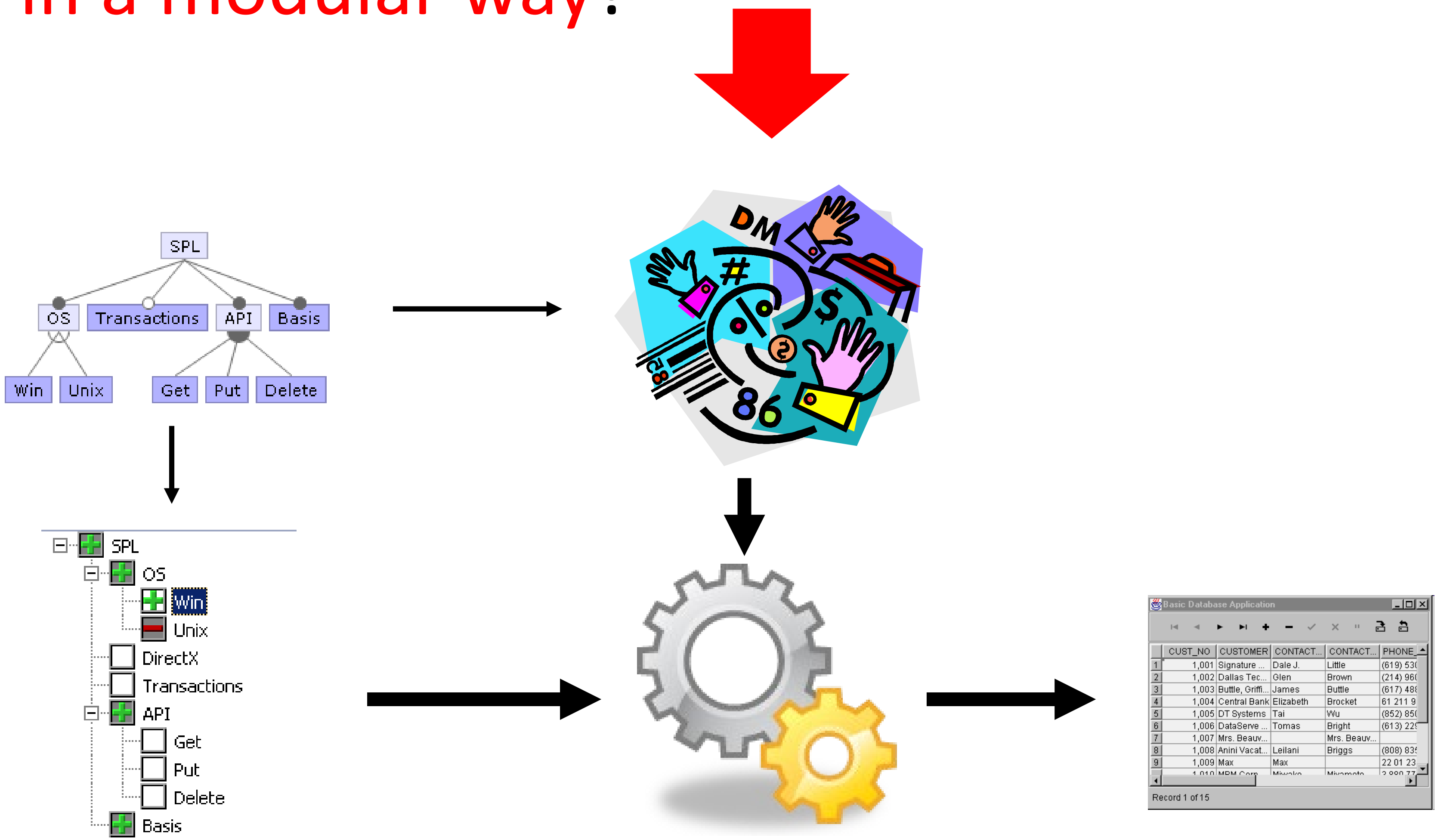
Software Product Lines
Part 6: Feature-Orientation

Daniel Strüber, Radboud University

with courtesy of: **Sven Apel, Christian Kästner, Gunter Saake**

How to implement variability in a modular way?



Goals

- Solve problems:
  - Feature Traceability
  - Crosscutting concerns
  - Preplanning
  - Inflexible extension mechanisms (inheritance)
- Modular feature implementation
- New types of implementation techniques

Agenda

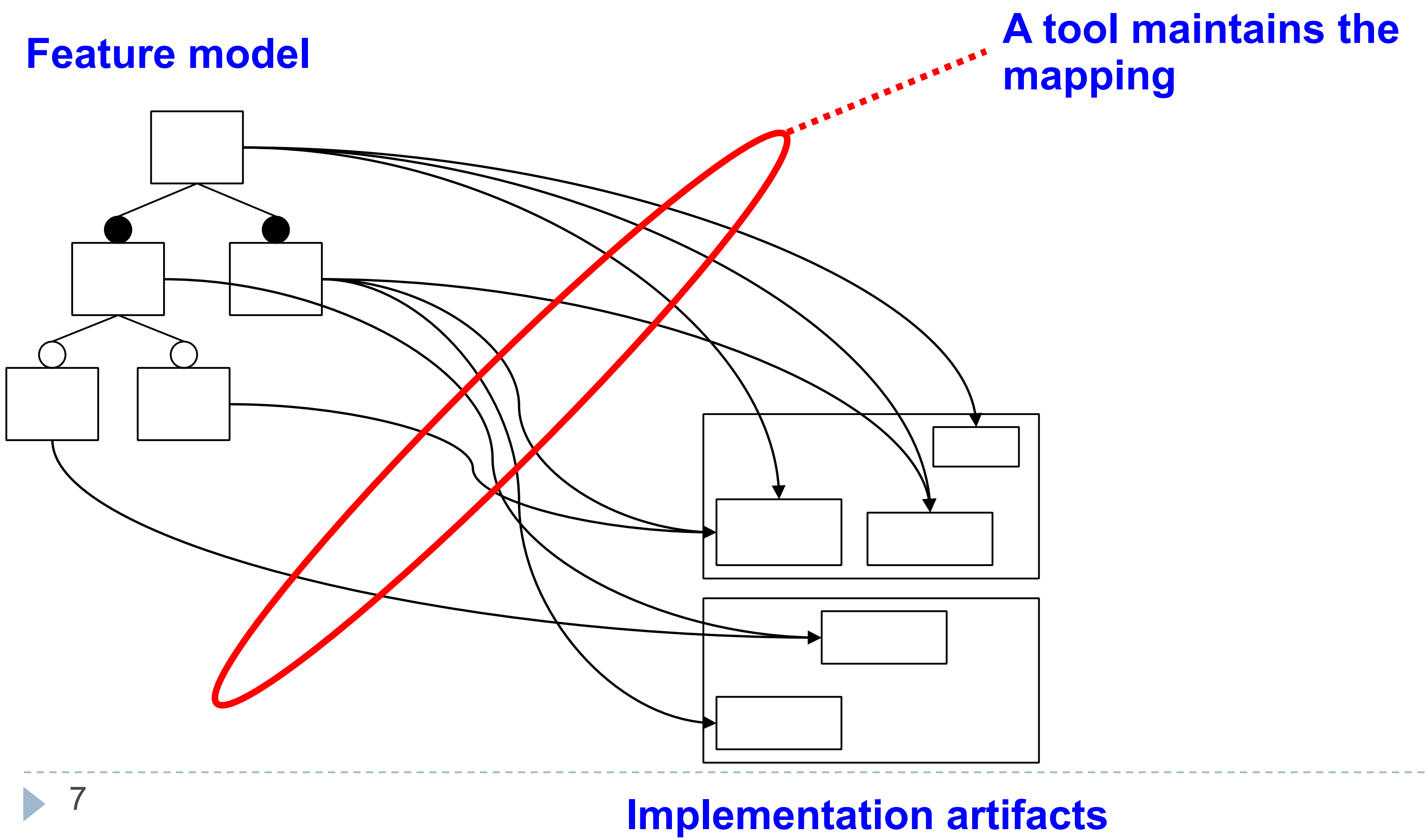
- Key idea
- Implementation with AHEAD and FeatureHouse
- Uniformity principle

Key idea

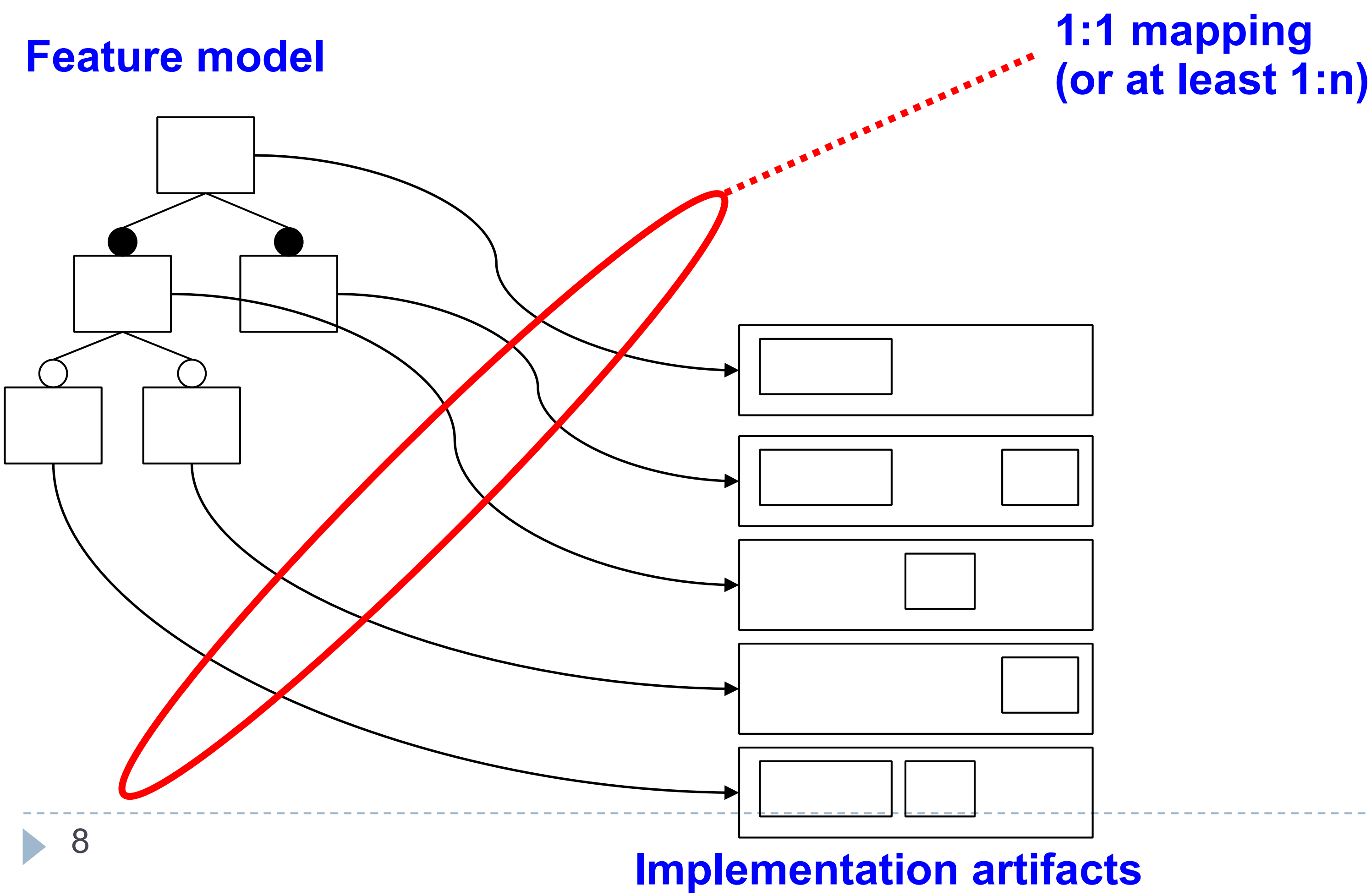
Goal: feature cohesion

- we want to have all implementation artifacts for a feature a single location in the code
  - features explicit in code
- A question of programming language and programming environment
  - physical vs. virtual cohesion
- Automatically gives us traceability as well

Feature Traceability with Tool Support



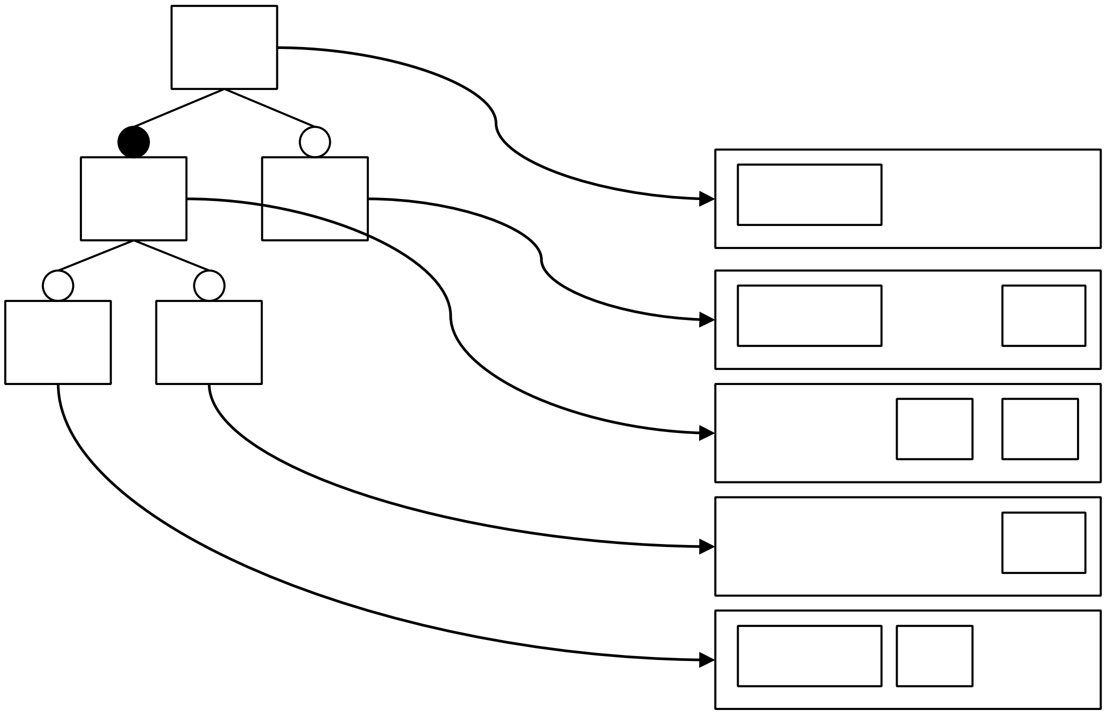
Feature Traceability with Language Support



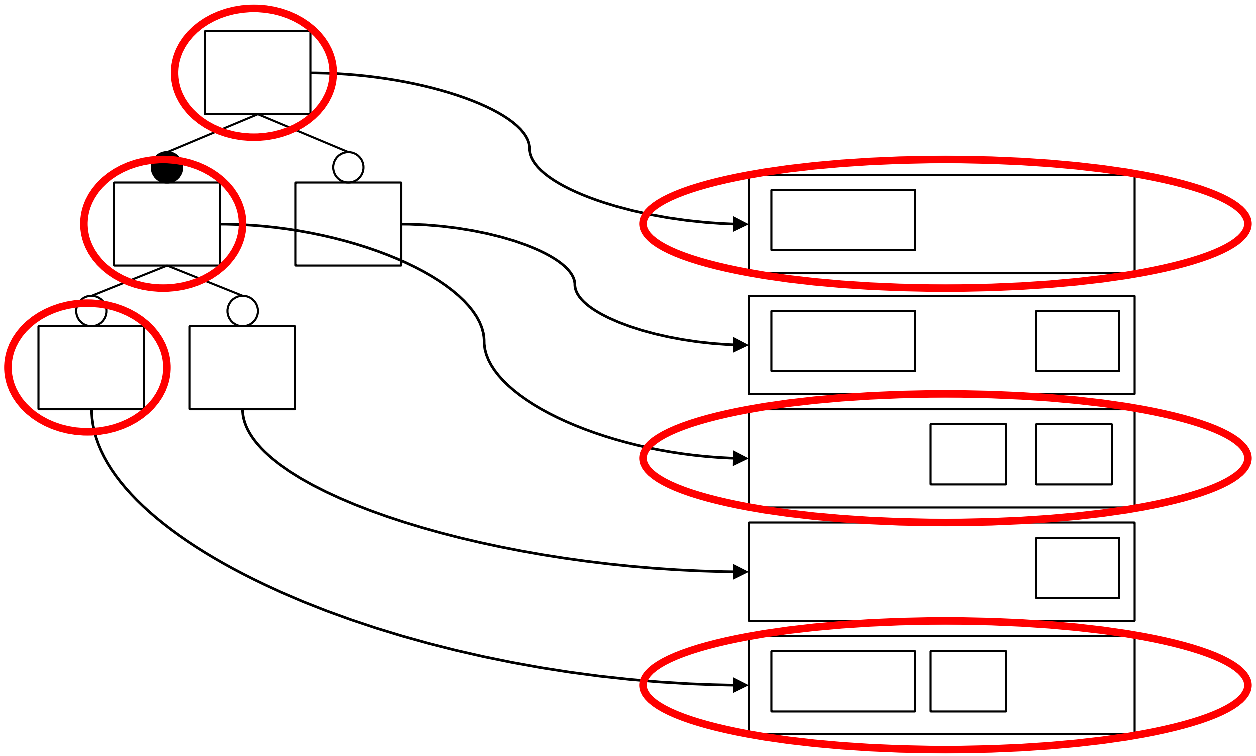
Feature-Oriented Programming

- Language-based approach for taming the feature traceability problem
- Implement each feature in a feature module
  - Perfect feature traceability
  - Separation and modularization of features
- Feature-based program generation
  - Programs are generated via composition of feature modules
- As a research idea, introduced 20 years ago
  - Prehofer, ECOOP‘97 and Batory, ICSE'03

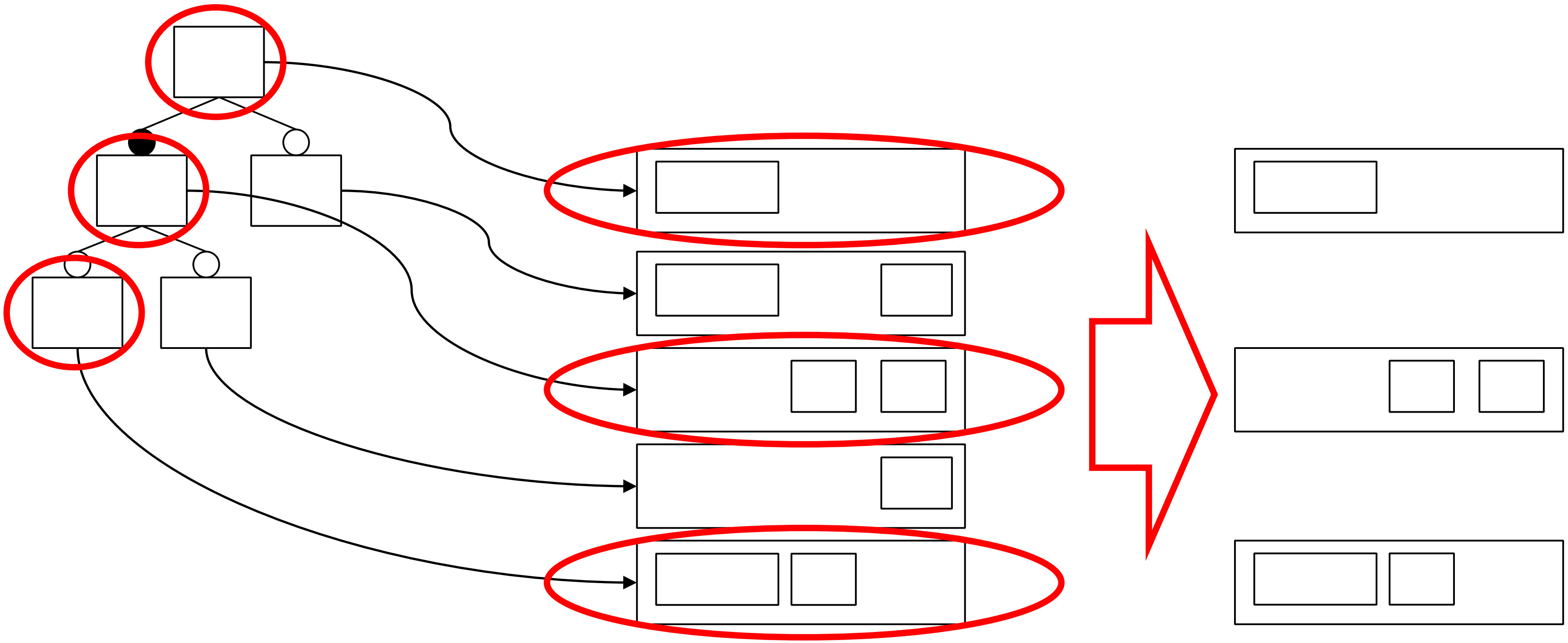
Feature Composition



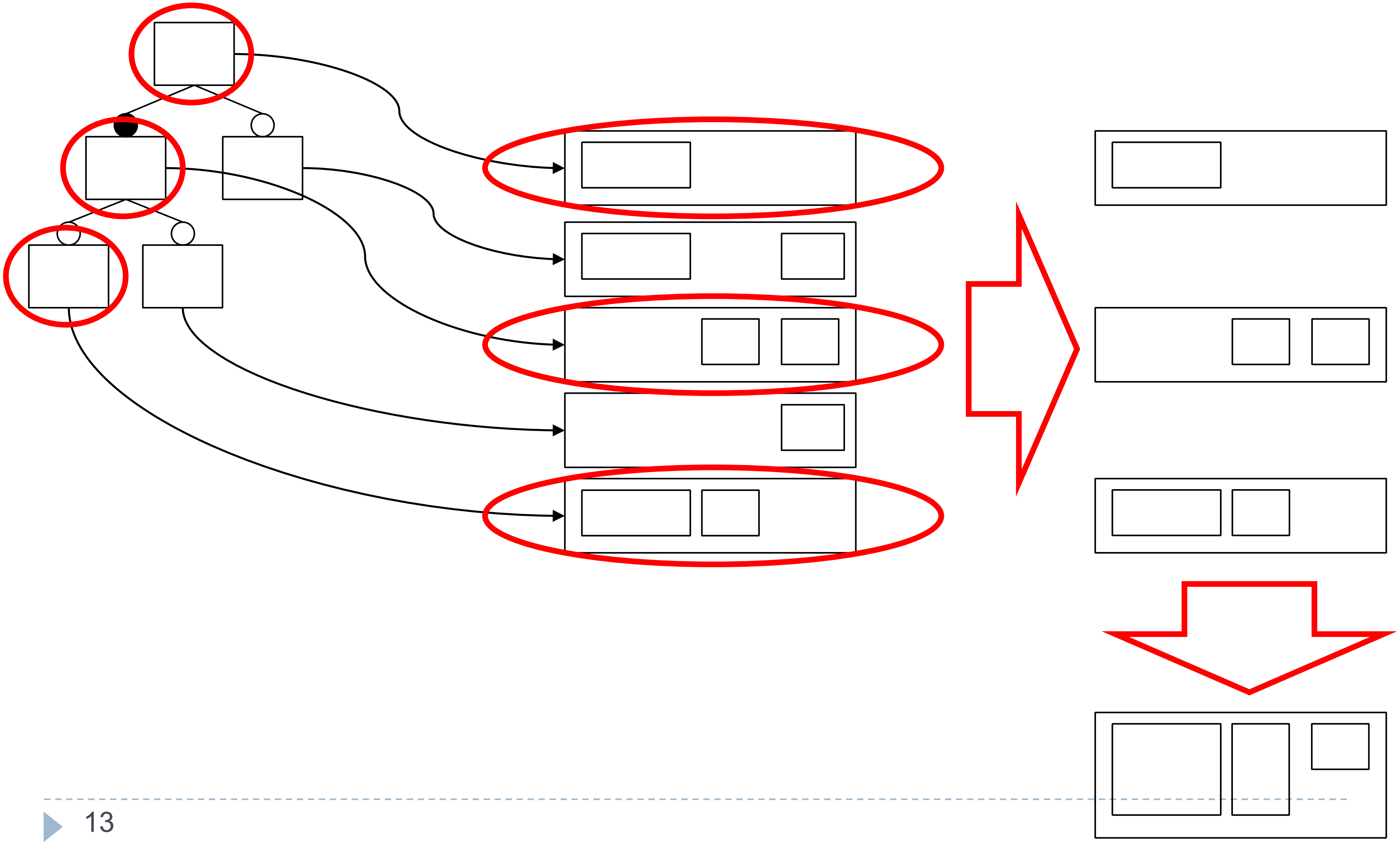
Feature Composition



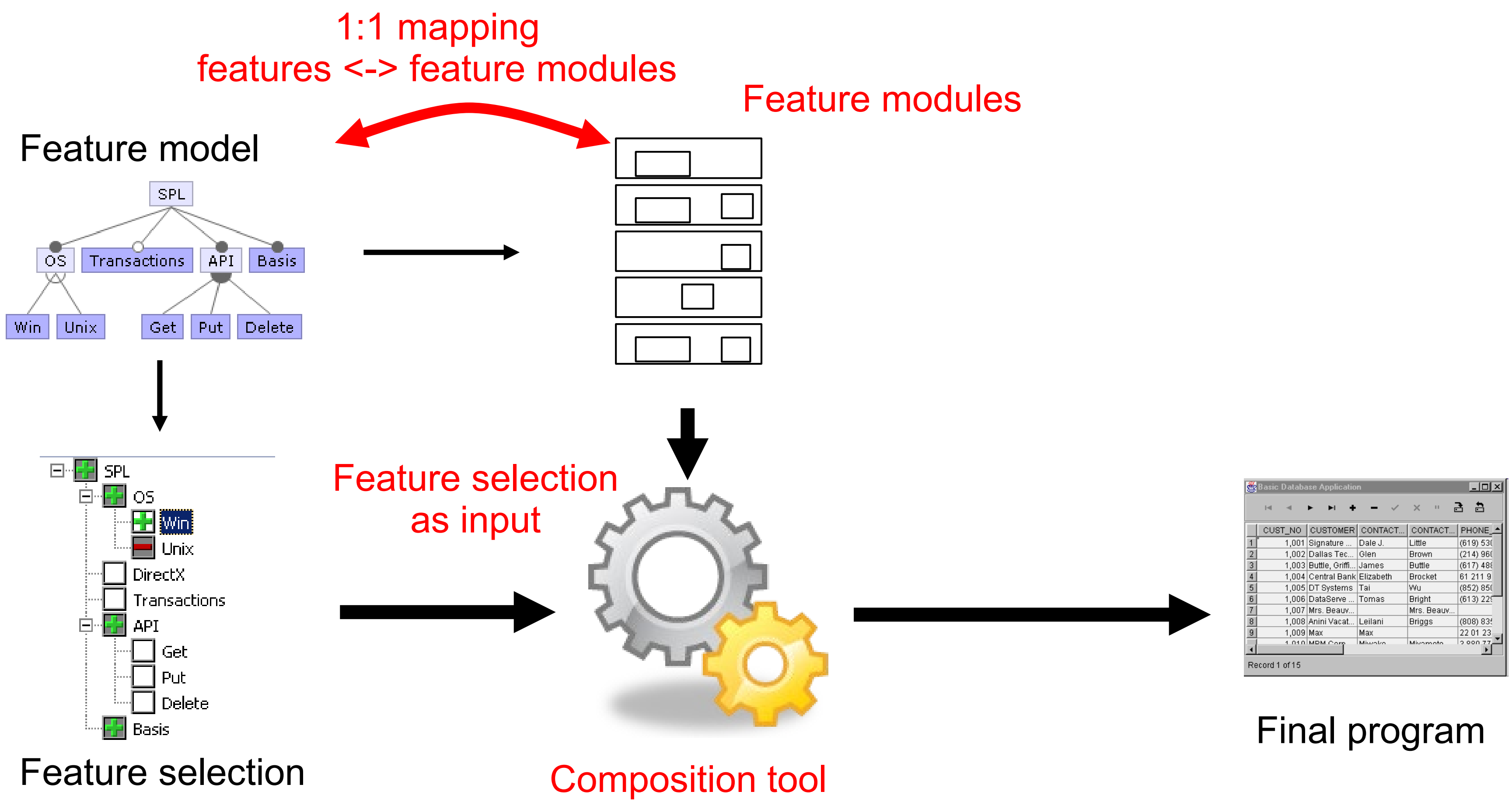
Feature Composition



Feature Composition



Product lines with feature modules



Implementation with AHEAD and FeatureHouse

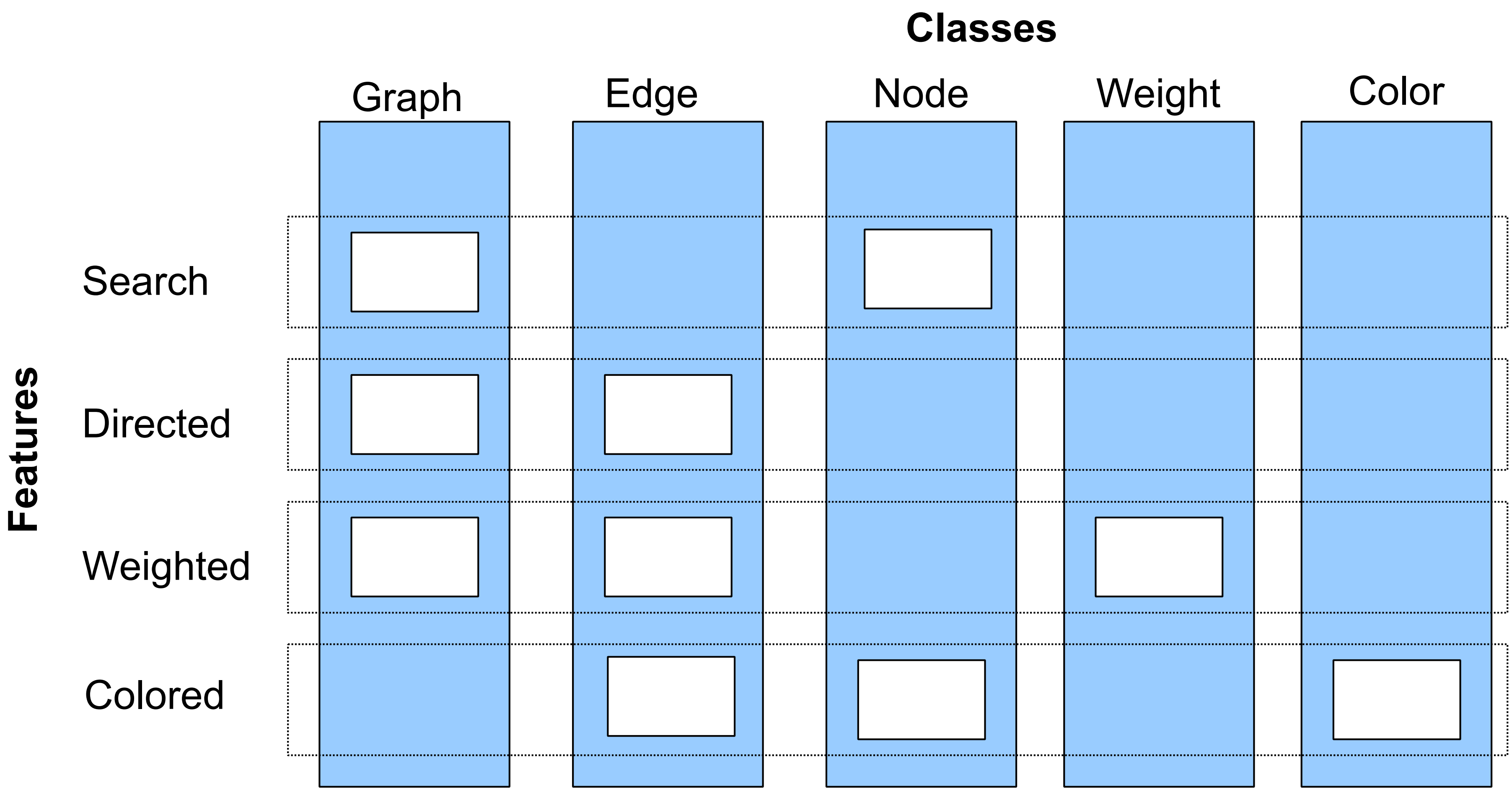
Implementing feature modules

- Starting point: code base structured into classes
- Features often implemented by several classes
- Classes often implement more than one feature

- Idea: keep class structure, but split classes along features

- Implemented in tools AHEAD (Algebraic Hierarchical Equations for Application Design) and FeatureHouse

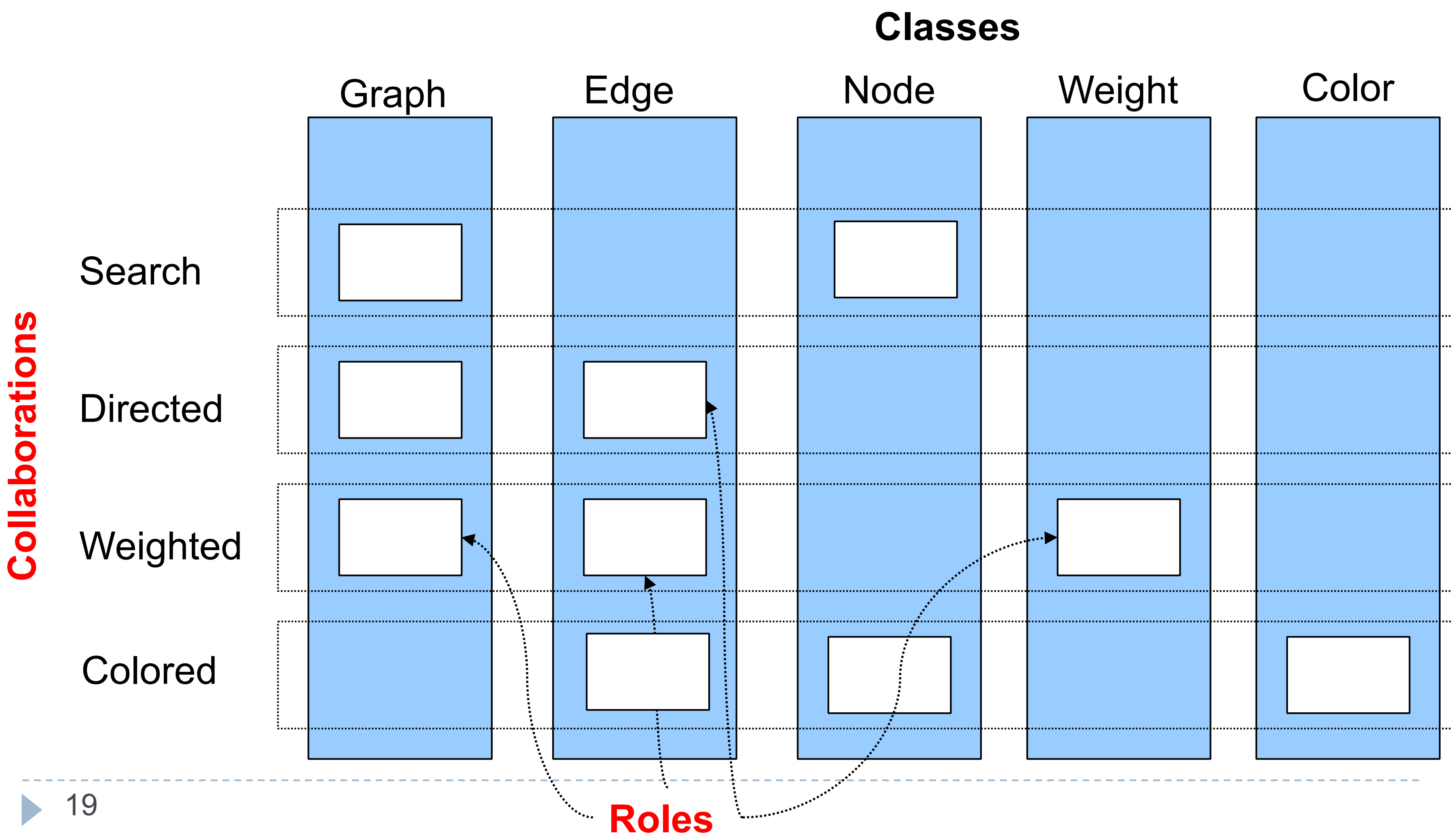
Splitting of classes



Collaborations & roles

- Collaboration: a set of classes that interact to implement a feature
- Different classes play different roles within collaborations
- One class plays different roles in different collaborations
- A role encapsulates the functionality (methods, fields) of a class that is relevant for the collaboration

Collaborations & roles



Collaborations in graph example

```
class Graph {
  List nodes = new List();
  List edges = new List();
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m);
    edges.add(e); return e;
  }
  void print() {
    for(int i = 0; i < edges.size(); i++)
      ((Edge)edges.get(i)).print();
  }
}
```

```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

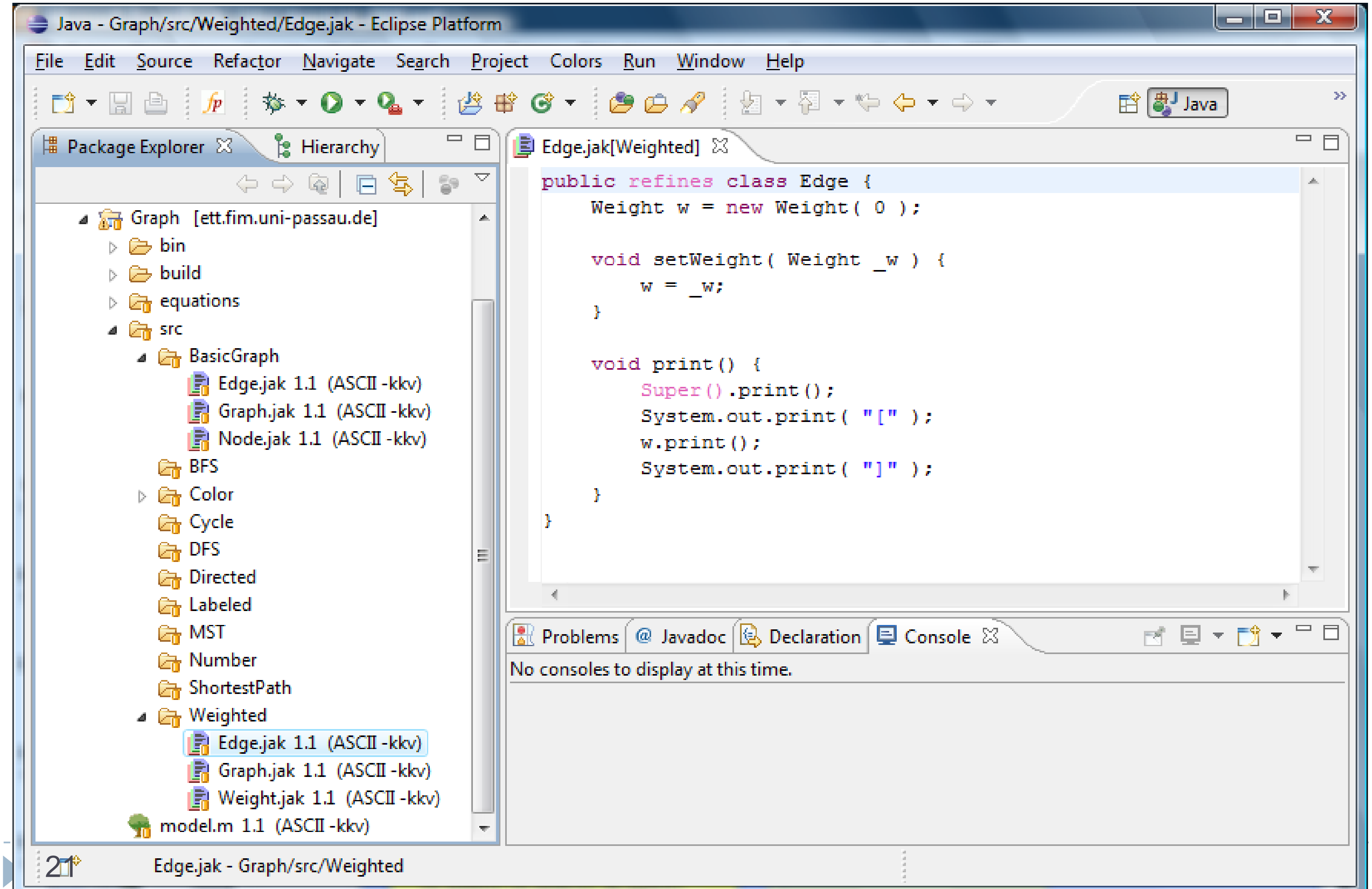
```
class Node {
  int id = 0;
  void print() {
    System.out.print(id);
  }
}
```

```
refines class Graph {
  Edge add(Node n, Node m) {
    Edge e = Super.add(n, m);
    e.weight = new Weight();
  }
  Edge add(Node n, Node m, Weight w)
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m);
    edges.add(e);
    e.weight = w; return e;
  } }
```

```
refines class Edge {
  Weight weight = new Weight();
  void print() {
    Super.print(); weight.print();
  }
}
```

```
class Weight {
  void print() { ... }
}
```

Directory hierarchy: features -> roles



Example: class refinements

Successive extension of base implementation by means of refinements

Edge.jak

```
class Edge {
    private Node start; ...
}
```

Edge.jak

```
refines class Edge {
    private int weight;
    ...
}
```

Edge.jak

```
refines class Edge {
    private Color color;
    ...
}
```

Method refinements (AHEAD)

- Each extension can refine and introduce methods
- Methods can be overriden
- Methods from the next refinement level can be called with **`Super`***
- Similar to inheritance

```
class Edge {
    void print() {
        System.out.print(
          " Edge between " + node1 +
          " and " + node2);
    }
}
```

```
refines class Edge {
    private Node start;
    void print() {
        Super().print();
        System.out.print(
            " directed from " + start);
    }
}
```

```
refines class Edge {
    private int weight;
    void print() {
        Super().print();
        System.out.print(
            " weighted with " + weigth);
    }
}
```

* For technical reasons, it's necessary to specify the input parameter types in the call of `Super`, e.g. `Super(String,int).print('abc', 3)`

Method refinement (FeatureHouse)

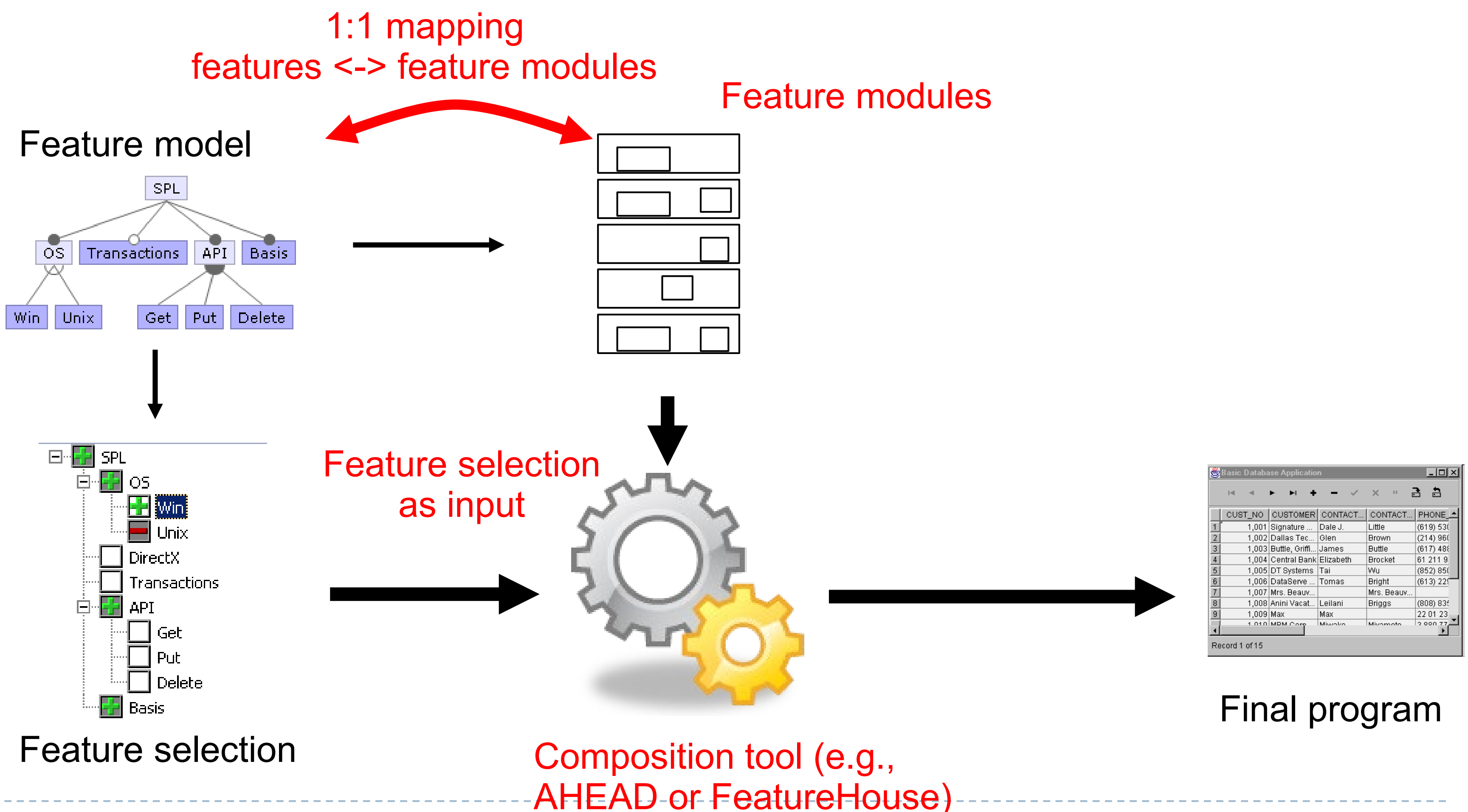
- No explicit keyword
- Each extension can refine and introduce methods
- Methods can be overriden
- Methods from the next refinement level can be called with **`original`**
- Similar to inheritance

```
class Edge {
    void print() {
        System.out.print(
          " Edge between " + node1 +
          " and " + node2);
    }
}
```

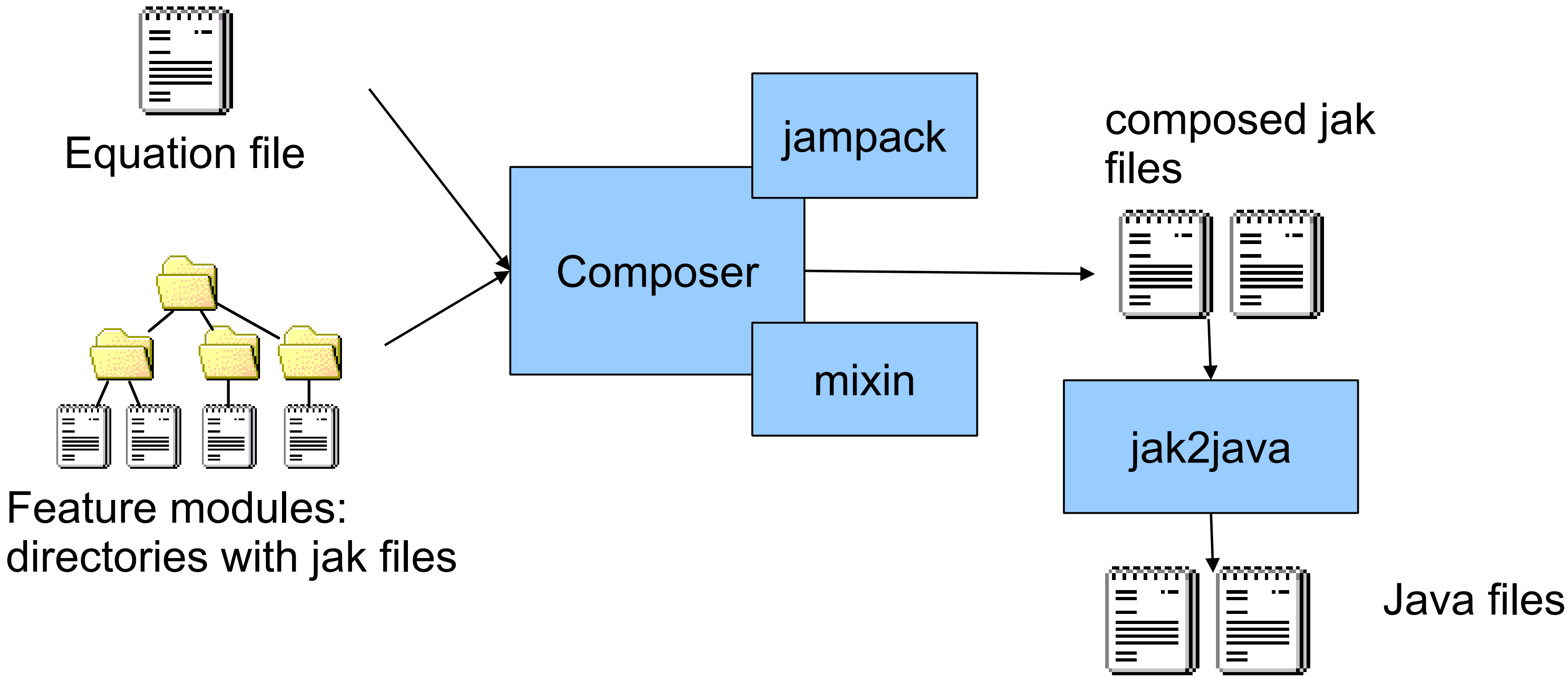
```
class Edge {
    private Node start;
    void print() {
        original();
        System.out.print(
            " directed from " + start);
    }
}
```

```
class Edge {
    private int weight;
    void print() {
        original();
        System.out.print(
            " weighted with " + weight);
    }
}
```

Product lines with feature modules

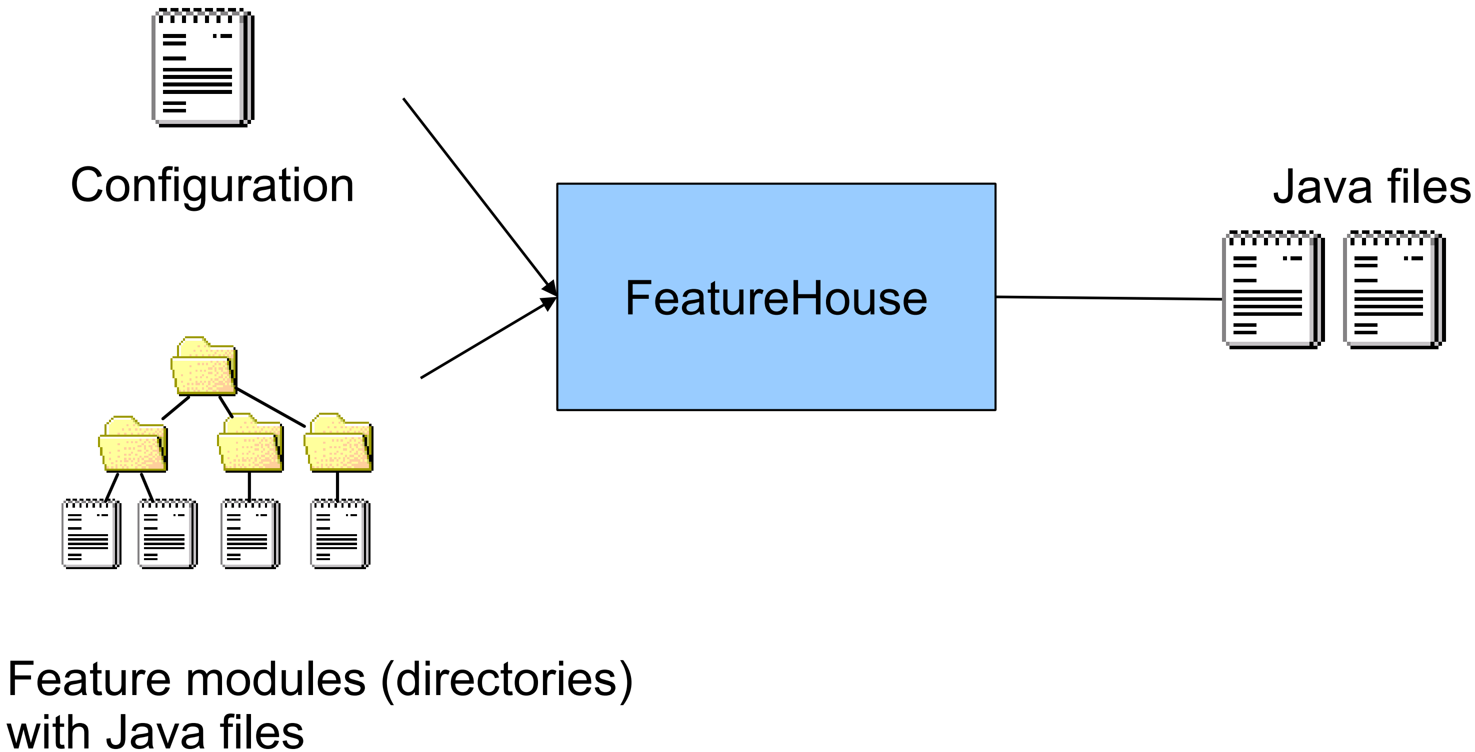


Composition in AHEAD



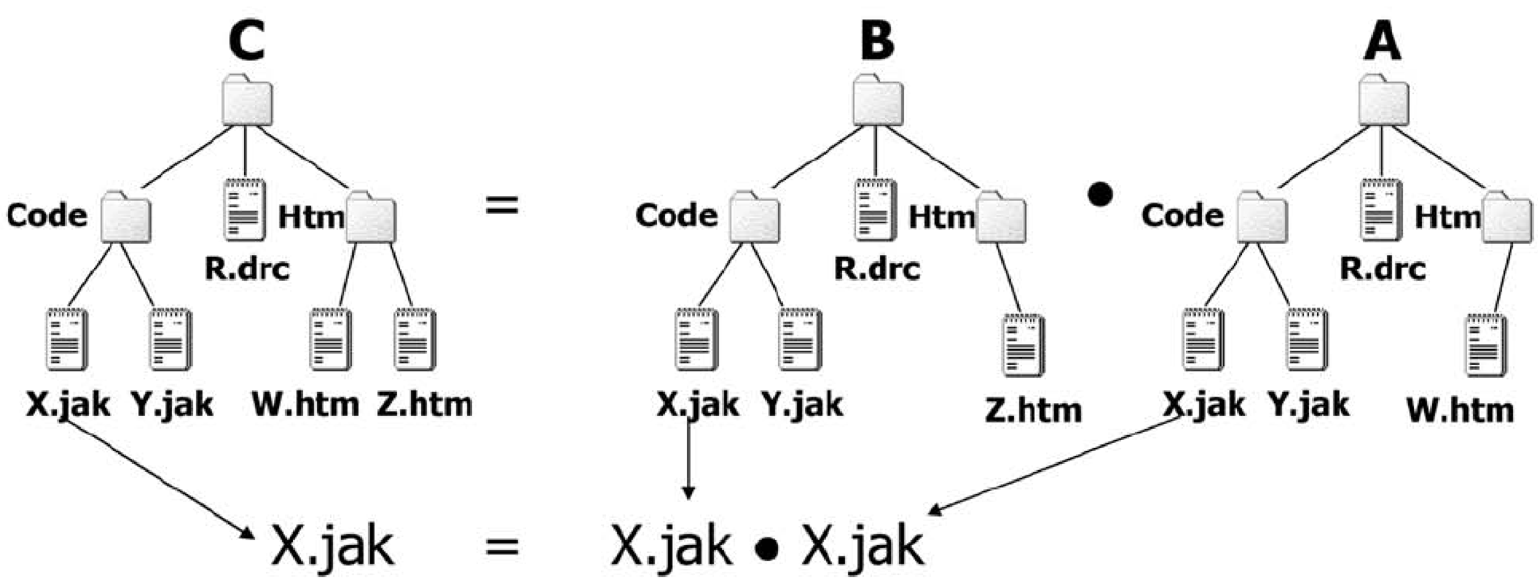
- The composer creates per class one jak file
  - **jampack**: refinement hierarchy of roles „flattened“
  - **mixin**: refinement hierarchy of roles represented by inheritance

Composition in FeatureHouse

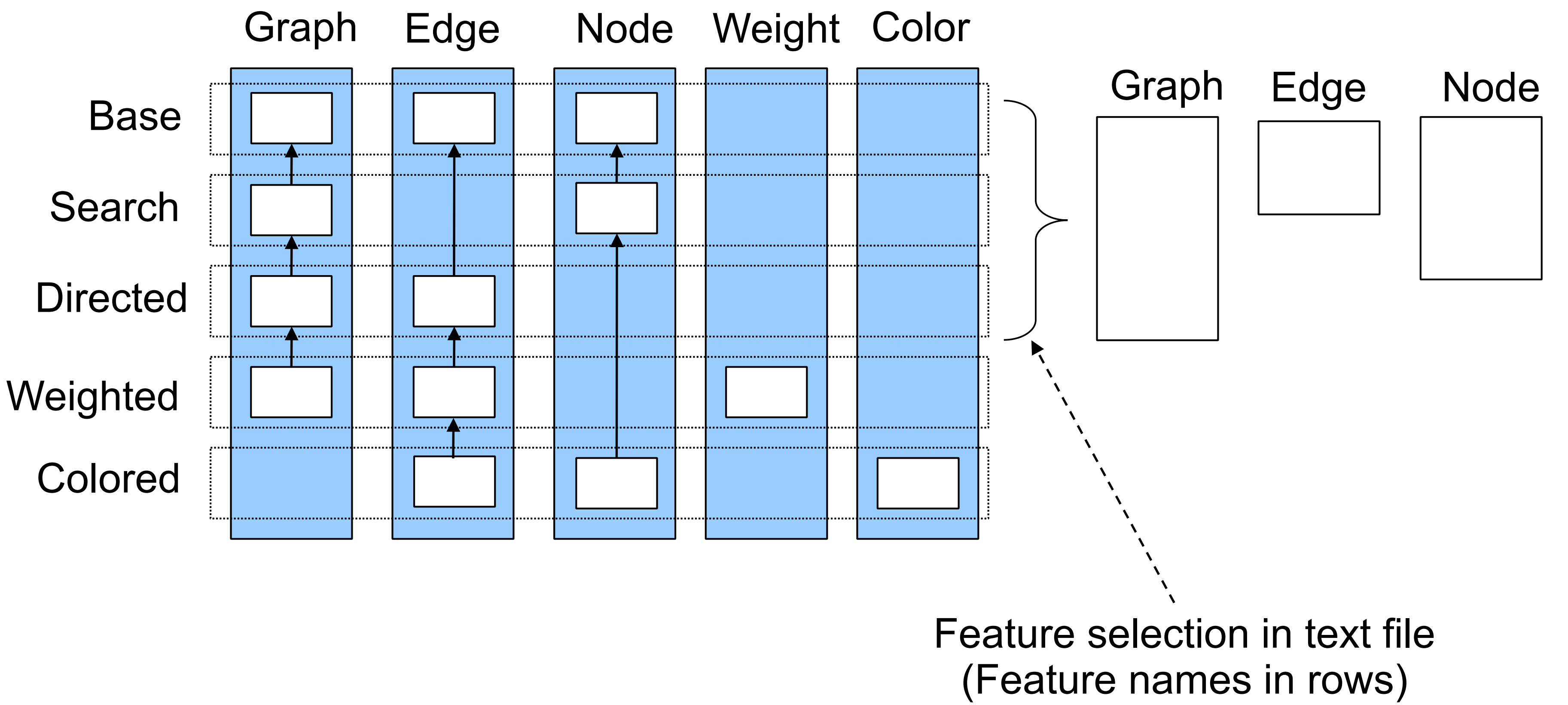


Composition of directories

- All roles of a collaboration are stored in a package/module, typically in a directory
- Composition of collaborations by composition of classes with all contained refinements of equal name



Example composition

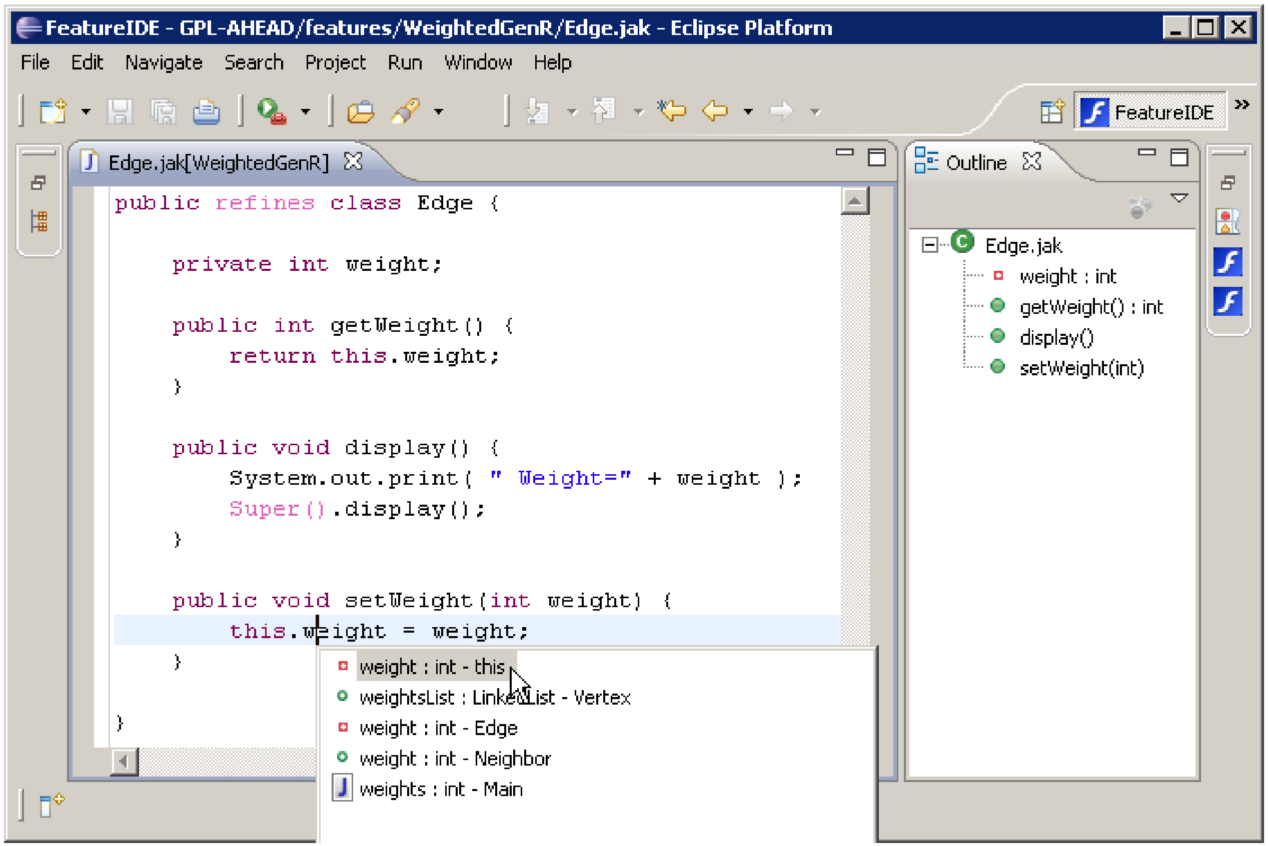


Tools

- AHEAD Tool Suite + Documentation
  - Command line tools for Jak (Java 1.4 extension)
  - http://www.cs.utexas.edu/users/schwartz/ATS.html
- FeatureHouse
  - Command line tool for Java, C#, C, Haskell, UML, …
  - http://www.fosd.de/fh
- FeatureC++
  - Alternative to AHEAD für C++
  - http://www.fosd.de/fcpp
- FeatureIDE
  - Eclipse-Plugin for AHEAD, FeatureHouse und FeatureC++
  - Automated build, syntax highlighting, etc…
  - http://www.fosd.de/featureide

FeatureIDE – Demo

- Video-Tutorial



https://www.youtube.com/watch?v=yRF0Kfs1NRA

Summary AHEAD and FeatureHouse

- One base class + arbitrary refinements (roles)
- Class refinements can…
  - Introduce fields
  - Introduce methods
  - Change (extend) method implementations
- Feature module (collaboration): directory with base classes and/or refinements
- During composition, base class and refinements for selected features are plugged together

Uniformity principle

Uniformity principle

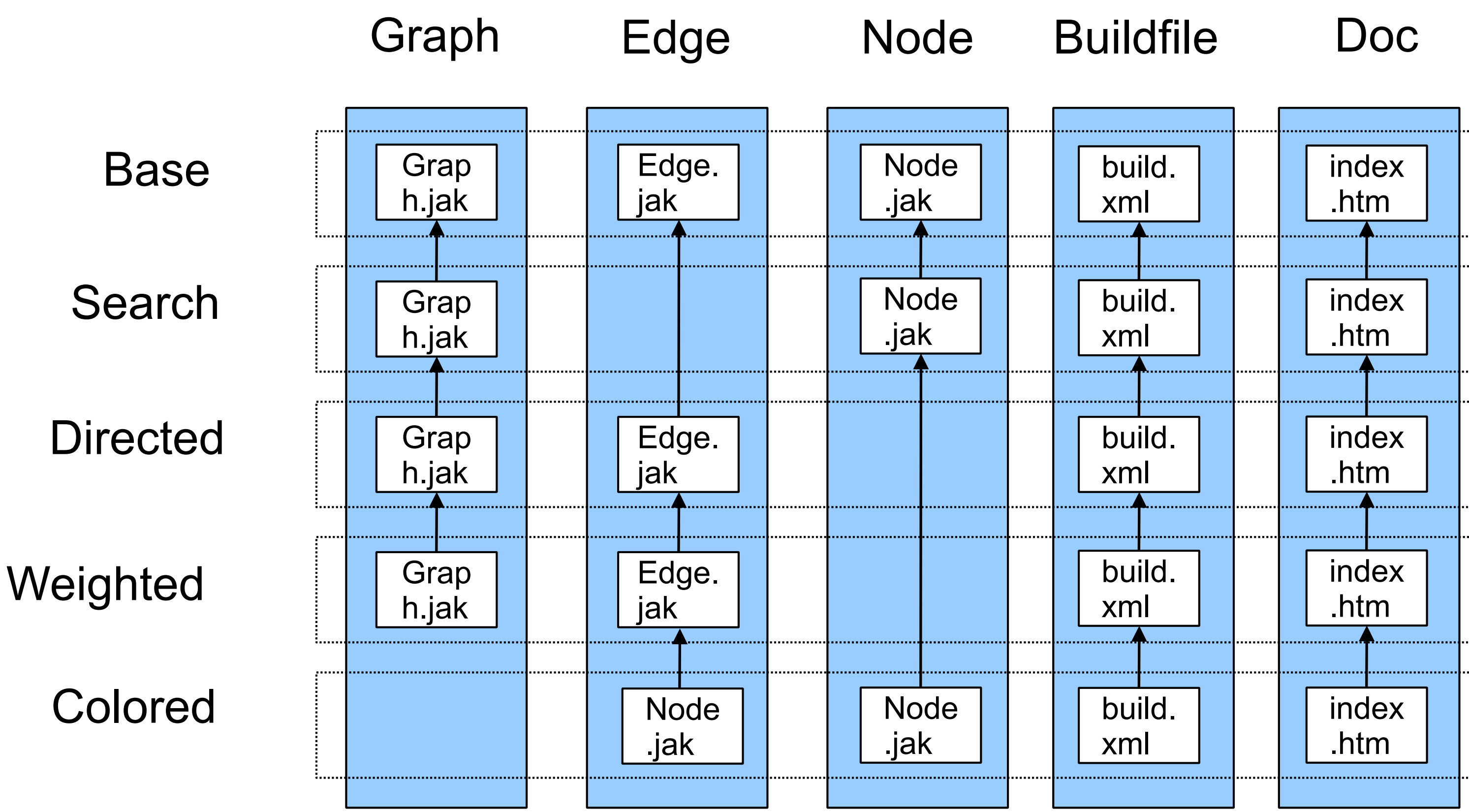
- Not all software is Java code
  - Other programming languages (e.g., C++, Javascript)
  - Build scripts (e.g, Make, XML)
  - Documentation (e.g., XML, HTML, PDF, Text, Word)
  - Grammars (e.g., BNF, ANTLR, JavaCC, Bali)
  - Models (e.g., UML, XMI, ...)
  - ...
- Need to be able to refine all software artifacts
- Integration of different artifacts types in collaborations

Uniformity principle

Features are implemented by a diverse selection of software artifacts and any kind of software artifact can be subject of subsequent refinement.

– Don Batory

Example: uniformity principle



Additional files: grammars, unit-tests, models, specification, and many more

Tool support

- AHEAD – language-independent concept, need customization for each language. Separate tools for:
  - Jak (Java 1.4)
  - Xak (XML)
  - Bali grammar
- FeatureHouse – language-independent tool, easily extensible. Implementations exist for:
  - Java 1.5
  - C#
  - C
  - Haskell
  - JavaCC and Bali grammars
  - UML

Zoom quiz

- How many roles can a program with three classes and four features have
  (a) maximally and (b) minimally?

Model building

An abstract model: why?

- So far focused on specific language constructs
- Model shows common ideas while abstracting away „irrelevant“ details
- Abstracts from details of AHEAD, FeatureHouse or other languages and tools
- Enables discussion about concepts regardless of a specific programming languages
  (→ uniformity principle)

An abstract model: why? II

- Will allow us to define and discuss operations on features (e.g., type checking or interaction analysis) in a formal and language-independent way
- Makes it easier to have reusable implementations for these operations
- Analysis of algebraic properties of feature composition → might support optimizations

Feature composition

- Features can be „composed“ with other features to form more complex features
- Programs are (composed) features, too
- Set F of features; composition operator $\bullet$

$$\bullet : F \times F \rightarrow F$$

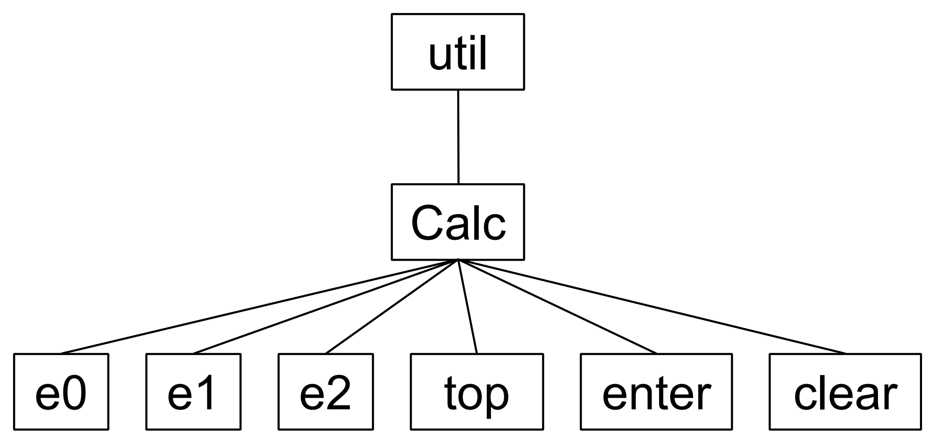
$$p = f_n \bullet f_{n-1} \bullet \ldots \bullet f_2 \bullet f_1$$

(associative, but not commutative)

Modeling features as trees

- A feature consists out of one or several code artefacts, each of them with an internal structure
- Features are modelled as trees (Feature Structure Tree – FST) that reflect the structure of the involved artifacts

```
package util;
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```



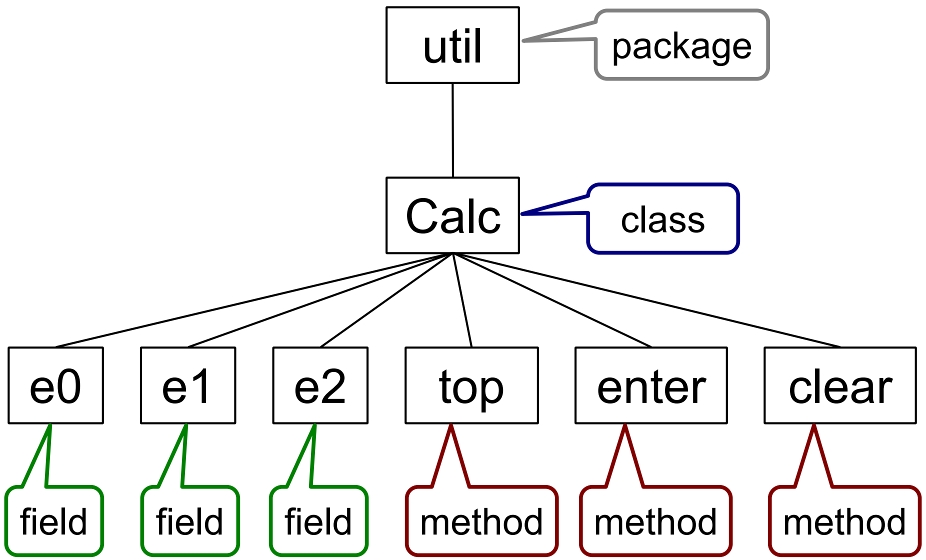
Structure of FSTs

- FST represents the essential structure of each artifact
- Example Java:
  - Packages, Classes, Methods, and Fields
  - Not in FST: Statements, Parameters, Initial Values of Fields
- Other granularity possible;
  choose based on programming language and task

Properties of FSTs

- Nodes in FSTs have a name and a type
- Order of children can matter

```
package util;
class Calc {
  int e0 = 0, e1 = 0, e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0; e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.valueOf(e0);
  }
}
```



Composition via tree superimposition

```
package util;
class Calc {
  void add() {
    e0 = e1 + e0;
    e1 = e2;
  }
}
```

feature: Add

•

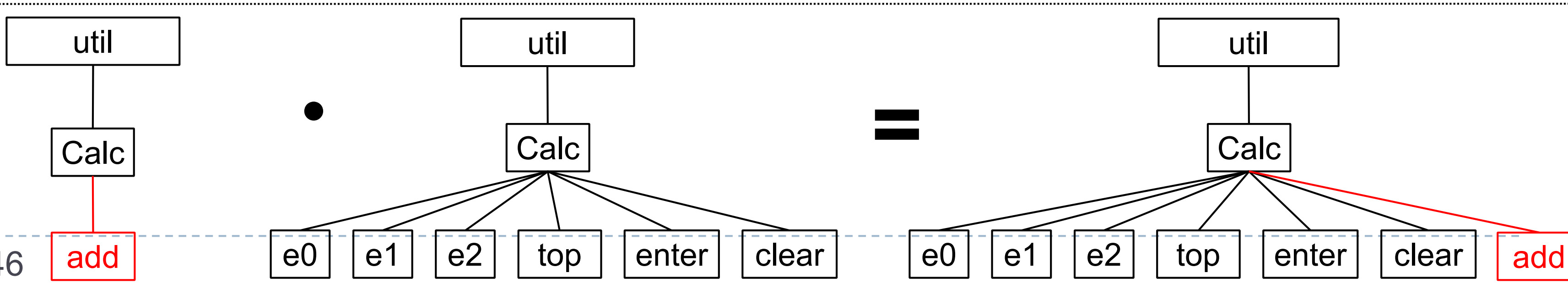
```
package util;
class Calc {
  int e0 = 0, e1 = 0,
      e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0;
    e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    return String.
      valueOf(e0);
  }
}
```

feature: CalcBase

=

```
package util;
class Calc {
  int e0 = 0, e1 = 0,
      e2 = 0;
  void enter(int val) {
    e2 = e1; e1 = e0;
    e0 = val;
  }
  void clear() {
    e0 = e1 = e2 = 0;
  }
  String top() {
    //...
  }
  void add() {
    e0 = e1 + e0;
    e1 = e2;
  }
}
```

feature: CalcAdd



Tree superimposition

- Recursive superimposition of tree‘s nodes, starting with the roots
- Two nodes get superimposed if...
  - ...they have the same node and type and
  - ...their parent nodes have been superimposed
- After the superimposition of two nodes, their children are superimposed where possible
- All nodes (those that *have* and those that have *not* been superimposed) are added to the result tree

Terminal and non-terminal nodes

- Non-terminal nodes
  - Transparent nodes
  - Can have children
  - Name and type but no further content
  - Superimposition generally does not lead to problems
- Terminal nodes
  - Do not have children
  - Name and type
  - Can have additional contents; therefore, superimposition can be nontrivial

Feature composition

- Recursive composition of FST elements
  - package ● package → package (applies to subpackages as well)
  - class ● class → class (applies to inner classes as well)
  - method ● method → ?
  - field ● field → ?

Superimposition of terminal nodes

- Option 1: Two terminal nodes with the same name and type can never be superimposed
- Option 2: Two terminal nodes with the same name and type can be superimposed in well-defined circumstances
  - method ● method → method, if the one method refines the other, for example, by invoking *Super* or *original*
  - field ● field → field, if at least one has no initial value

Composition of terminal nodes

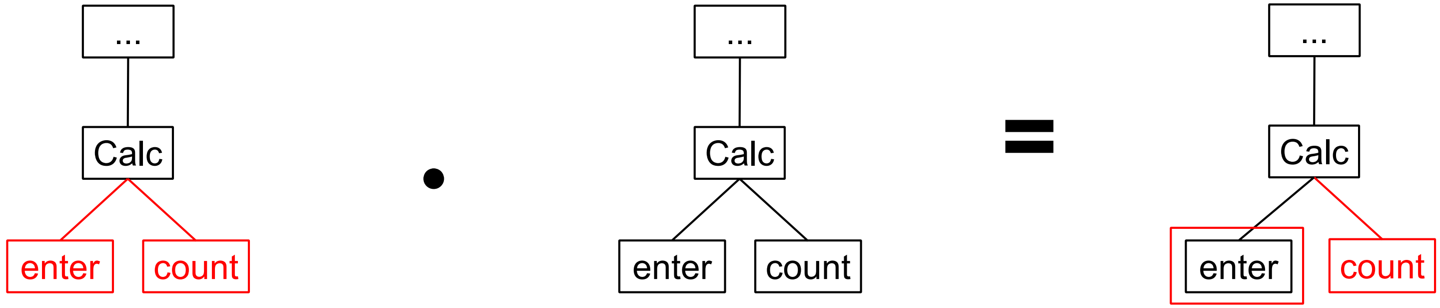
```
class Calc {
  int count = 0;
  void enter(int val) {
    original(val);
    count++;
  }
}
```

•

```
class Calc {
  int count;
  void enter(int val){
    e2 = e1;
    e1 = e0;
    e0 = val;
  }
}
```

=

```
class Calc {
  int count = 0;
  void enter(int val) {
    e2 = e1;
    e1 = e0;
    e0 = val;
    count++;
  }
}
```



Assumptions so far

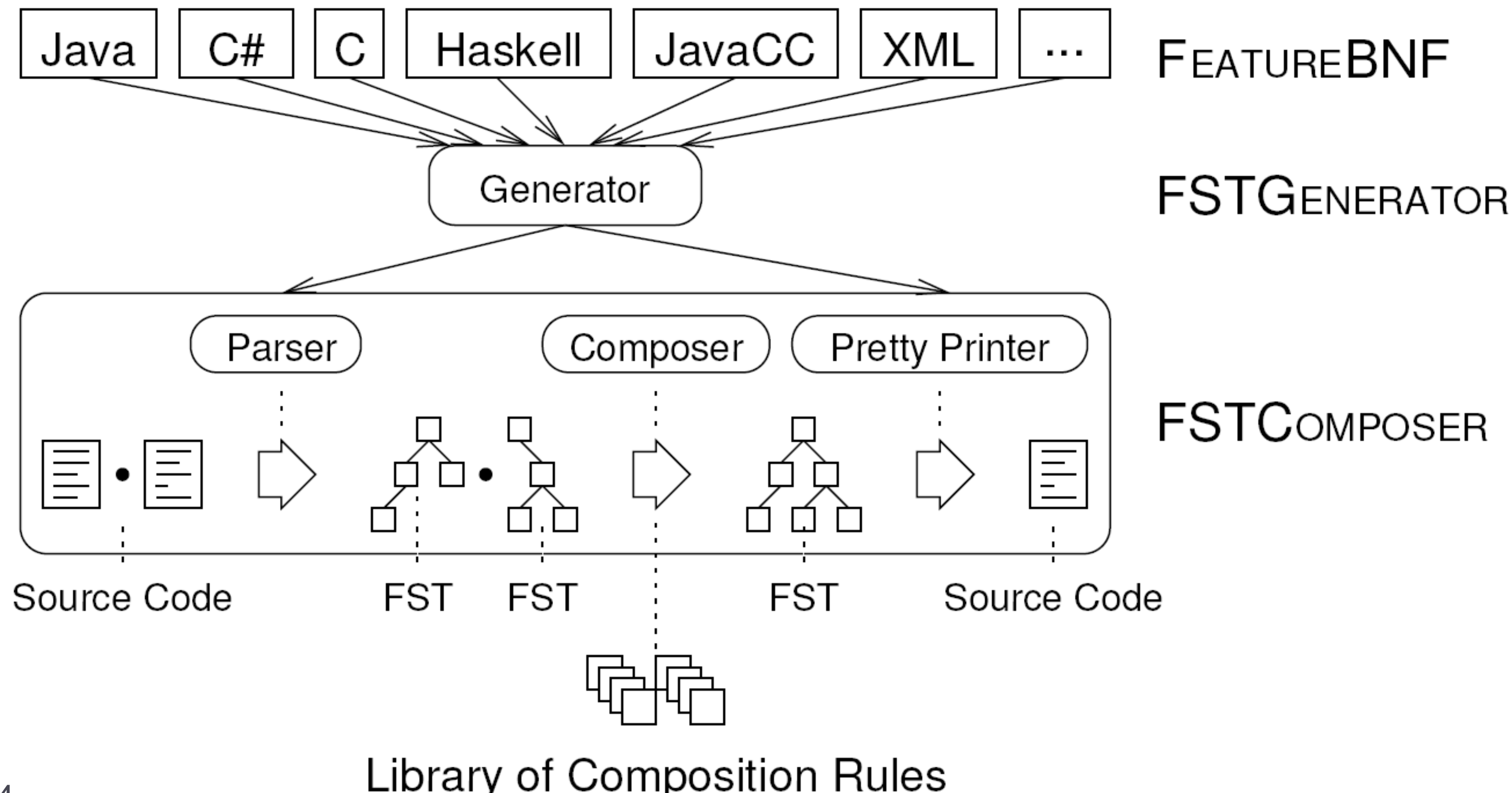
- The structure of a feature is hierarchical (tree)
- Each structure element has a name and a type
- Never two children of the same name and type
- For elements without a hierarchical substructure (terminal nodes), a composition rule is available
  - otherwise composition not possible

Which languages can be modelled with FSTs?

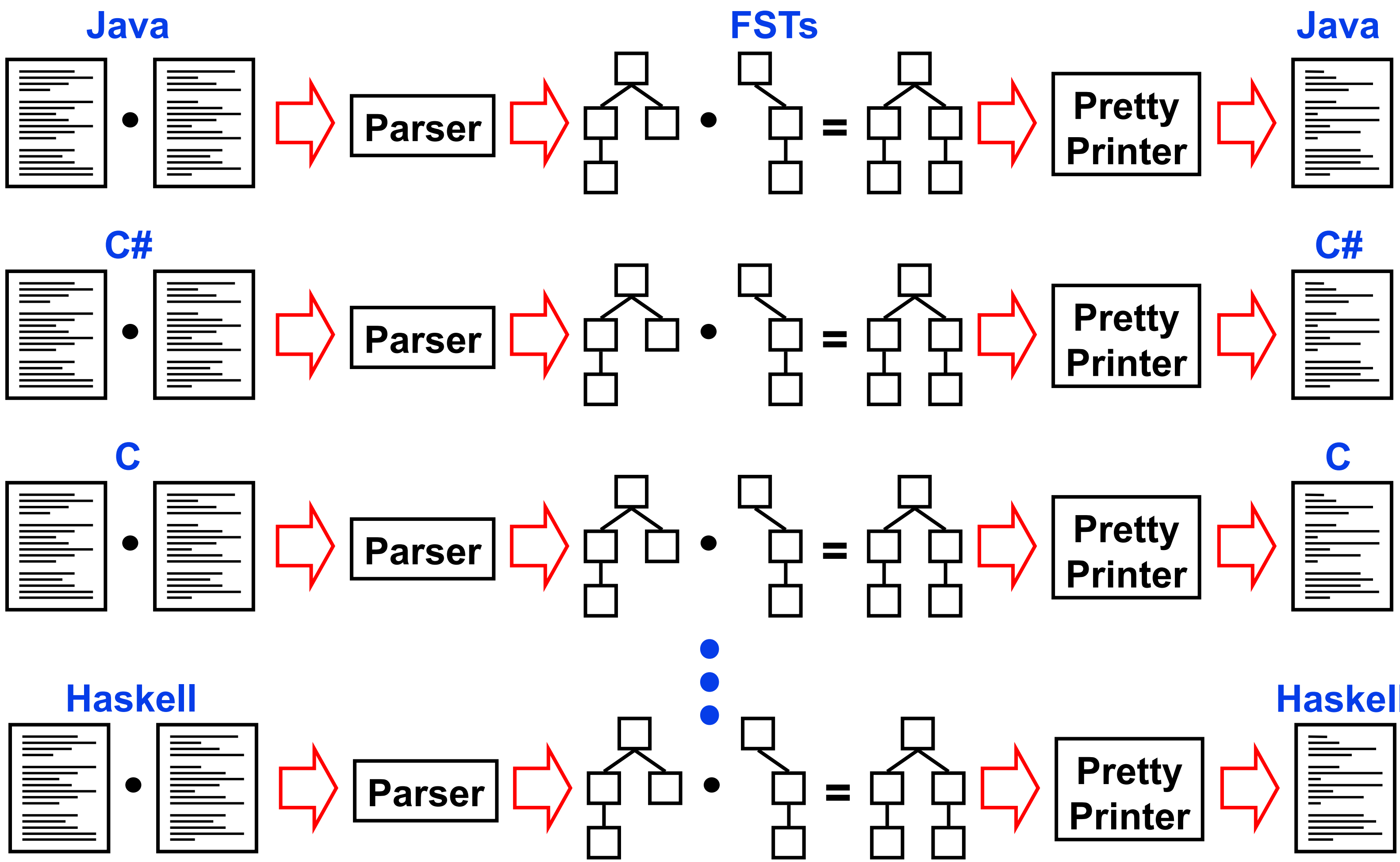
- Object-oriented languages usually satisfy the assumptions
- Some other languages do so as well, e.g., grammars
- Language that do not fulfill the assumptions are considered as not „feature-ready", do not exhibit enough structure
- Some languages can be enriched with additional structure, e.g., XML

FeatureHouse

- FeatureHouse was created based on this formalization



FeatureHouse



Perspectives of model building

- Discussion of language concepts independent of specific language, for example:
  - What would it mean if a feature can participate multiple times in a composition (e.g. X ● Y ● X)?
  - How can we compose structures where the order of children matters (e.g. XML)?
  - Under which conditions does feature composition commute?
  - How can we design a language to be „feature-ready“ (especially. how to define terminal superimposition)?
  - What happens if we want to allow deletion of elements (methods, fields)?

Summary

- Feature-oriented programming solves the feature fraceability problem via collaborations and roles
  - 1:1 mapping
- Implementation based on refinement
- Uniformity principle

Outlook

- Implementing cross-cutting concerns can be quite involved
- Features are not always independent. How to implement dependent collaborations?
- Assessment / distinction

Literature

- D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, 30(6), 2004.
  [The paper that introduced AHEAD]
- S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. IEEE Transactions on Software Engineering, 39(1), 2013.
  [Overview of FSTs and FeatureHouse]