12a. Graphs for Models and Programs

- Prof. Dr. U. Aßmann
- Technische Universität Dresden
- Institut für Software- und Multimediatechnik
- Gruppe Softwaretechnologie
- http://st.inf.tu-dresden.de/teaching/swt2

- Version 16-1.1, 02.12.16

Obligatory Reading

Goals

- Understand that graphs can be used to represent software, models and programs
- Understand value-flow, control-flow graphs, call graphs

Motivation

- Programs are represented by graphs
- Models and specifications are graph-based
  - We have to deal with basic graph theory to be able to measure well

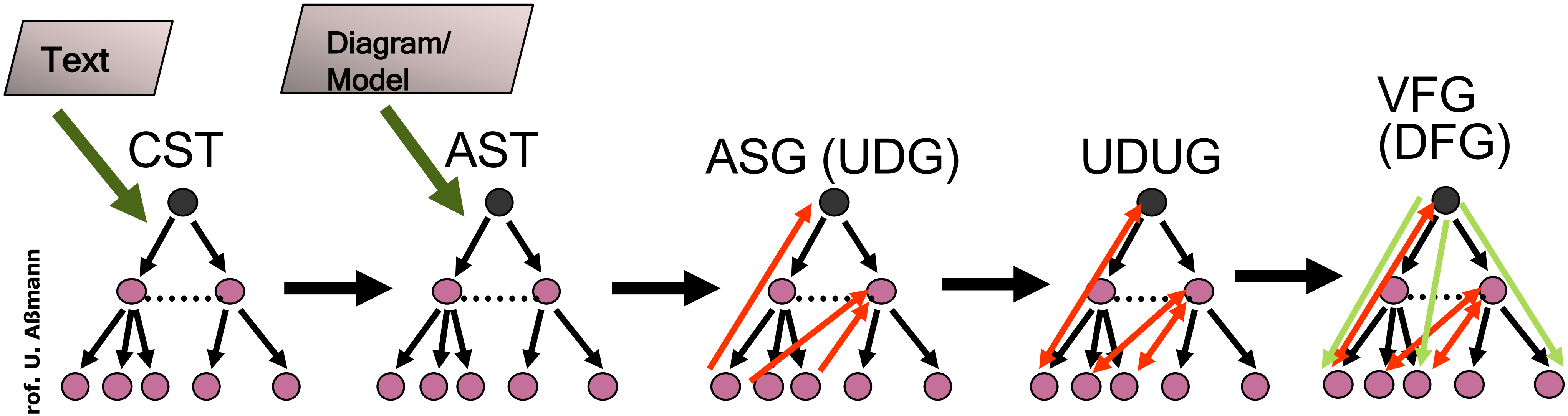
How are models and programs represented in a Software Tool?

Some Relationships (Graphs) in Software Systems

12A.1 GENERATING GRAPHS FROM DIAGRAMS AND PROGRAMS

All Models, Specifications and Programs Have an Internal Graph-Based Representation

- Texts are parsed to abstract syntax trees (AST). Two-step procedure:
  1. Concrete Syntax Tree (CST)
  2. Abstract Syntax Tree (AST) (also directly from diagram editors)
- Through name analysis, they become abstract syntax graphs (ASG) or Use-Def-Graphs (UDG)
- Through def-use-analysis, they become Use-def-Use Graphs (UDUG)
- If value flow (data flow) between variables is analysed, the value flow graph (VFG) or data-flow graph (DFG) result



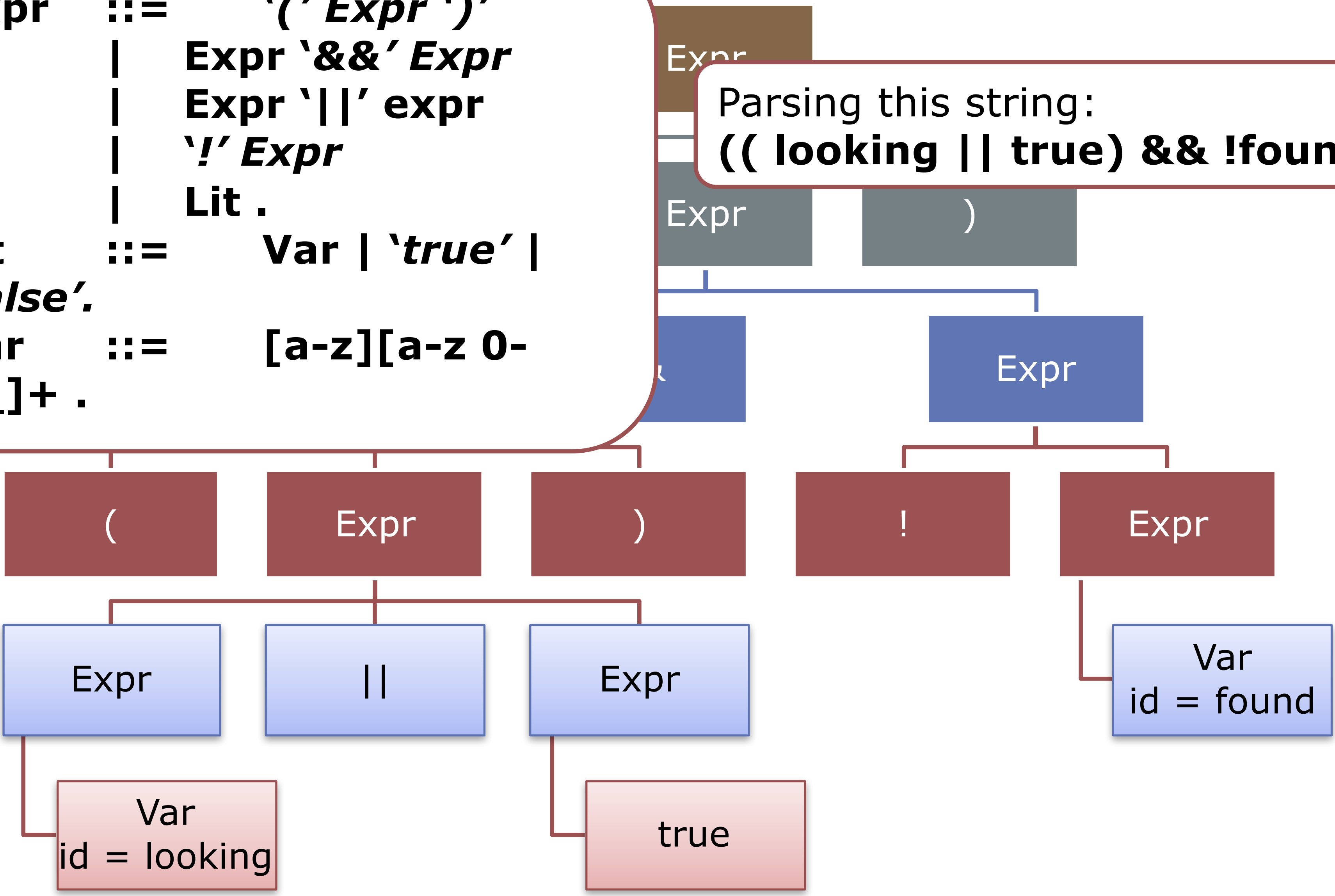
Concrete Syntax Tree (CST) – Example


```
Expr   ::= '(' Expr ')'
         |   Expr '&&' Expr
         |   Expr '||' expr
         |   '!' Expr
         |   Lit .
Lit    ::=      Var | 'true' | 'false'.
Var    ::=      [a-z][a-z 0-9_]+ .
```

Parsing this string:
((looking || true) && !found)

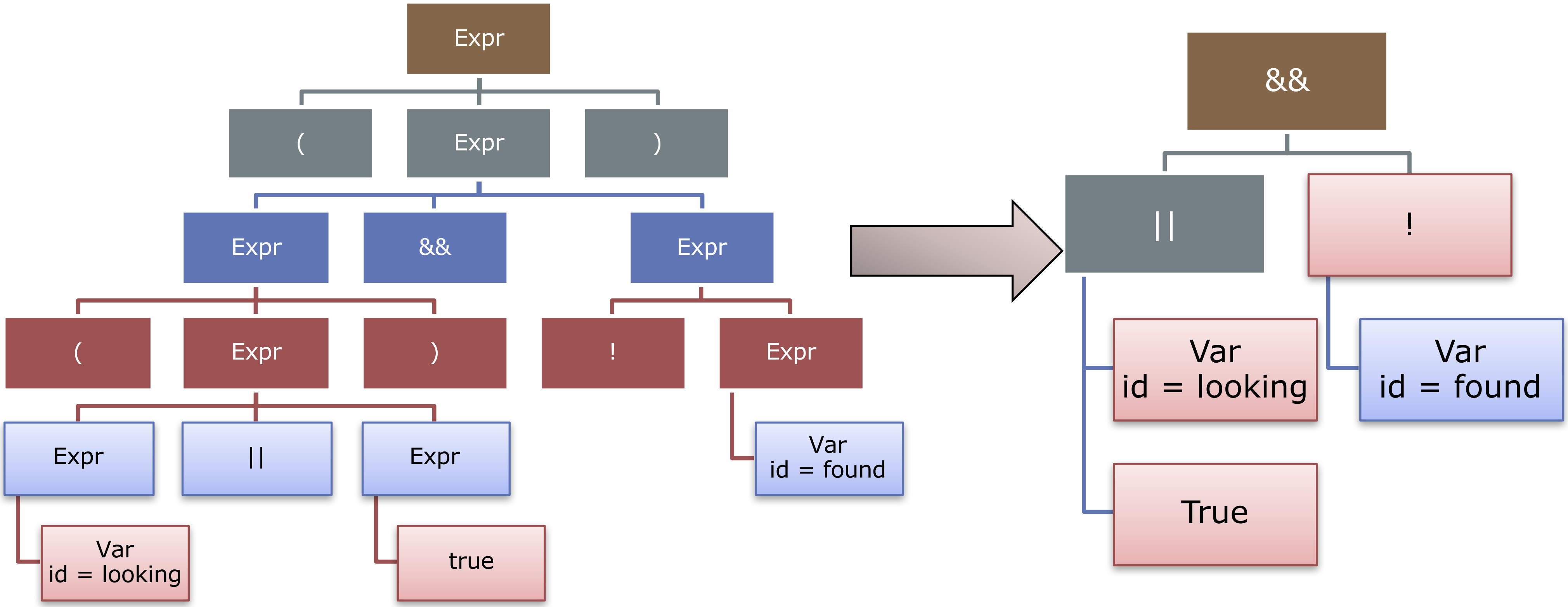
Expr ::= *'(' Expr ')'*
| Expr *'&&' Expr*
| Expr '||' expr
| *'!' Expr*
| Lit .
Lit ::= Var | *'true'* | *'false'*.
Var ::= [a-z][a-z 0-9_]+ .

Parsing this string:
((looking || true) && !found)



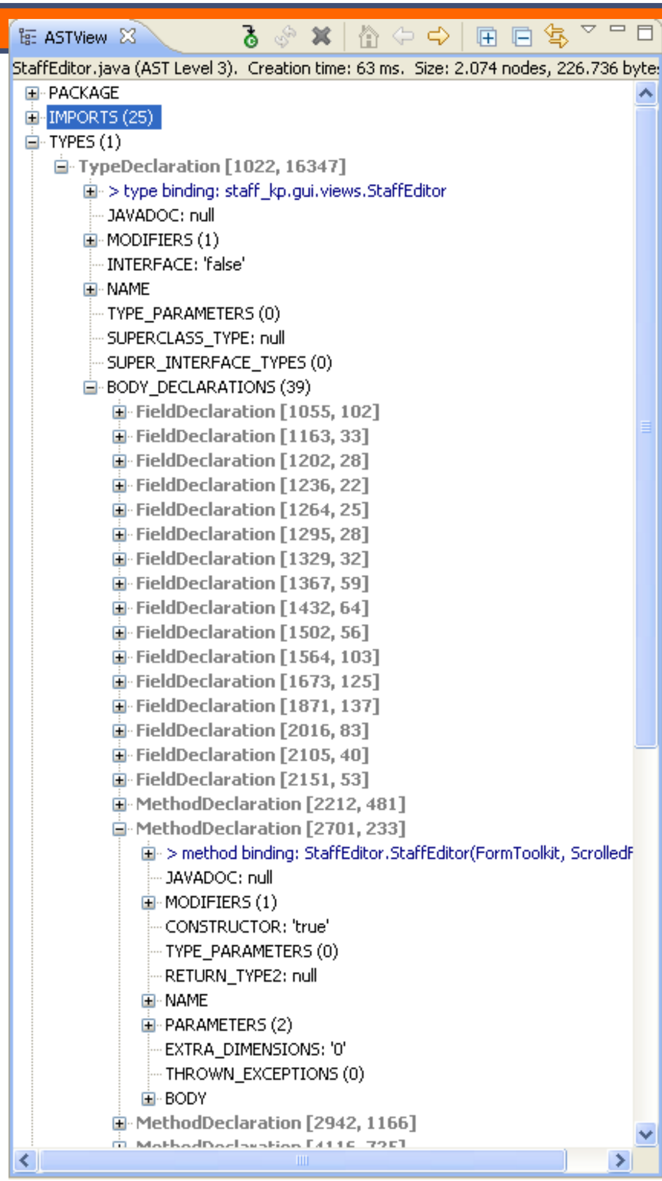
Prof. U. Aßmann

From the CST to the AST



Abstract Syntax Trees (AST)

- Parse trees (CST) waste a fair amount of space for terminal symbols and productions
- Compilers post-process parse trees into ASTs
- ASTs are the fundamental data structure of IDEs (ASTView in Eclipse JDT)



Abstract Syntax Trees (AST)

- Problem with ASTs: They do not support static semantic checks, re-factoring and browsing operations, e.g:
  - Name semantics:
    - Have all used variables been declared? Are they declared once?
    - Have all Classes used been imported?
  - Type semantics (type checking): are all types used in expressions / assignments compatible?
  - Type inference: can all types for variables – if not given – be inferenced?
  - Referencing:
    - Navigate to the declaration of method call / variable reference / type
  - Pretty-printing: How can I pretty-print the AST to a CST again, so that the CST looks like the original CST
    - Necessary for *hygenic refactoring*

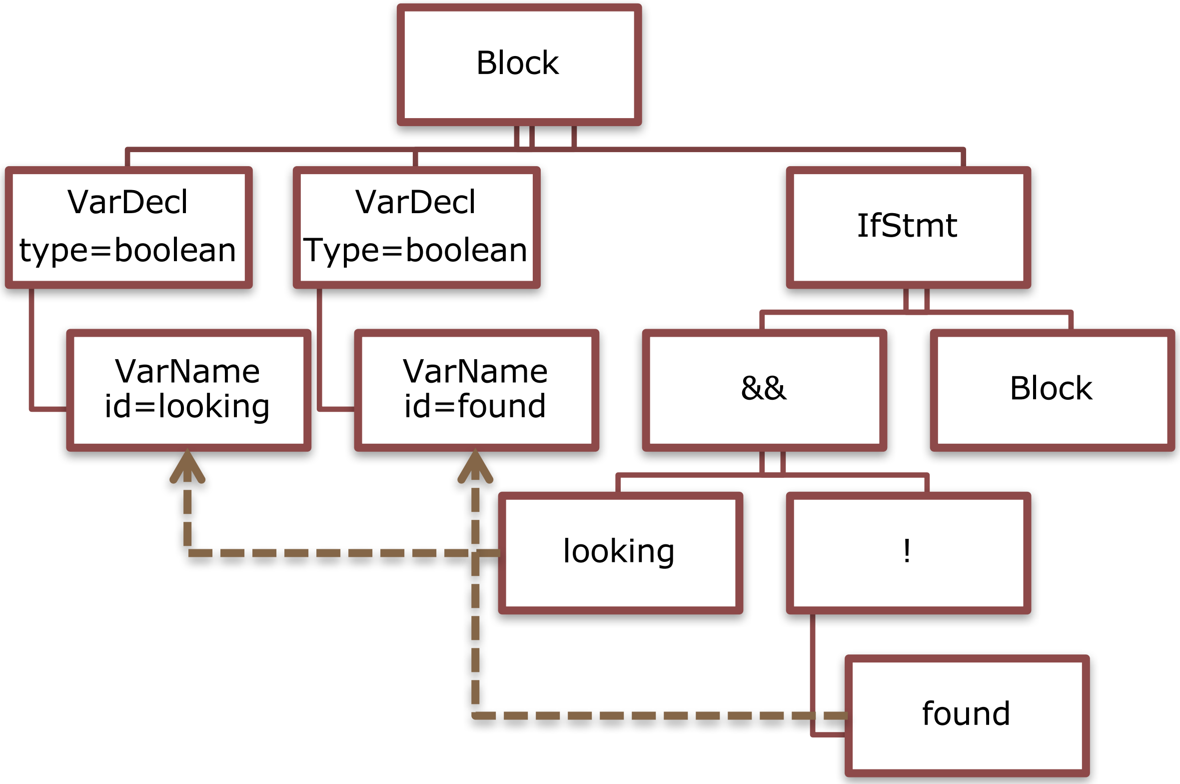
Def-Use Graphs (DUG) and Use-Definition-Use Graphs (UDUG)

- Every language and notation has
  - ***Definitions*** of items (definition of the variable Foo), who add type or other metadata
  - ***Uses*** of items (references to Foo)
- We talk in specifications or programs about *names of objects* and their use
  - Definitions are done in a data definition language (DDL)
  - Uses are part of a data query language (DQL) or data manipulation language (DML)
- Starting from the abstract syntax tree, *name analysis* finds out about the definitions of uses of names
  - Building the *Use-Def graph*
  - This revolves the meaning of used names to definitions
  - Inverting the Use-Def graph to a Use-Def-Use graph (UDUG)
  - This links all definitions to their uses

Abstract Syntax Graphs (ASG) are UDGs

- Abstract Syntax Graphs have *use-def edges* that reflect semantic relationships
  - from uses of names to definitions of names
- These edges are used for static semantic checks
  - Type checking
  - Casts and coercions
  - Type inference

boolean looking, found;
...
if (looking && !found) {...}



Refactoring on Complete Name-Resolved Graphs (Use-Def-Use Graphs)

- UDUGs are used in refactoring operations (e.g. renaming a class or a method consistently over the entire program).

- For renaming of a definition, all uses have to be changed, too
  - We need to trace all uses of a definition in the Use-Def-graph, resulting in its inverse, the *Def-Use-graph*
  - Refactoring works always on Def-Use-graphs *and* Use-Def-graphs, the *complete name-resolved graph* (the *Use-Def-Use graphs*)

Example: Rename Refactorings in Programs

Refactor the name Person to Human, using bidirectional use-def-use links:

```
class Person { .. }
class Course {
Person teacher = new Person("Jim");
Person student = new Person("John");
}
```

Definition

Reference (Use)

```
class Human { .. }
class Course {
    Human teacher = new Human("Jim");
    Human student = new Human("John");
}
```

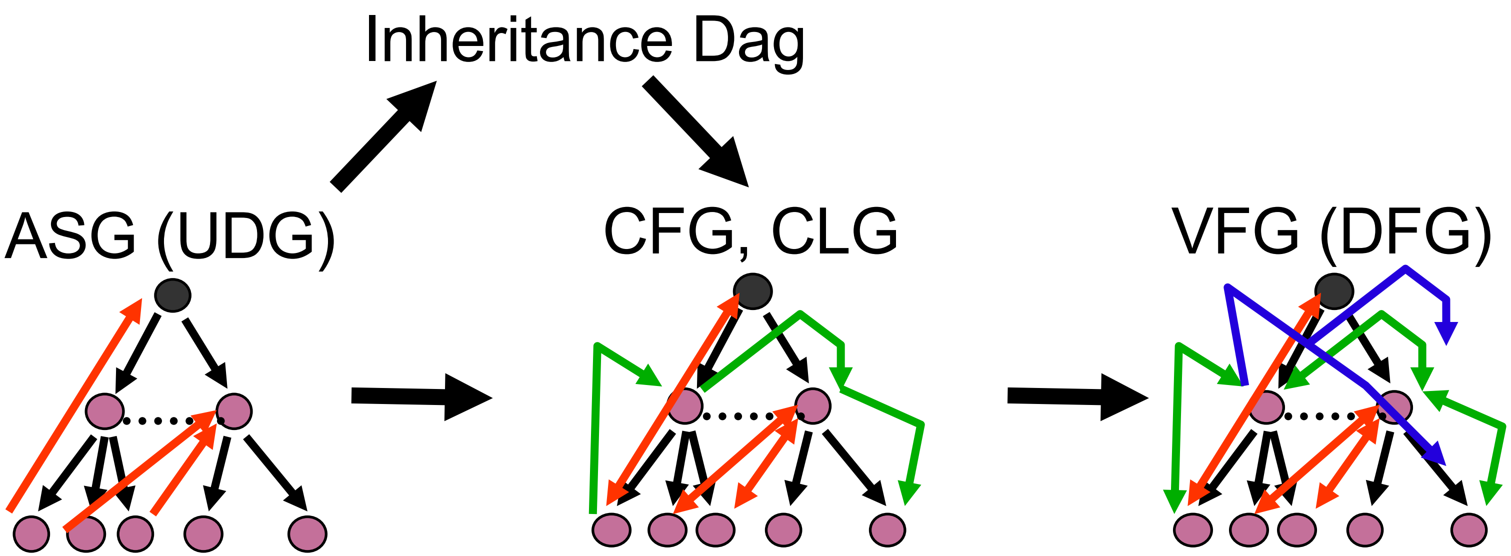
Refactoring

- Refactoring works always in the same way:
  - Change a definition
  - Find all dependent references
  - Change them
  - Recurse handling other dependent definitions
- Refactoring can be supported by tools
  - The Use-Def-Use-graph forms the basis of refactoring tools
- However, building the Use-Def-Use-Graph for a complete program costs a lot of space and is a difficult program analysis task
  - Every method that structures this graph benefits immediately the refactoring
  - either simplifying or accelerating it
- UDUGs are large
  - Efficient representation important

Further Representations for Flow Analysis

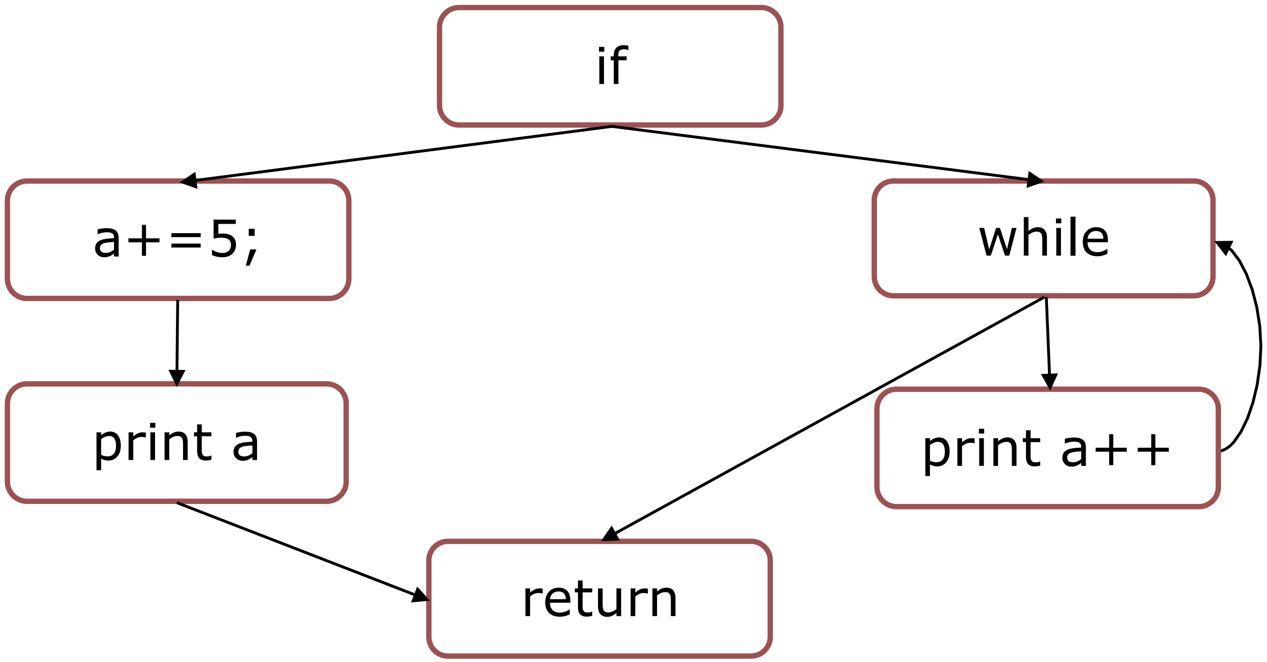
From the ASG or an UDUG, more graph-based program representations can be derived

- Inheritance Analsis
- Control-flow Analysis -> Control-Flow Graph (CFG), Call graph (CLG)
  - Records control-flow relationships
- Data-Flow Analysis -> Data-Flow Graph (DFG) or Value-Flow Graph (VFG)
  - Records flow relationships for data values



Control-Flow Graphs

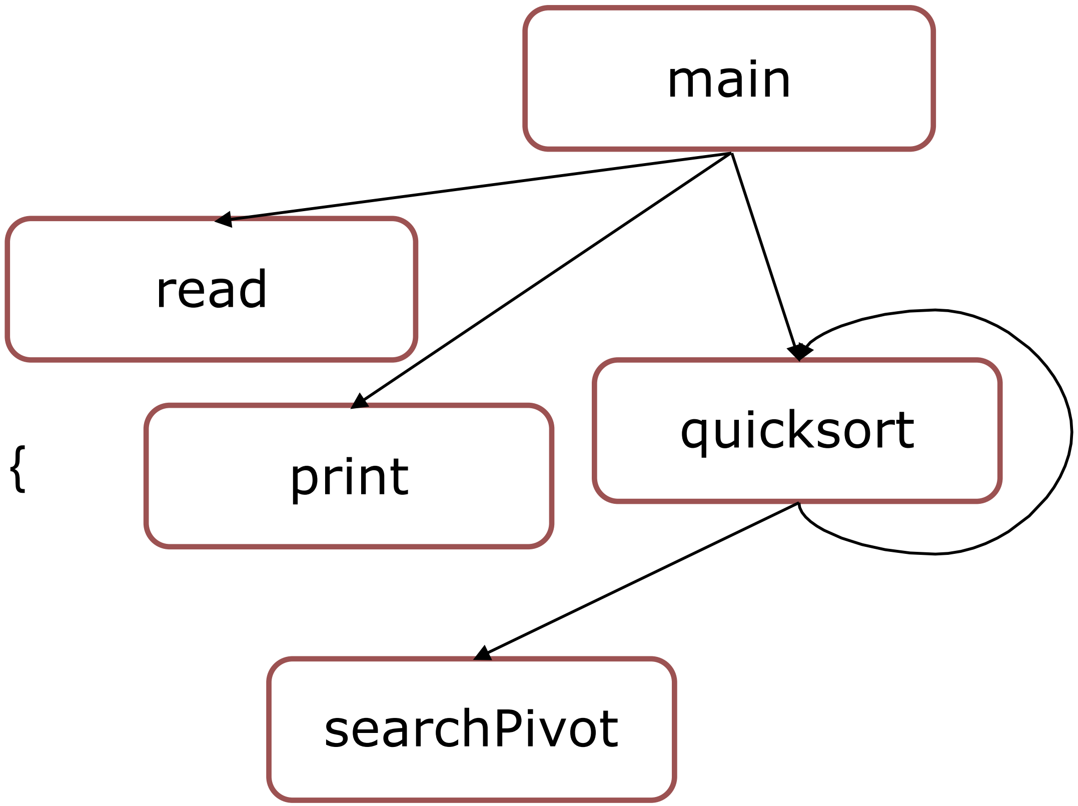
- Describe the control flow in a program
- Typically, if statements and switch statements split control flow
  - Their ends join control flow
- Control-Flow Graphs *resolve* symbolic labels
  - Perform name analysis on labels
- Nested loops are described by nested control flow graphs



Simple (Flow-Insensitive) Call Graph (CLG)

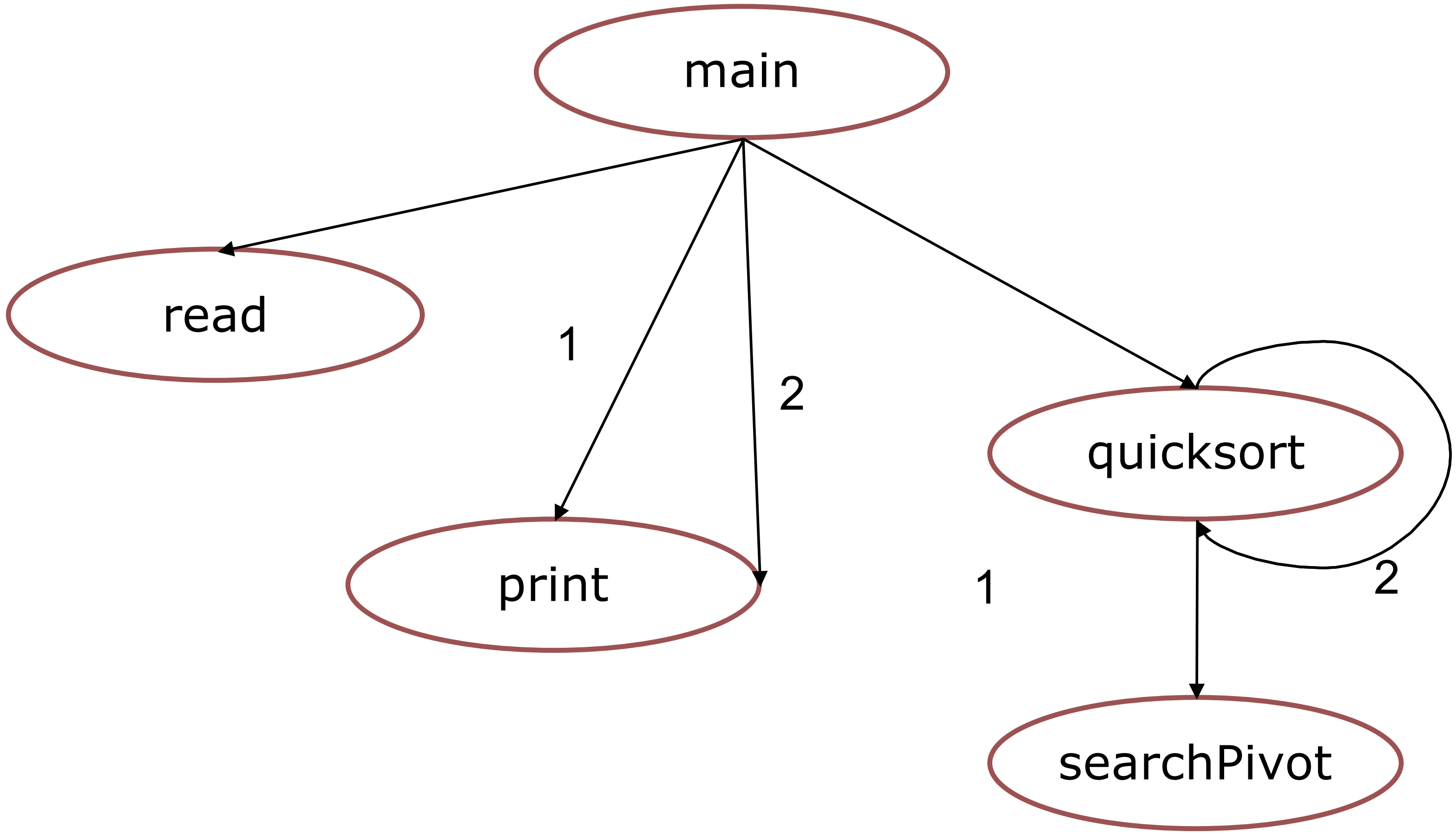
- Describe the call relationship between the procedures
  - Interprocedural control-flow analysis performs name analysis on called procedure names

```
main = procedure () {
  array int[] a = read();
  print(a);
  quicksort(a);
  print(a);
}
quicksort = procedure(a: array[0..n]) {
  int pivot = searchPivot(a);
  quicksort(a[0], a[pivot-1]);
  quicksort(a[pivot+1,n]);
}
```



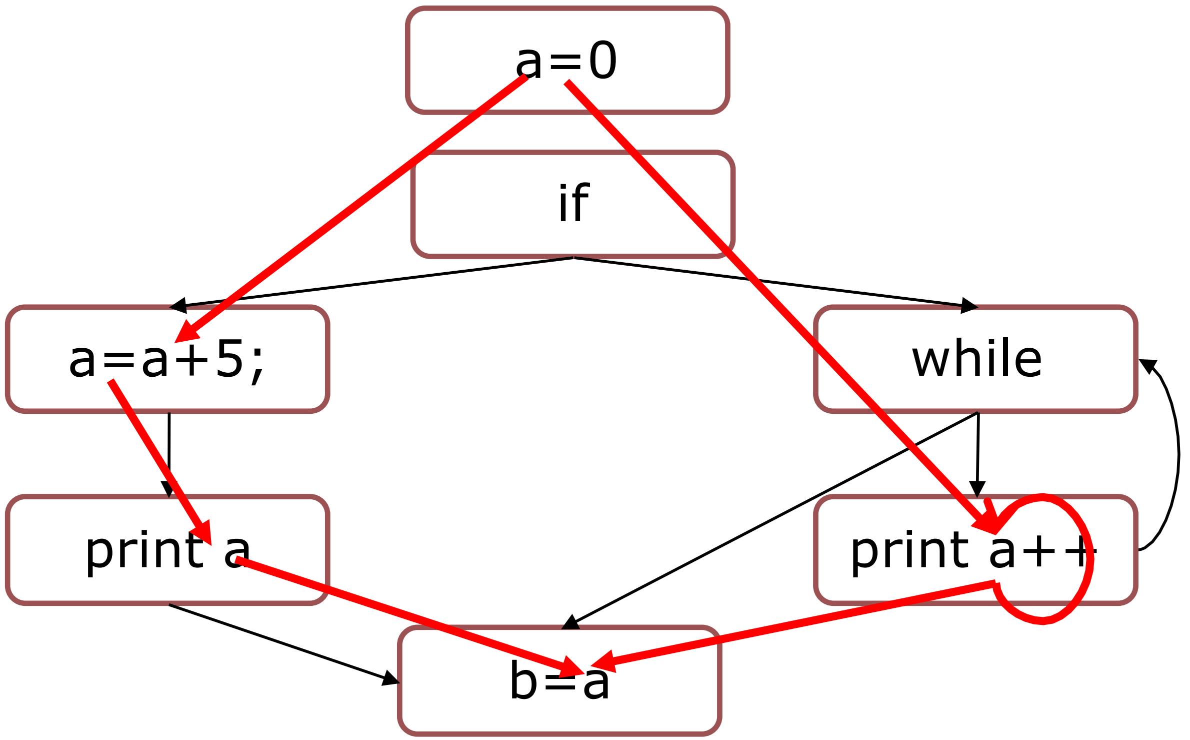
(Flow-Insensitive) Call Graph (CLG)

- Describe the call relationship between the procedures including call sites
  - Flow-insensitive
  - Flow-sensitive versions consider the control flow graph



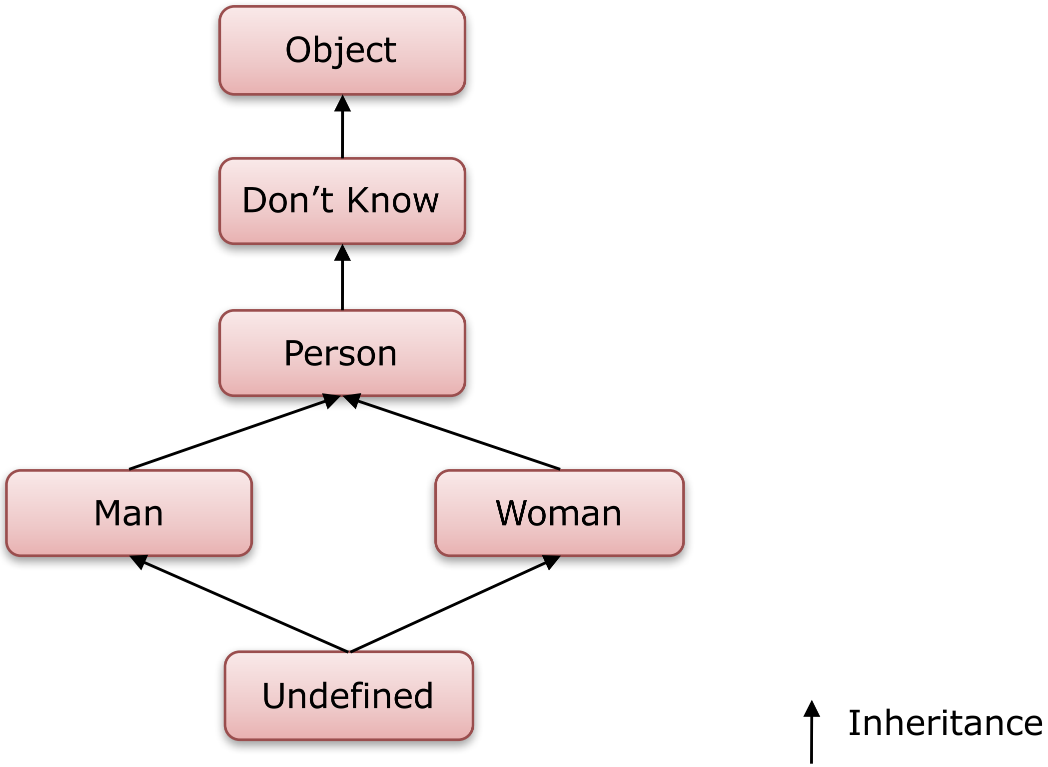
Value-Flow Graphs (VFG) aka Data-Flow Graphs (DFG)

- A *data-flow graph (DFG)* aka *value-flow graph (VFG)* describes the flow of data through the variables
  - DFG are based on control-flow graphs
- Building the data-flow graph is called *data-flow analysis*
  - Data-flow analysis is often done by *abstract interpretation,* the symbolic execution of a program at compile time



Inheritance Analysis: Building an Inheritance Tree or Inheritance Lattice

- A *lattice* is a partial order with largest and smallest element
- Inheritance hierarchies can be generalized to inheritance lattices
- An *inheritance analysis* builds the transitive closure of the inheritance lattice



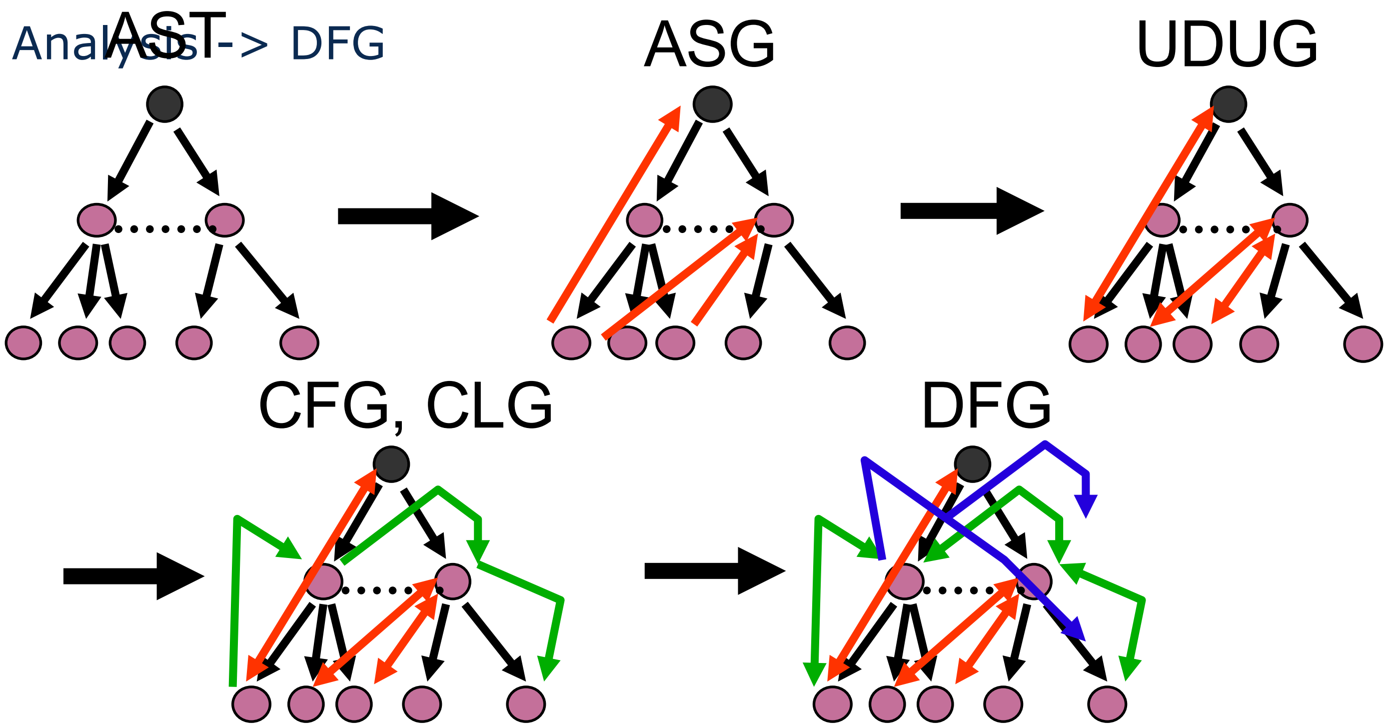
UML Graphs

- All diagram sublanguages of UML generate internal graph representations
  - They can be analyzed and checked with graph techniques
  - Graphic languages, such as UML, need a graph parser to be recognized, or a specific GUI who knows about graphic elements

- Hence, graph techniques are an essential tool of the software engineer

Remark: All Specifications Have a Graph-Based Representation

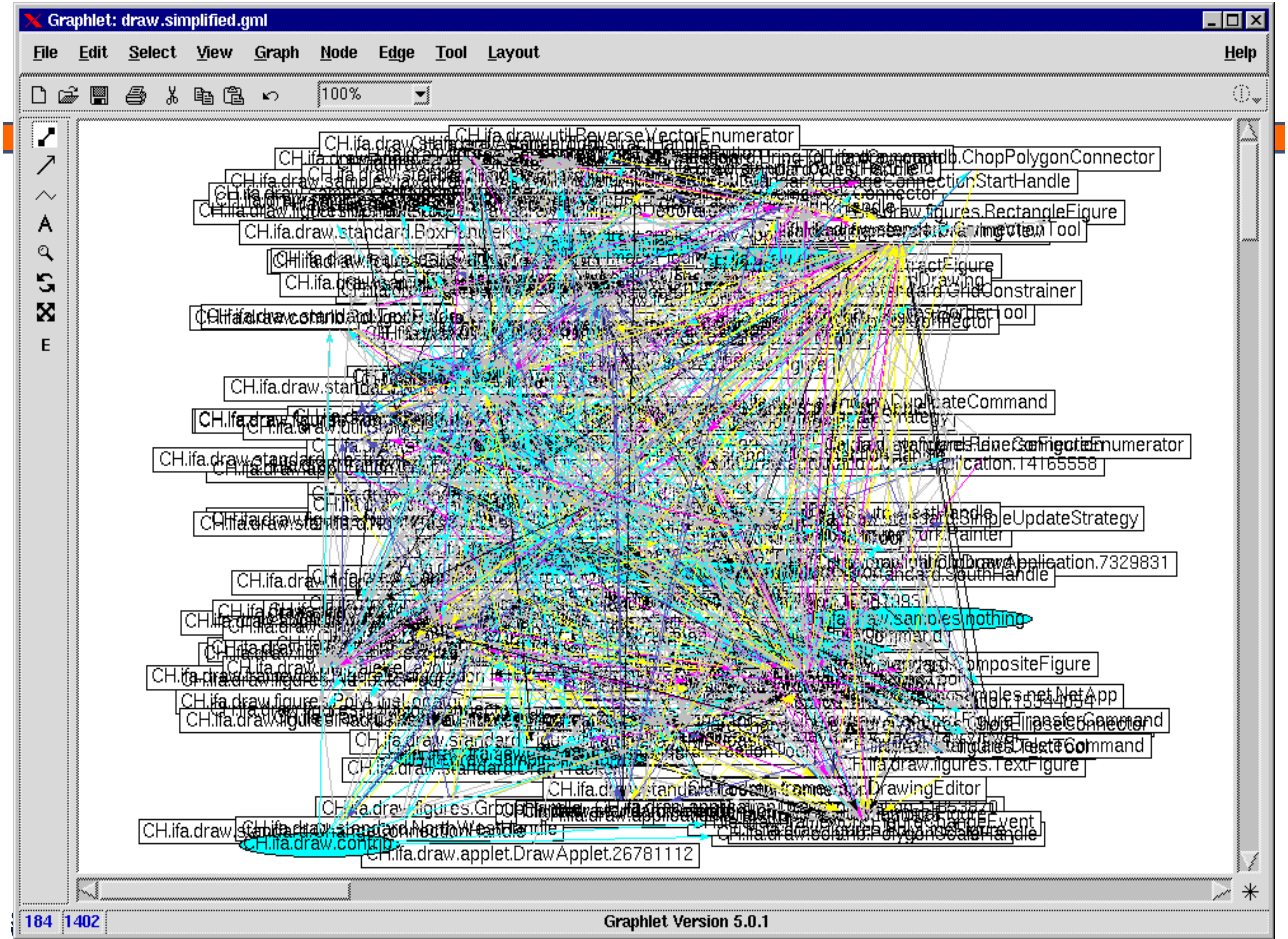
- Texts are parsed to abstract syntax trees (AST)
- Graphics are parsed by GUI or graph parser to AST also
- Through name analysis, they become abstract syntax graphs (ASG)
- Through def-use-analysis, they become Use-def-Use Graphs (UDUG)
- Control-flow Analysis -> CFG, CLG
- Data-Flow Analysis -> DFG



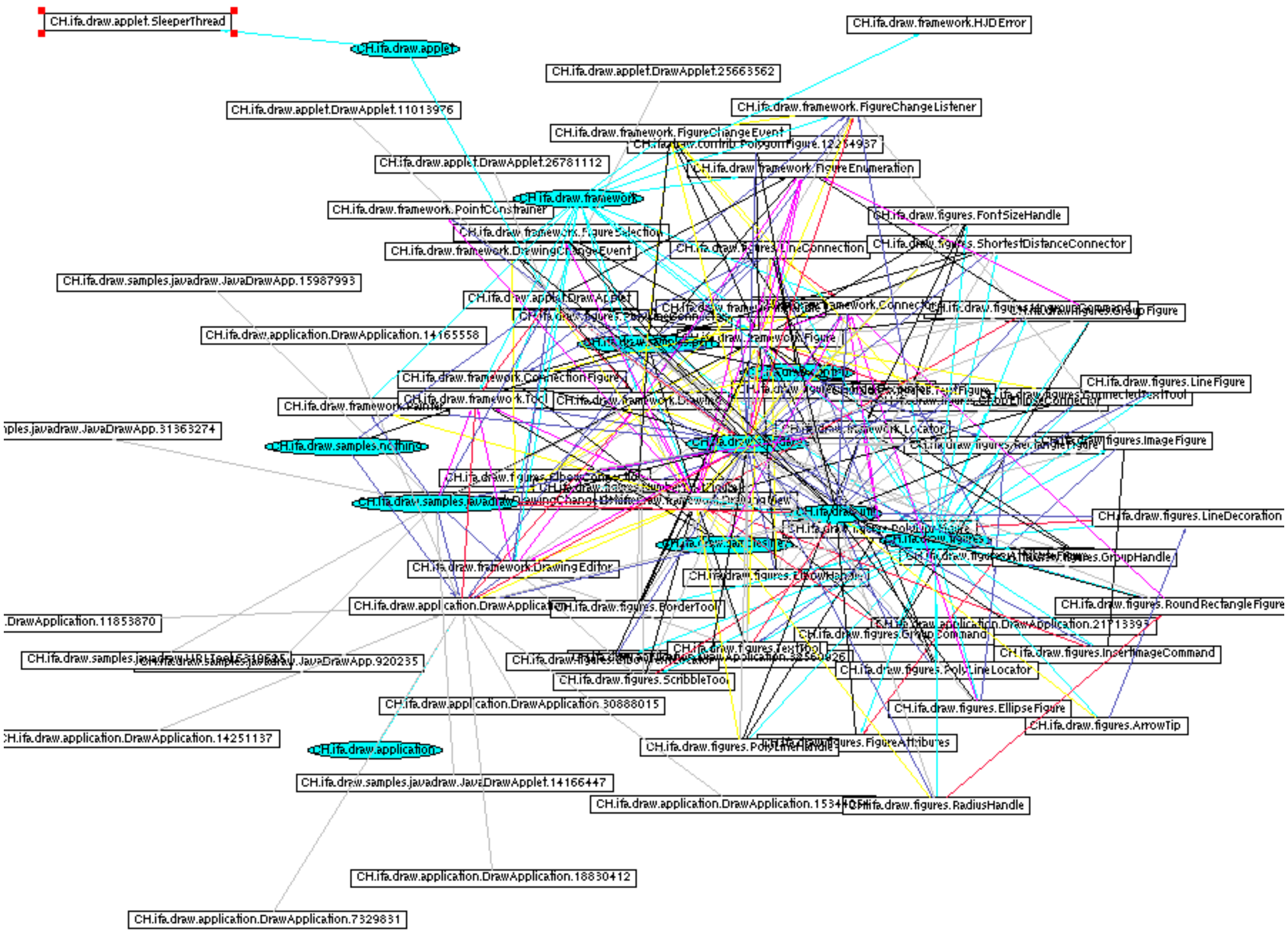
- Large models have large graphs
- They can be hard to understand

- Figures taken from Goose Reengineering Tool, analysing a Java class system [Goose, FZI Karlsruhe]

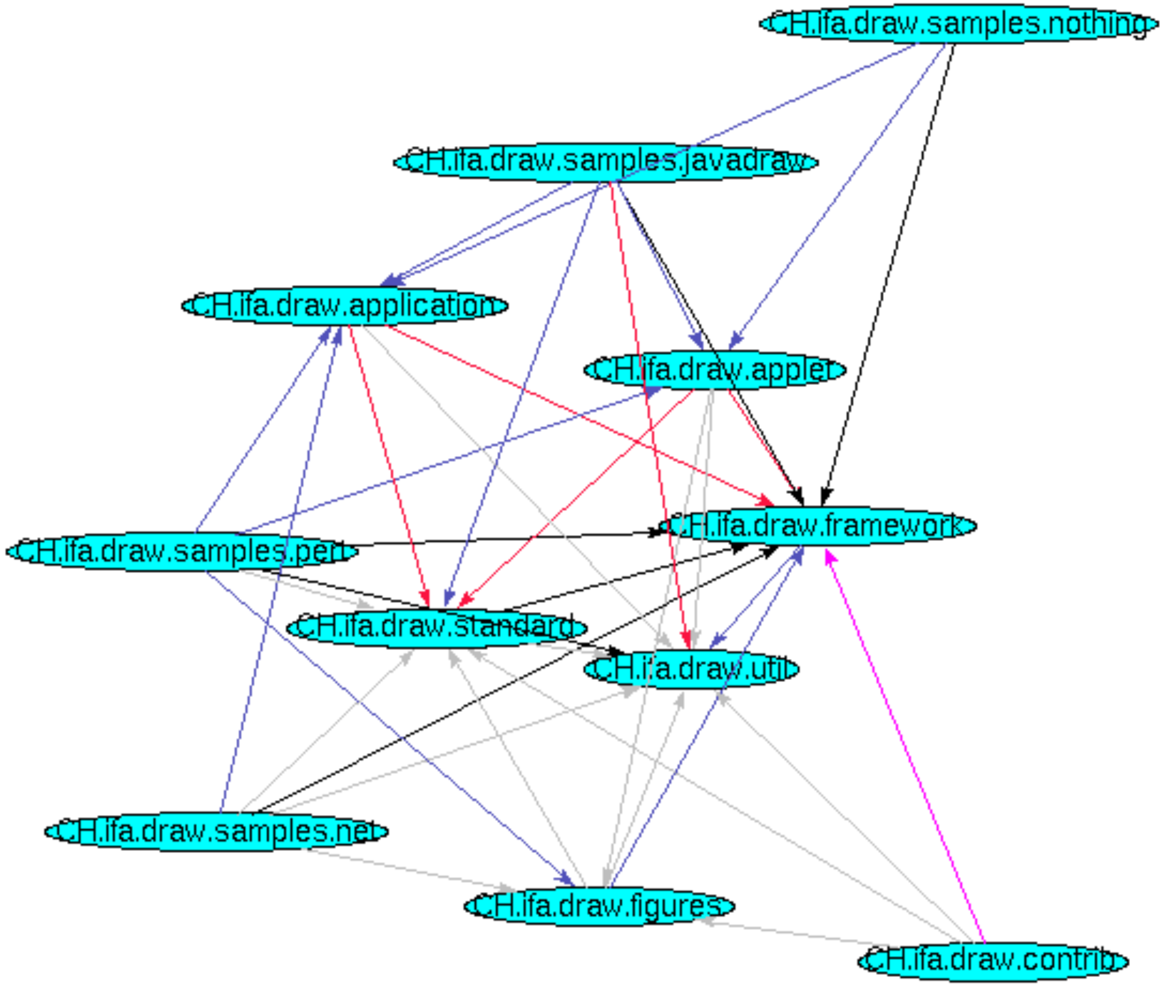
12A.2 THE PROBLEM: HOW TO MASTER LARGE GRAPHS OF MODELS AND PROGRAMS


Graphlet: draw.simplified.gml
File
Edit
Select
View
Graph
Node
Edge
Tool
Layout
Help
100%
CH.ifa.draw.util.ReverseVectorEnumerator
CH.ifa.draw.applet.DrawApplet.26781112
DuplicateCommand
eUpdateStrategy
mpositeFigure
DrawingEditor
184
1402
Graphlet Version 5.0.1

Partially Collapsed



Totally Collapsed



Requirements for Modeling in Requirements and Design

- We need guidelines how to develop simple models
- We need analysis techniques to
  - Analyze models
    - Find out about their complexity
    - Find out about simplifications
  - Search in models
  - Check the consistency of the models

The End

- Why are EARS and binary Datalog equivalent?
- Explain the graph-logic isomorphism
- Why does the „SameGeneration“ Program compute layers?
- Describe how you dump a UML classs diagram into a logic fact base
- What can be done if a model becomes too large?