Recent Developments in Feature-Oriented Software Development

Sven Apel
Chair for Programming
University of Passau, Germany

Department of
Informatics and
Mathematics

Programming
Group

A Joint Effort

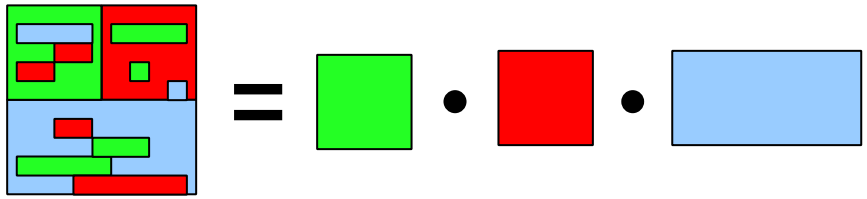
- Don Batory (University of Texas at Austin)
- DeLesley Hutchins (University of Edinburgh)
- Christian Kästner (University of Magdeburg)
- Martin Kuhlemann (University of Magdeburg)
- Thomas Leich (Metop Inc.)
- Christian Lengauer (University of Passau)
- Roberto Lopez-Herrejon (Oxford University)
- Bernhard Möller (University of Augsburg)
- Marko Rosenmüller (University of Magdeburg)
- etc.

Feature-Oriented Software Development

- What is a feature?
  - Increment in program functionality
  - Implements a requirement
  - Provides a configuration option
  - Represents a domain concept
  - Is used to distinguish programs of a product line
- Idea: represent features explicitly in design and code
  - Each feature is encapsulated in a module
  - A feature refines a (possibly empty) program
  - A final program is composed of a number of features

Feature Decomposition and Composition

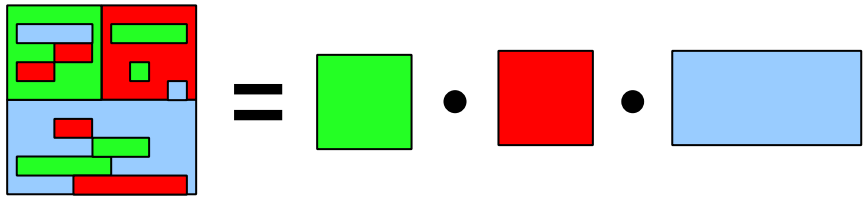
Legacy Program



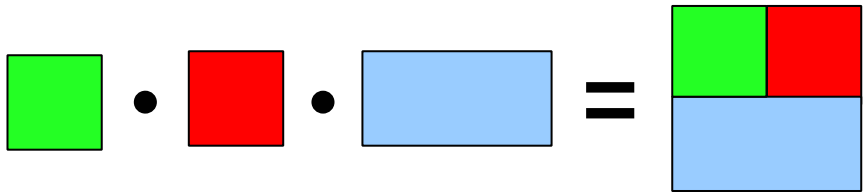
Set of Features and Rules

Feature Decomposition and Composition

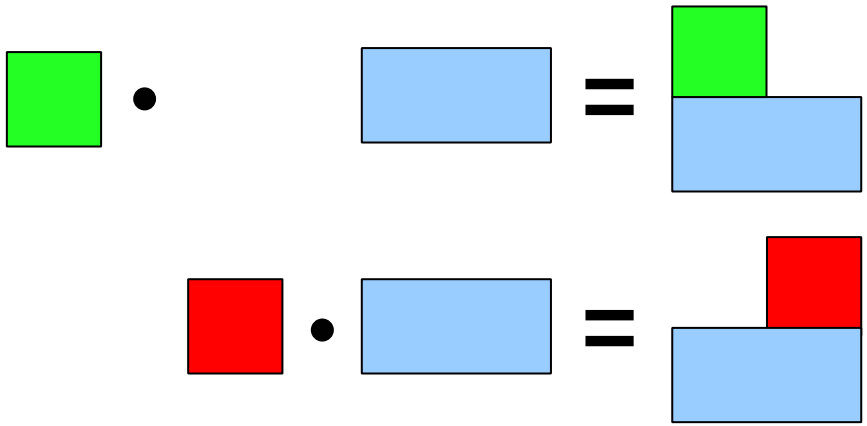
Legacy Program



Set of Features and Rules



Software Product Line



Tailored Program Variants

Challenges

- How untangle code of different features?
- How to compose code of different features?
- How to refactor legacy software systems into features?
- How to ensure correctness of feature composition and refactoring?
- How to represent and manage features at different stages of software development?
- How to incorporate different kinds of software artifacts?
- How to reason about features formally?
- …

Agenda

- Feature modularity through aspectual feature modules
- Superimposition as language-independent feature composition technique
- Feature algebra, as formal foundation for features
- Type systems for feature composition
- Feature analysis and decomposition with Colored IDE

Agenda

- Feature modularity through aspectual feature modules
- Superimposition as language-independent feature composition technique
- Feature algebra, as formal foundation for features
- Type systems for feature composition
- Feature analysis and decomposition with Colored IDE

Features are Crosscutting Concerns

ApplicationSession

StandardSession

ServerSession

- Example: Session expiration in the Apache Tomcat Server

SessionInterceptor

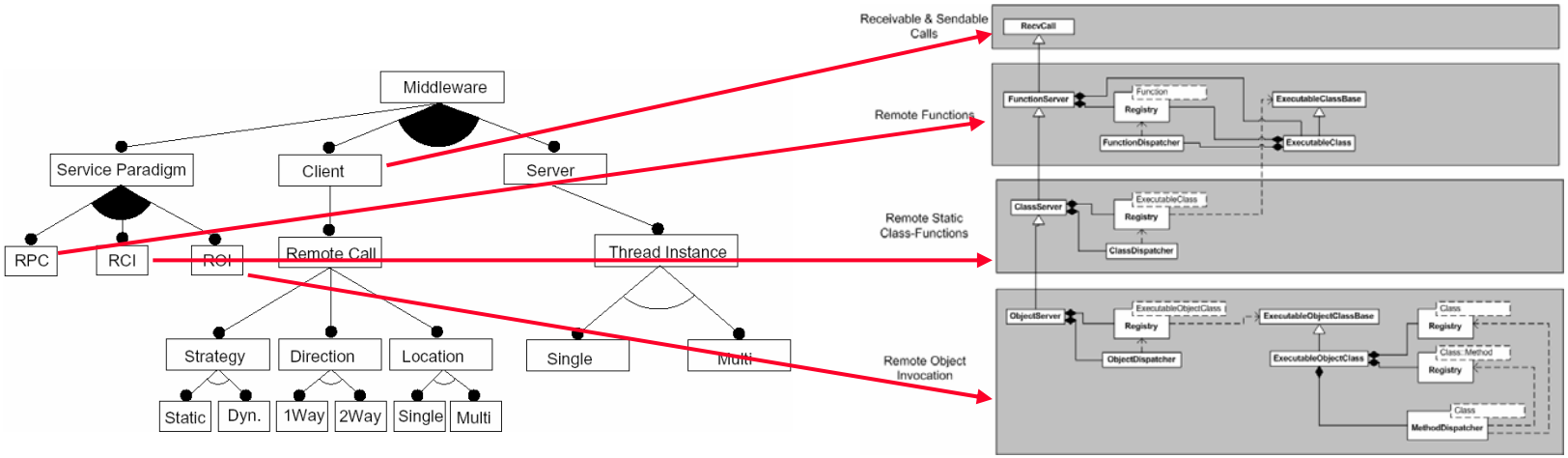
ServerSessionManager

StandardManager

StandardSessionManager

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

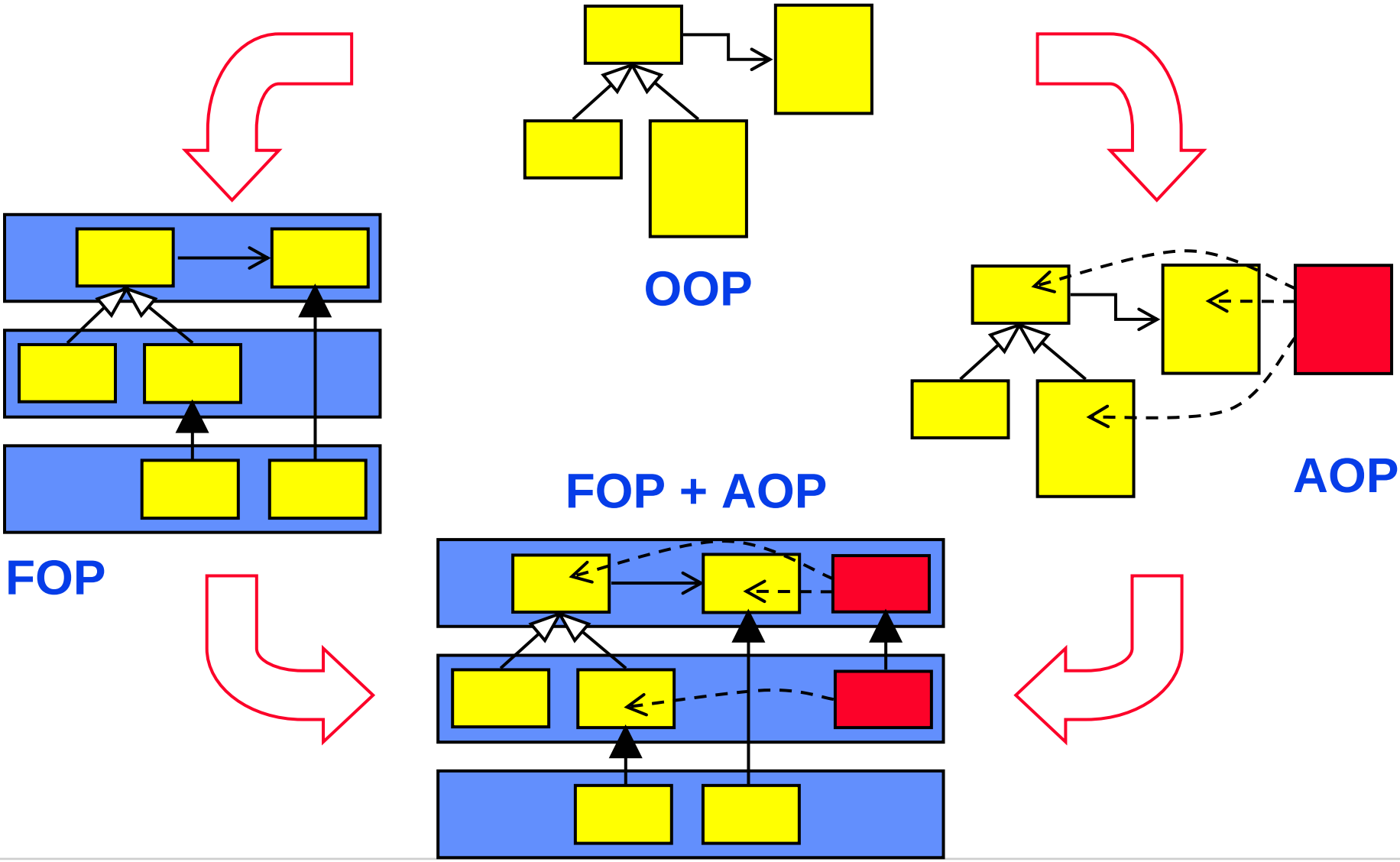
Feature Modularity



Two Programming Paradigms

	FOP	AOP
static	***good support*** – fields, methods, classes	***limited support*** – fields, methods
dynamic	***weak support*** – basic dynamic (method extensions)	***good support*** – advanced dynamic
heterogeneous	***good support*** – refinements and collaborations	***limited support*** – no explicit collaborations
homogeneous	***no support*** – one refinement per join point (code replication)	***good support*** – wildcards and enumerated pointcuts

A Symbiosis of FOP and AOP



Agenda

- Feature modularity through aspectual feature modules
- Superimposition as language-independent feature composition technique
- Feature algebra, as formal foundation for features
- Type systems for feature composition
- Feature analysis and decomposition with Colored IDE

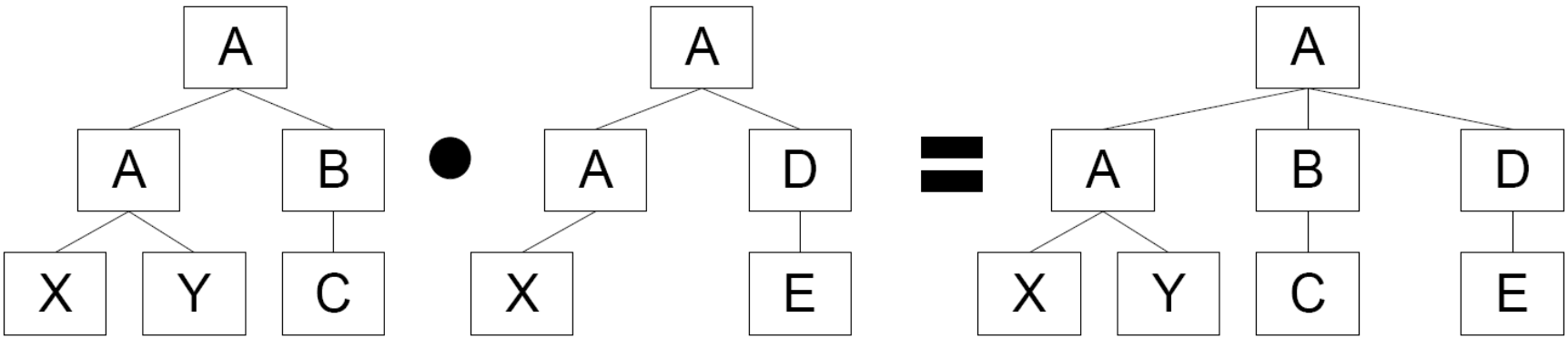
An Observation

- Numerous languages provide abstraction and modularity mechanisms for features
  - CaesarJ, Classbox/J, ContextL, FeatureC++, Hyper/J, Jak, Jiazzi, Lasagne/J, ObjectTeams/J, Scala
- Despite all differences there is a common pattern

  ➔ **Superimposition**

Superimposition

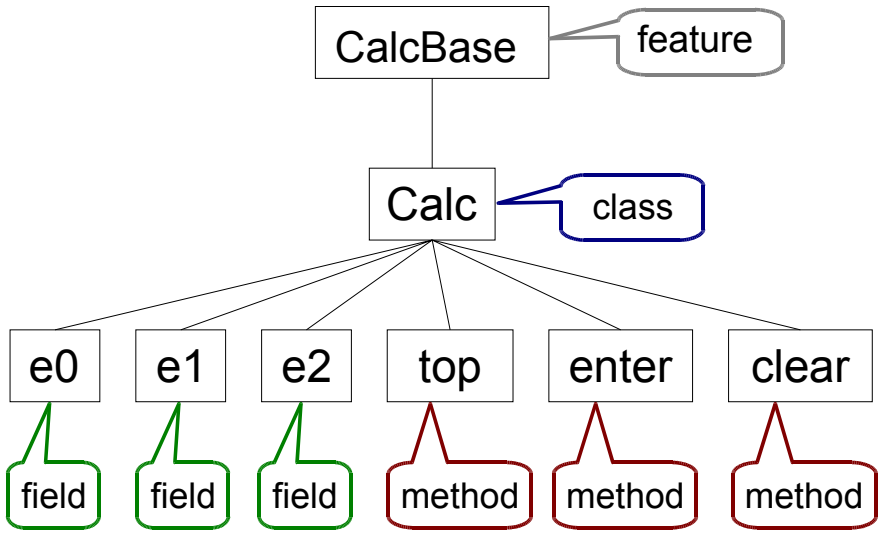
- Informally,
  - ...the composition of software components (features)
  - ...by merging their corresponding substructures hierarchically
  - ...by matching name, type, and relative position



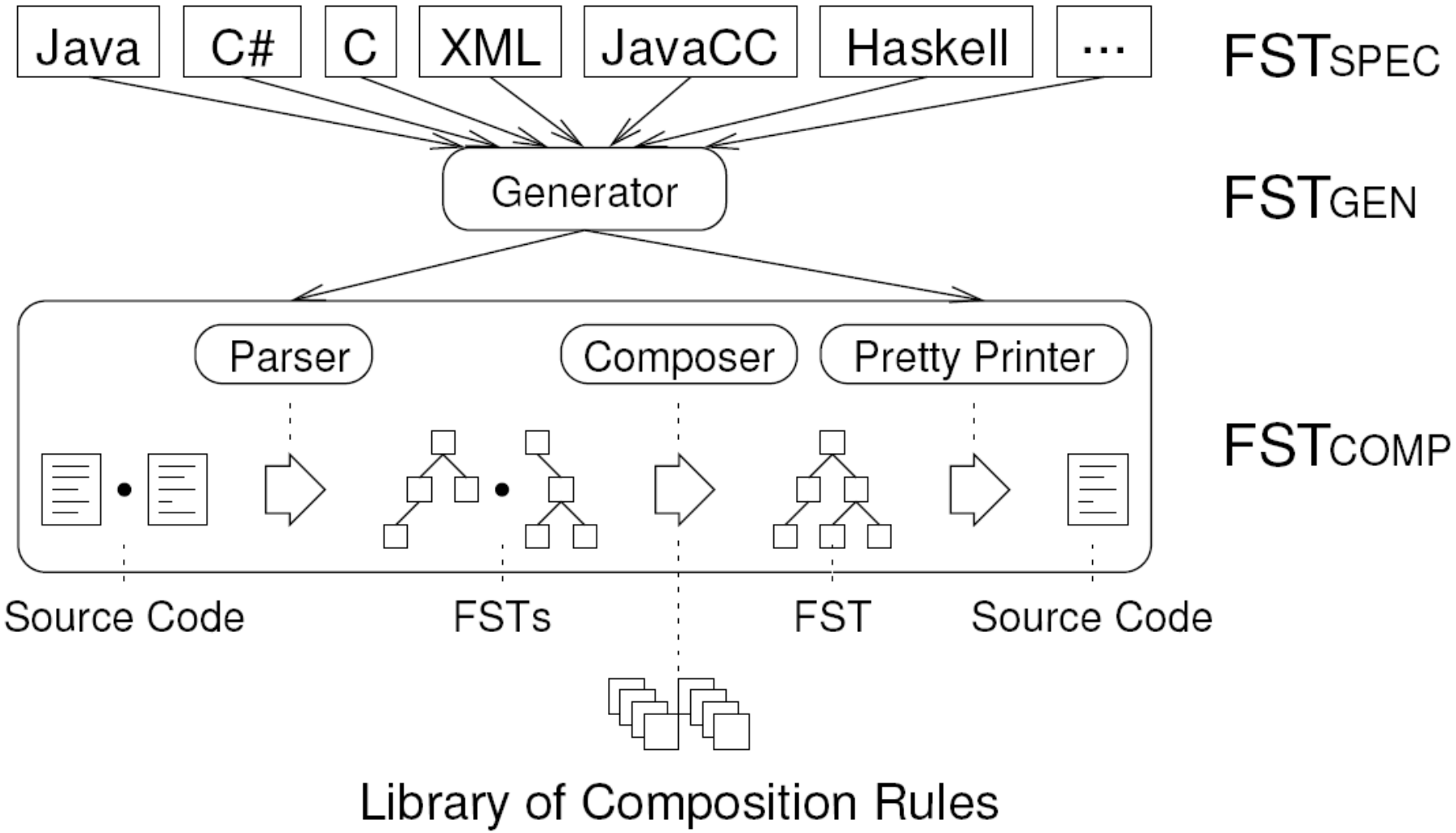
Feature F Feature G Feature F • G

An Idea

- Capture the essential properties of superimposition in a model (**Feature Structure Tree**) and tool (**FSTComposer**)
  - Language independence
  - Understand the principles of feature composition
  - Plugging in the language of your choice



Tool Chain


Java
C#
C
XML
JavaCC
Haskell
...
FSTSPEC
Generator
FSTGEN
Parser
Composer
Pretty Printer
FSTCOMP
Source Code
FSTs
FST
Source Code
Library of Composition Rules

Case Studies

- Supported languages: Java, C#, C, XML, JavaCC

	FEA	CLA	LOC	TYP	TIM	Description
FFJ	2	–	289	JavaCC	< 1 s	Feature Featherweight Java Grammar [4]
GraphLib	13	–	934	C	1 s	Low-level graph library[a]
GPL	26	57	2,439	Java, XML	2 s	Graph Product Line (Java Version) [29]
GPL	20	55	2,148	C#	2 s	Graph Product Line (C# Version)
Violet	88	157	9,660	Java, Text	7 s	Configuration management tool [9]
GUIDSL	26	294	13,457	Java	10 s	UML Editor[b]
Berkeley DB	99	765	58,030	Java	24 s	Oracle's Embedded Storage Engine [25]

FEA: number of features; CLA: number of classes; LOC: lines of code; TYP: types of contained artifacts; TIM: time to compose

[a] http://keithbriggs.info/graphlib.html

[b] http://www.horstmann.com/violet/

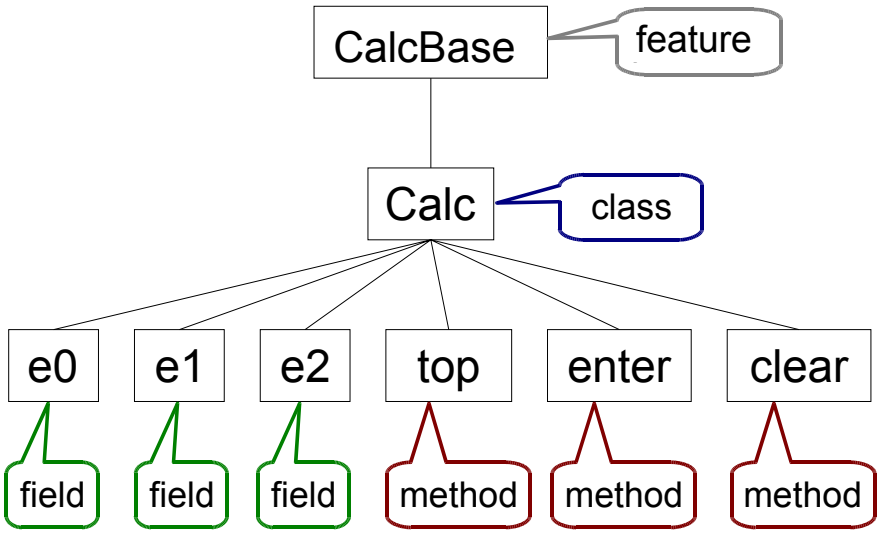
Agenda

- Feature modularity through aspectual feature modules
- Superimposition as language-independent feature composition technique
- Feature algebra, a formal foundation for features
- Type systems for feature composition
- Feature analysis and decomposition with Colored IDE

Feature Algebra

- Formal reasoning about feature composition
  - Abstracts from details of programming languages
  - Alternatives in the algebra are alternatives in programming language mechanisms
  - Not only useful for the description of composition, e.g., for feature interaction analysis
  - Architectural metaprogramming

Superimposition is Introduction Sum



$$F = CalcBase \oplus CalcBase.Calc \oplus CalcBase.Calc.e0 \oplus CalcBase.Calc.e1 \oplus CalcBase.Calc.e2 \oplus CalcBase.Calc.enter \oplus CalcBase.Calc.clear \oplus CalcBase.Calc.top$$

Quantification and Weaving is Modification Application

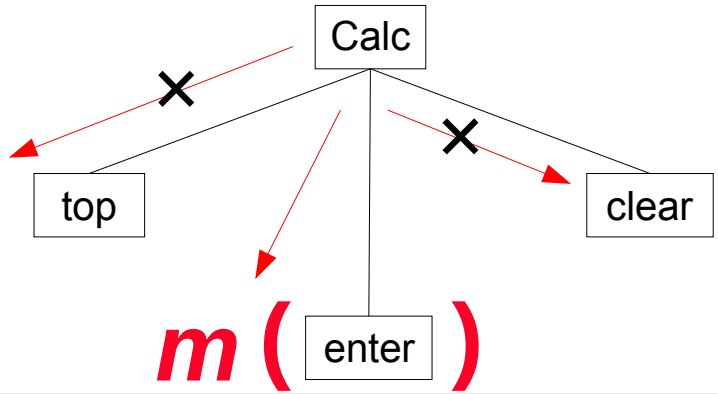
Modification Application

$$m(Calc \oplus Calc.top \oplus Calc.enter \oplus Calc.clear)$$

Quantification

$$m(Calc) \oplus m(Calc.top) \oplus m(Calc.enter) \oplus m(Calc.clear)$$

Weaving



Agenda

- Feature modularity through aspectual feature modules
- Superimposition as language-independent feature composition technique
- Feature algebra, as formal foundation for features
- Type systems for feature composition
- Feature analysis and decomposition with Colored IDE

State of the Art

- Current FOP languages and tools do not care much about type safety
- Usually, feature code is translated to a lower-level representation, e.g., Jak to Java or FeatureC++ to C++
- Typing is postponed to the target language compiler
  - Information is lost for typing and debugging
- Solution: type systems for FOP languages
  - FFJ: a case study extending FJ by features
  - gDeep: a language-independent type system framework

Feature Featherweight Java (FFJ)

- Extending FJ toward FOP

CD ::= *class declarations:*
class C extends C { $\overline{C}$ $\overline{f}$; KD $\overline{MD}$ }

CR ::= *class refinements:*
refines class C { $\overline{C}$ $\overline{f}$; KR $\overline{MD}$ $\overline{MR}$ }

KD ::= *constructor declarations:*
C($\overline{D}$ $\overline{g}$, $\overline{C}$ $\overline{f}$) { super($\overline{g}$); this.$\overline{f}$=$\overline{f}$; }

KR ::= *constructor refinements:*
refines C($\overline{E}$ $\overline{h}$, $\overline{C}$ $\overline{f}$) { original($\overline{h}$); this.$\overline{f}$=$\overline{f}$; }

MD ::= *method declarations:*
C m($\overline{C}$ $\overline{x}$) { return t; }

MR ::= *method refinements:*
refines C m($\overline{C}$ $\overline{x}$) { return t; }

t ::= *terms:*
x *variable*
t.f *field access*
t.m($\overline{t}$) *method invocation*
new C($\overline{t}$) *object creation*
(C) t *cast*

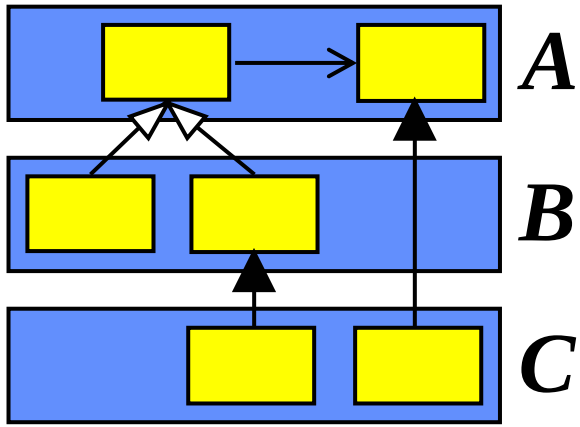
v ::= *values:*
new C($\overline{v}$) *object creation*

Extensions of FFJ

- *ME* — method extension
- *DV* — default values
- *SD* — superclass declaration
- *BR* — backward references

Extensions of FFJ

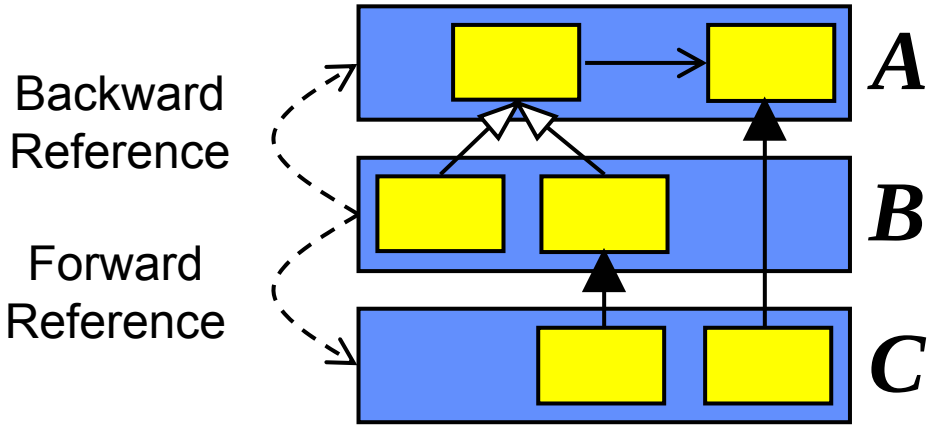
- *ME* — method extension
- *DV* — default values
- *SD* — superclass declaration
- *BR* — backward references



$$P = C\,(\,B\,(\,A\,))$$

Extensions of FFJ

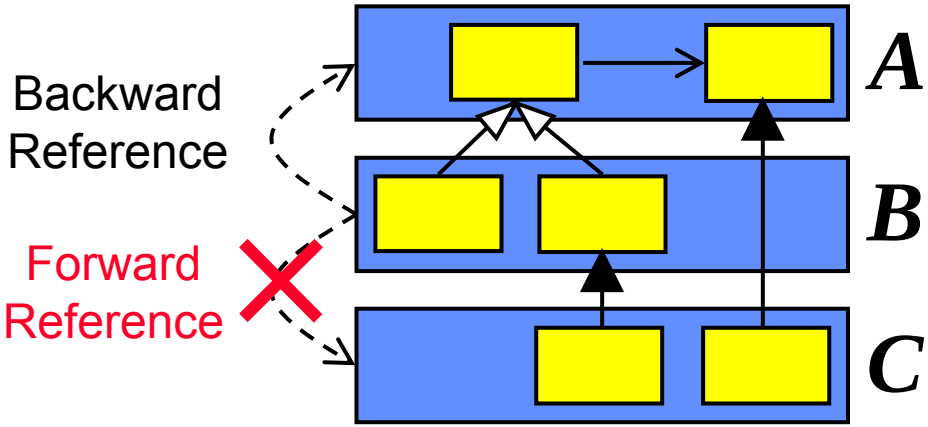
- *ME* — method extension
- *DV* — default values
- *SD* — superclass declaration
- *BR* — backward references



$$P = C\,(\,B\,(\,A\,))$$

Extensions of FFJ

- *ME* — method extension
- *DV* — default values
- *SD* — superclass declaration
- *BR* — backward references



$$P = C\,(\,B\,(\,A\,))$$

Extensions of FFJ

- *ME* — method extension
- *DV* — default values
- *SD* — superclass declaration
- *BR* — backward references

	FJ	FFJ	FFJ$_{ME}$	FFJ$_{DV}$	FFJ$_{SD}$	FFJ$_{BR}$
Java	√					
Jak$_1$	√	√	√[a]	√[b]		
Jak$_2$	√	√	√	√[b]		√[d]
FSTComposer	√	√	√	√[b]	√[c]	

gDeep

- Language-independent module system
  - Feature module ⇔ record
  - Feature composition ⇔ recursive record superimposition

$$\lambda^{+} X \leq A.\ \mu Y \text{ refines } X\{...\}$$

- Subtyping laws for features

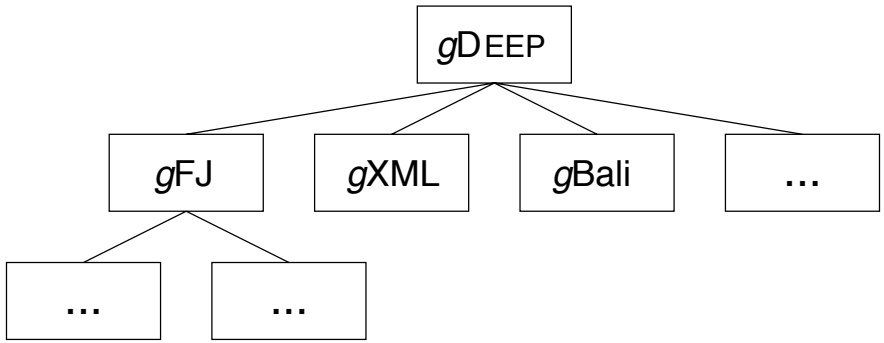
$$\forall A.\ F(A) \leq A$$

$$\forall A, B.\ A \leq B \text{ implies } F(A) \leq F(B)$$

$$F_1(F_2(...F_n(A))) \ \leq \ G_1(G_2(...G_m(A)))$$

gDeep

- Type system based on Deep (Hutchins, OOPSLA'06)
  - Combination of nominal and structural subtyping
  - Framework for plugging in sister calculi ➔ hierarchical type system
    - gDeep ➔ module checking
    - Sister calculus ➔ term checking
    - Currently supported: FJ, XML, Bali
  - Benefit: do not need to extend each artifact language!



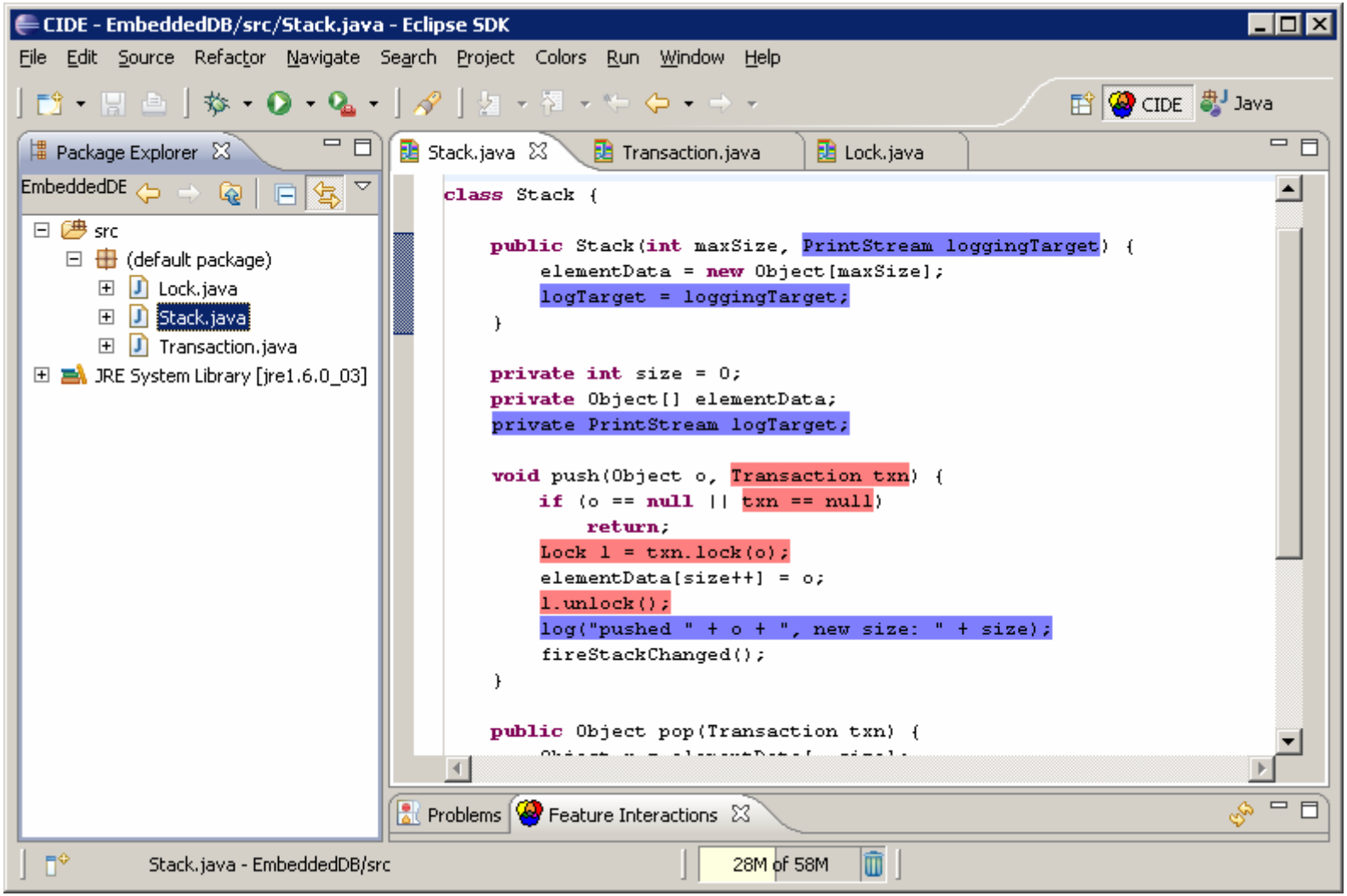
Agenda

- Feature modularity through aspectual feature modules
- Superimposition as language-independent feature composition technique
- Feature algebra, as formal foundation for features
- Type systems for feature composition
- Feature analysis and decomposition with Colored IDE

Observation

- Granularity of feature composition is sometimes too coarse
  - Extending a sequence of statements
  - Extending the signature of a method
  - Extending expressions and control structures
- A solution: preprocessors, frames, annotations?
  - Pollute source code
  - Error-prone

Our Approach: Colored IDE (CIDE)



Projection and Generation

```
public class Stack {

    public Stack(int maxSize, StatisticObject s, PrintStream loggingTarget) {
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;

    private Object[] elementData;

    private PrintStream logTarget;

    public boolean push(Object o) {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for " + o);
            return false;
        }
        elementData[size++] = o;
        log("pushed " + o + ", new size: " + size);
        unlock(lock);
        return true;
    }

    public Object pop() {
        Lock lock = lock();
        if (lock == null) {
            log("lock failed for pop");
            return null;
        }
        Object r = elementData[--size];
        unlock(lock);
        return r;
    }

    private void log(String msg) {
        // .....
        //...............
        logTarget.println(msg);
    }

    private void unlock(Lock lock) {
        // .....
        //...............
        // .....
        //...............
    }

    private Lock lock() {
        // ...
        // .....
        //...............
        // .....
        //...............
        // .....
        //...............
        return new Lock();
    }
```

Show/Hide

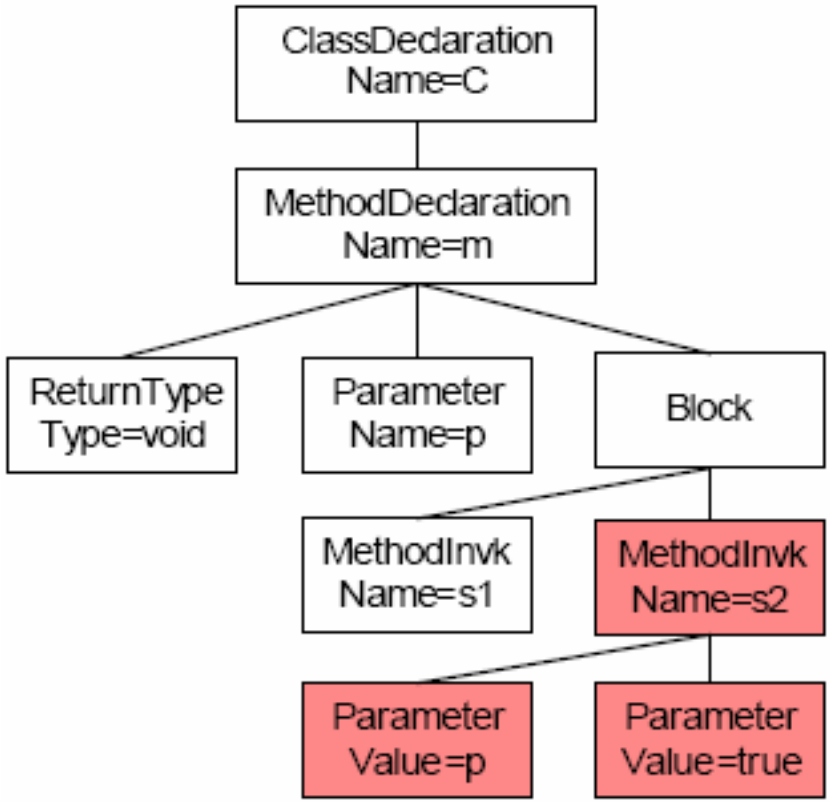
Generate (Product or entire SPL)

```
import java.io.PrintStream;

public class Stack {

    public Stack(int maxSize, StatisticObject s, PrintStream loggingTarget) {
        elementData = new Object[maxSize];
        logTarget = loggingTarget;
    }

    private int size = 0;

    private Object[] elementData;

    private PrintStream logTarget;

    public boolean push(Object o) {
        elementData[size++] = o;
        log("pushed " + o + ", new size: " + size);
        return true;
    }

    public Object pop() {
        Object r = elementData[--size];
        return r;
    }

    private void log(String msg) {
        // .....
        //...............
        logTarget.println(msg);
    }

}
```

Safe Decomposition

- Features are assigned to structural elements (to optional AST-subtrees)
- AST hidden from developer, mapped by CIDE

```
class C {
   void m(int p) {
      s1();
      s2(p, true);
   }
}
```



Dualities

- Attribute grammar-based integration of new languages
  - FSTComposer ⇔ CIDE
- Type system for colored Java code
  - FFJ ⇔ CFJ

- S. Apel, T. Leich, and G. Saake. **Aspectual Feature Modules**. *IEEE Trans. Softw. Eng. (TSE)*, 34(2), 2008.

Aspectual Feature Modules

- S. Apel and C. Lengauer. **Superimposition: A Language-Independent Approach to Software Composition**. In *Proc. Int. Symp. Software Composition (SC)*, 2008.

Superimposition

- S. Apel, C. Lengauer, B. Möller, and C. Kästner. **An Algebra for Features and Feature Composition**. In *Proc. Int. Conf. Algebraic Methodology and Software Technology (AMAST)*, 2008.

Feature Algebra

- S. Apel, C. Kästner, and C. Lengauer. **An Overview of Feature Featherweight Java**. Technical Report MIP-0802, University of Passau, 2008.
- S. Apel and D. Hutchins. **An Overview of the gDeep Calculus**. Technical Report MIP-0712, University of Passau, 2007.

Type Systems

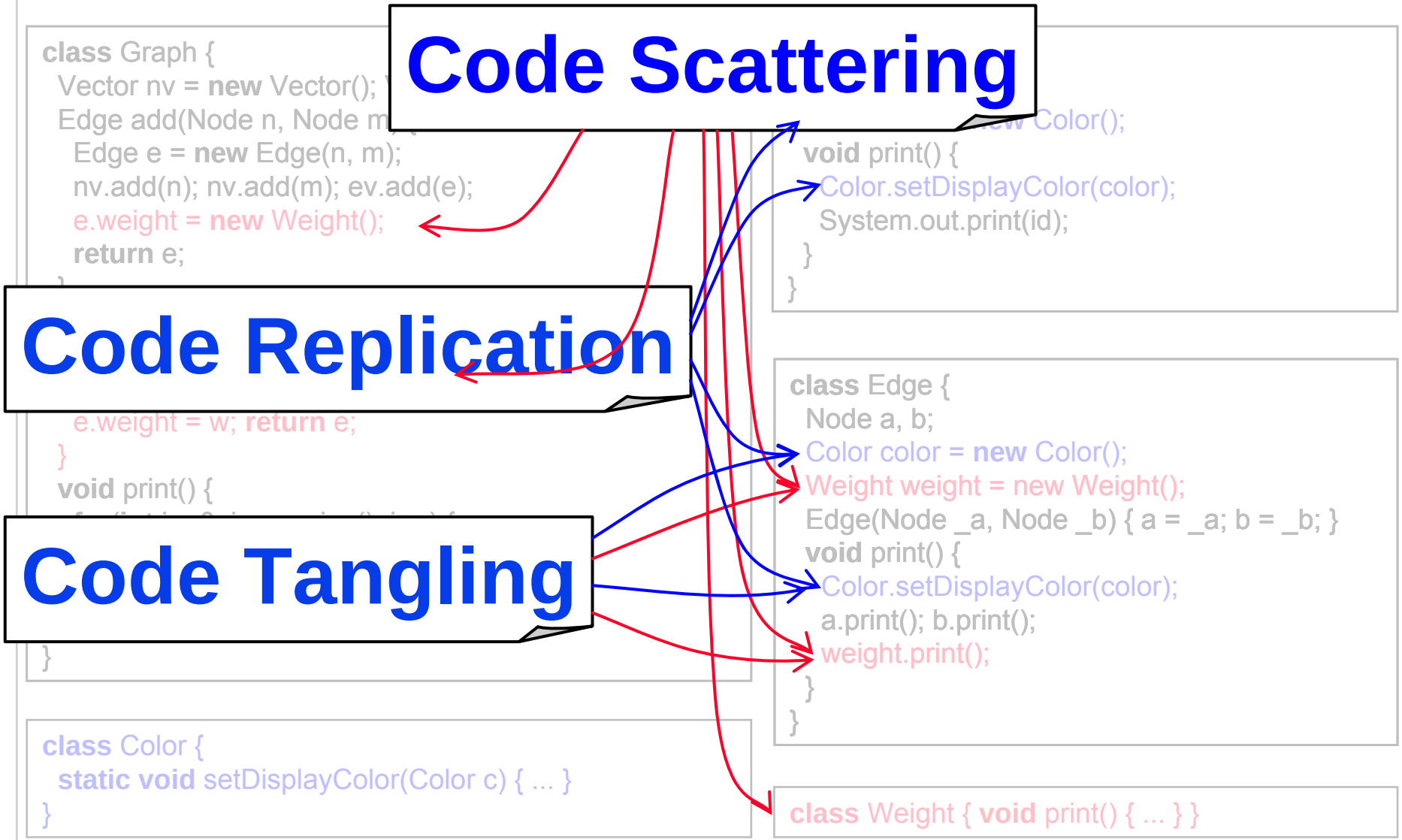
- C. Kästner, S. Apel, and M. Kuhlemann. **Granularity in Software Product Lines**. In *Proc. Int. Conf. Software Engineering (ICSE)*, 2008.
- C. Kästner and S. Apel. **Type-checking Software Product Lines – A Formal Approach**. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, 2008.

CIDE

The Big Picture

- The Standish Group Chaos Reports
  - 16% of all software projects were successful (1995)
  - 34% of all software projects were successful (2003)
- Software crisis and software engineering
  - 1st NATO Software Engineering Conference (1968)
  - Software Engineering Institute, ICSE, ESEC, FSE, …
- Ways out of the crisis ➔ fundamental principles
  - Modularity and separation of concerns
  - Stepwise refinement / development
  - …

An Introductory Example



Crosscutting Concerns

- *"A concern is an area of interest or focus in a system. Concerns are the primary criteria for decomposing software into smaller, more manageable and comprehensible parts that have meaning to a software engineer."* (AOSD.NET Glossary)

- *"Crosscutting (is) a structural relationship between representations of a concern. In this way it is similar to other kinds of structure, like hierarchical structure and block structure."* (AOSD.NET Glossary)

Is There a Software Crisis?

[…] only about 16% of software projects were successful, 53% were fraught with problems (cost or budget overruns, content deficiencies), and 31% were cancelled; the average software project ran 222% late, 189% over budget and delivered only 61% of the specified functions.

– Chaos Report, Standish Group 1995

[…] only about 34% of all software projects were deemed to be successful.

– Chaos Report, Standish Group 2003

Feature Modules à la Jak

Basic Graph

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

Weight

```
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);
    nv.add(n); nv.add(m); ev.add(e);
    e.weight = w; return e;
  }
}
```

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

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

Aspects à la AspectJ

Basic Graph

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

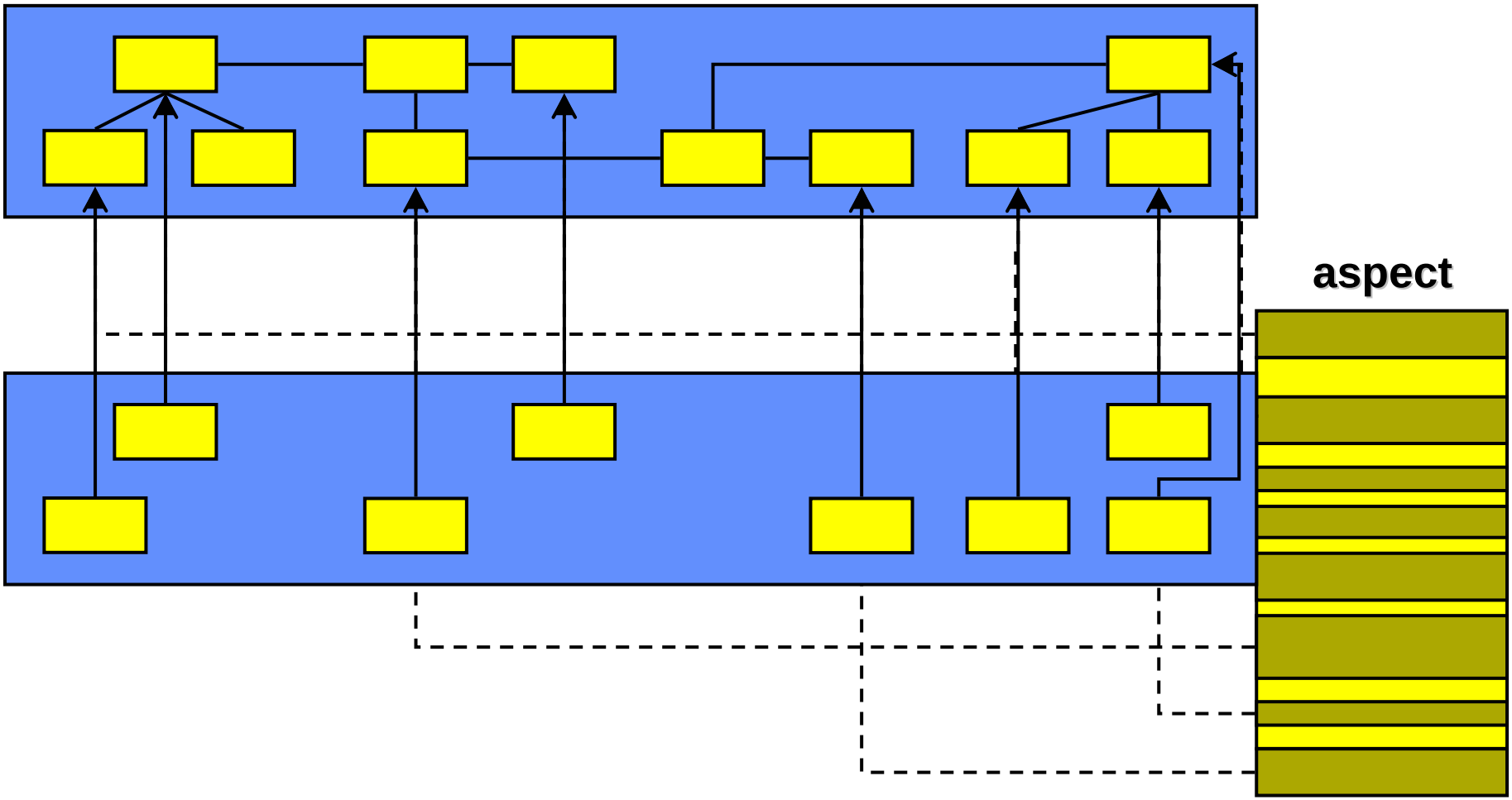
```
class Edge {
  Node a, b;
  Edge(Node _a, Node _b) {
    a = _a; b = _b;
  }
  void print() {
    a.print(); b.print();
  }
}
```

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

Color

```
aspect ColorAspect {
  interface Colored { ... }
  declare parents: (Node || Edge) implements Colored;
  Color (Node || Edge).color = new Color();
  before(Colored c) : execution(void print()) && this(c) {
    Color.setDisplayColor(c.color);
  }
  static class Color { ... }
}
```

AOP versus FOP



Tyranny of the Dominant Decomposition

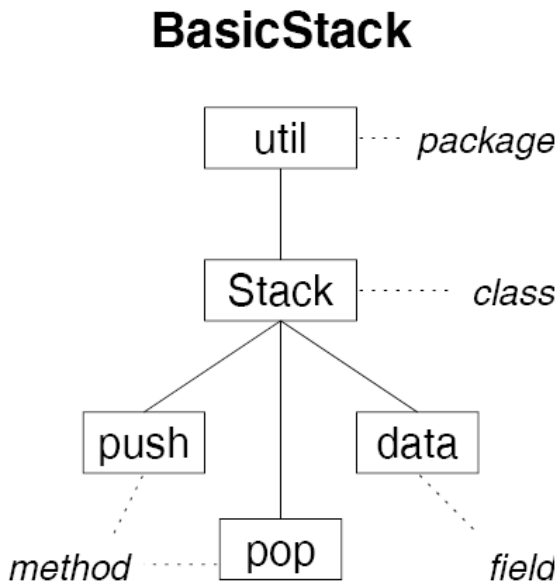
- *[a] program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another.*

- *Multi-dimensional separation of concerns is aimed at breaking the tyranny, allowing separation of all kinds of concerns of importance simultaneously, including overlapping, interacting and crosscutting concerns.*

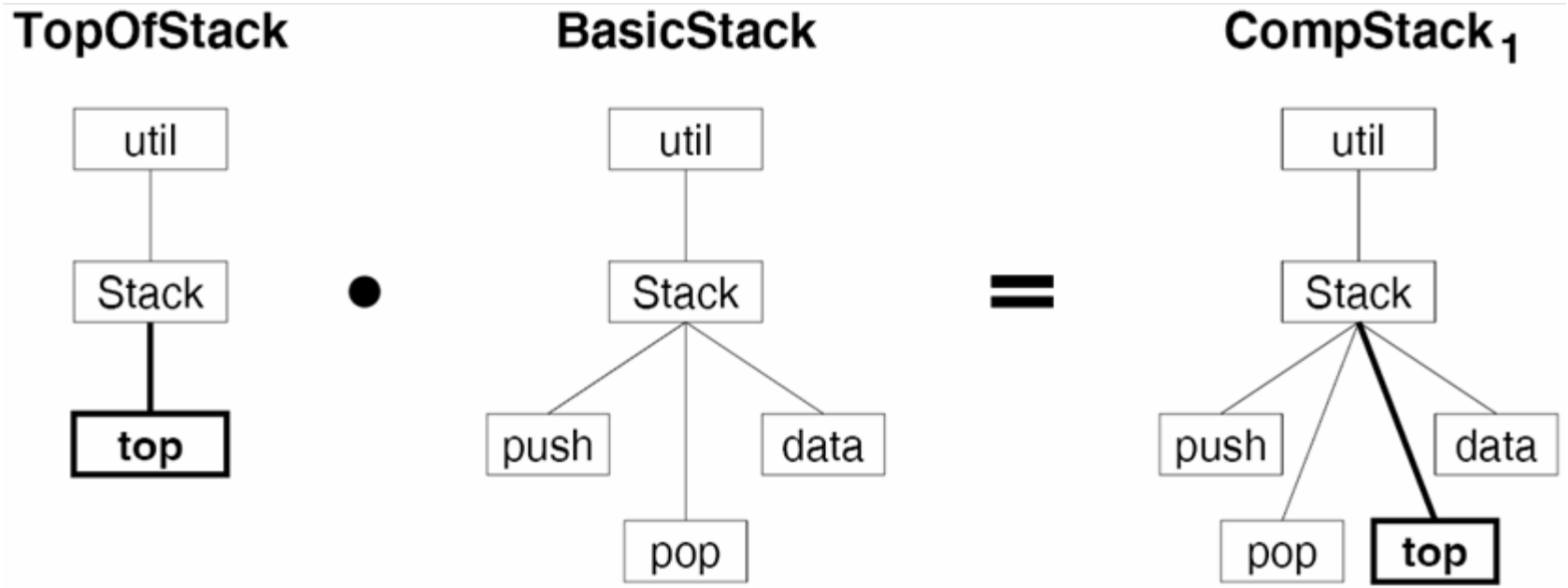
-- Software by Composition group at the IBM T.J. Watson Research Center

Example: FSTComposer plus Java

```
1  package util;
2  class Stack {
3    LinkedList data = new LinkedList();
4    void push(Object obj) {
5      data.addFirst(obj);
6    }
7    Object pop() {
8      return data.removeFirst();
9    }
10 }
```



Example: FSTComposer plus Java



Example: FSTComposer plus Java

```
1 package util;
2 class Stack {
3   Object top() { return data.getFirst(); }
4 }
```

```
1 package util;
2 class Stack {
3   LinkedList data = new LinkedList();
4   void push(Object obj) { data.addFirst(obj); }
5   Object pop() { return data.removeFirst(); }
6 }
```

```
1 package util;
2 class Stack {
3   LinkedList data = new LinkedList();
4   void push(Object obj) { data.addFirst(obj); }
5   Object pop() { return data.removeFirst(); }
6   Object top() { return data.getFirst(); }
7 }
```

Example: FFJ

```
1  class Expr extends Object {          // Feature Add ...
2    Expr() { super(); }
3  };
4  class Add extends Expr {
5    int a; int b;
6    Add(int a, int b) { super(); this.a=a; this.b=b; }
7  }
```

```
8  class Sub extends Expr {             // Feature Sub ...
9    int a; int b;
10   Sub(int a, int b) { super(); this.a=a; this.b=b; }
11 }
```

```
12 refines class Expr {                 // Feature Eval ...
13   refines Expr() { original(); }
14   int eval() { return 0; }
15 }
16 refines class Add {
17   refines Add(int a, int b) { original(a,b); }
18   refines int eval() { return this.a+this.b; }
19 }
20 refines class Sub {
21   refines Sub(int a, int b) { original(a,b); }
22   refines int eval() { return this.a−this.b; }
23 }
```

Example: gDeep

```
A = μX refines Top{
   a: int a = 0;
};
B = μX refines A {
   foo: int foo(int i) { return a + i; };
};
F = λ+X ≤ A. μY refines X {
   override foo: int foo(int i) { return 2 * original.foo(i); };
};
C = F(B);
```

Type Checking in gDeep

- Ensure that every function (i.e., feature) is monotone
- Ensure that for $F(A)$, A has the right type
- For delegation $M@(N).L$, ensure that $N \leq M$ and M is record with a slot label L
- For M **refines** V, ensure that every overriding slot in M is a subtype of the corresponding slot in V

How to Plug FJ into gDeep?

- Replace the class table; classes are looked up via paths

$$X.c \text{ where } X \text{ is self}$$

- Introduce a syntax for delegating behavior

$$C@(t).m(\bar{u}) \qquad \text{original}.m(\bar{u})$$

- Define translation function
  - Classes ➔ gDeep records
  - Methods and fields ➔ gDeep declarations
- Modular type checking
  - gFJ's type rules use gDeep's subtype rules

Example: gDeep + gFJ

```
1  AddMod = μX refines Top {
2    Expr: μY refines Top { } [ (extends Object, Expr() {super();}) ];
3    Add: μZ refines Top {
4      a: X.Expr a;
5    b: X.Expr b;
6    } [ (extends X.Expr, Add() {super(); this.a=null; this.b=null;}) ];
7  }
```

```
8  SubMod = μX refines AddMod {
9    Sub: μY refines Top {
10     a: X.Expr a;
11   b: X.Expr b;
12   } [ (extends X.Expr, Sub() {super(); this.a=null; this.b=null;}) ];
13 }
```

```
14 EvalMod = μX refines SubMod {
15   Expr: μY refines SubMod@(X).Expr { } {
16     eval: int eval() { return 0; };
17   };
18   Add: μZ refines SubMod@(X).Add {
19     eval: int eval() { return this.a.eval() + this.b.eval(); };
20   };
21   Sub: μT refines SubMod@(X).Sub {
22     eval: int eval() { return this.a.eval() − this.b.eval(); };
23   };
24 }
```

Example: gDeep + gBali

```
1  AddGram = μX refines Top {
2     Expr: X.Val | X.Val X.Oper X.Expr;
3     Oper: '+';
4     Val : INTEGER;
5  };
```

```
6  SubGram = μX refines AddGram {
7     override Oper: original_X .Oper | '+';
8  };
```

Example: gDeep + gXak

```
1  CalcDoc = μX refines Top {
2    operations: (<li> Addition of integers </li>);
3    main:
4      (<html><head><title>Calculator Documentation</title></head>
5          <body bgcolor="white">
6            <h1>Calculator Documentation</h1>
7            <ul> <?gdeep expr="X.operations" ?> </ul></body></html>);
8  };
```

```
9  SubCalc = μX refines CalcDoc {
10   operations:
11     (<?gdeep expr="original_X .operations ?> <li>Subtraction of integers</li>);
12 };
```