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

Background (i)

- Software product line engineering

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

— SEI, CMU

Background (ii)

- Feature-oriented software development

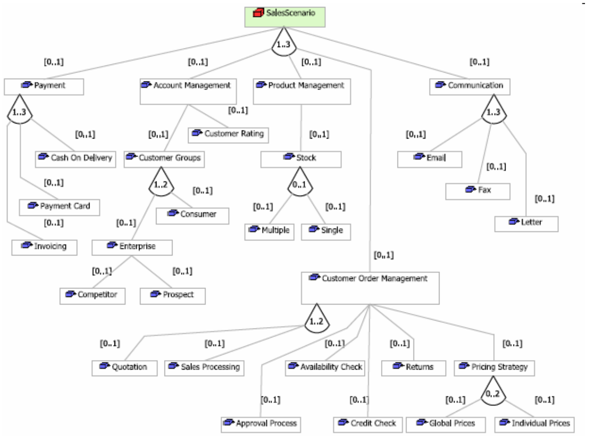
[A feature is] a distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.

— Czarnecki and Eisenecker

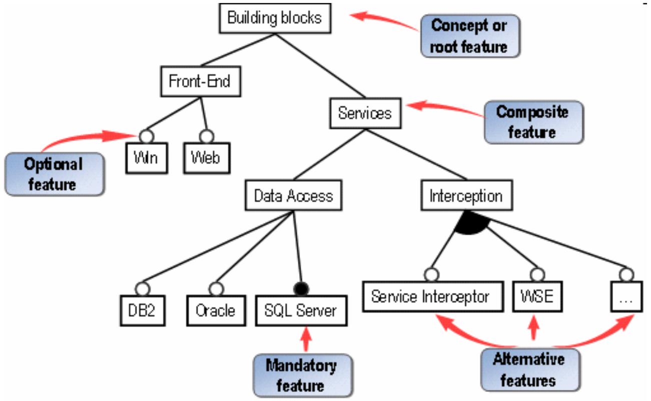
A feature is a product characteristic that is used in distinguishing programs within a family of related programs.

— Don Batory

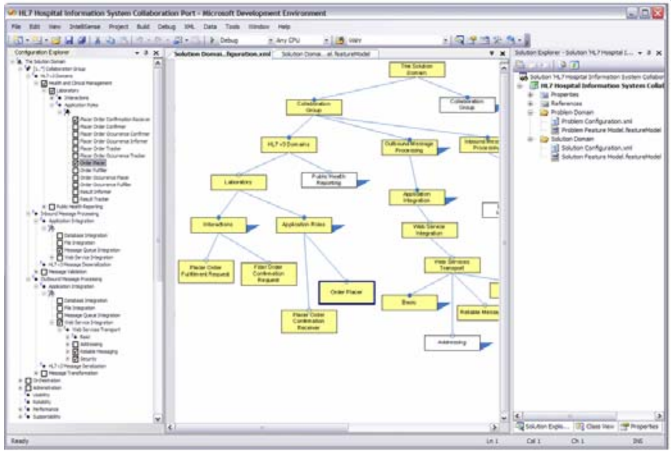
The Good…



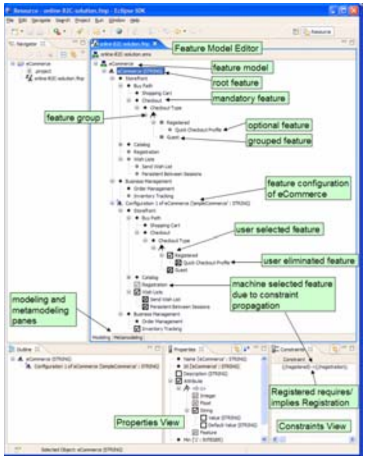
Sales Management (feasiPLe Project)



Enterprise Administrative Application (Microsoft)


HL7 Hospital Information System Collaboration Port - Microsoft Development Environment
Configuration Explorer
Solution Explorer

Hospital Information System (Microsoft)

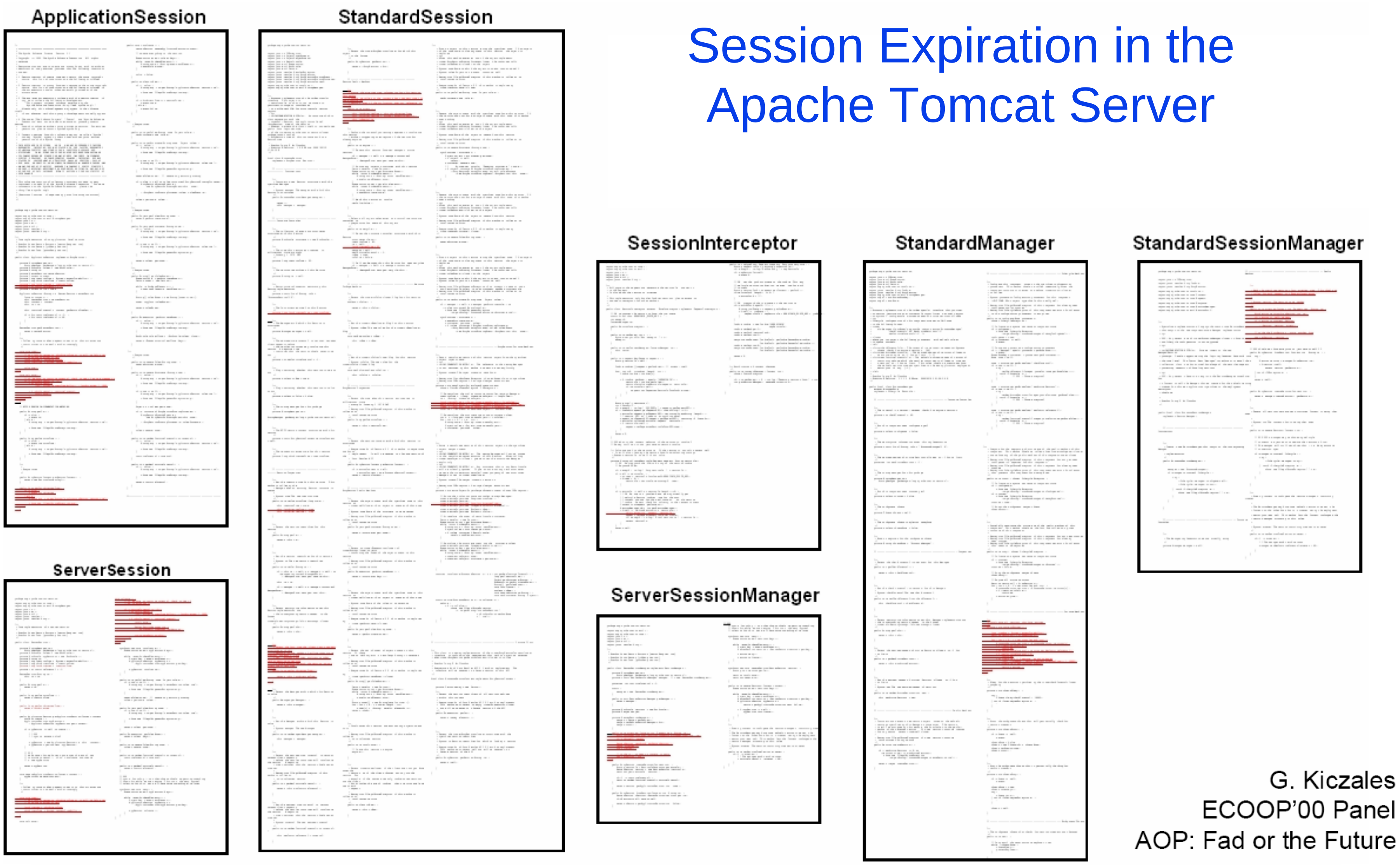


eCommerce System (Uni. Waterloo)

…the Bad…

- Best practice in software product line engineering
  - Imperative and object-oriented programming
  - C preprocessor, conditional compilation
  - C++ template metaprogramming
  - Version control systems
  - Make, Configure, Ant, Maven
  - Ad-hoc code generators
  - …

…and the Ugly



Session Expiration in the Apache Tomcat Server

G. Kiczales
ECOOP'00 Panel
AOP: Fad or the Future

…and the Ugly

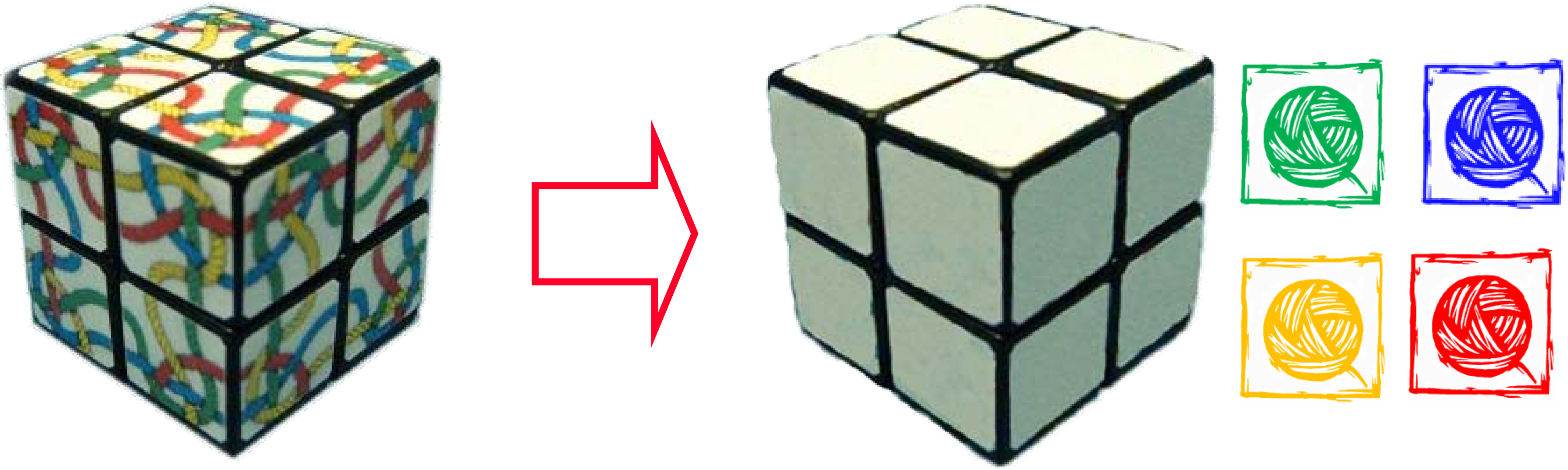
```
#ifndef _STDARG_H
#ifndef _ANSI_STDARG_H_
#ifndef __need___va_list
#define _STDARG_H
#define _ANSI_STDARG_H_
#endif /* not __need___va_list */
#undef __need___va_list

/* Define __gnuc_va_list.  */

#ifndef __GNUC_VA_LIST
#define __GNUC_VA_LIST
typedef __builtin_va_list __gnuc_va_list;
#endif

#ifdef _STDARG_H

#define va_start(v,l)     __builtin_va_start(v,l)
#define va_end(v)         __builtin_va_end(v)
#define va_arg(v,l)       __builtin_va_arg(v,l)
#if !defined(__STRICT_ANSI__) || __STDC_VERSION__ + 0 >= 199900L
#define va_copy(d,s)      __builtin_va_copy(d,s)
#endif
#define __va_copy(d,s)                          __builtin_va_copy(d,s)

/* Define va_list, if desired, from __gnuc_va_list. */

#ifdef _HIDDEN_VA_LIST
#undef _VA_LIST
#endif

#ifdef _BSD_VA_LIST
#undef _BSD_VA_LIST
#endif

#if defined(__svr4__) || (defined(_SCO_DS) && !defined(__VA_LIST))
#ifndef _VA_LIST_
#define _VA_LIST_
#ifdef __i860__
#ifndef _VA_LIST
#define _VA_LIST va_list
#endif
#endif /* __i860__ */
typedef __gnuc_va_list va_list;
#ifdef _SCO_DS
#define __VA_LIST
#endif
#endif /* _VA_LIST_ */
#else /* not __svr4__ || _SCO_DS */
```

```
#if !defined (_VA_LIST_) || defined (__BSD_NET2__) || defined (____386BSD____) ||
   defined (__bsdi__) || defined (__sequent__) || defined (__FreeBSD__) || defined(WINNT)
/* The macro _VA_LIST_DEFINED is used in Windows NT 3.5  */
#ifndef _VA_LIST_DEFINED
/* The macro _VA_LIST is used in SCO Unix 3.2.  */
#ifndef _VA_LIST
/* The macro _VA_LIST_T_H is used in the Bull dpx2  */
#ifndef _VA_LIST_T_H
/* The macro __va_list__ is used by BeOS.  */
#ifndef __va_list__
typedef __gnuc_va_list va_list;
#endif /* not __va_list__ */
#endif /* not _VA_LIST_T_H */
#endif /* not _VA_LIST */
#endif /* not _VA_LIST_DEFINED */
#if !(defined (__BSD_NET2__) || defined (____386BSD____) || defined (__bsdi__) ||
   defined (__sequent__) || defined (__FreeBSD__))
#define _VA_LIST_
#endif
#ifndef _VA_LIST
#define _VA_LIST
#endif
#ifndef _VA_LIST_DEFINED
#define _VA_LIST_DEFINED
#endif
#ifndef _VA_LIST_T_H
#define _VA_LIST_T_H
#endif
#ifndef __va_list__
#define __va_list__
#endif

#endif /* not _VA_LIST_, except on certain systems */

#endif /* not __svr4__ */

#endif /* _STDARG_H */

#endif /* not _ANSI_STDARG_H_ */
#endif /* not _STDARG_H */
```

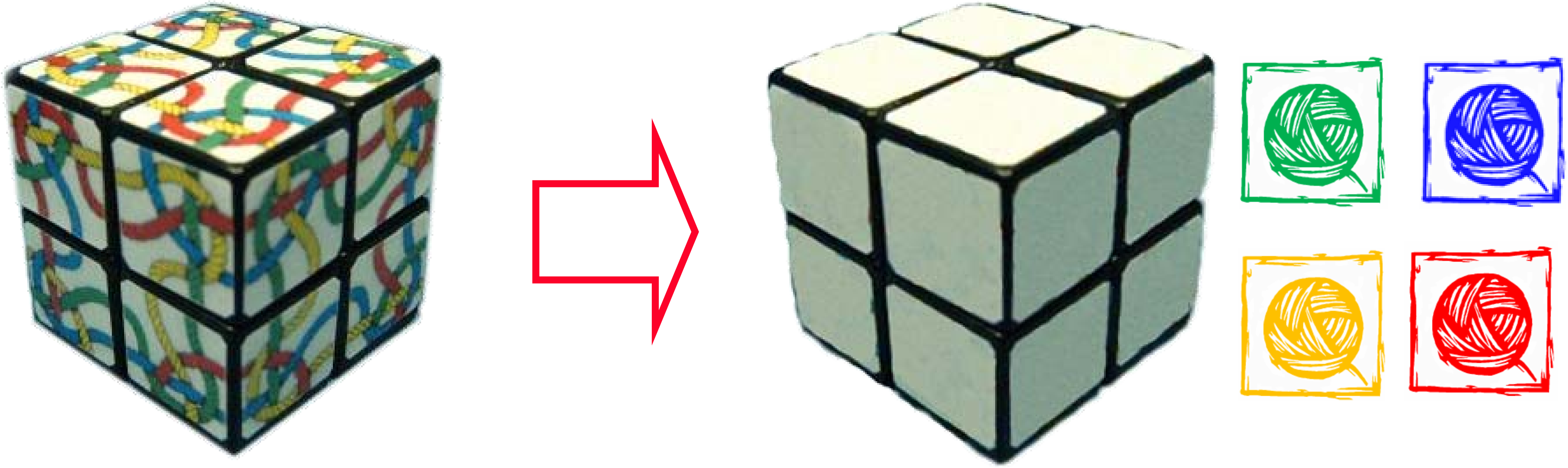
gcc/…/stdarg.h (130 LOC)

Problem: Lack of Separation of Concerns

Vision: Implement Features Modularly



Vision: Implement Features Modularly



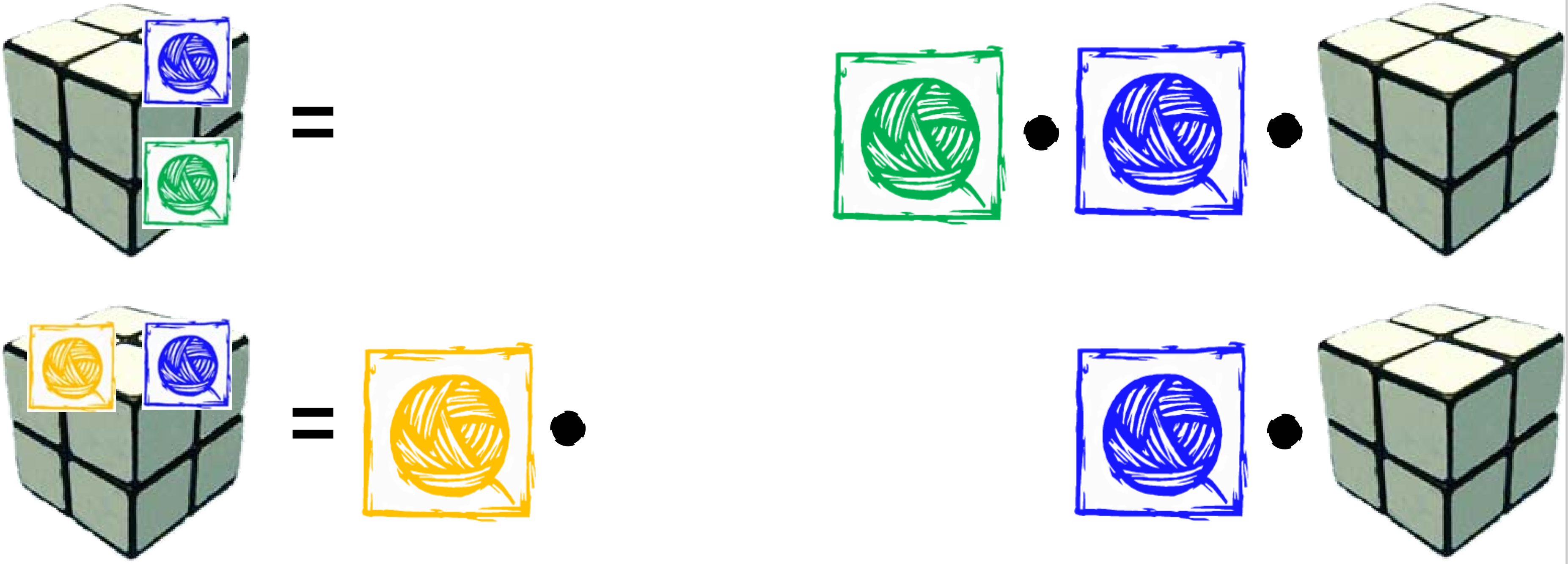
A feature is implemented by ***a structure that extends and modifies the structure of a given program*** *in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option.*

— Apel et al.

Software Product Generation based on Features

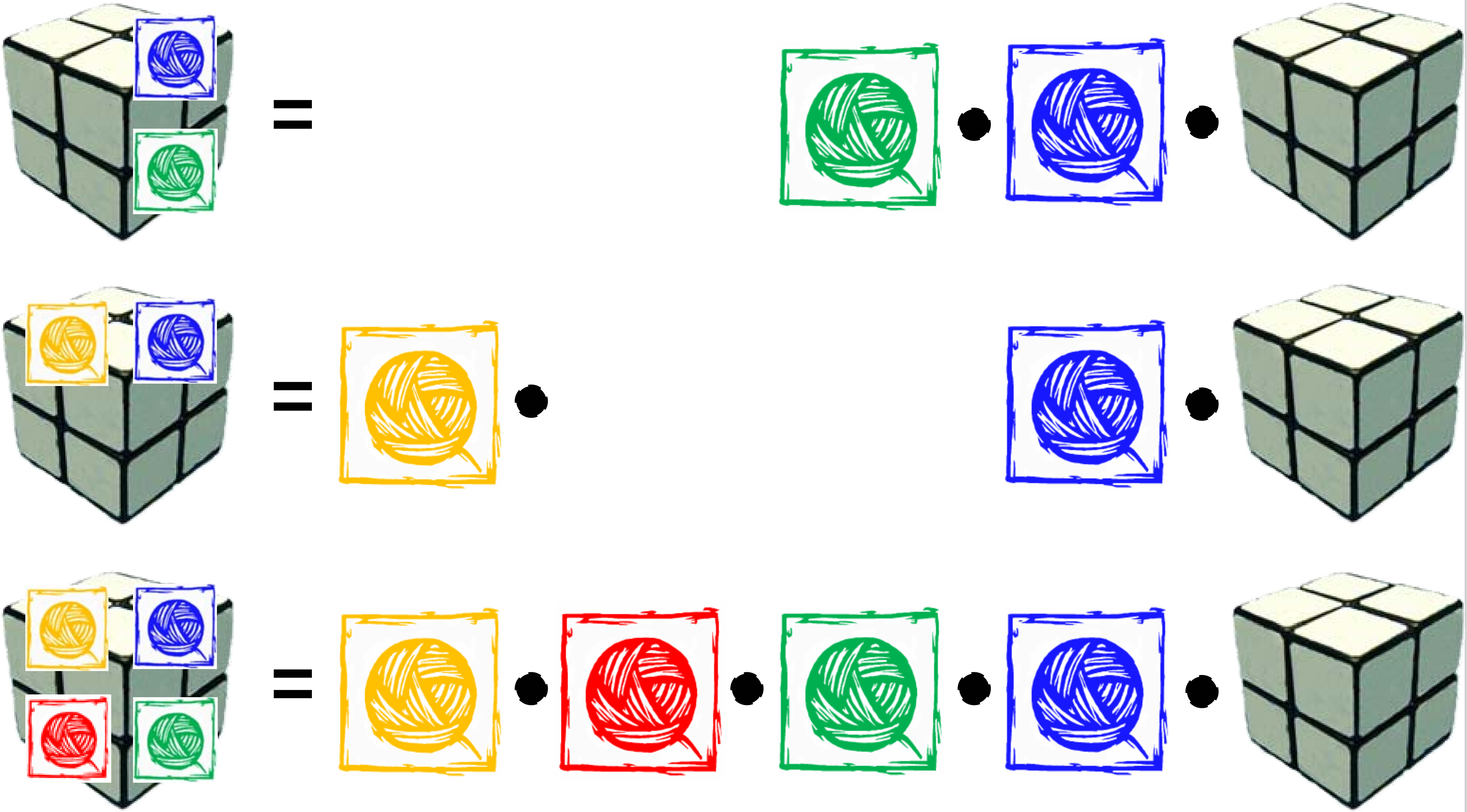
• feature composition

Software Product Generation based on Features



• feature composition

Software Product Generation based on Features



● feature composition

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

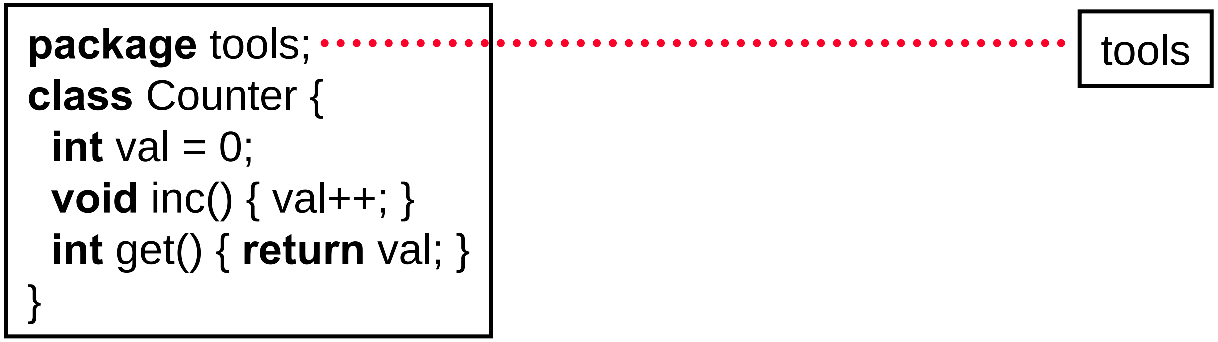
- Informally,
  - the composition of two software artifacts (features),
  - by merging recursively the artifacts' structures,
  - based on nominal and structural similarity.
- Think of software artifacts in terms of their hierarchical, modular structure, e.g.:

```
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.:



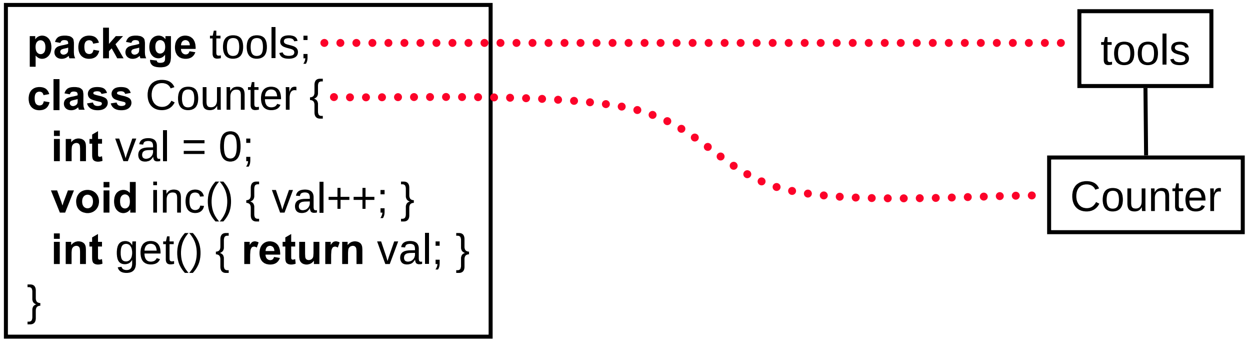
```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

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.:

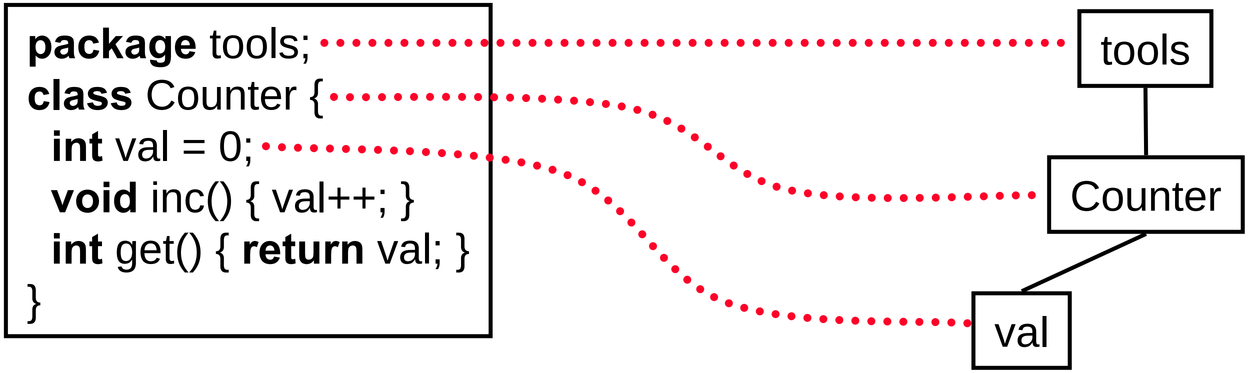


```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

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.:

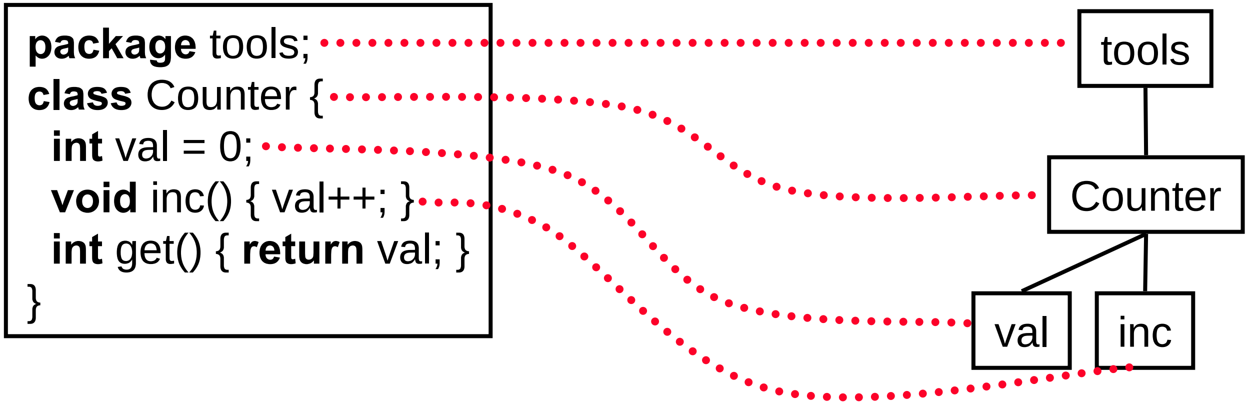


```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

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.:



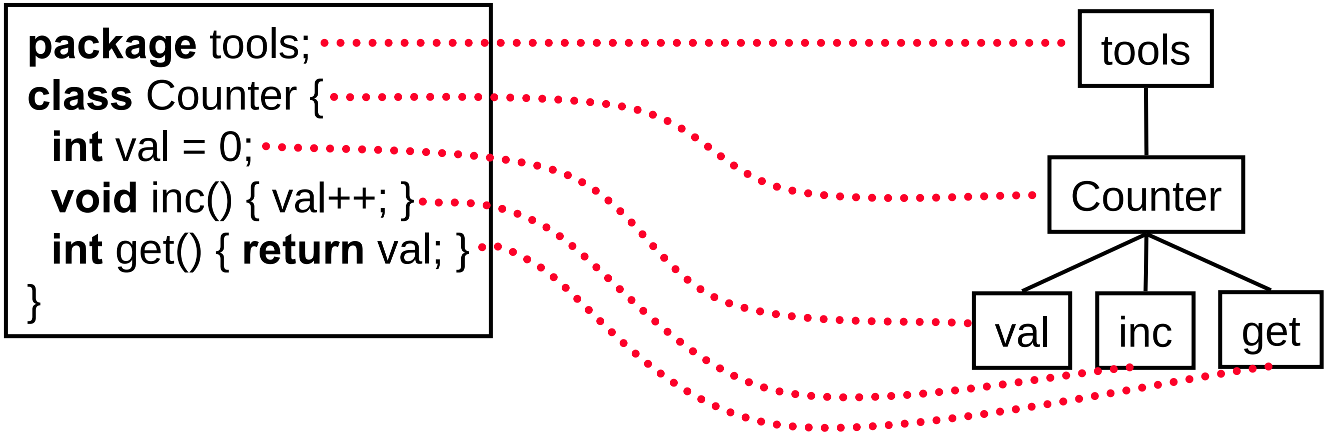
```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

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.:



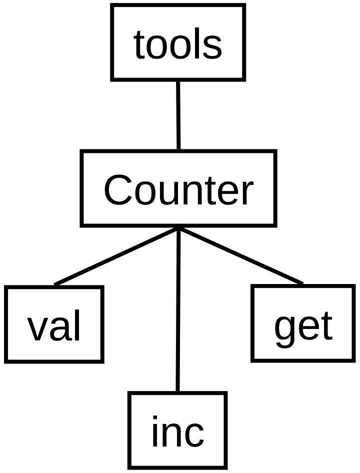
```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

software artifact

hierarchical structure

Superimposition

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



tools

Counter

val

inc

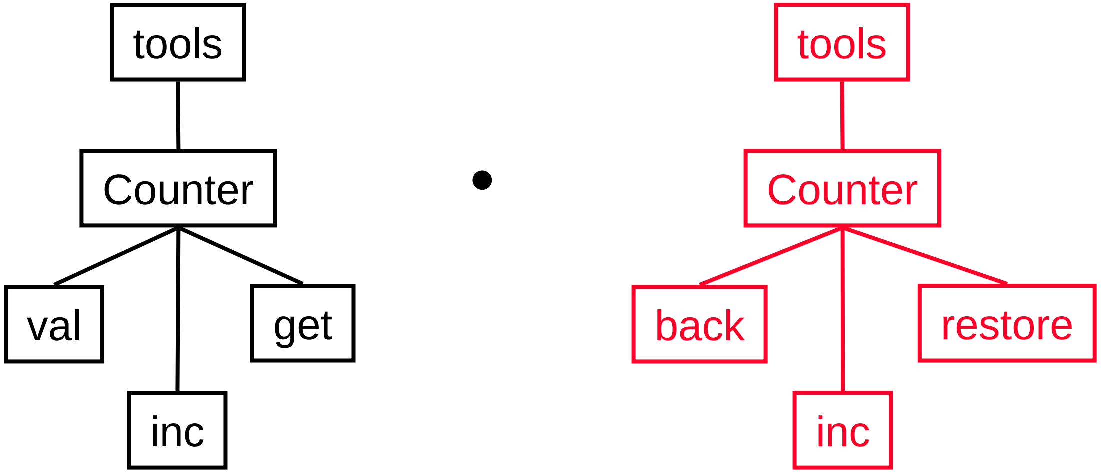
get

Superimposition

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

•

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



Superimposition

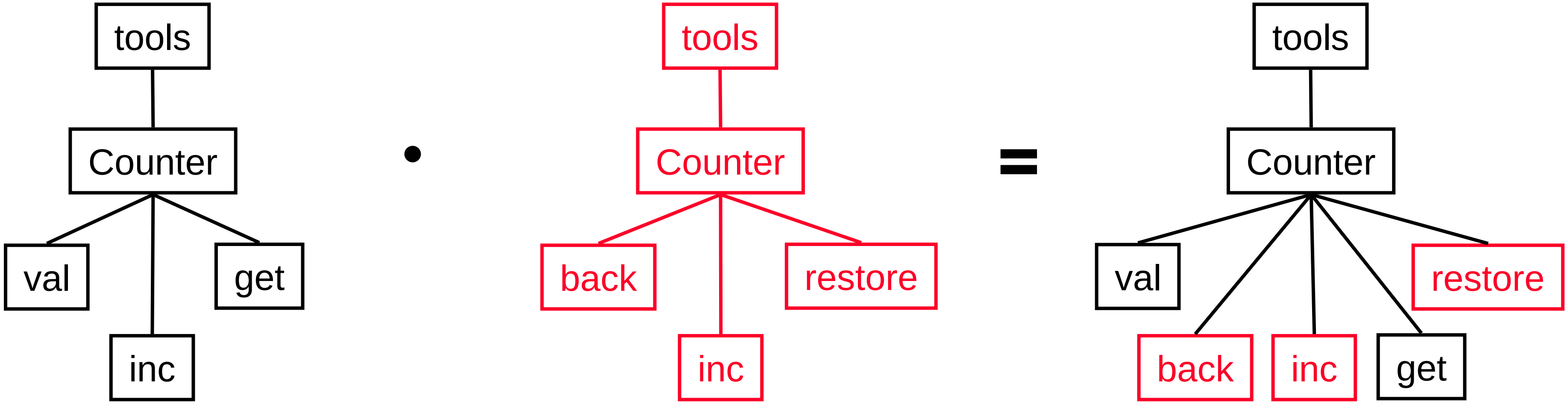
```
package tools;
class Counter {
  int val = 0;
  void inc() { val++; }
  int get() { return val; }
}
```

•

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

=

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



Supporting Superimposition

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

●

```
package backup; import base.Counter;
refine class Counter {
  int back = 0;
  void inc() { back=val; original(); }
  void restore() { val=back; }
}
```

Classbox/J

Supporting Superimposition

Classbox/J

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

●

```
package backup; import base.Counter;
refine class Counter {
  int back = 0;
  void inc() { back=val; original(); }
  void restore() { val=back; }
}
```

Jak

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

●

```
layer backup;
refines class Counter {
  int back = 0;
  void inc() { back=val; Super().inc(); }
  void restore() { val=back; }
}
```

Supporting Superimposition

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

●

```
package backup; import base.Counter;
refine class Counter {
  int back = 0;
  void inc() { back=val; original(); }
  void restore() { val=back; }
}
```

Classbox/J

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

●

```
layer backup;
refines class Counter {
  int back = 0;
  void inc() { back=val; Super().inc(); }
  void restore() { val=back; }
}
```

Jak

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

●

```
team class Backup extends Base {
  class Counter {
    int back = 0;
    void inc() { back=val; tsuper.inc(); }
    void restore() { val=back; }
  }
}
```

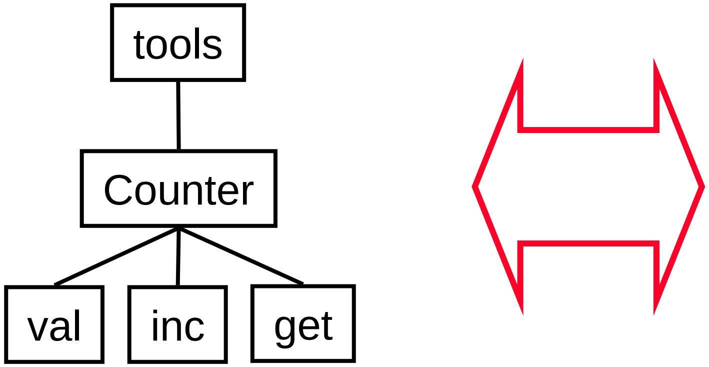
ObjectTeams/J

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

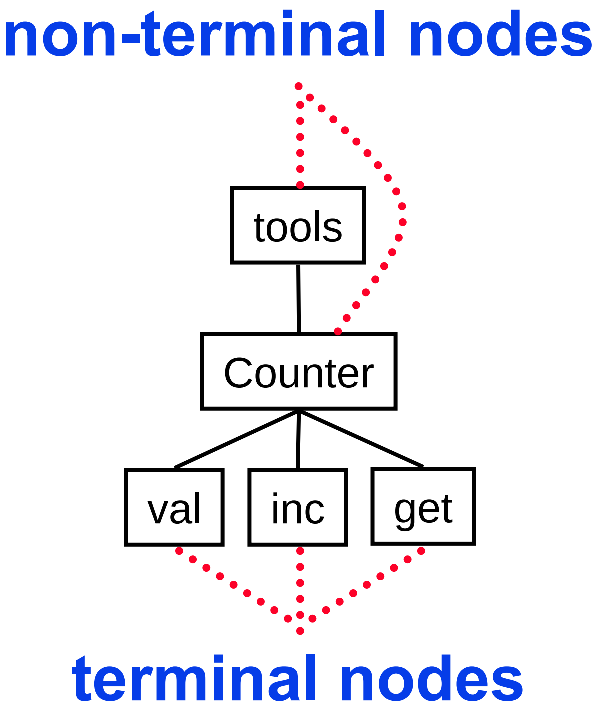


$$\begin{aligned} Base \; = \; & tools : Package \\ & \oplus\; tools.Counter : Class \\ & \oplus\; tools.Counter.val : Field \\ & \oplus\; tools.Counter.inc : Method \\ & \oplus\; tools.Counter.get : Method \end{aligned}$$

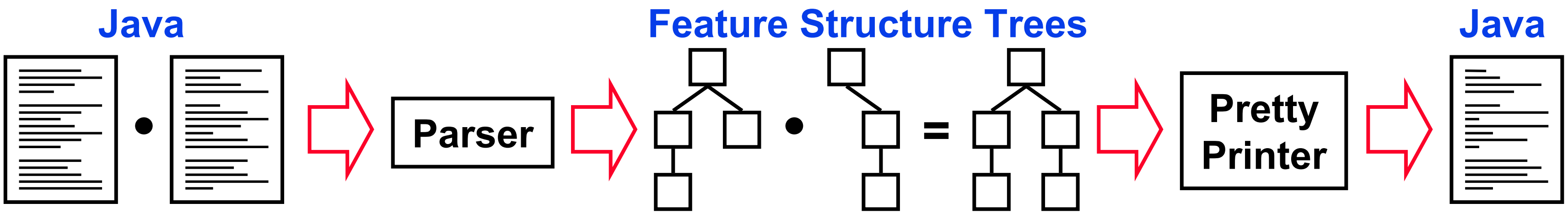
Nodes have names and types; names are mangled; leaves have content.

Non-Terminal vs. Terminal Nodes

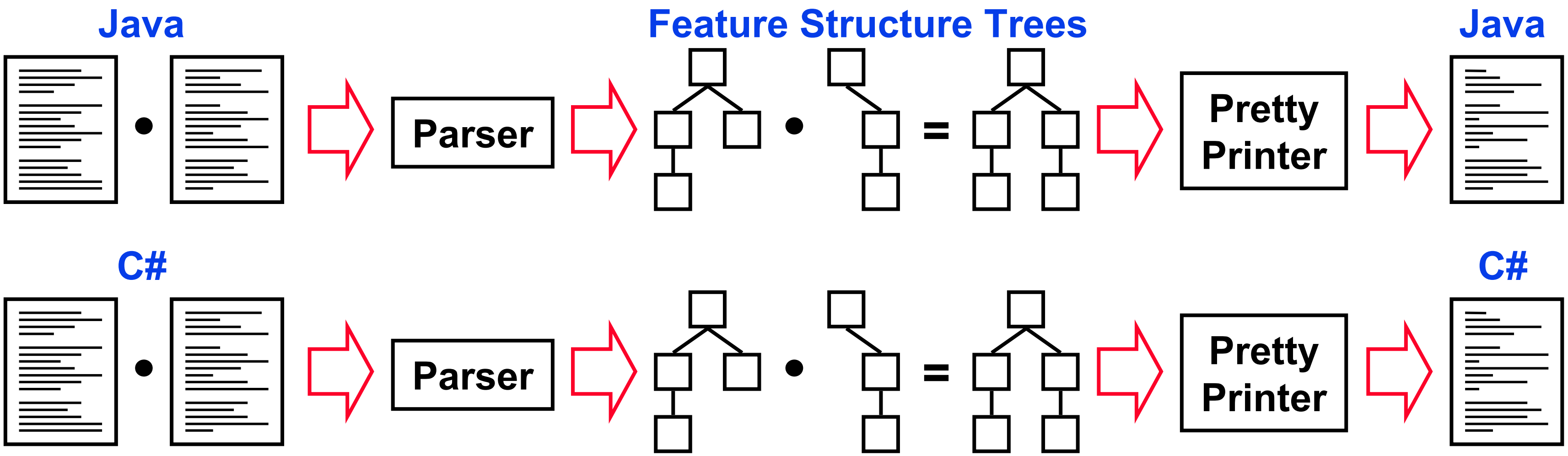
- Non-terminal nodes
  - Identified by name and type
  - Superimposition proceeds recursively with the children
- Terminal nodes
  - Identified by name and type
  - Carry further language-specific content
  - Superimposition terminates with composing contents



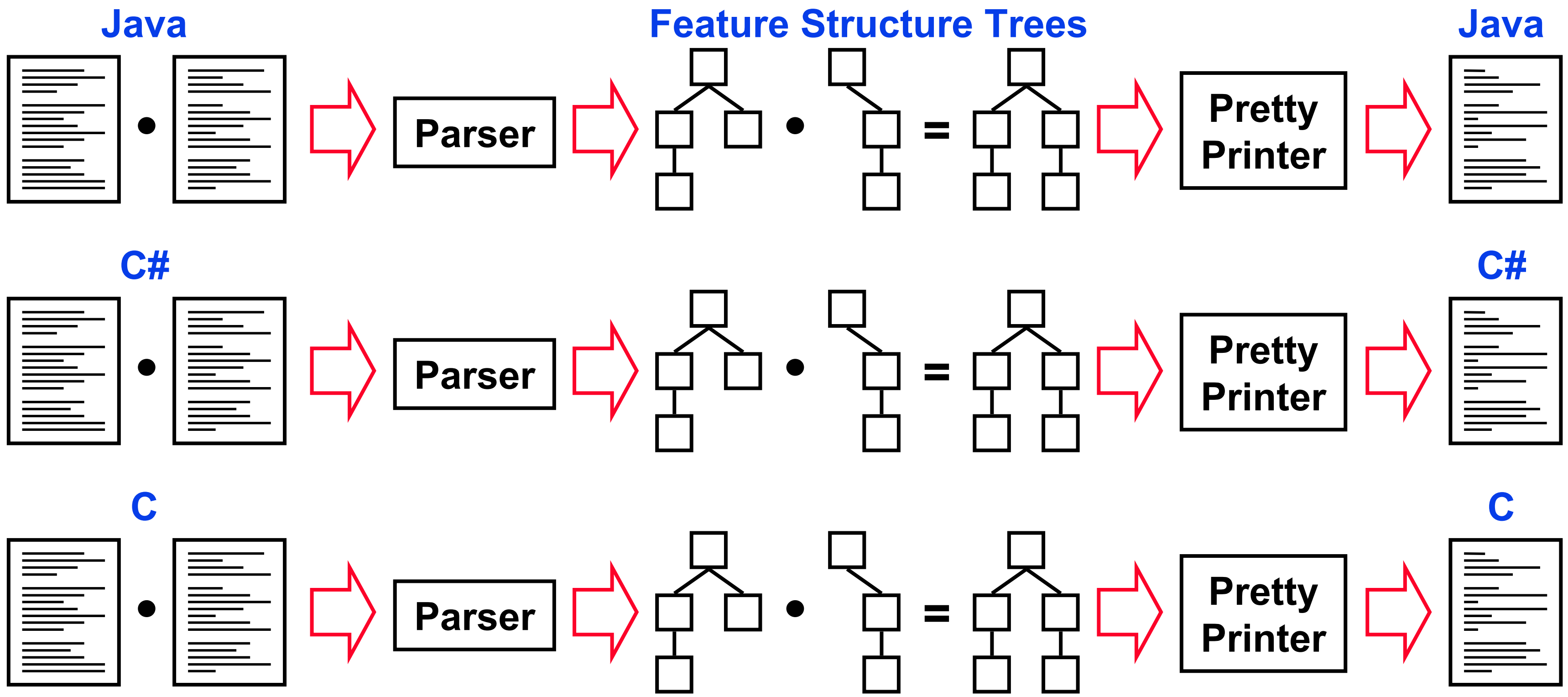
FSTComposer



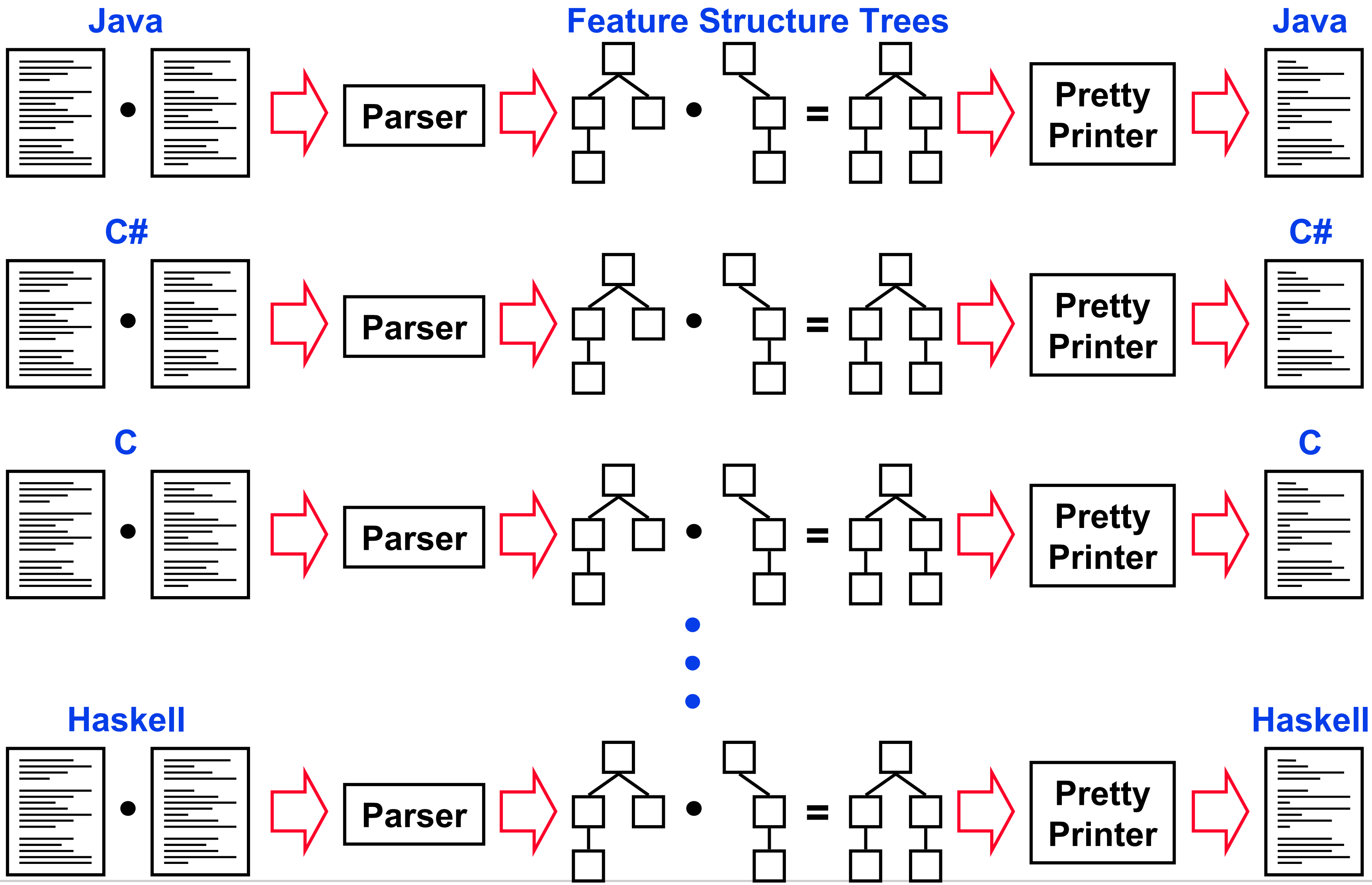
FSTComposer



FSTComposer



FSTComposer



Problems

- Manual integration of Java, C#, XMI/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
    - – $\mathit{method} \times \mathit{method} \rightarrow \mathit{method}$ (overriding)
    - – $\mathit{constructor} \times \mathit{constructor} \rightarrow \mathit{constructor}$ (concatenation)
    - – $\mathit{implements} \times \mathit{implements} \rightarrow \mathit{implements}$ (union)
    - – $\mathit{extends} \times \mathit{extends} \rightarrow \mathit{extends}$ (replacement)
    - – ...

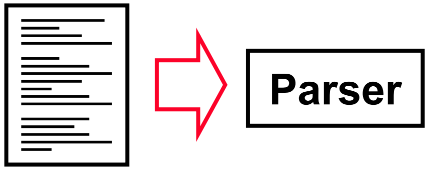
➔ Benefit of language independence is almost lost

An Observation

- Code for supporting different languages is very similar

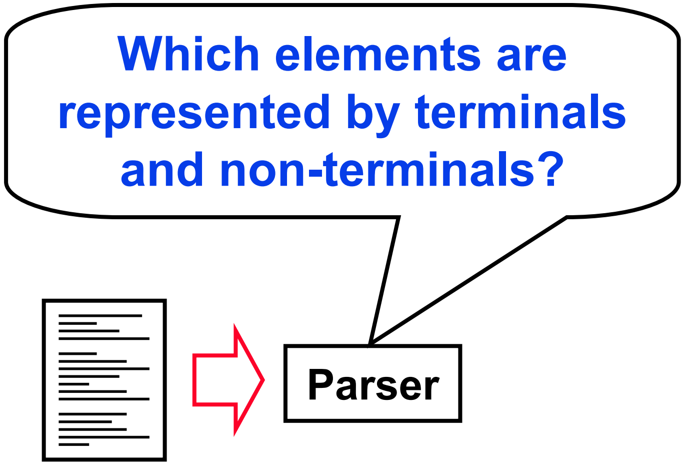
An Observation

- Code for supporting different languages is very similar



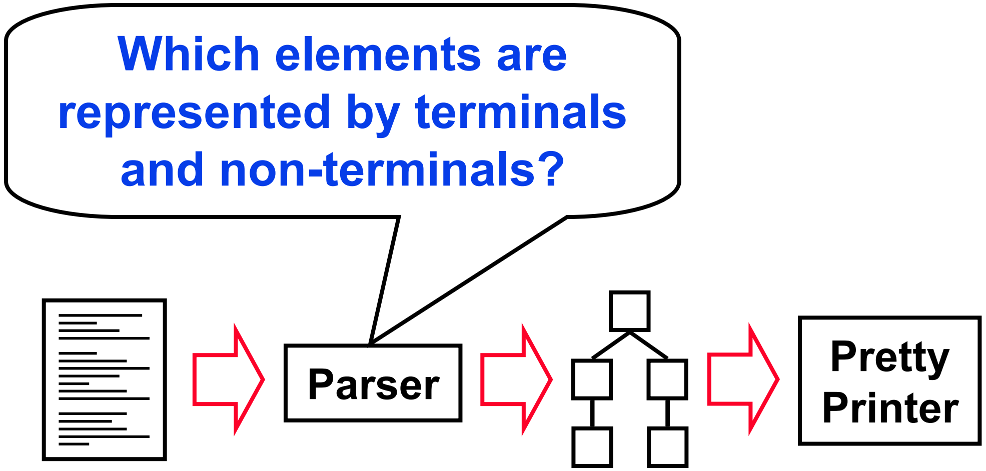
An Observation

- Code for supporting different languages is very similar


Which elements are represented by terminals and non-terminals?
Parser

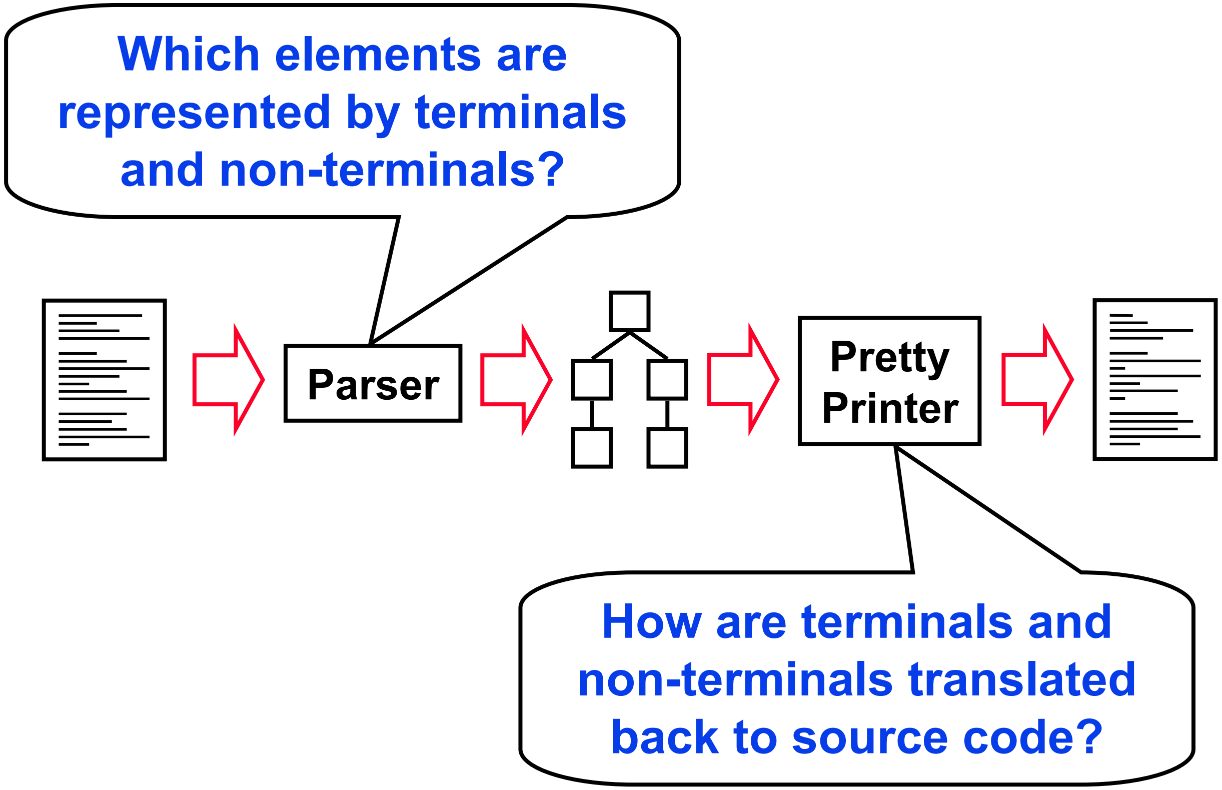
An Observation

- Code for supporting different languages is very similar


Which elements are represented by terminals and non-terminals?
Parser
Pretty Printer

An Observation

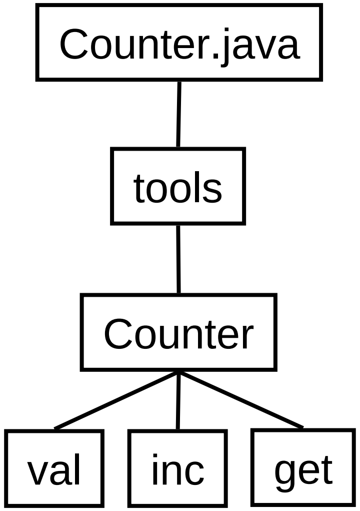
- Code for supporting different languages is very similar


Which elements are represented by terminals and non-terminals?
Parser
Pretty Printer
How are terminals and non-terminals translated back to source code?

An Insight

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

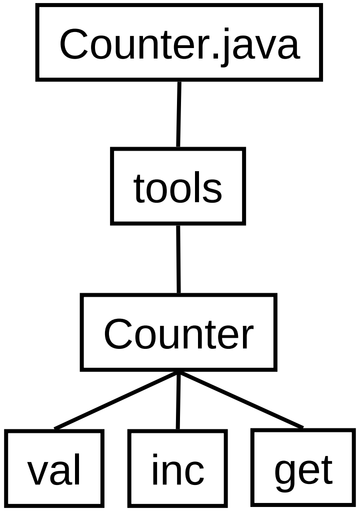
FST



An Insight

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

FST



Grammar

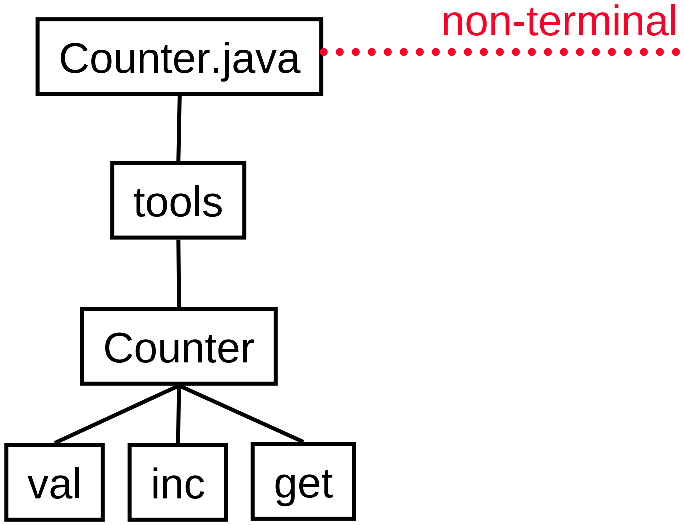
```
JavaFile : [PackageDecl] (ClassDecl)*;

PackageDecl : "package" PackageName ";";

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

FST

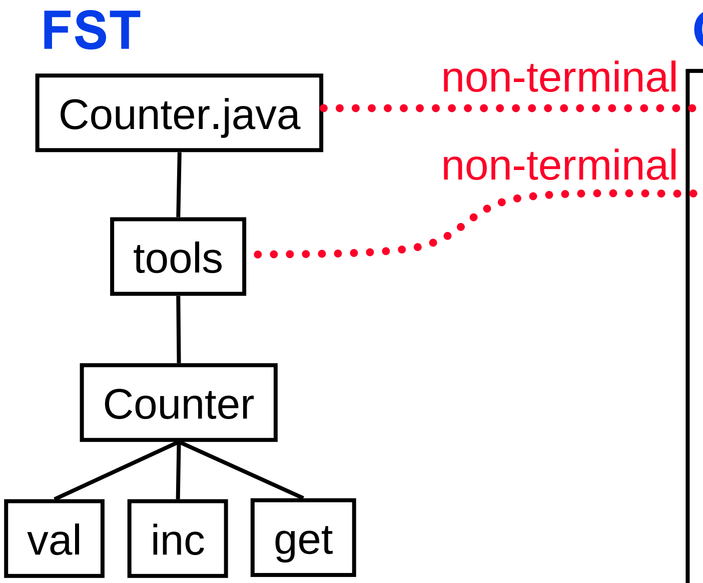
Grammar



```
JavaFile : [PackageDecl] (ClassDecl)*;

PackageDecl : "package" PackageName ";";

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

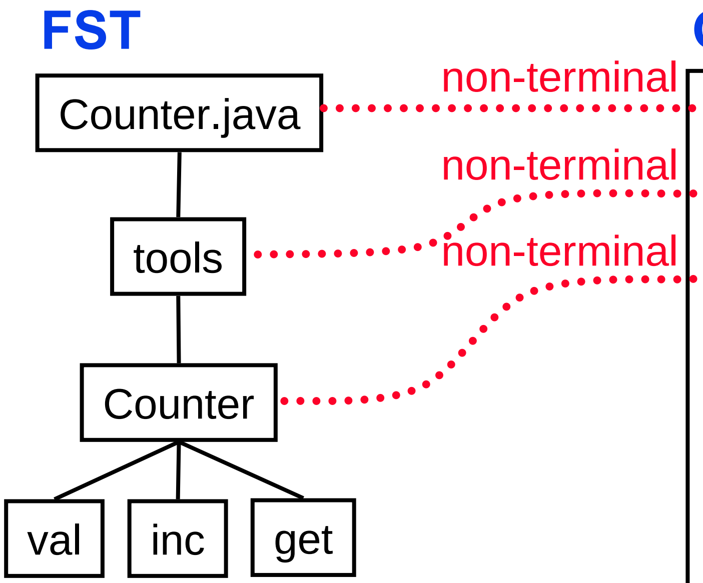


Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;
PackageDecl : "package" PackageName ";";
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

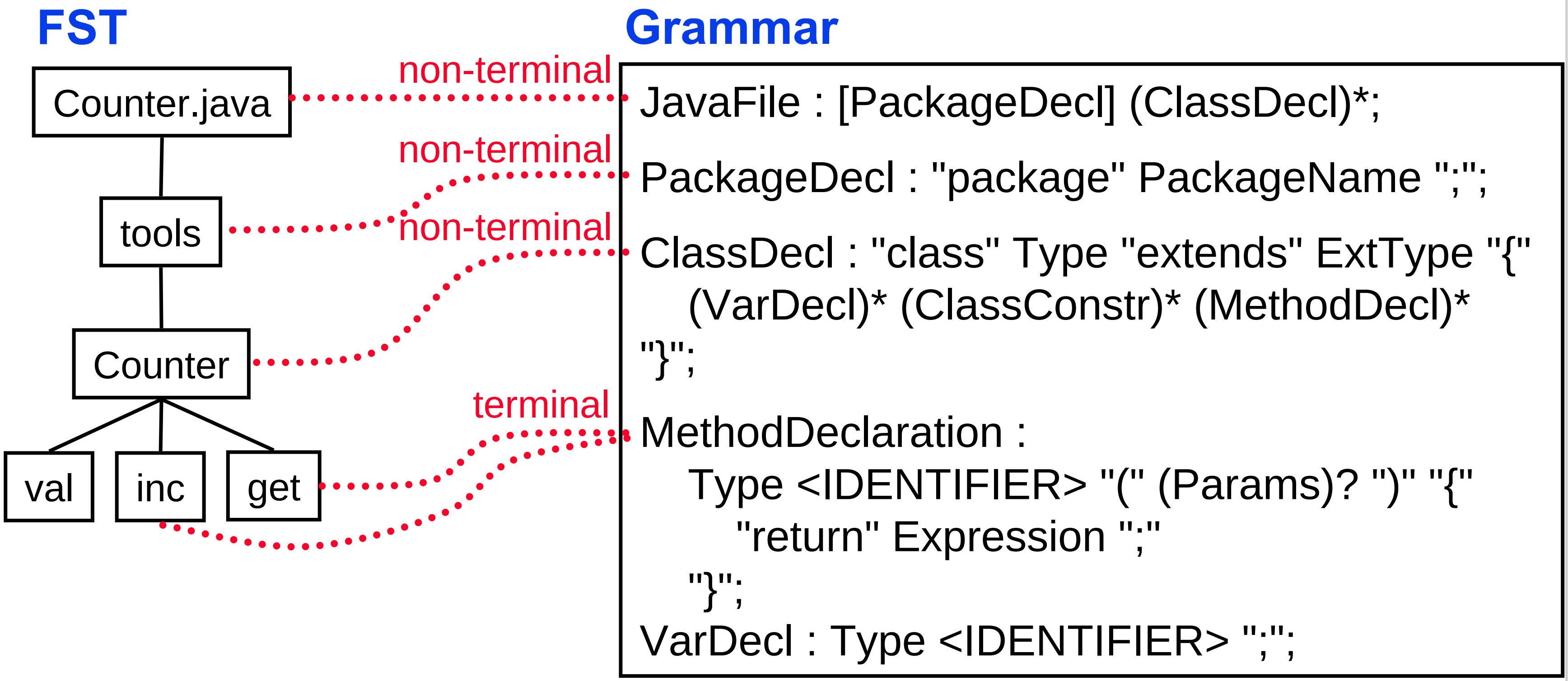


Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;
PackageDecl : "package" PackageName ";";
ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";
MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";
VarDecl : Type <IDENTIFIER> ";";
```

An Insight

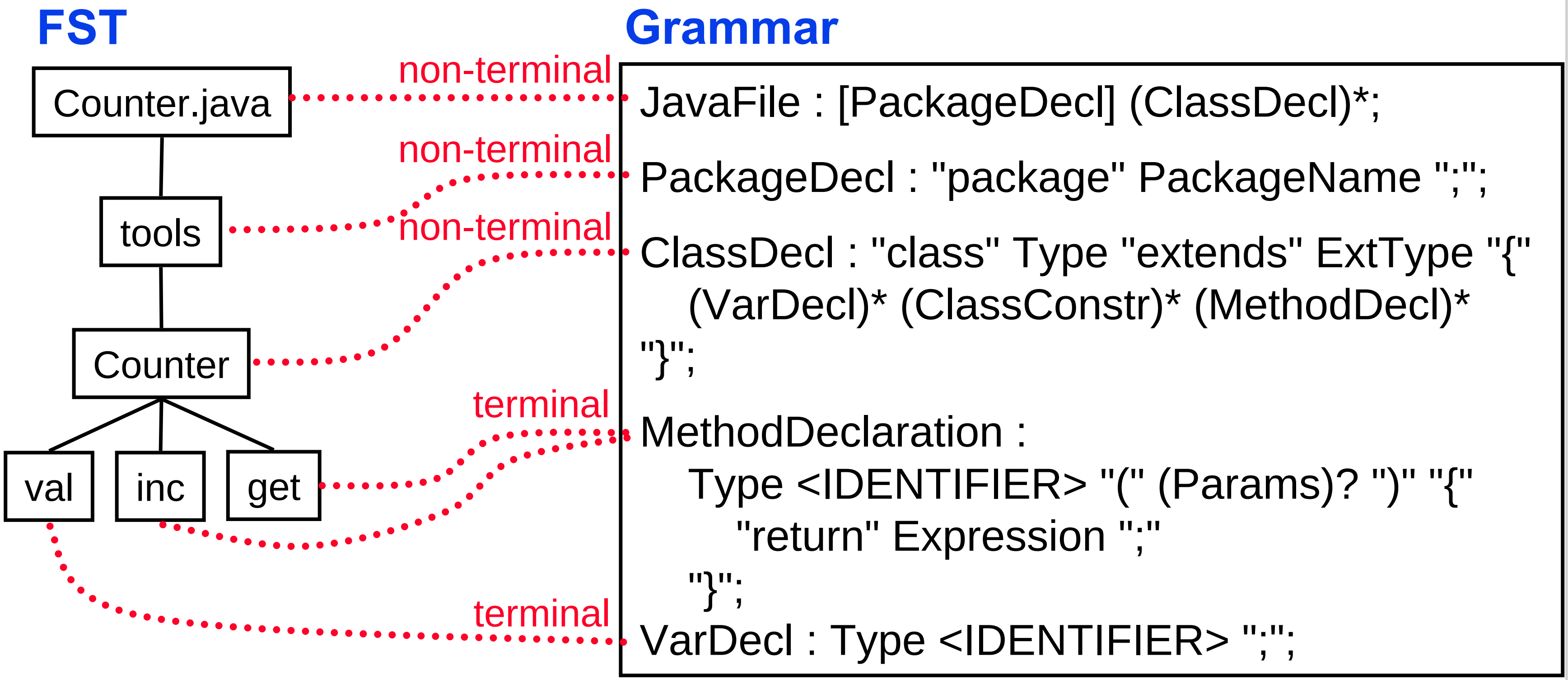
- Non-terminals and terminals correspond to production rules in the artifact language's grammar



```
JavaFile : [PackageDecl] (ClassDecl)*;
PackageDecl : "package" PackageName ";";
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



```
JavaFile : [PackageDecl] (ClassDecl)*;
PackageDecl : "package" PackageName ";";
ClassDecl : "class" Type "extends" ExtType "{"
    (VarDecl)* (ClassConstr)* (MethodDecl)*
"}";
MethodDeclaration :
    Type <IDENTIFIER> "(" (Params)? ")" "{"
        "return" Expression ";"
    "}";
VarDecl : Type <IDENTIFIER> ";";
```

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 : [PackageDecl] (ClassDecl)*;


PackageDecl : "package" PackageName ";";


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



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



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

Annotating a Simplified Java Grammar

Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;


PackageDecl : "package" PackageName ";";


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



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



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

Example

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

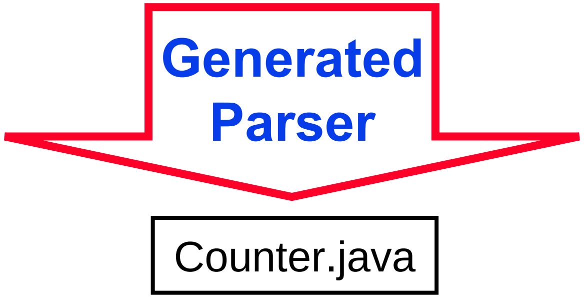
Annotating a Simplified Java Grammar

Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;


PackageDecl : "package" PackageName ";";


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



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



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

Example

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



Generated Parser

Counter.java

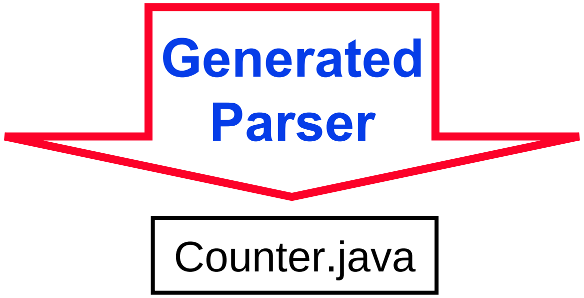
Annotating a Simplified Java Grammar

Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;


PackageDecl : "package" PackageName ";";


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



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



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

Example

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



Generated Parser

Counter.java

Don't know which elements are non-terminals and terminals!

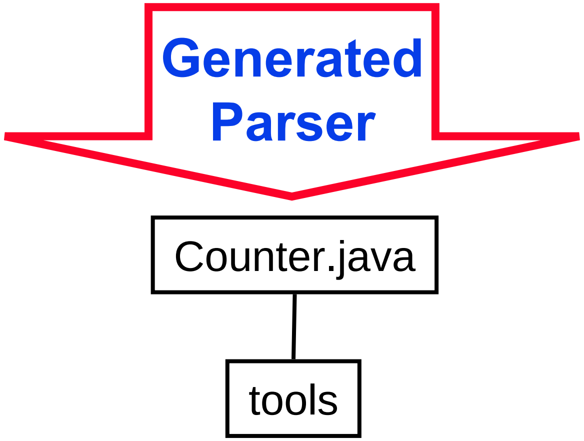
Annotating a Simplified Java Grammar

Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;

@FSTNonTerminal(name="{PackageName}")
PackageDecl : "package" PackageName ";";


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



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



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

Example

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


Generated Parser

Counter.java

tools

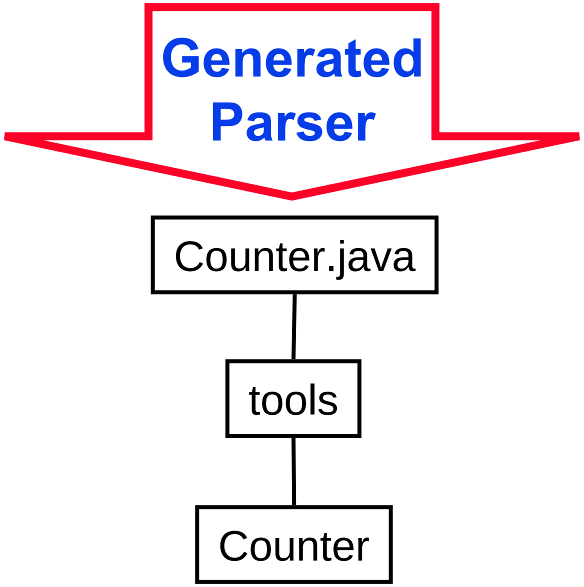
Annotating a Simplified Java Grammar

Grammar

```
JavaFile : [PackageDecl] (ClassDecl)*;

@FSTNonTerminal(name="{PackageName}")
PackageDecl : "package" PackageName ";";

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



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


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

Example

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



Generated Parser

Counter.java

tools

Counter

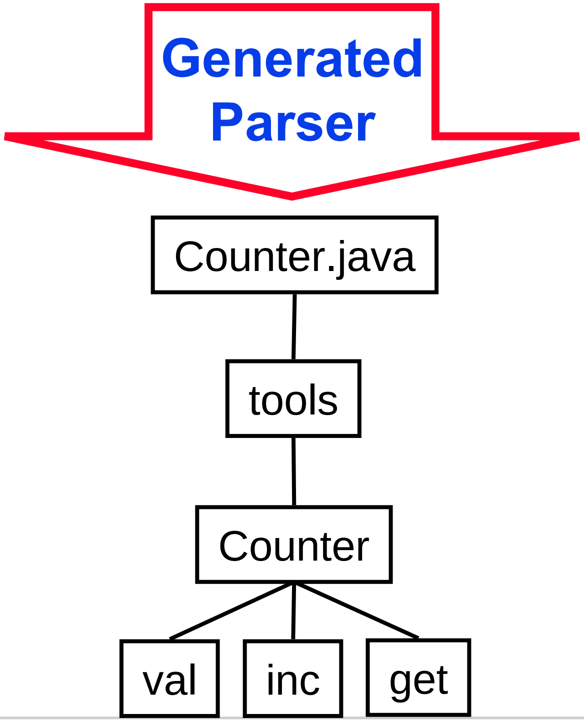
Annotating a Simplified Java Grammar

Grammar

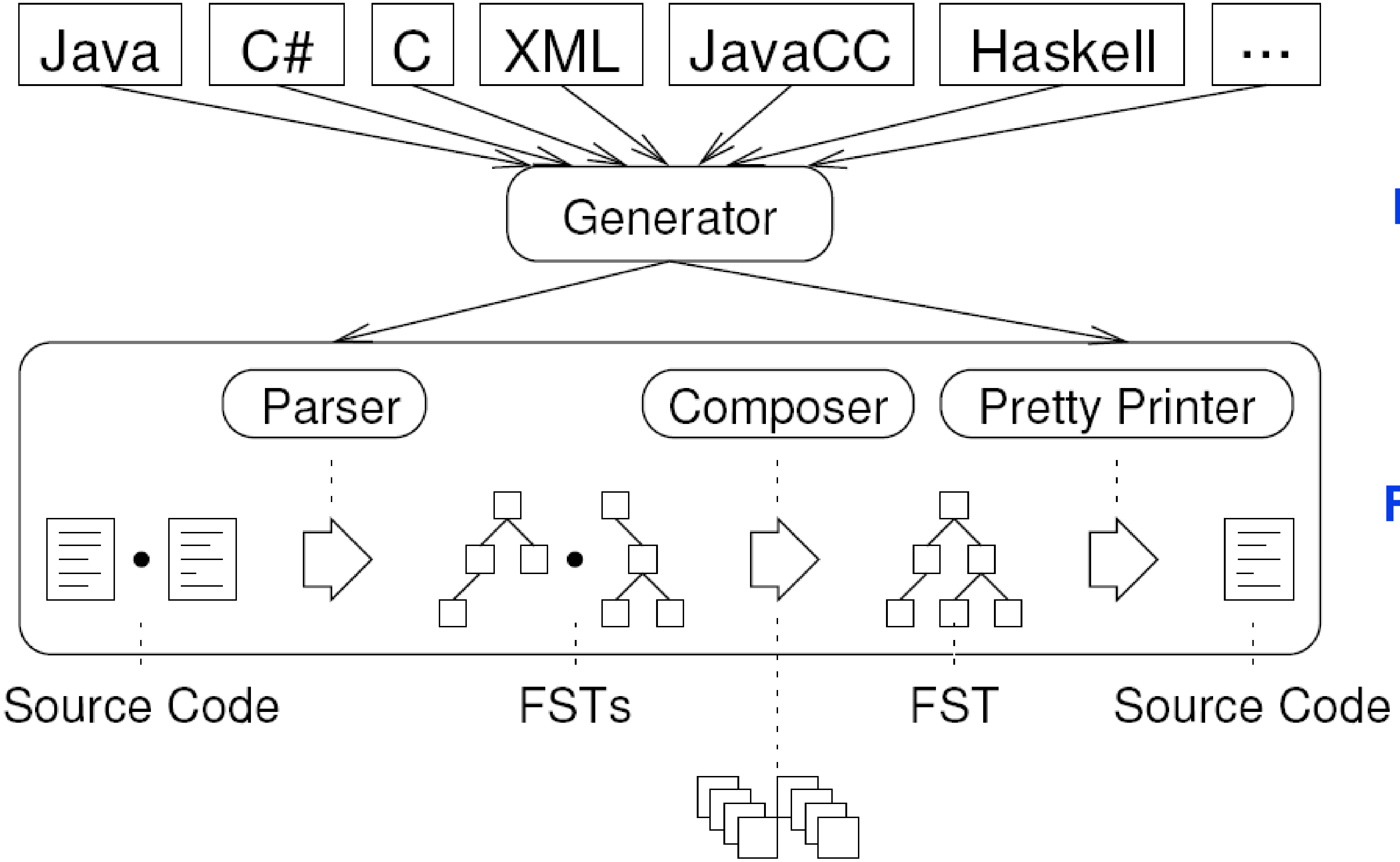
```
JavaFile : [PackageDecl] (ClassDecl)*;

@FSTNonTerminal(name="{PackageName}")
PackageDecl : "package" PackageName ";";

@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

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



FeatureHouse



FeatureBNF

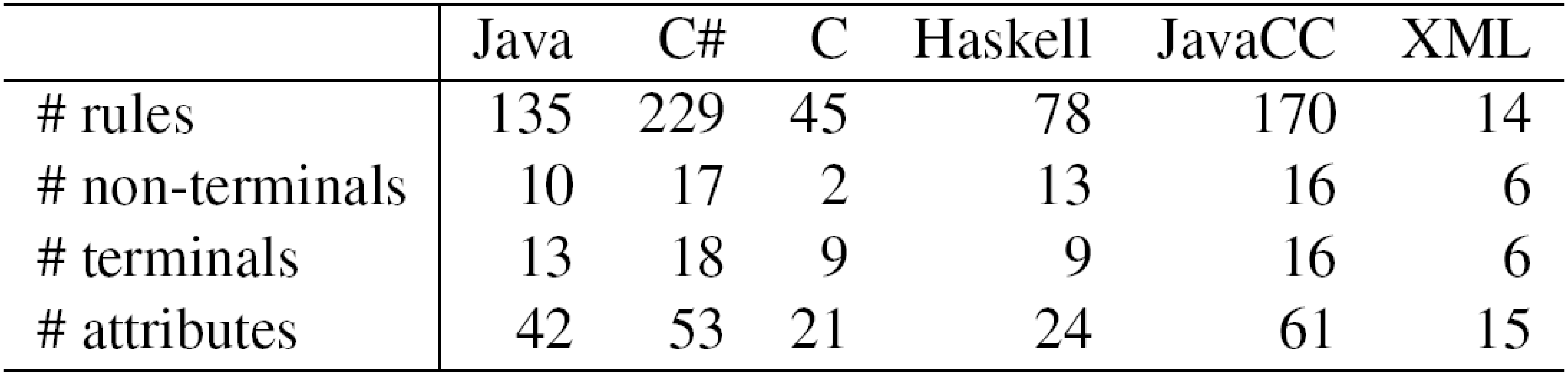
FSTGenerator

FSTComposer

library of composition rules

Integrating Languages

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



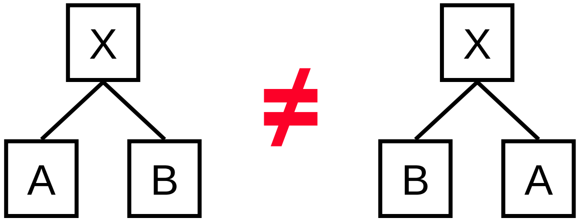
	Java	C#	C	Haskell	JavaCC	XML
# rules	135	229	45	78	170	14
# non-terminals	10	17	2	13	16	6
# terminals	13	18	9	9	16	6
# attributes	42	53	21	24	61	15

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	XMI/UML	Phone system
ACS	4	2.080	XMI/UML	Audio control system
CMS	10	2.037	XMI/UML	Conference management system
GPL (C#)	20	2.148	C#	Graph product line (C# version)
GBS	29	2.380	XMI/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

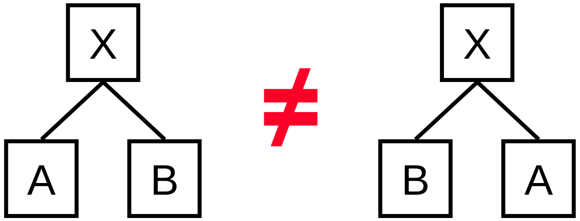
Problems

- Lexical order:

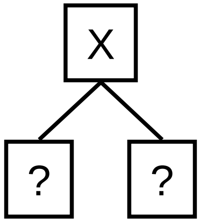


Problems

- Lexical order:

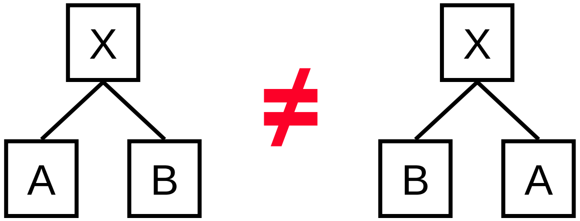


- Unnamed elements:

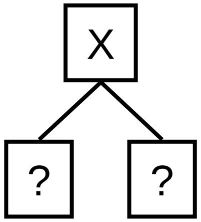


Problems

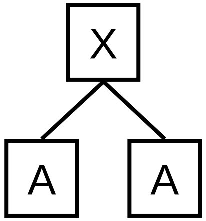
- Lexical order:



- 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)
- Languages can be integrated almost **automatically**
  - A wide variety of very different languages are integrated
- Problems with lexical order, unnamed elements, and ambiguous names

Questions?

- Sven Apel, Christian Kästner, and Christian Lengauer. **FeatureHouse: Language-Independent, Automated Software Composition**. In *Proc. Int. Conf. Software Engineering (ICSE)*. Mai 2009.

  FeatureHouse

- Sven Apel, Florian Janda, Salvador Trujillo, and Christian Kästner. **Model Superimposition in Software Product Lines**. In *Proc. Int. Conf. Model Transformation (ICMT)*. July 2009.

  UML

- Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. **Feature (De)composition in Functional Programming**. In *Proc. Int. Conf. Software Composition (SC)*. July 2009.

  Haskell

- 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.

  Feature Algebra

Mandatory Language Properties

- Properties a language must have to be ready for superimposition
  - The substructure of an software artifact must be representable as a tree
  - Every element of an artifact must provide a name and must belong to a syntactical category
  - An element must not contain multiple elements with identical names and types
  - There must be composition rules for elements that are represented as terminals

➔Even plain text can be composed; but the more structure is exposed, the more fine-grained superimposition can be

An Exception

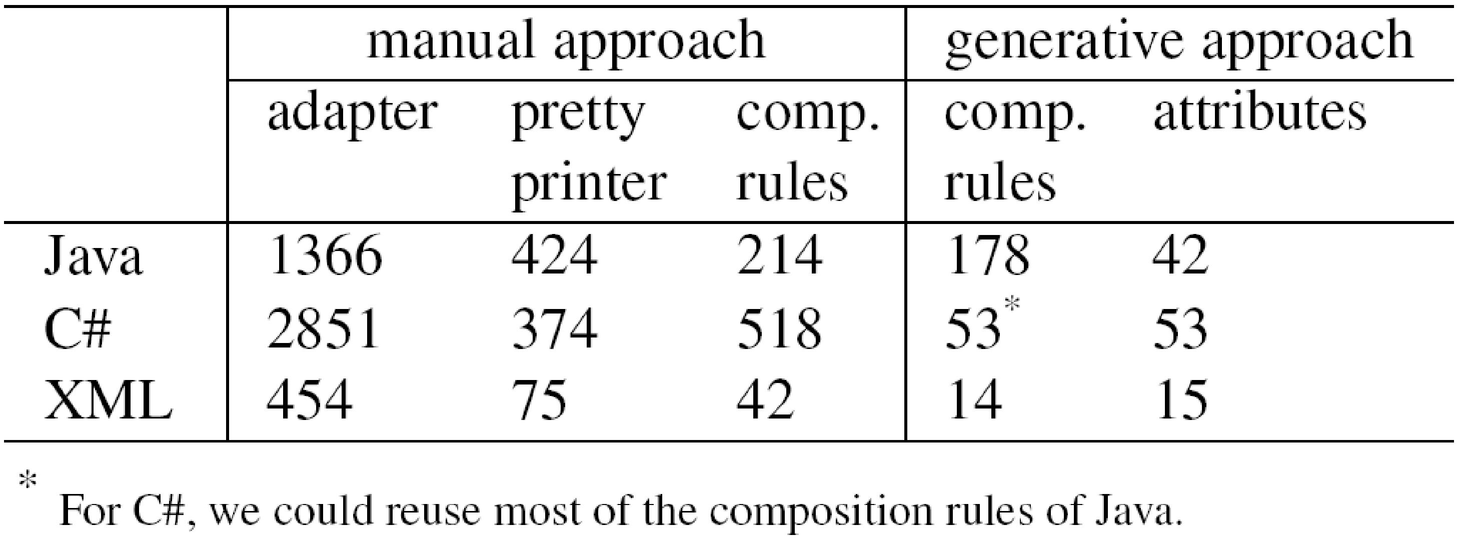
- Parsing and superimposing XML documents on the basis of elements' names is often not appropriate
  - For example, in XHTML: **`<ul>`**, **`<li>`**, …
- Actually, an XML schema describes the syntax of an XML language, not the XML grammar itself
  - ➔ FeatureXSD is needed (analogous to FeatureBNF)

    (implemented and tested with XHTML, XMI/UML, Ant)

Manual vs. Generative Approach (i)

- Granularity – Which elements are (non-)terminals?
  - Fixed in the manual and flexible in the generative approach
  - E.g. should abstract data type in Haskell be non-terminals?

- Boilerplate code



	manual approach			generative approach	
	adapter	pretty printer	comp. rules	comp. rules	attributes
Java	1366	424	214	178	42
C#	2851	374	518	53[*]	53
XML	454	75	42	14	15

[*] For C#, we could reuse most of the composition rules of Java.

Manual vs. Generative Approach (ii)

- Composition rules
  - ◆ Implicit in the manual approach; a library in the generative approach
  - ◆ Possibility to reuse in the generative approach, e.g. the rule **`MethodOverriding`**
- Expenditure of time
  - ◆ In the order of weeks in the manual approach; in the order of days in the generative approach
- Susceptibility to error
  - ◆ Many bugs in the manual approach due to forgotten or misinterpreted AST nodes
  - ◆ Top-down annotation in the general approach is more systematic

Some Interesting Examples: Arith (Haskell)

- Arithmetic expression evaluator (by Armin Größlinger)
  - 27 features, 532 lines of Haskell code
  - Variables, if-then-else, binary and unary operations, lambdas, lazy / strict evaluation, dynamic / static scoping, …
- Further Haskell case study
  - Functional graph library

```
module Expr where {
    data Expr = Num Int | Add Expr Expr | Sub Expr Expr deriving Show;
}
```

```
module Expr where {
   data Expr = Num Int | Add Expr Expr | Sub Expr Expr deriving Show;
}
```

●

```
module Expr where {
   eval :: Expr -> Int;
   eval (Num x) = x;
   eval (Add x y) = (eval x) + (eval y);
   eval (Sub x y) = (eval x) - (eval y);
}
```

```
module Expr where {
    data Expr = Num Int | Add Expr Expr | Sub Expr Expr deriving Show;
}
```

●

```
module Expr where {
    eval :: Expr -> Int;
    eval (Num x) = x;
    eval (Add x y) = (eval x) + (eval y);
    eval (Sub x y) = (eval x) - (eval y);
}
```

●

```
module Expr where {
    data Expr = Mul Expr Expr deriving Show;
    eval (Mul x y) = (eval x) * (eval y);
}
```

```
module Expr where {
    data Expr = Num Int | Add Expr Expr | Sub Expr Expr deriving Show;
}
```

●

```
module Expr where {
    eval :: Expr -> Int;
    eval (Num x) = x;
    eval (Add x y) = (eval x) + (eval y);
    eval (Sub x y) = (eval x) - (eval y);
}
```

●

```
module Expr where {
    data Expr = Mul Expr Expr deriving Show;
    eval (Mul x y) = (eval x) * (eval y);
}
```

=

```
module Expr where {
    data Expr = Num Int | Add Expr Expr | Sub Expr Expr | Mul Expr Expr deriving Show;
    eval :: Expr -> Int;
    eval (Num x) = x;
    eval (Add x y) = (eval x) + (eval y);
    eval (Sub x y) = (eval x) - (eval y);
    eval (Mul x y) = (eval x) * (eval y);
}
```

Some Interesting Examples: Haskell

- Arithmetic expression evaluator (by Armin Größlinger)
  - 27 features, 532 lines of Haskell code
  - Variables, if-then-else, binary and unary operations, lambdas, lazy / strict evaluation, dynamic / static scoping, …
- Further Haskell case study
  - Functional graph library
- Observations
  - Superimposition works naturally together with modules, type classes, and abstract data types
  - Problem with the lexical order of equations

```
eval env (Bin op exp1 exp2) = ...
eval _ _ = ...
```

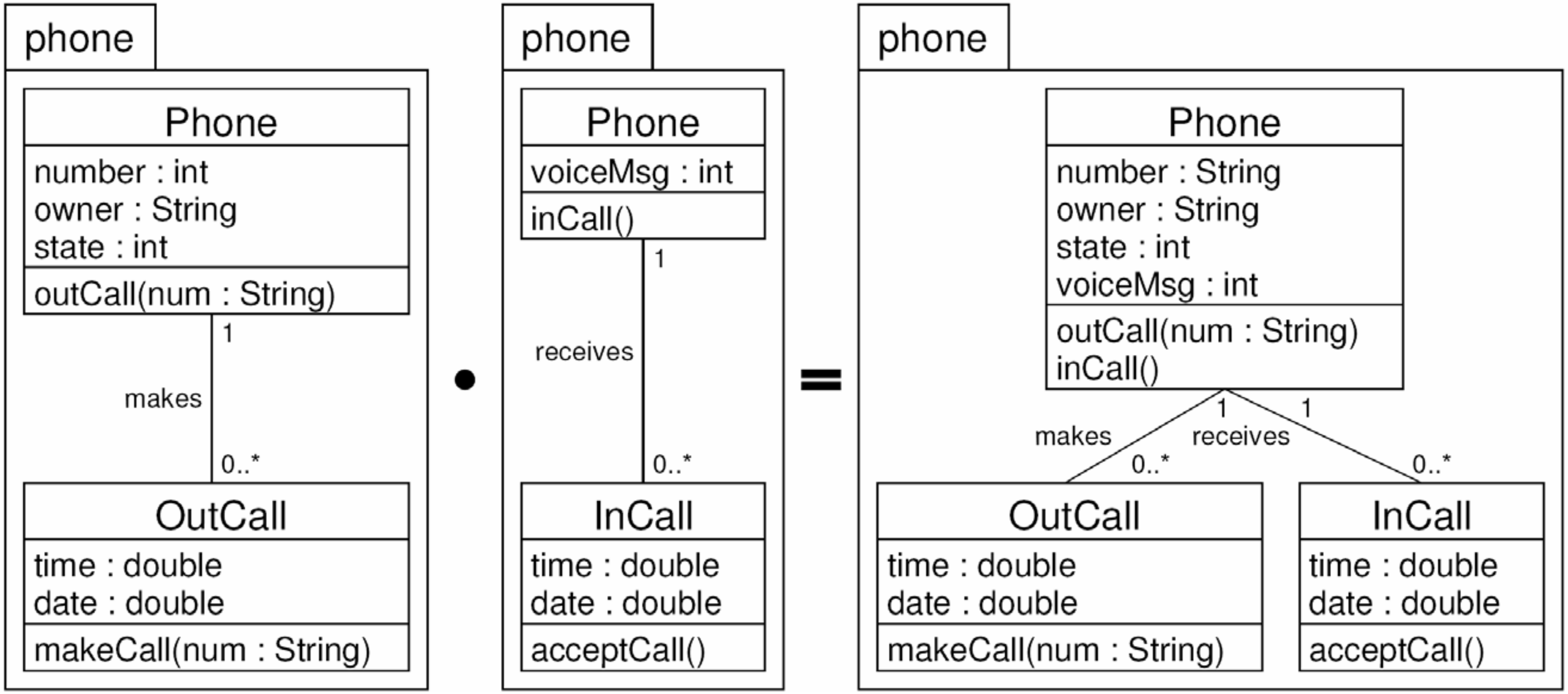
≠

```
eval _ _ = ...
eval env (Bin op exp1 exp2) = ...
```

  - Cannot extend signatures of functions or equations

Some Interesting Examples: XMI/UML

- Phone system
  - 2 features, 1004 lines of XMI code
  - Receive incoming and make outgoing calls
- Further XMI/UML case studies
  - Audio control system, conference management system, gas boiler control system (from IKERLAN Research Centre)



phone

Phone

number : int
owner : String
state : int

outCall(num : String)

1

makes

0..*

OutCall

time : double
date : double

makeCall(num : String)

•

phone

Phone

voiceMsg : int

inCall()

1

receives

0..*

InCall

time : double
date : double

acceptCall()

=

phone

Phone

number : String
owner : String
state : int
voiceMsg : int

outCall(num : String)
inCall()

1

1

makes

receives

0..*

0..*

OutCall

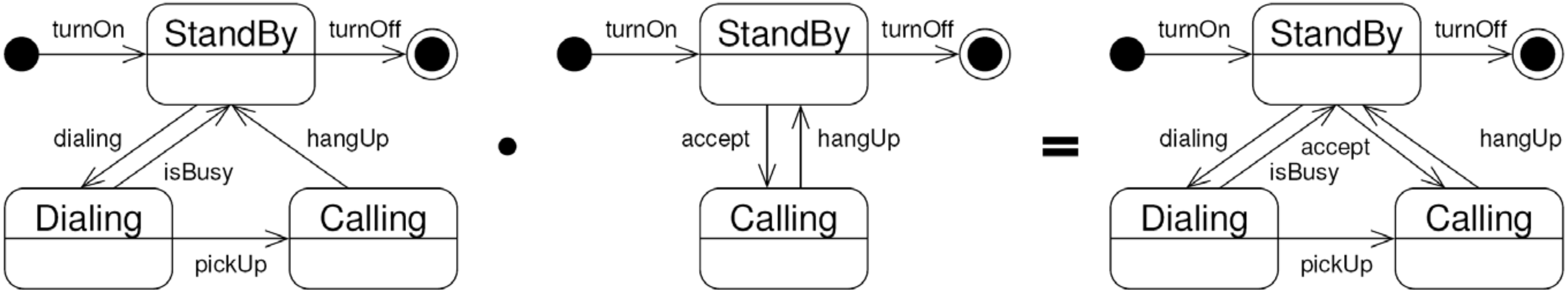
time : double
date : double

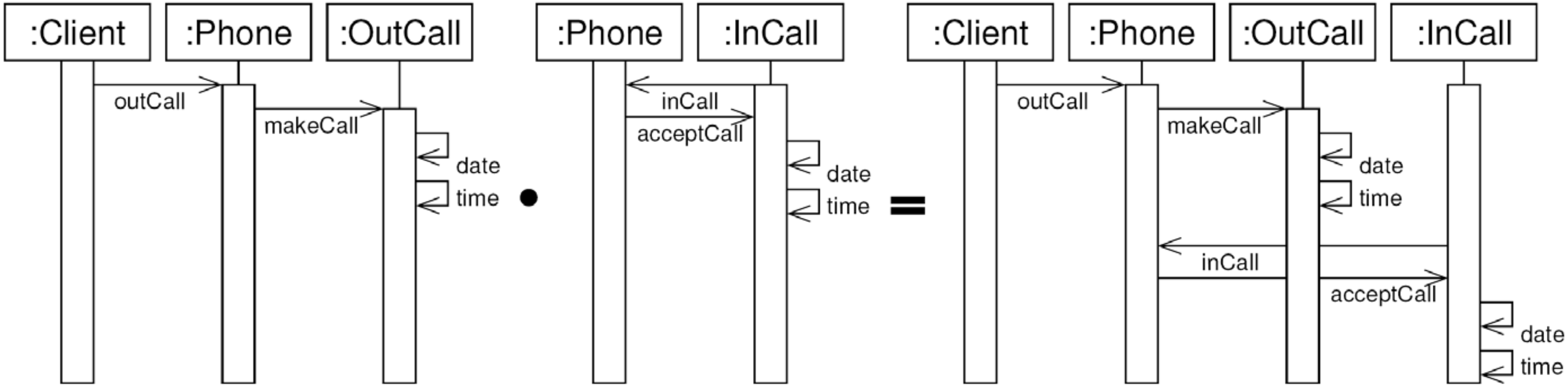
makeCall(num : String)

InCall

time : double
date : double

acceptCall()


turnOn
StandBy
turnOff
dialing
isBusy
hangUp
Dialing
Calling
pickUp
•
turnOn
StandBy
turnOff
accept
hangUp
Calling
=
turnOn
StandBy
turnOff
dialing
accept
isBusy
hangUp
Dialing
Calling
pickUp

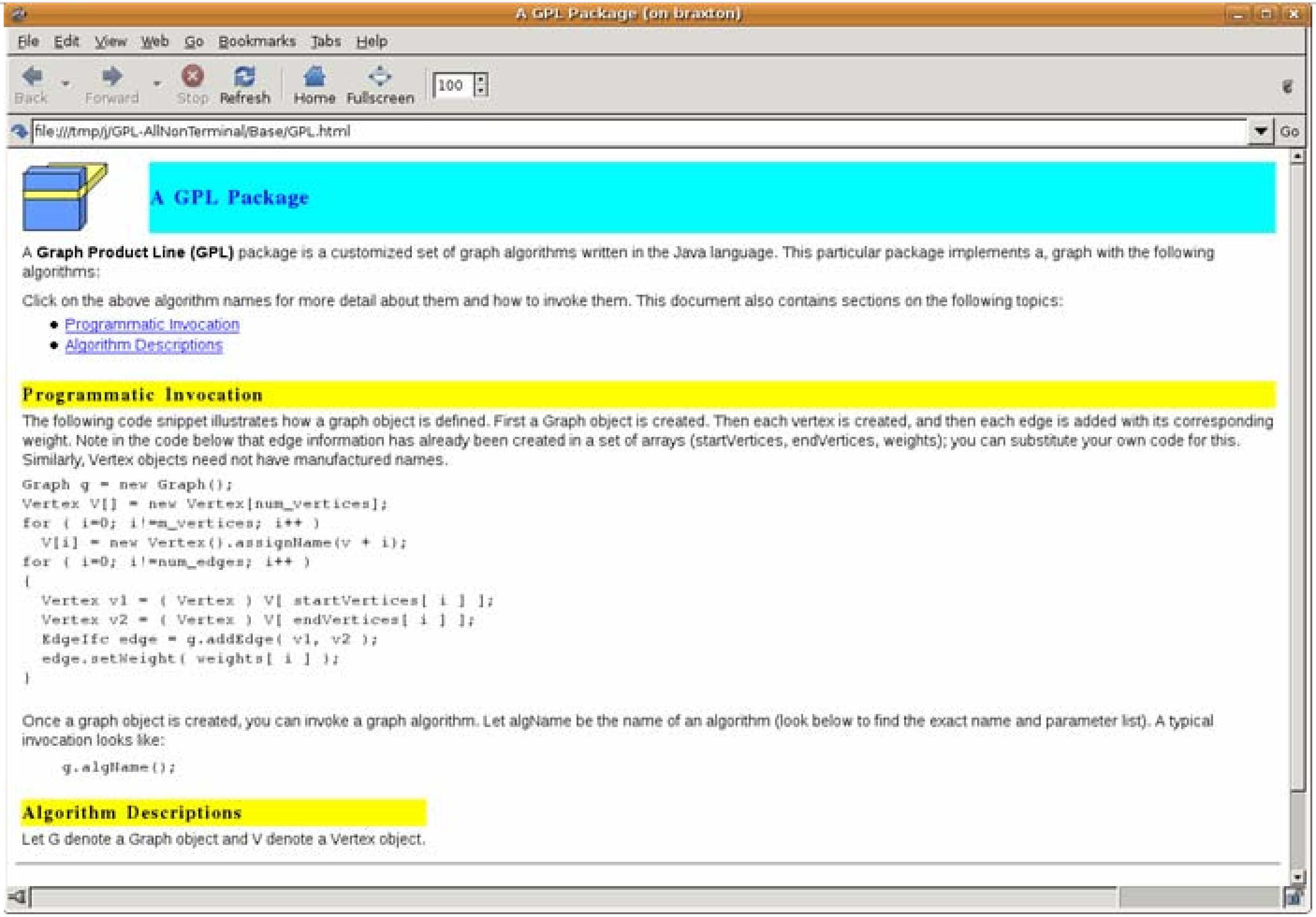

:Client
:Phone
:OutCall
outCall
makeCall
date
time
●
:Phone
:InCall
inCall
acceptCall
date
time
=
:Client
:Phone
:OutCall
:InCall
outCall
makeCall
date
time
inCall
acceptCall
date
time

Some Interesting Examples: UML

- Phone system
  - 2 features, 1004 lines of XMI code
  - Receive incoming and make outgoing calls
- Further XMI/UML case studies
  - Audio control system, conference management system, gas boiler control system (from IKERLAN Research Centre)
- Observations
  - Packages, classes, objects, states, etc. align well with superimposition
  - Problem with extending timelines in sequence diagrams
  - Cannot make fine-grained extensions, e.g., add cardinalities
  - No semantic checks

Some Interesting Examples: XHTML

- Documentation of the graph product line
  - 9 features, 419 lines of (annotated) XHTML code
  - BFS, DFS, Prim, Kruskal, cycle check, number of vertices, connected components, …


A GPL Package (on braxton)
File Edit View Web Go Bookmarks Tabs Help
Back
Forward
Stop
Refresh
Home
Fullscreen
100
file:///tmp/j/GPL-AllNonTerminal/Base/GPL.html
Go

A GPL Package

A **Graph Product Line (GPL)** package is a customized set of graph algorithms written in the Java language. This particular package implements a, graph with the following algorithms:

Click on the above algorithm names for more detail about them and how to invoke them. This document also contains sections on the following topics:

- Programmatic Invocation
- Algorithm Descriptions

Programmatic Invocation

The following code snippet illustrates how a graph object is defined. First a Graph object is created. Then each vertex is created, and then each edge is added with its corresponding weight. Note in the code below that edge information has already been created in a set of arrays (startVertices, endVertices, weights); you can substitute your own code for this. Similarly, Vertex objects need not have manufactured names.

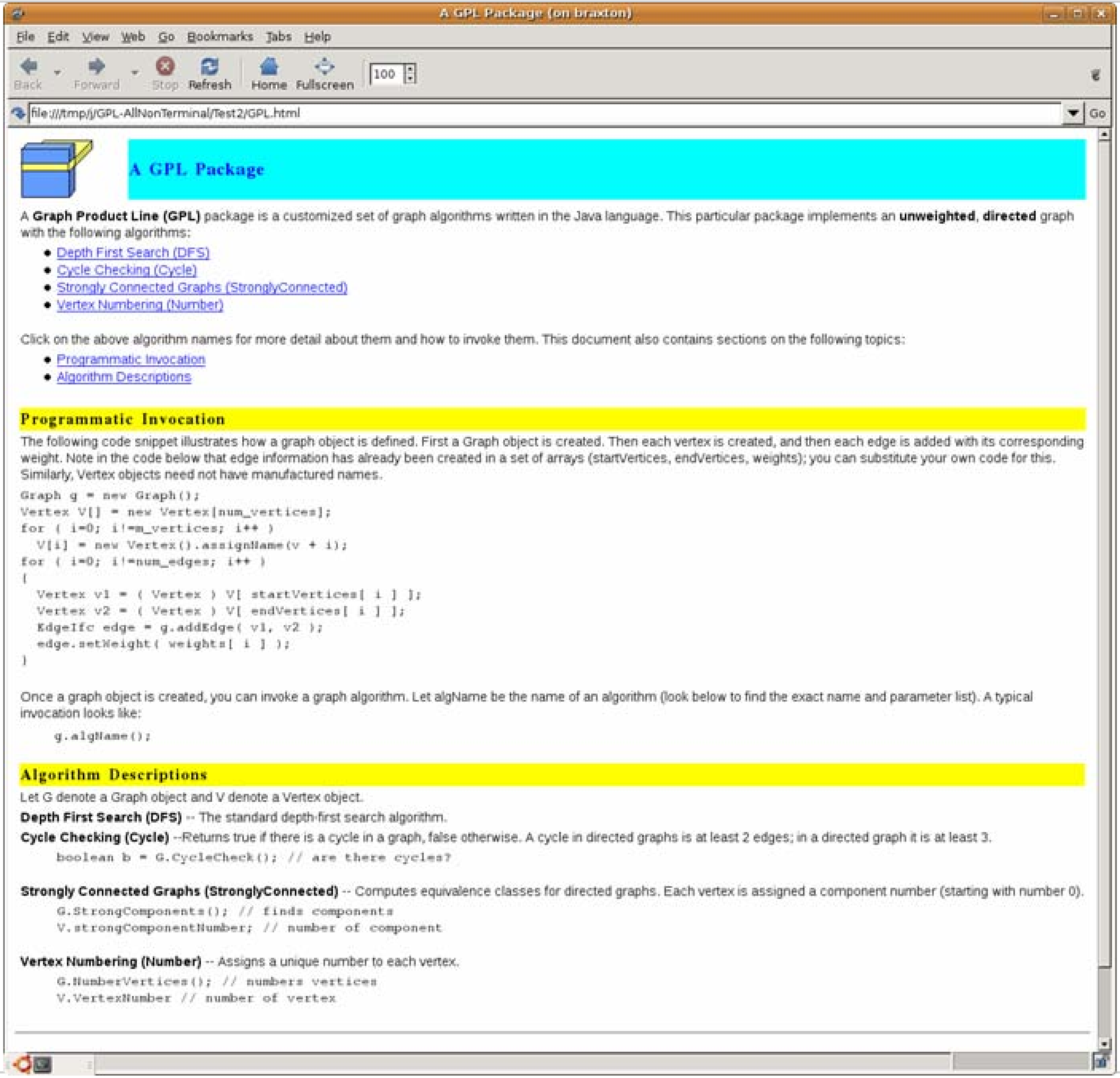
```
Graph g = new Graph();
Vertex V[] = new Vertex[num_vertices];
for ( i=0; i!=m_vertices; i++ )
  V[i] = new Vertex().assignName(v + i);
for ( i=0; i!=num_edges; i++ )
{
  Vertex v1 = ( Vertex ) V[ startVertices[ i ] ];
  Vertex v2 = ( Vertex ) V[ endVertices[ i ] ];
  EdgeIfc edge = g.addEdge( v1, v2 );
  edge.setWeight( weights[ i ] );
}
```

Once a graph object is created, you can invoke a graph algorithm. Let algName be the name of an algorithm (look below to find the exact name and parameter list). A typical invocation looks like:

```
    g.algName();
```

Algorithm Descriptions

Let G denote a Graph object and V denote a Vertex object.



A GPL Package (on braxton)
File Edit View Web Go Bookmarks Tabs Help
Back Forward Stop Refresh Home Fullscreen
100
file:///tmp/j/GPL-AllNonTerminal/Test2/GPL.html
Go

A GPL Package

A **Graph Product Line (GPL)** package is a customized set of graph algorithms written in the Java language. This particular package implements an **unweighted**, **directed** graph with the following algorithms:

- Depth First Search (DFS)
- Cycle Checking (Cycle)
- Strongly Connected Graphs (StronglyConnected)
- Vertex Numbering (Number)

Click on the above algorithm names for more detail about them and how to invoke them. This document also contains sections on the following topics:

- Programmatic Invocation
- Algorithm Descriptions

Programmatic Invocation

The following code snippet illustrates how a graph object is defined. First a Graph object is created. Then each vertex is created, and then each edge is added with its corresponding weight. Note in the code below that edge information has already been created in a set of arrays (startVertices, endVertices, weights); you can substitute your own code for this. Similarly, Vertex objects need not have manufactured names.

```
Graph g = new Graph();
Vertex V[] = new Vertex[num_vertices];
for ( i=0; i!=m_vertices; i++ )
  V[i] = new Vertex().assignName(v + i);
for ( i=0; i!=num_edges; i++ )
{
  Vertex v1 = ( Vertex ) V[ startVertices[ i ] ];
  Vertex v2 = ( Vertex ) V[ endVertices[ i ] ];
  EdgeIfc edge = g.addEdge( v1, v2 );
  edge.setWeight( weights[ i ] );
}
```

Once a graph object is created, you can invoke a graph algorithm. Let algName be the name of an algorithm (look below to find the exact name and parameter list). A typical invocation looks like:

```
    g.algName();
```

Algorithm Descriptions

Let G denote a Graph object and V denote a Vertex object.

Depth First Search (DFS) -- The standard depth-first search algorithm.

Cycle Checking (Cycle) --Returns true if there is a cycle in a graph, false otherwise. A cycle in directed graphs is at least 2 edges; in a directed graph it is at least 3.

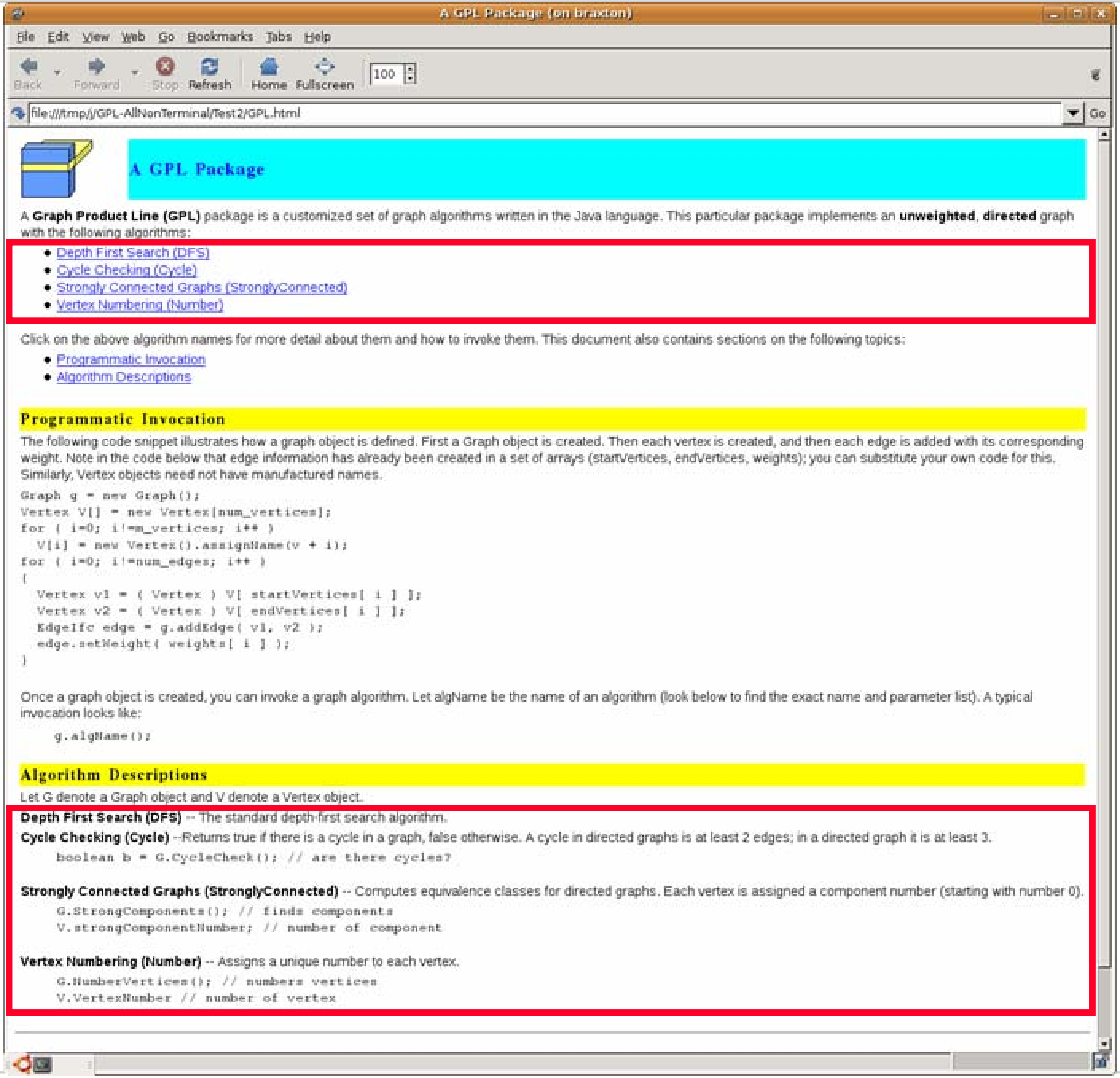
```
    boolean b = G.CycleCheck(); // are there cycles?
```

Strongly Connected Graphs (StronglyConnected) -- Computes equivalence classes for directed graphs. Each vertex is assigned a component number (starting with number 0).

```
    G.StrongComponents(); // finds components
    V.strongComponentNumber; // number of component
```

Vertex Numbering (Number) -- Assigns a unique number to each vertex.

```
    G.NumberVertices(); // numbers vertices
    V.VertexNumber // number of vertex
```


A GPL Package (on braxton)
File Edit View Web Go Bookmarks Tabs Help
Back
Forward
Stop
Refresh
Home
Fullscreen
100
file:///tmp/j/GPL-AllNonTerminal/Test2/GPL.html
Go

A GPL Package

A **Graph Product Line (GPL)** package is a customized set of graph algorithms written in the Java language. This particular package implements an **unweighted**, **directed** graph with the following algorithms:

- Depth First Search (DFS)
- Cycle Checking (Cycle)
- Strongly Connected Graphs (StronglyConnected)
- Vertex Numbering (Number)

Click on the above algorithm names for more detail about them and how to invoke them. This document also contains sections on the following topics:

- Programmatic Invocation
- Algorithm Descriptions

Programmatic Invocation

The following code snippet illustrates how a graph object is defined. First a Graph object is created. Then each vertex is created, and then each edge is added with its corresponding weight. Note in the code below that edge information has already been created in a set of arrays (startVertices, endVertices, weights); you can substitute your own code for this. Similarly, Vertex objects need not have manufactured names.

```
Graph g = new Graph();
Vertex V[] = new Vertex[num_vertices];
for ( i=0; i!=m_vertices; i++ )
  V[i] = new Vertex().assignName(v + i);
for ( i=0; i!=num_edges; i++ )
{
  Vertex v1 = ( Vertex ) V[ startVertices[ i ] ];
  Vertex v2 = ( Vertex ) V[ endVertices[ i ] ];
  EdgeIfc edge = g.addEdge( v1, v2 );
  edge.setWeight( weights[ i ] );
}
```

Once a graph object is created, you can invoke a graph algorithm. Let algName be the name of an algorithm (look below to find the exact name and parameter list). A typical invocation looks like:

```
    g.algName();
```

Algorithm Descriptions

Let G denote a Graph object and V denote a Vertex object.

Depth First Search (DFS) -- The standard depth-first search algorithm.

Cycle Checking (Cycle) --Returns true if there is a cycle in a graph, false otherwise. A cycle in directed graphs is at least 2 edges; in a directed graph it is at least 3.

```
    boolean b = G.CycleCheck(); // are there cycles?
```

Strongly Connected Graphs (StronglyConnected) -- Computes equivalence classes for directed graphs. Each vertex is assigned a component number (starting with number 0).

```
    G.StrongComponents(); // finds components
    V.strongComponentNumber; // number of component
```

Vertex Numbering (Number) -- Assigns a unique number to each vertex.

```
    G.NumberVertices(); // numbers vertices
    V.VertexNumber // number of vertex
```

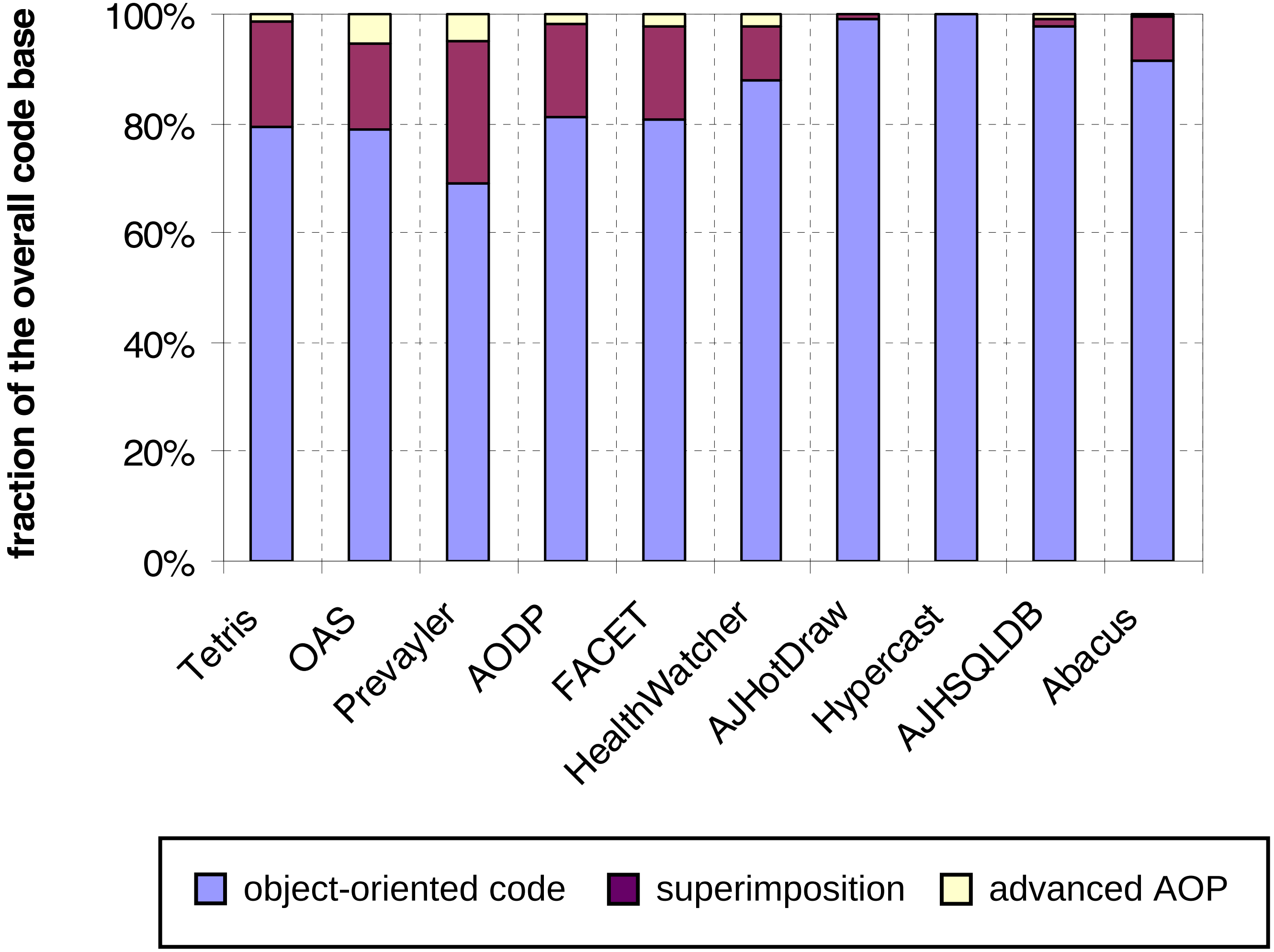
Some Interesting Examples: XHTML

- Documentation of the graph product line
  - 9 features, 419 lines of (annotated) XHTML code
  - BFS, DFS, Prim, Kruskal, cycle check, number of vertices, connected components, …
- Observations
  - Superimposition of XHTML documents is possible ➔ lists, sections, headers, tables, etc. are the non-terminals
  - Need to assign unique names to XHTML elements that are to be extended, e.g., to add an item to a list
  - Lexical order matters ➔ cannot add element in the middle of other elements

Analyzed AspectJ Projects

Tetris	The popular game	1 KLOC	Blekinge Institute of Technology
OAS	An online auction system	2 KLOC	Lancaster University
Prevayler	Transparent persistence for Java	4 KLOC	University of Toronto
AODP	Aspect-oriented implementation of the Gang-of-Four design patterns	4 KLOC	University of British Columbia
FACET	An aspect-based CORBA event channel	6 KLOC	Washington University
HealthWatcher	Web-based information system for public health systems	7 KLOC	Lancaster University
AJHotDraw	2D graphics framework	22 KLOC	Sourceforge project
Hypercast	Multicast overlay network communication	67 KLOC	University of Virginia, Microsoft
AJHSQLDB	SQL relational database engine	76 KLOC	University of Passau
Abacus	A CORBA middleware framework	130 KLOC	University of Toronto

Superimposition vs. Advanced AOP



Supporting Superimposition

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

●

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

Hyper/J

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

●

```
class Backup<T extends Base> extends T {
  class Counter extends T.Counter {
    int back = 0;
    void inc() { back=val; super.inc(); }
    void restore() { val=back; }
  }
}
```

Java Layers

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

●

```
cclass Backup extends Base {
  cclass Counter {
    int back = 0;
    void inc() { back=val; val++; }
    void restore() { val=back; }
  }
}
```

CaesarJ