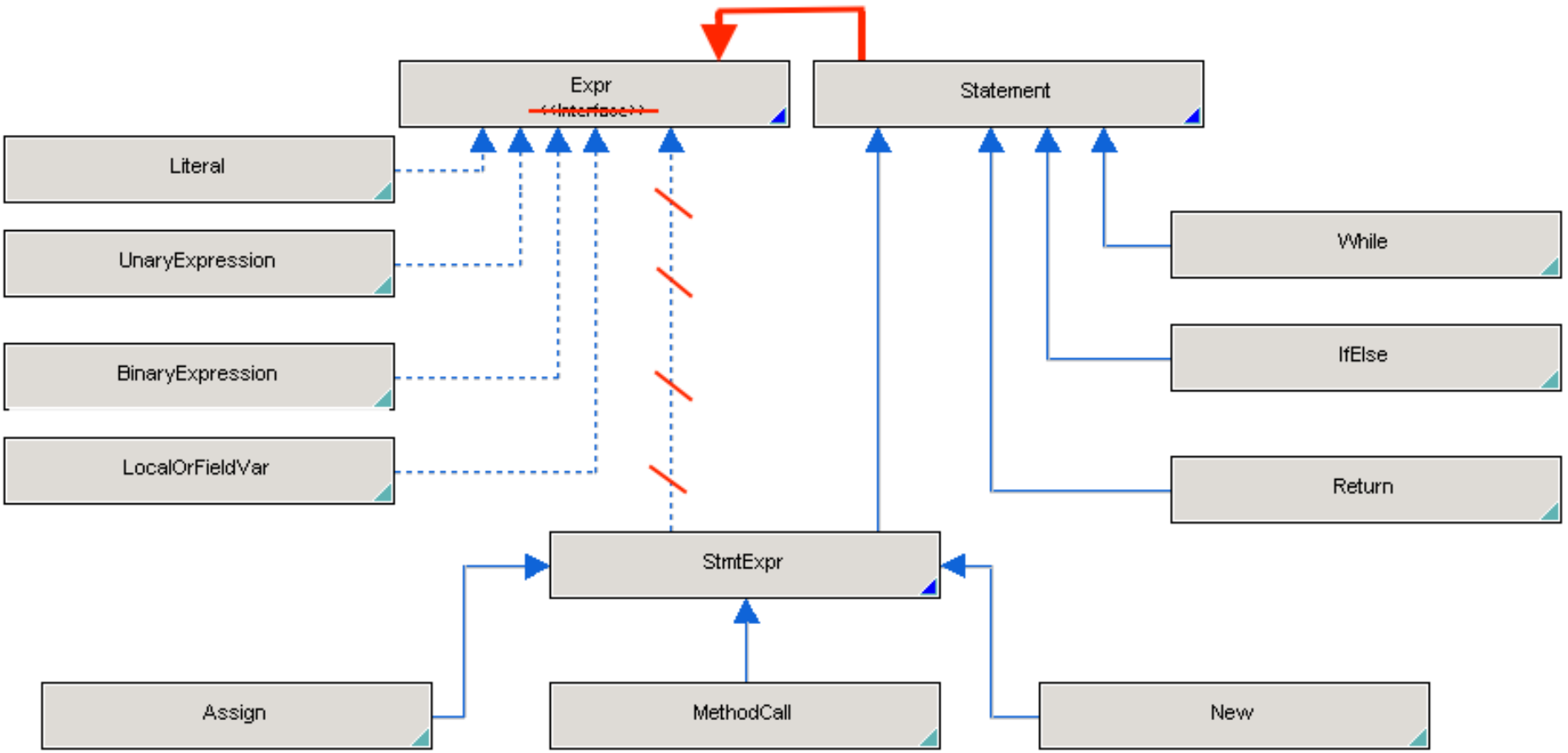
Architektur abstrakte Syntax



Design Expression/Statements (alt)



Design Expression/Statements (neu)



Probleme

- technisches Zusammenspiel: Objectif, Eclipse, Java (Versionen, 32-Bit, 64-Bit)
- Parserprobleme:
  - Diamond–Operator
  - voll qualifizierte Typen
- Java–Code generieren:
  - komplett neue Formatierung
  - unnötige Kommentare

Structural Types – experiences

Nominal Type – Type Inference

```
import java.util.Vector;

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Nominal Type – Type Inference

```
import java.util.Vector;

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Result: `m : Vector<T>` → `T`.

Nominal Type – Type Inference

Extension

```
import java.util.*

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Nominal Type – Type Inference

Extension

```
import java.util.*

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Result: `m : List<T>` $\rightarrow$ `T` $\wedge \ldots \wedge$ `Vector<T>` $\rightarrow$ `T` $\wedge$ `Stack<T>` $\rightarrow$ `T`.

Multiple results! Is there a principal type?

Structural Type – approach

```
class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Structural Type – approach

```
interface I<T> { T elementAt(int x); }

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Result: `m : I<T> → T`.

Only one result!!!

Structural Type – approach

```
import java.util.*

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Result: Three Possibilities:

- `m : List<T> → T ∧ ... ∧ Vector<T> → T ∧ Stack<T> → T.`
- `m : I<T> → T.`
  where `interface I<T> { T elementAt(int x); }` is generated.
- `m : List<T> → T ∧ ... ∧ Vector<T> → T ∧ Stack<T> → T`
  `∧ I<T> → T.`
  where `interface I<T> { T elementAt(int x); }` is generated.

Structural Type – approach

```
import java.util.*

class A {

    m (v) {
        return v.elementAt(0);
    }
}
```

Result: Second Possibility:

- m : List<T> → T ∧ ... ∧ Vector<T> → T ∧ Stack<T> → T.
- m : I<T> → T.
  where interface I<T> { T elementAt(int x); } is generated.
- m : List<T> → T ∧ ... ∧ Vector<T> → T ∧ Stack<T> → T
  ∧ I<T> → T.
  where interface I<T> { T elementAt(int x); } is generated.

⇒ Each field/variable has an interface as principal type!

More complex example

```
class A {

    mt(x, y, z) {
        return x.sub(y).add(z);
    }
}
```

The result of the type inference algorithm

```
interface Sub<R, T> { R sub(T x); }

interface Add<R, T> { R add(T x); }

class A <ν1,ν3,ν4,ν6>
        [ν3 extends ν5,
         ν4 extends ν7,
         ν1 extends Sub<ν2,ν5>,
         ν2 extends Add<ν6,ν7>] {

    ν6 mt(ν1 x, ν3 y, ν4 z) {
        return x.sub(y).add(z);
    }
}
```

Interface implementation: Sub and Add

```
class myInteger extends Sub<myInteger, myInteger>,
                        Add<myInteger, myInteger> {

    Integer i;

    myInteger sub(myInteger x) {
        return new myInteger(i - x.i);
    }

    myInteger add(myInteger x) {
        return new myInteger(i + x.i);
    }
}
```

Instance of A

```
class A <ν1,ν3,ν4,ν5>
        [ν3 ⩽ ν5, ν4 ⩽ ν7, ν1 ⩽ Sub<ν2,ν5>, ν2 ⩽ Add<ν6,ν7>]{
    ν6 mt(ν1 x, ν3 y, ν4 z) {
        return x.sub(y).add(z);
    }
}
```

```
class Main {

    main() {
        return new A<>()
                .mt(new myInteger(2),
                    new myInteger(1),
                    new myInteger(3));
    }
}
```

Situation of the programmer

```
class A {

    mt(x, y, z) {
        return x.sub(y).add(z);
    }
}
```

```
class Main {

    main() {
        return new A<>()
                .mt(??, ??, ??); } }
```

Situation of the programmer

```
class A {

    mt(x, y, z) {
        return x.sub(y).add(z);
    }
}
```

```
class Main {

    main() {
        return new A<>()
                .mt(??, ??, ??); } }
```

- Object contains method sub.
- Method sub has as return type an object that contains a method add.

Situation of the programmer

```
class A {

    mt(x, y, z) {
        return x.sub(y).add(z);
    }
}
```

```
class Main {

    main() {
        return new A<>()
                .mt(??, ??, ??); } }
```

- Object contains method sub.
- Method sub has as return type an object that contains a method add.
- Object is of the arument type of sub.

Situation of the programmer

```
class A {

    mt(x, y, z) {
        return x.sub(y).add(z);
    }
}
```

```
class Main {

    main() {
        return new A<>()
                .mt(??, ??, ??); } }
```

- Object contains method sub.
- Method sub has as return type an object that contains a method add.
- Object is of the arument type of sub.
- Object is of the argument type of add.

Discussion

Structural type inference

- pro:
  - *one* principal type
- con:
  - each field/variable has an interface as type
  - programms without type annotations are less readable

Conclusion and future work

Conclusion

- Round trip engineering
- Structural type as principal type?

Conclusion and future work

Conclusion

- Round trip engineering
- Structural type as principal type?

Future work

- finalize reenginering
- optimize constraints