FeatureHouse: Language-Independent, Automated Software Composition

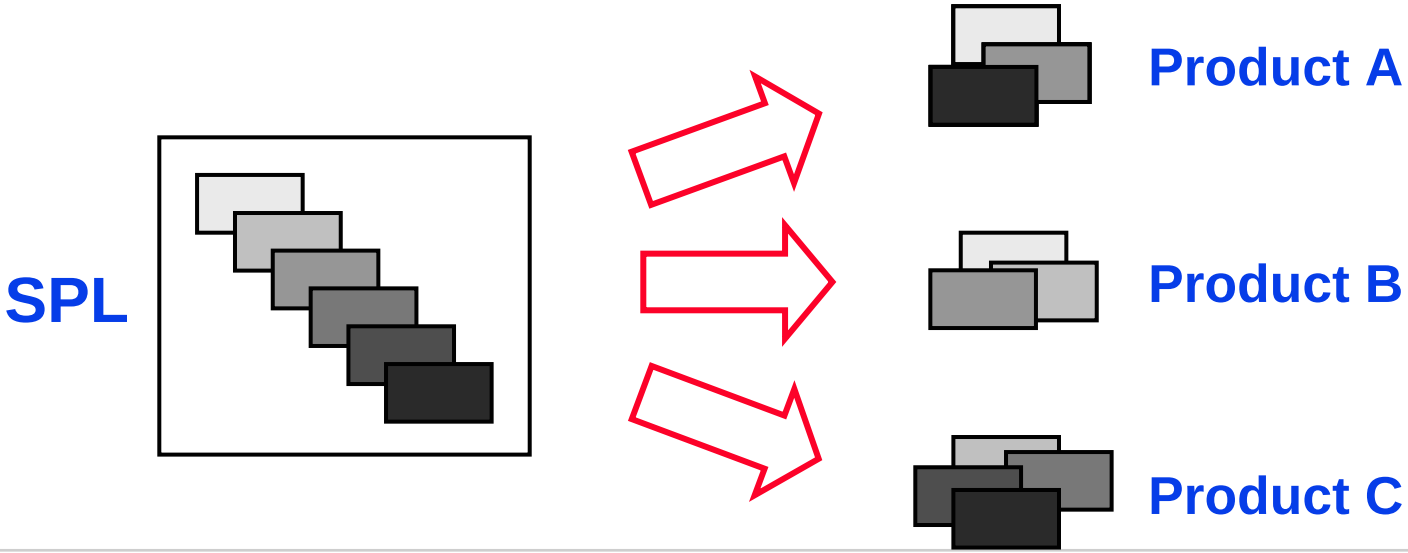
Sven Apel
Department of Informatics and Mathematics
University of Passau

Christian Kästner
School of Computer Science
University of Magdeburg

Christian Lengauer
Department of Informatics and Mathematics
University of Passau

Software Product Lines (SPLs)

- Set of **related software products** for **one domain** generated from common code base
- Products distinguished in terms of **features**
- What is a feature?
  - Product characteristic
  - Domain abstraction relevant to stakeholders

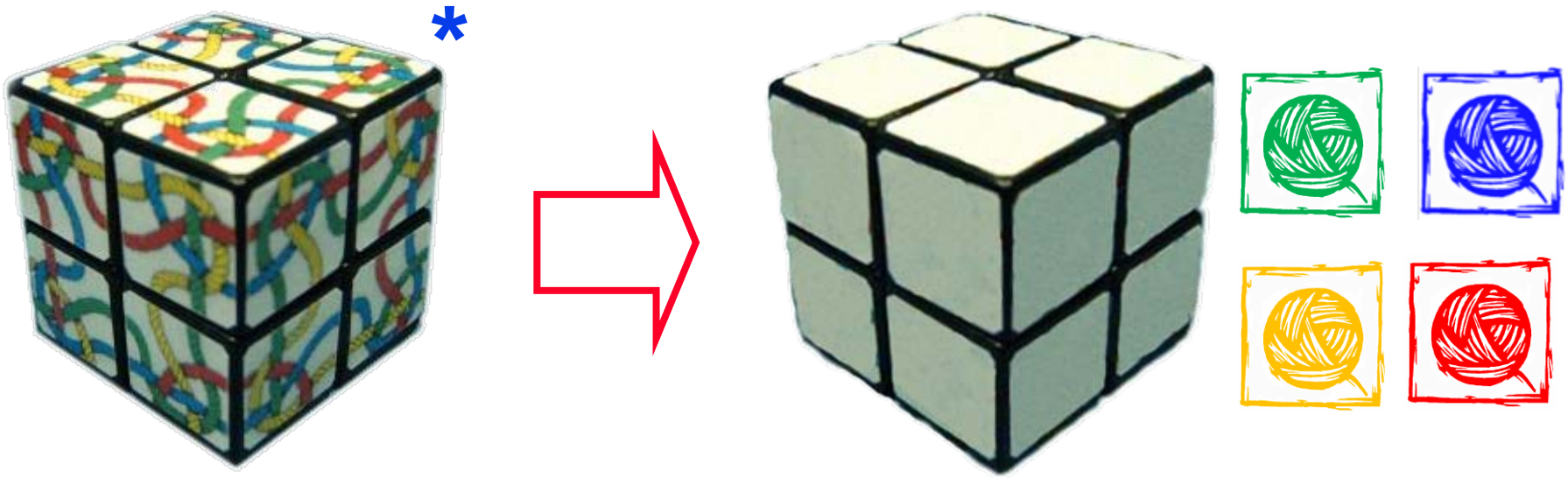


Problem: Lack of Separation of Concerns

*

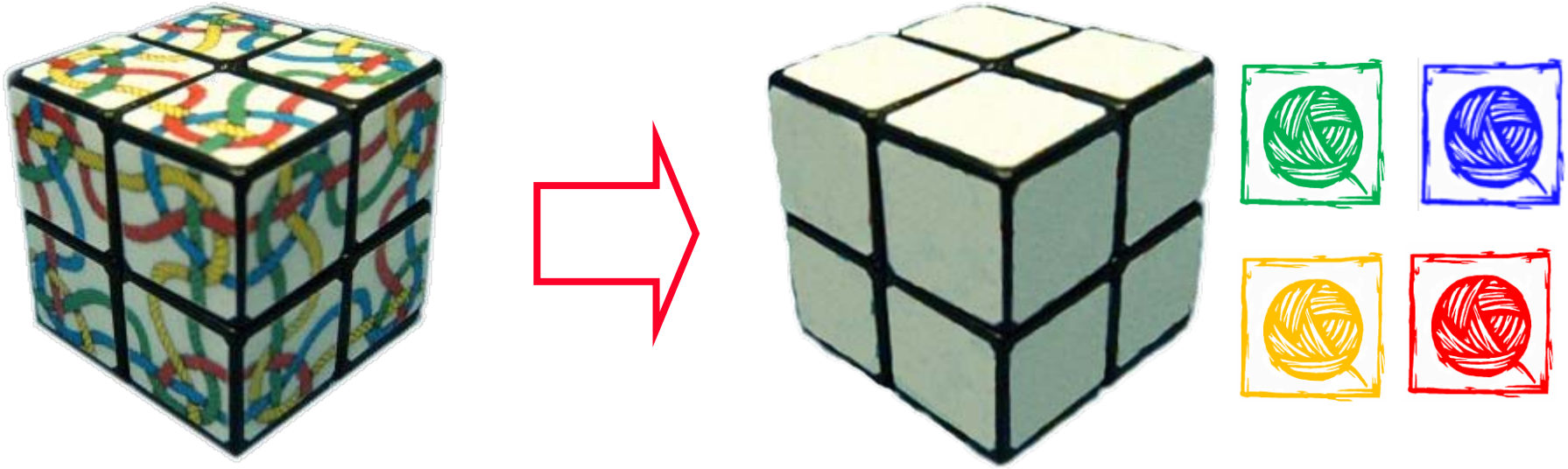
* Graphics taken from K. Hoffman and P. Eugster, ICSE 2008.

Vision: Implement Features Modularly



* Graphics taken from K. Hoffman and P. Eugster, ICSE 2008.

Vision: Implement Features Modularly

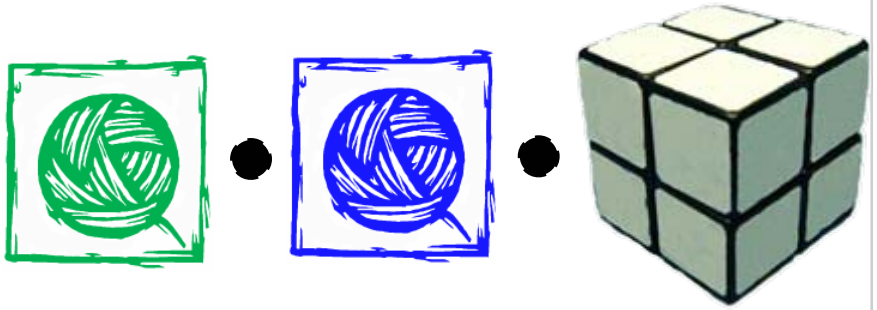


*A feature module is **a structure that extends and modifies the structure of a given program** in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option.*

— Apel et al.

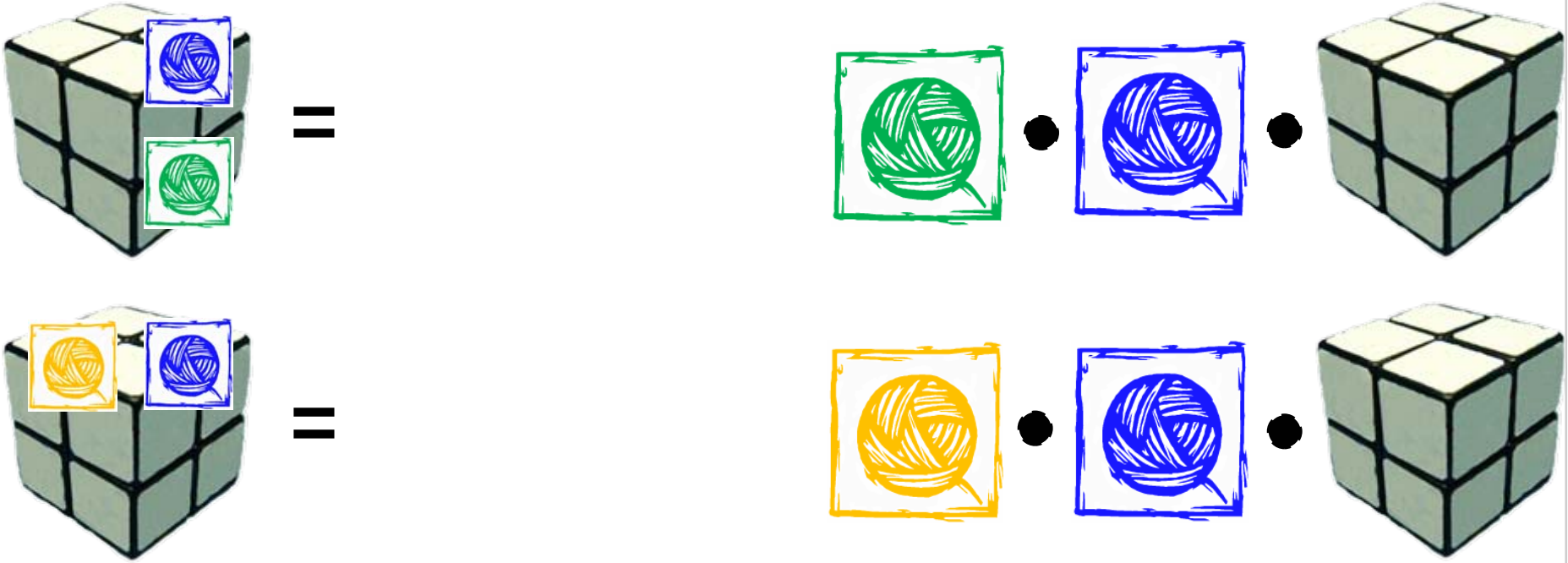
Software Product Generation based on Features

=



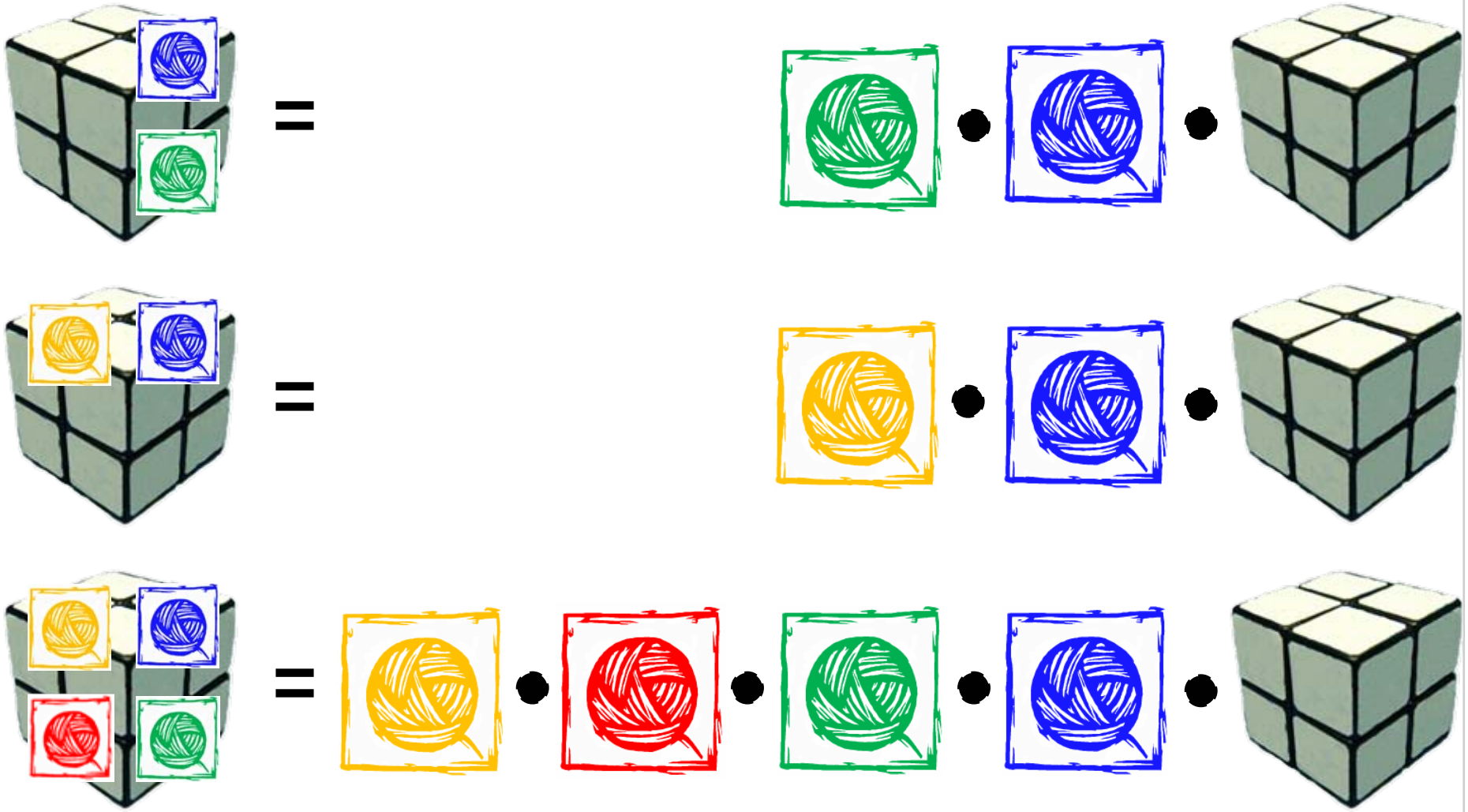
• feature composition

Software Product Generation based on Features



• feature composition

Software Product Generation based on Features



Feature (De)Composition

- Aggregation
- Generation
- Model transformation
- Class and plug-in loading
- Superimposition
- Aspect weaving
- ...

Feature (De)Composition

- Aggregation
- Generation
- Model transformation
- Class and plug-in loading
- Superimposition
- Aspect weaving
- ...

Superimposition

- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.

Superimposition

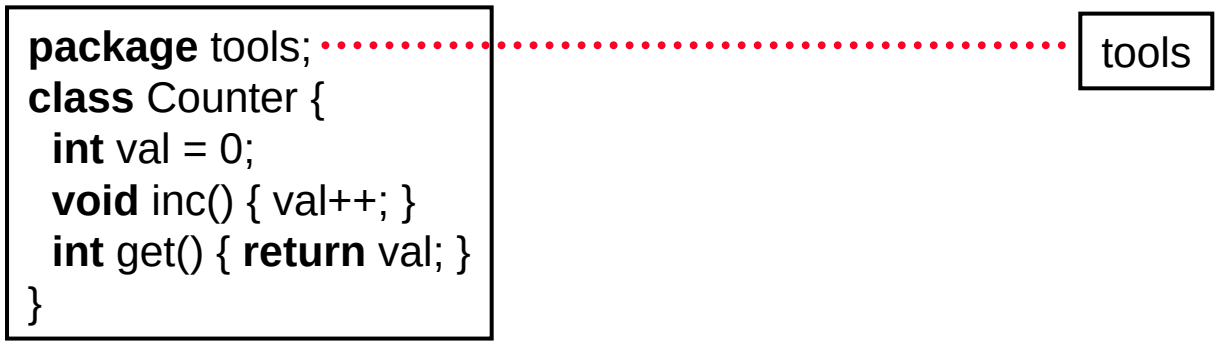
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

software artifact

Superimposition

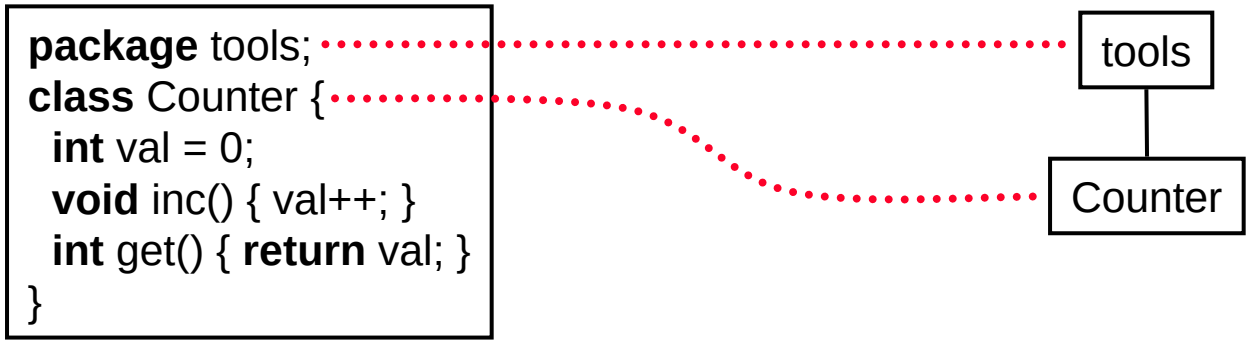
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:



software artifact hierarchical structure

Superimposition

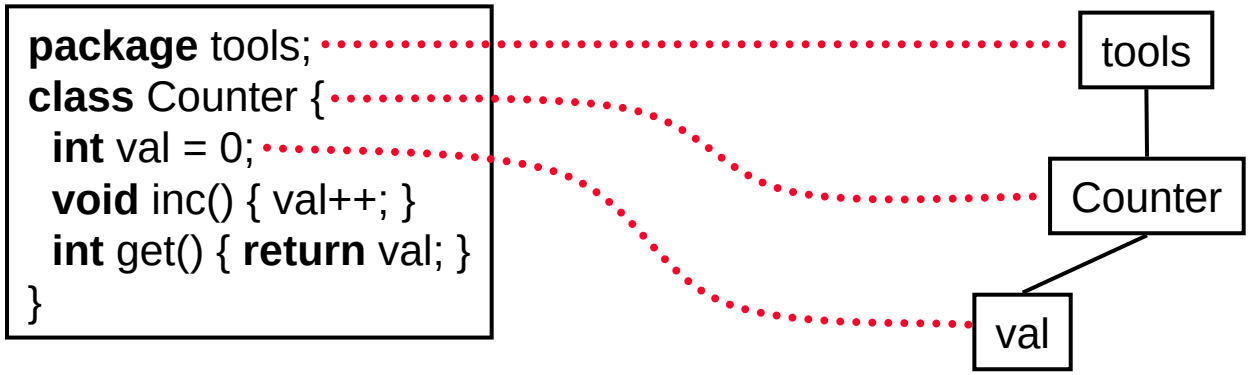
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:



software artifact hierarchical structure

Superimposition

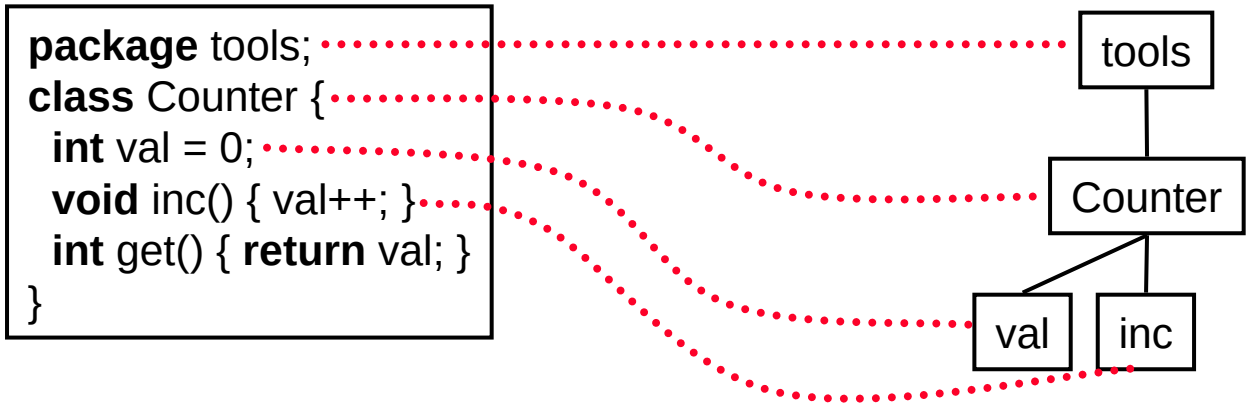
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:



software artifact hierarchical structure

Superimposition

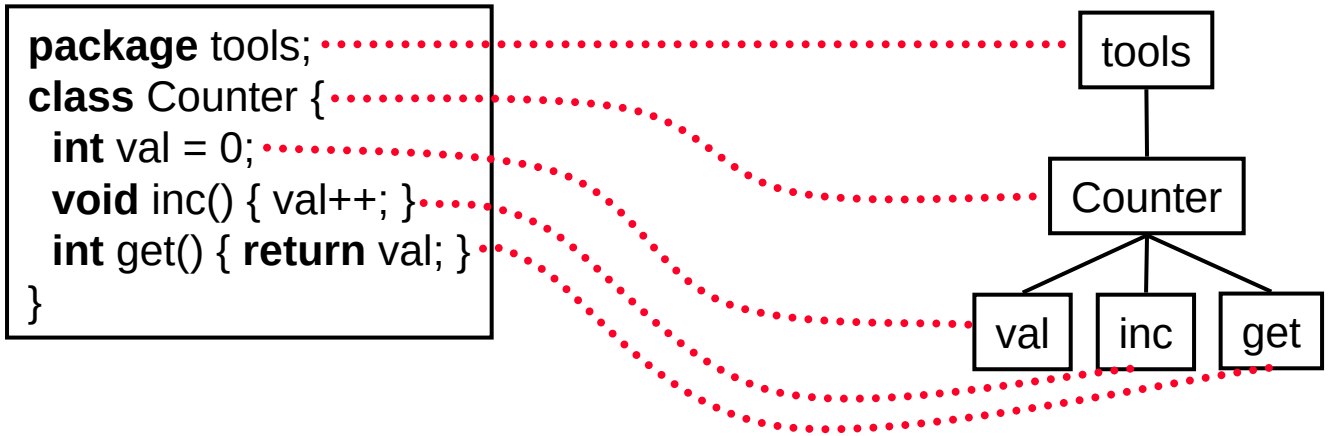
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:



software artifact hierarchical structure

Superimposition

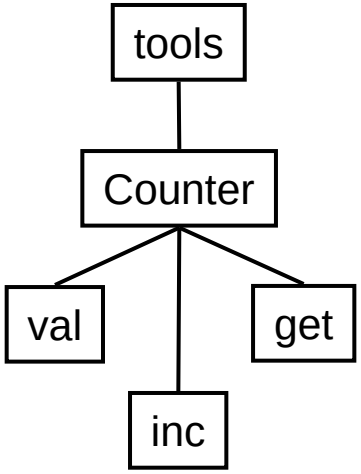
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:



Superimposition

Counter

```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```



Superimposition

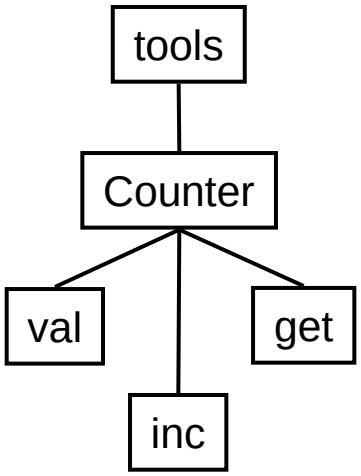
Counter

```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

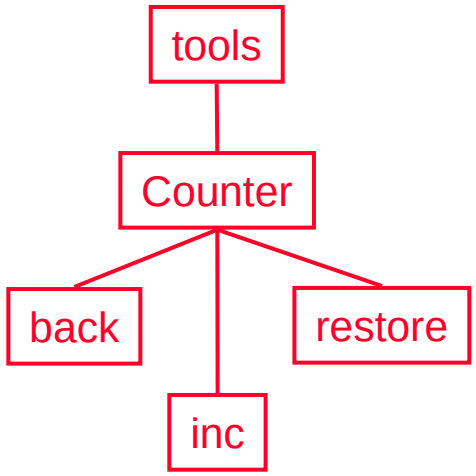
•

Backup

```
package tools;
class Counter {
  int back = 0;
  void inc() { back=val; original(); }
  void restore() { val=back; }
}
```



•



Superimposition

Counter

```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

•

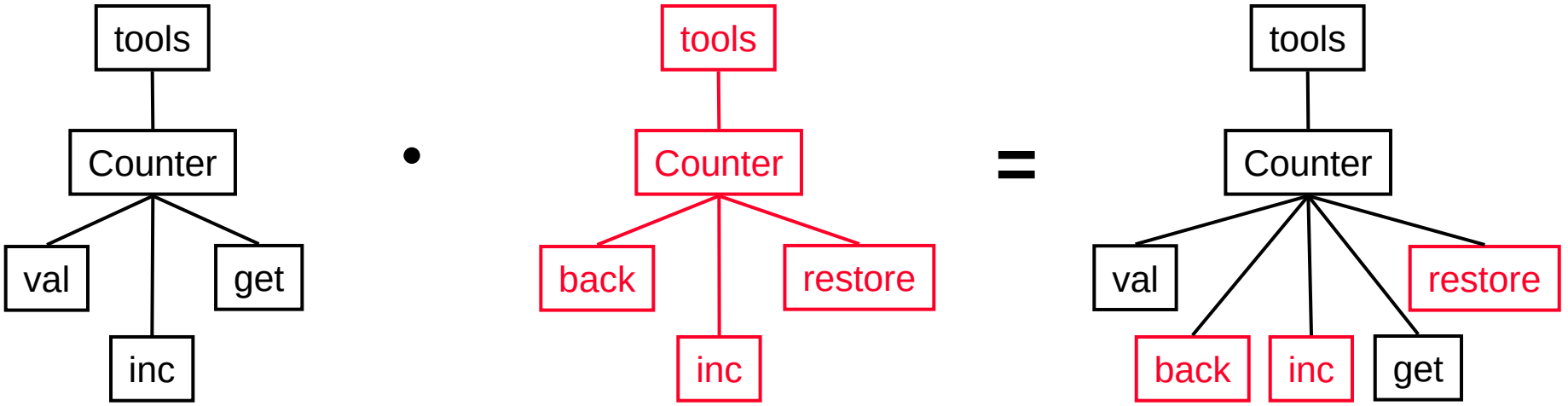
Backup

```
package tools;
class Counter {
  int back = 0;
  void inc() { back=val; original(); }
  void restore() { val=back; }
}
```

=

BackupCounter

```
package tools;
class Counter {
  int val = 0;
  int back = 0;
  void inc() { back=val; val++; }
  int get() { return val; }
  void restore() { val=back; }
}
```

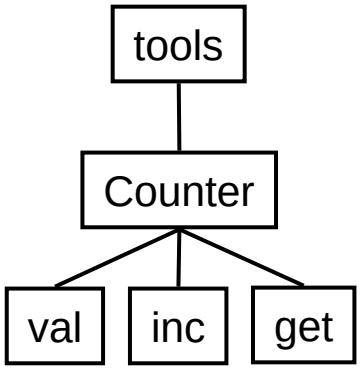


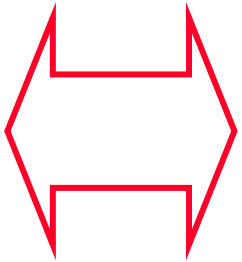
Languages, Tools, and Formal Systems

- Languages
  - Jak, Scala, CaesarJ, FeatureC++, Java Layers, Classbox/J, ObjectTeams/J, Lasagne/J
- Tools
  - Hyper/J, AHEAD Tool Suite, Jiazzi, Xak
- Formal Systems
  - Jx, J&, vc, vObj, Tribe, .FJ, FFJ, FLJ, Deep, gDeep

An Idea

- Capture the essential properties of superimposition in a model and composition tool
  - Language independence
  - General theory of software composition by superimposition
  - Integrate a language of your choice
- **Feature Structure Tree** Model



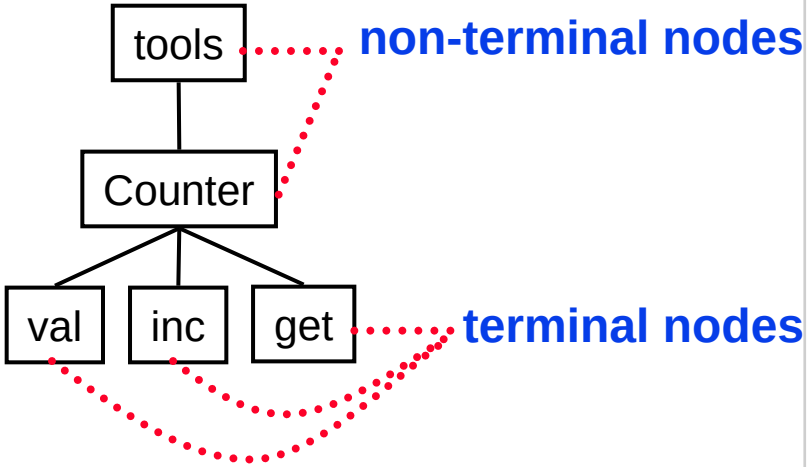


$$
\begin{aligned}
Base \;=\; & tools : Package \\
& \oplus\, tools.Counter : Class \\
& \oplus\, tools.Counter.val : Field \\
& \oplus\, tools.Counter.inc : Method \\
& \oplus\, tools.Counter.get : Method
\end{aligned}
$$

Feature Algebra (Apel et al., AMAST'08)

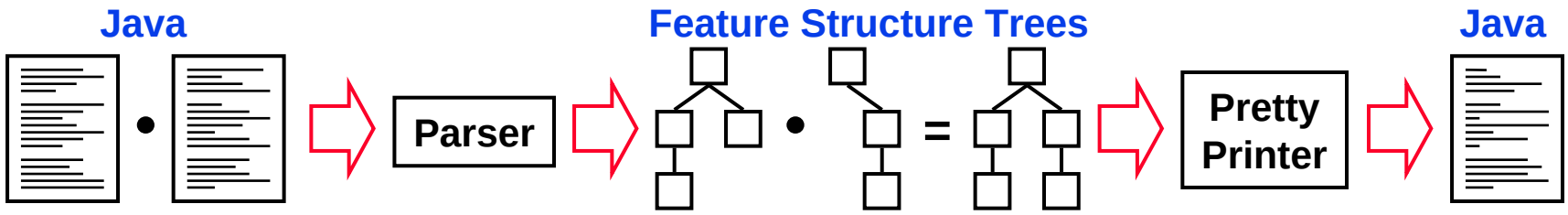
Non-Terminal vs. Terminal Nodes

- Non-terminal nodes
  - Identified by name* and type
  - Superimposition proceeds recursively with the children
- Terminal nodes
  - Identified by name* and type
  - Carry further language-specific content
  - Superimposition terminates with composing contents

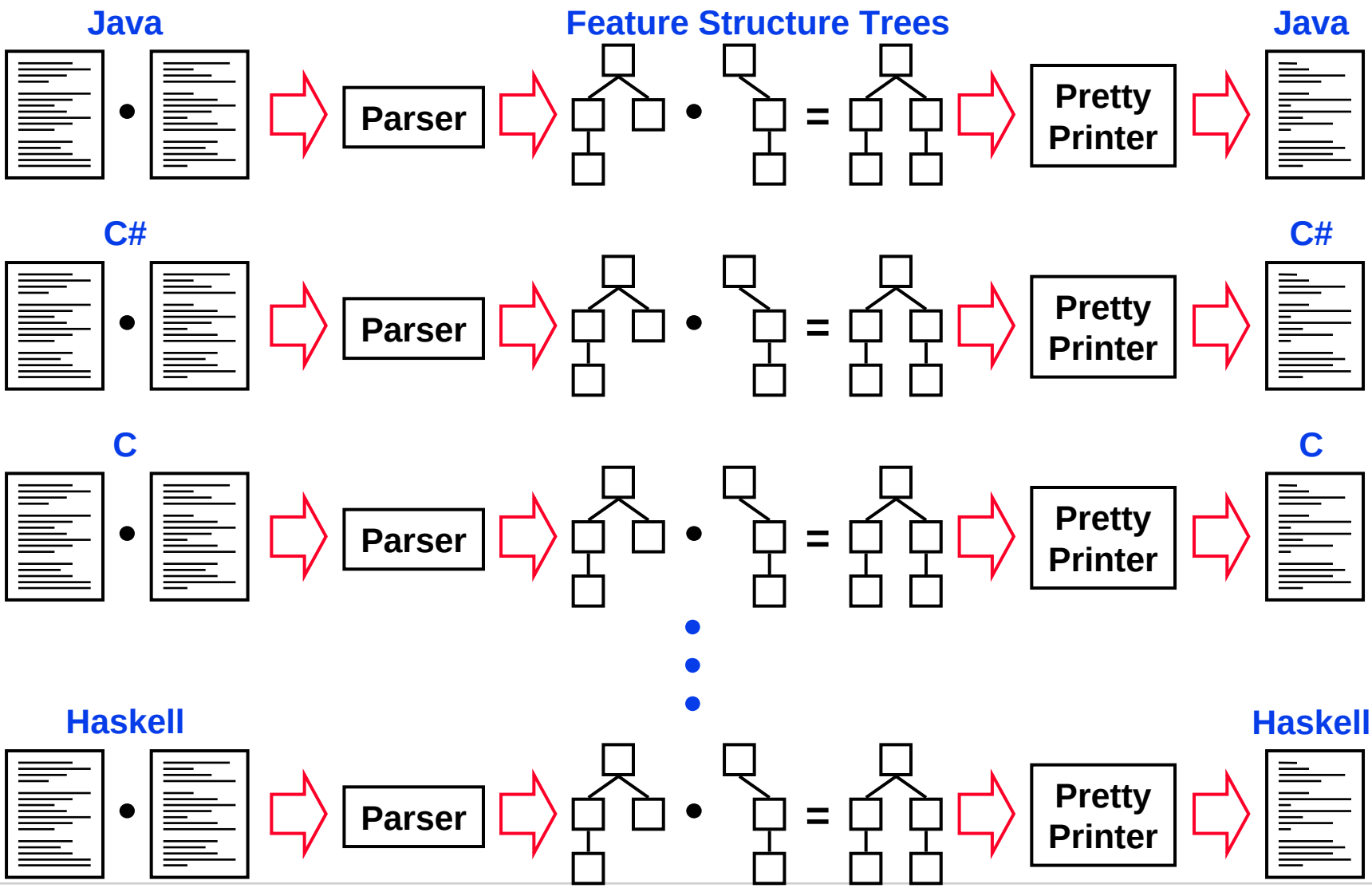


* Names are mangled.

FSTComposer



FSTComposer



Problems

- Manual integration of Java, C#, UML, and Bali
  - Implementation of a parser and a pretty printer per language
    - Tedious (several weeks effort)
    - Error-prone (many bugs)
  - Composition of terminal nodes requires special language-dependent rules
    - $method \times method \rightarrow method$ (overriding)
    - $constructor \times constructor \rightarrow constructor$ (concatenation)
    - $implements \times implements \rightarrow implements$ (union)
    - $extends \times extends \rightarrow extends$ (replacement)
    - ...

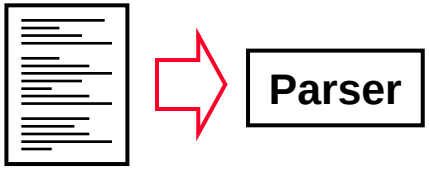
➔ Benefit of language independence is almost lost

An Observation

- Code for supporting different languages is very similar

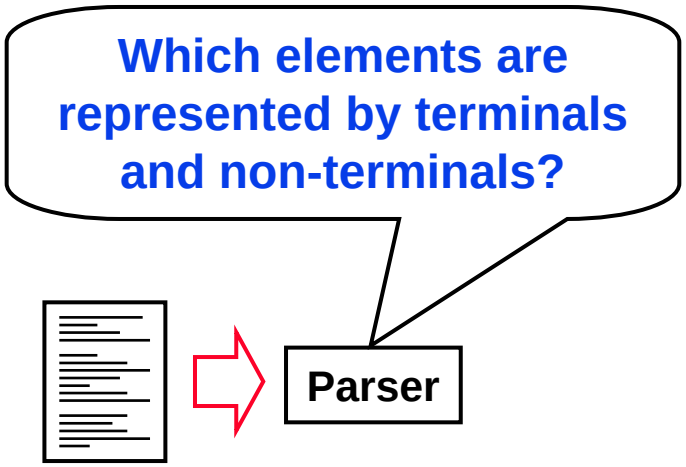
An Observation

- Code for supporting different languages is very similar



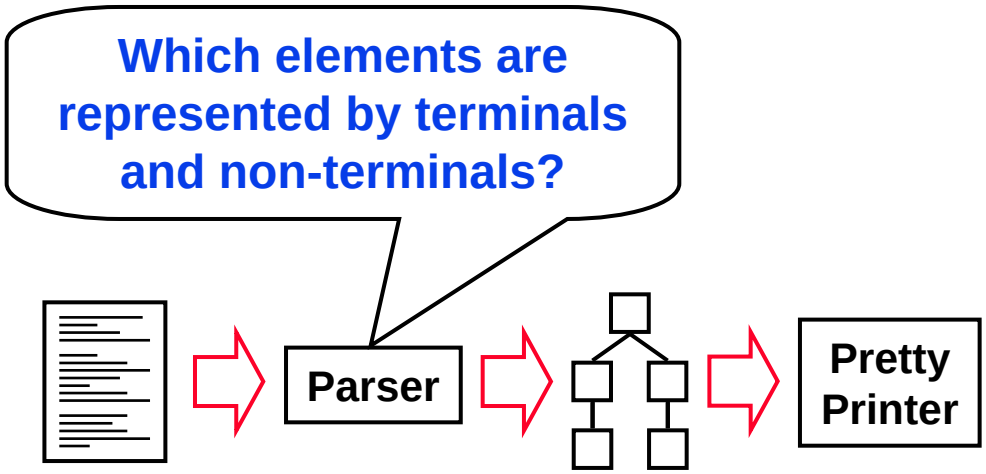
An Observation

- Code for supporting different languages is very similar



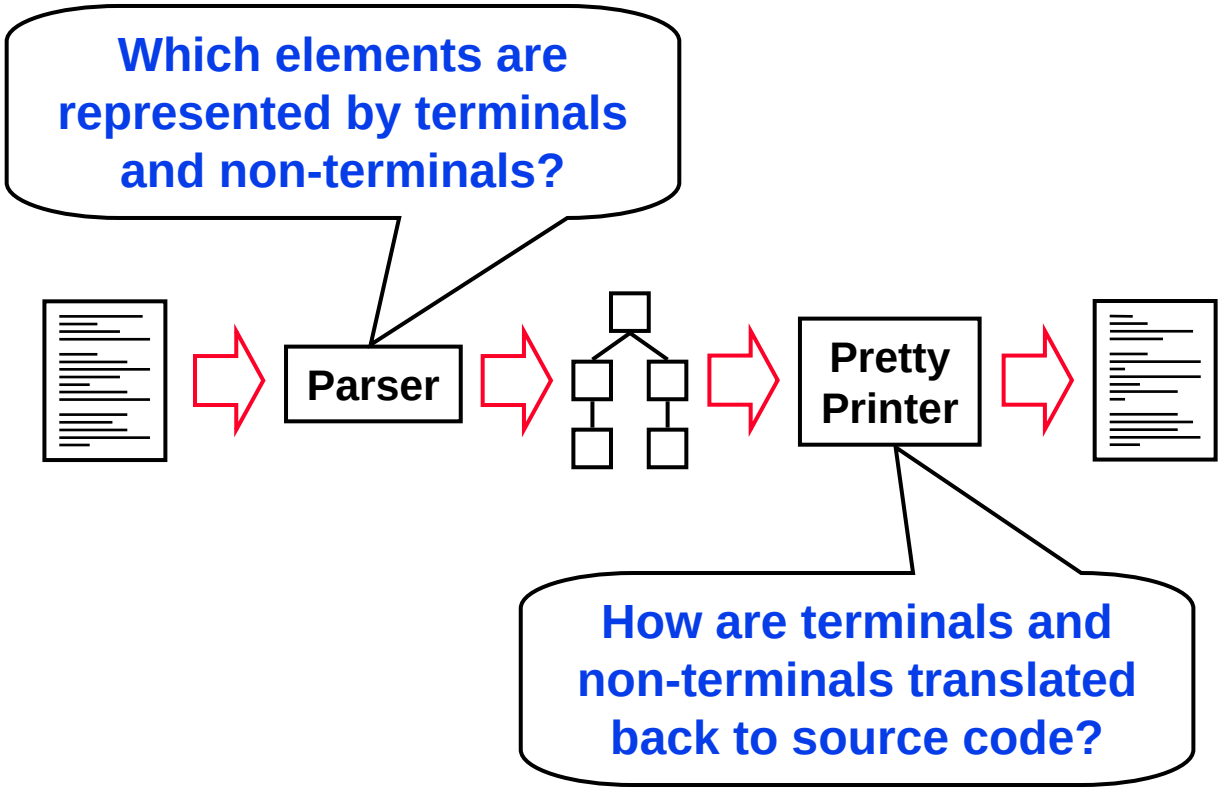
An Observation

- Code for supporting different languages is very similar



An Observation

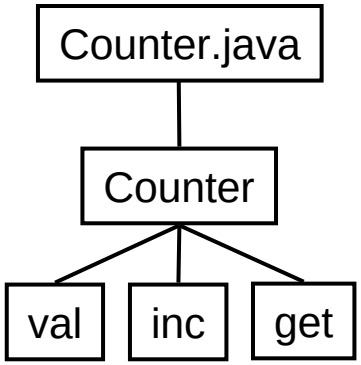
- Code for supporting different languages is very similar



An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

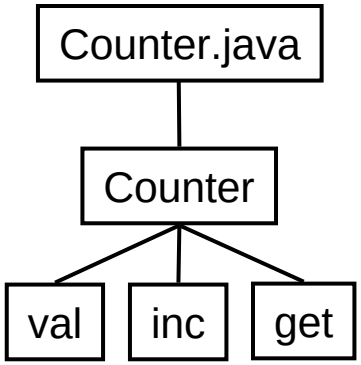
FST



An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar

FST

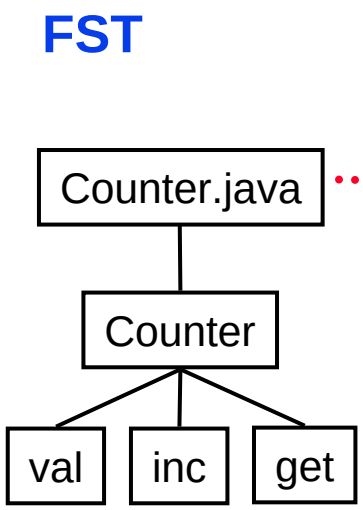


Grammar

```
JavaFile : (ClassDecl)*;

ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";

MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";
VarDecl : Type <IDENTIFIER> ";";
```

An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar



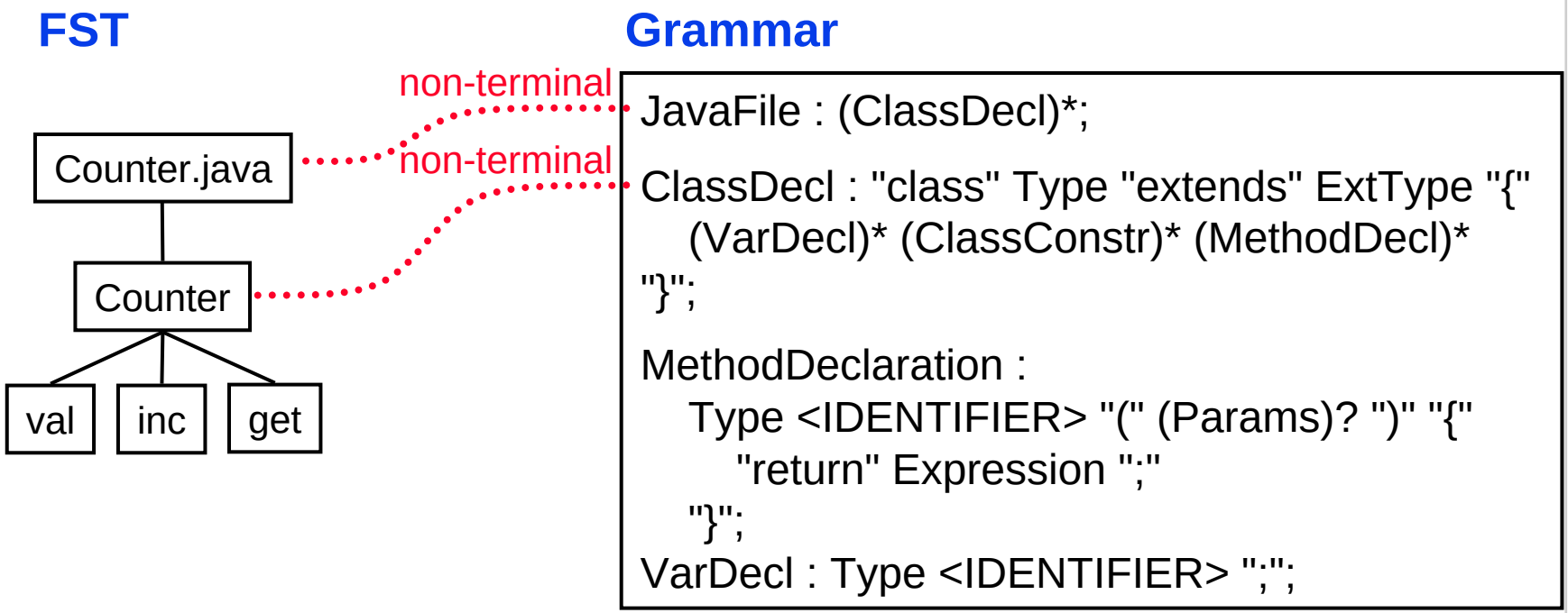
non-terminal

Grammar

```
JavaFile : (ClassDecl)*;

ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";

MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";
VarDecl : Type <IDENTIFIER> ";";
```

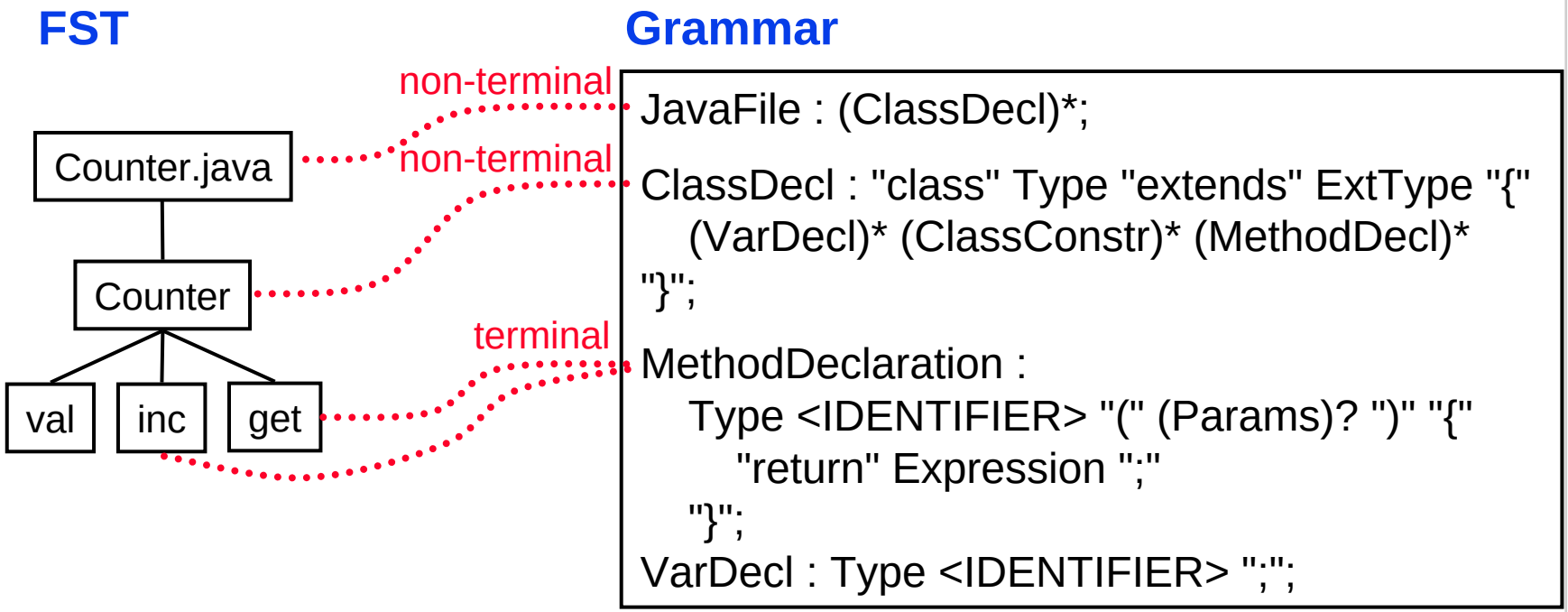
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar



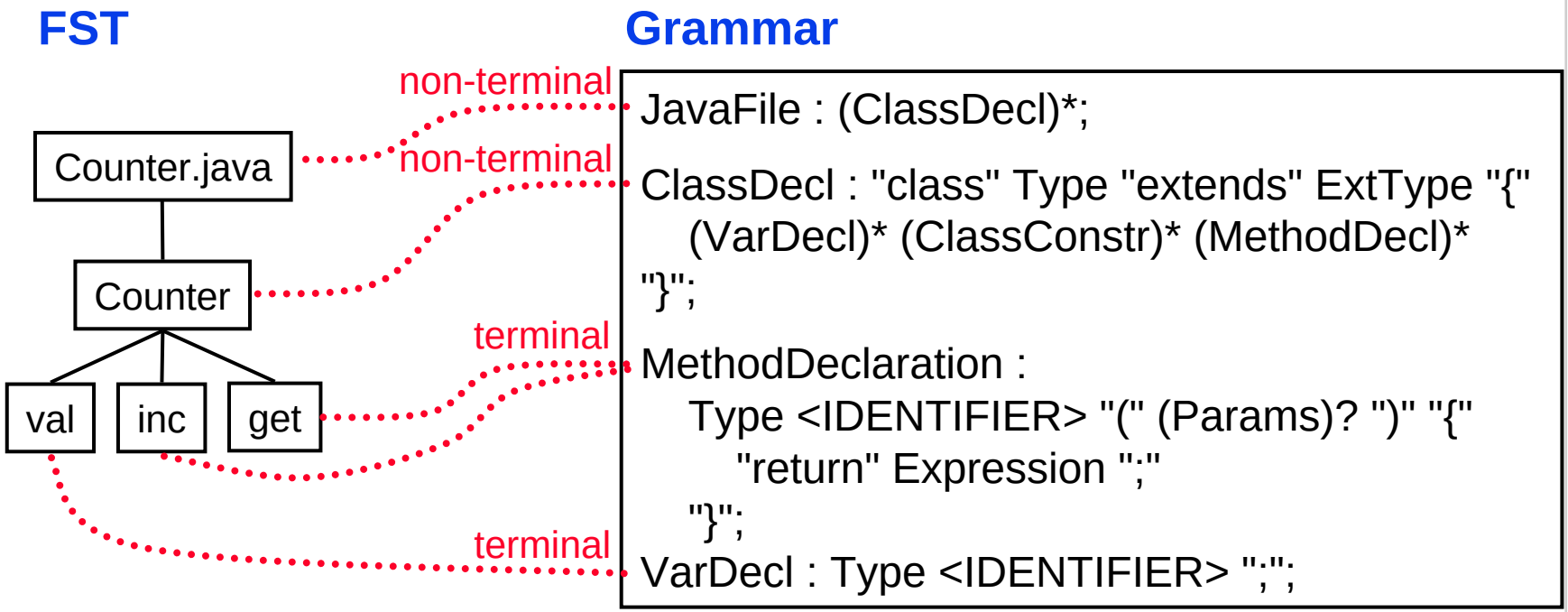
An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar



An Insight

- Non-terminals and terminals correspond to production rules in the artifact language's grammar



An Idea

- Automate the integration of a new language on the basis of the language's grammar
- Use **annotations/attributes** to define…
  - which production rules map to non-terminals,
  - which production rules map to terminals, and
  - how the contents of terminals of a certain type are composed
- Generate a parser and pretty printer automatically on the basis of an **annotated grammar**

Annotating a Simplified Java Grammar

Grammar

```
JavaFile : (ClassDecl)*;


ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";



MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";



VarDecl : Type <IDENTIFIER> ";";
```

Annotating a Simplified Java Grammar

Grammar

```
JavaFile : (ClassDecl)*;

ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";

MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";

VarDecl : Type <IDENTIFIER> ";";
```

Example

```
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

Counter.java

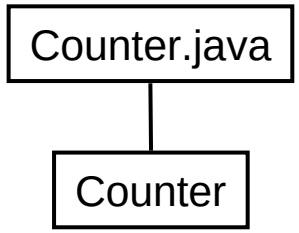
Annotating a Simplified Java Grammar

Grammar

```
@FSTNonTerminal()
JavaFile : (ClassDecl)*;


ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";



MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";



VarDecl : Type <IDENTIFIER> ";";
```

Example

```
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```



Annotating a Simplified Java Grammar

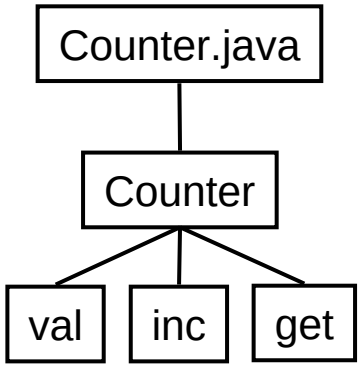
Grammar

```
@FSTNonTerminal()
JavaFile : (ClassDecl)*;

@FSTNonTerminal(name="{Type}")
ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";



MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";



VarDecl : Type <IDENTIFIER> ";";
```

Example

```
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

Generated Parser



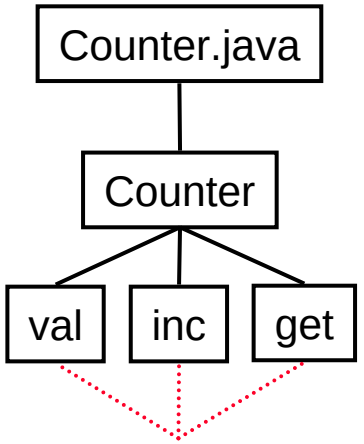
Annotating a Simplified Java Grammar

Grammar

```
@FSTNonTerminal()
JavaFile : (ClassDecl)*;

@FSTNonTerminal(name="{Type}")
ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";

@FSTTerminal(name="{<IDENTIFIER>}({Params})",
    compose="MethodOverriding")
MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";

@FSTTerminal(name="{<IDENTIFIER>}",
    compose="FieldSpecialization")
VarDecl : Type <IDENTIFIER> ";";
```

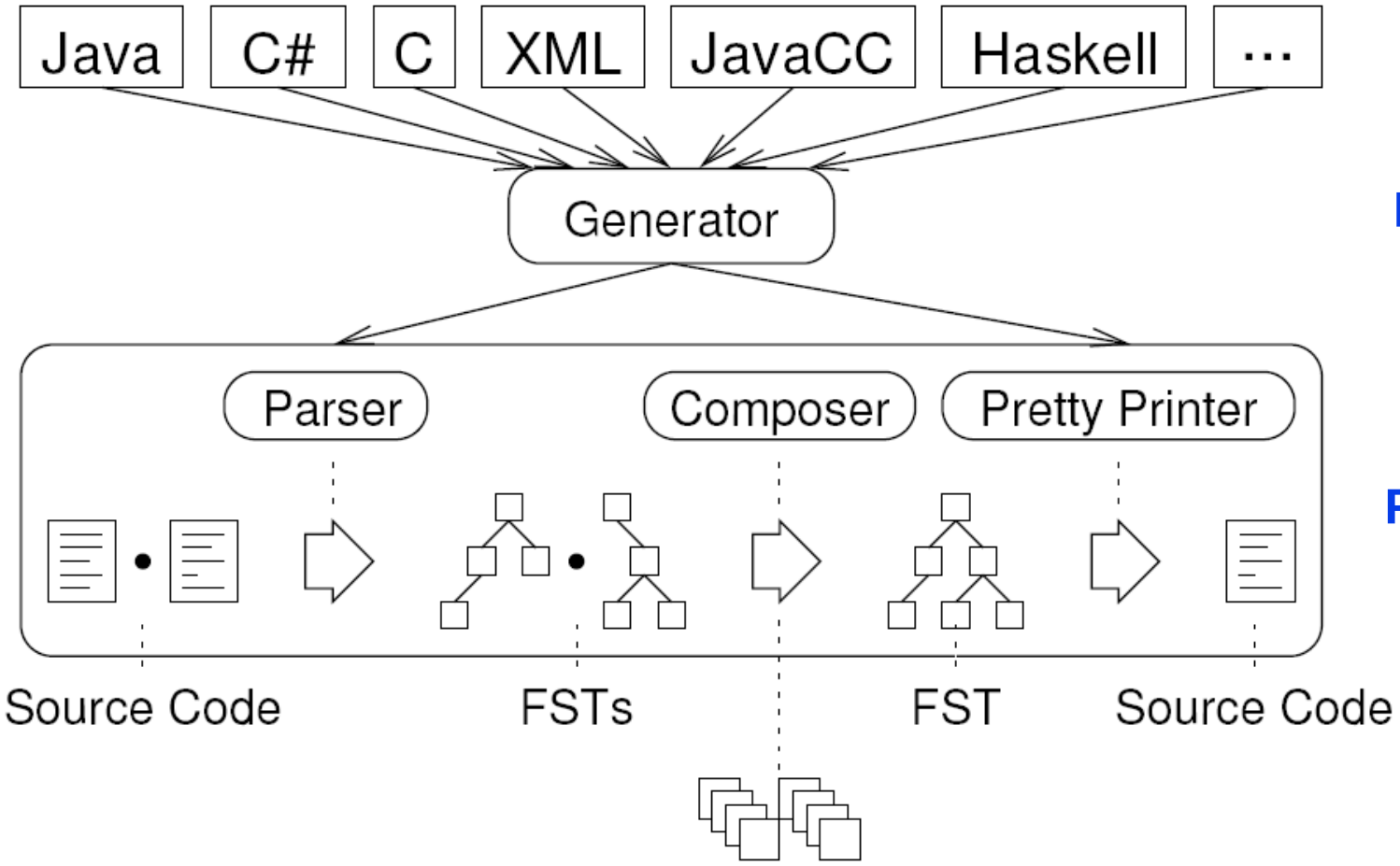
Example

```
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```



How to compose?

FeatureHouse



FeatureBNF

FSTGenerator

FSTComposer

library of composition rules

Integrating Languages

- Moderate effort for annotating grammars
- Only a few composition rules ➔ library and reuse

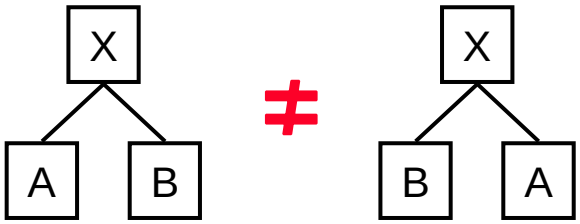
	Java	C#	C	Haskell	JavaCC	XML
# rules	135	229	45	78	170	14
# non-terminals	10	17	2	13	16	6
# terminals	13	18	9	9	16	6
# attributes	42	53	21	24	61	15

Case Studies

	Features	LOC	Artifact Types	Description
FFJ	2	289	JavaCC	Grammar of the FFJ language
Arith	27	532	Haskell	Arithmetic expression evaluator
GraphLib	13	934	C	Low level graph library
Phone	2	1.004	UML	Phone system
ACS	4	2.080	UML	Audio control system
CMS	10	2.037	UML	Conference management system
GPL (C#)	20	2.148	C#	Graph product line (C# version)
GBS	29	2.380	UML	Gas boiler control system (IKERLAN)
GPL (Java)	26	2.439	Java, XHTML	Graph product line (Java version)
FGL	20	2.730	Haskell	Functional graph library
Violet	88	9.660	Java, Text	Visual UML editor
GUIDSL	26	13.457	Java	Product line configuration tool
Berkeley DB	99	84.030	Java	Oracle's embedded DBMS

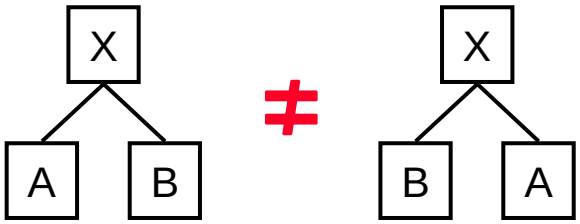
Problems

- Lexical order:

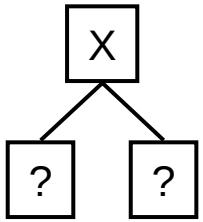


Problems

- Lexical order:

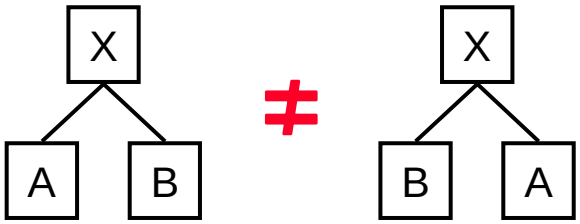


- Unnamed elements:

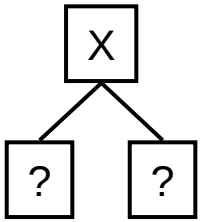


Problems

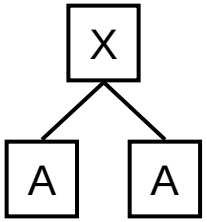
- Lexical order:



- Unnamed elements:



- Ambiguous names:



Conclusion

- **Superimposition** is a general mechanism to compose software artifacts (features)
- **Language independent** model captures the essence of superimposition (formal model: feature algebra)
- **FeatureHouse**: Languages can be integrated almost **automatically**
  - A wide variety of very different languages are integrated
- **Substantial cases studies** demonstrate practicality and reveal open issues
- Problems with lexical order, unnamed elements, and ambiguous names

Questions?

- Sven Apel, Christian Lengauer, Bernhard Möller, and Christian Kästner. **An Algebra for Features and Feature Composition**. In *Proc. Int. Conf. Algebraic Methodology and Software Technology (AMAST)*. July 2008.
- Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. **Model Superimposition in Software Product Lines**. In *Proc. Int. Conf. Model Transformation (ICMT)*. July 2009.
- Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. **Feature (De)composition in Functional Programming**. In *Proc. Int. Conf. Software Composition (SC)*. July 2009.

Feature Algebra

UML

Haskell