Welcome to the Family, Son!

Luca Favalli

Università degli Studi di Milano
Computer Science Department

@T-LADIES kick-off meeting

Pisa, July 6th 2022

Joint work with Walter Cazzola

A domain-specific language (DSL) is a programming language that mimics the terms, idioms and expressions used among the experts in the target domain

- ideally, a domian expert, with no experience in programming, could read, understand and validate such code.

A **domain-specific language** (DSL) is a programming language that mimics the terms, idioms and expressions used among the experts in the target domain

- ideally, a domian expert, with no experience in programming, could read, understand and validate such code.

We are used to use domain specific languages (DSLs)

- LaTeX to typeset scientific documents
- SQL to query relational databases
- make & ant to build up software systems

. . . but we don't have the custom to write our own DSL.

DSL benefits are evident

- problem-tailored solutions
  - i.e., solutions more concise and clearer
- domain-oriented solutions
  - i.e., solutions implementable by domain experts

Neverlang

Luca Favalli

DSL

LPL
SPLE
Our approach

Neverlang
Terminology
Language Decomposition
Language Composition
Example
Language Extension
Language Execution
IDE Generation

AiDE
Bottom-up
FeatureIDE

References

DSL benefits are evident

- problem-tailored solutions
  - i.e., solutions more concise and clearer
- domain-oriented solutions
  - i.e., solutions implementable by domain experts

- Encapsulation – a DSL hides implementation details.
- Productivity – domain coupling eases the coding phase.
- Communication – development is not limited to programmers.
- Quality – minor "impedance mismatch" between domain experts' requirements and implementing code.

Neverlang

Luca Favalli

DSL benefits are evident

- problem-tailored solutions
  - i.e., solutions more concise and clearer
- domain-oriented solutions
  - i.e., solutions implementable by domain experts

- Encapsulation – a DSL hides implementation details.
- Productivity – domain coupling eases the coding phase.
- Communication – development is not limited to programmers.
- Quality – minor "impedance mismatch" between domain experts' requirements and implementing code.

...but to implement them is hard!

- to develop a compiler/interpreter is long, complex and requires some skills;
- existing languages cannot be easily extended or modified; and
- there is a lack of tools easing their development

The main obstacle is

- the traditional approach to programming language implementation

The main obstacle is

- the traditional approach to programming language implementation

Compilers/Interpreters are

Monolithic and **Opaque**;

Domain Specific Languages
Obstacles

The main obstacle is

- the traditional approach to programming language implementation

Compilers/Interpreters are

Monolithic and **Opaque**;

Therefore, they are

- hard to change their code
- hard to extend them
- hard to **reuse** in the implementation of other languages

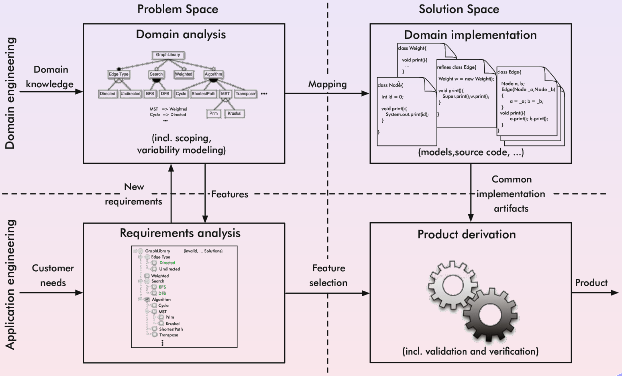
Language Product Lines

Our approach is based on product line engineering.
Product Lines are a staple in industrial production.

- A product line is a collection of features.
- Each product of the product family is a variant.
- Each variant is identified by a subset of all the available features.
- Product families emerge as a byproduct of product line development.

Software Product Lines (SPLs) and Feature-Oriented programming apply Product Lines concepts to software development.

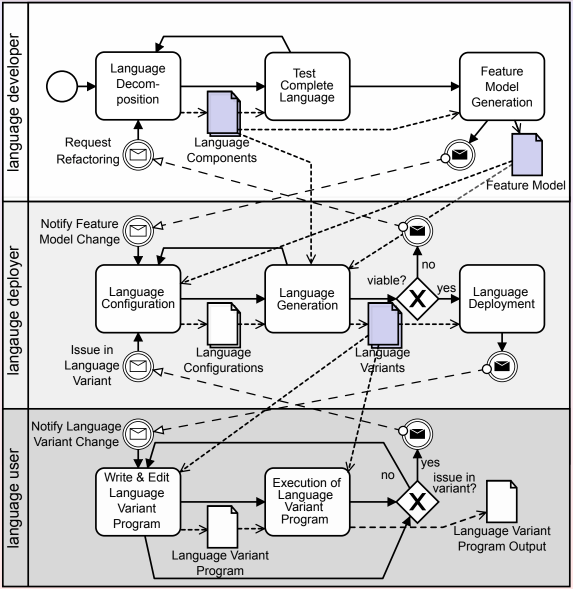
SPL development can be applied to the creation of language families, hence Language Product Lines (LPLs).



Meinicke, Thüm, Schröter, Benduhn, Leich and Saake 2017 [6]

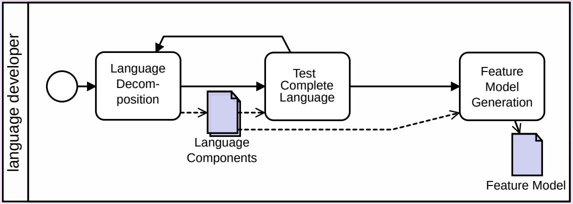
The order in which decisions are made

Favalli, Kühn, and Cazzola 2020 [3]



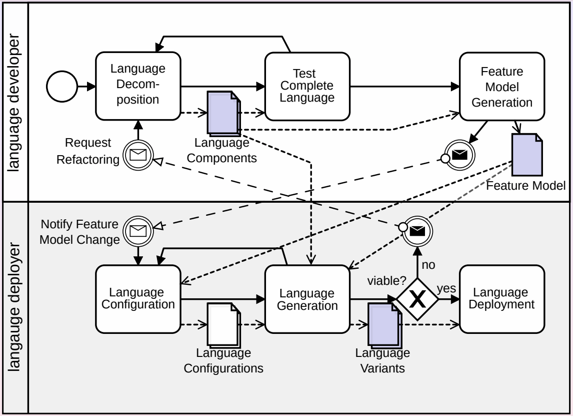
The order in which decisions are made

Favalli, Kühn, and Cazzola 2020 [3]



The order in which decisions are made

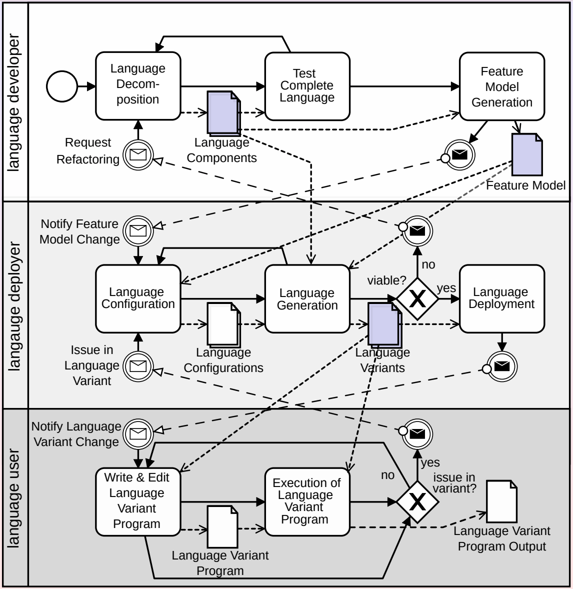
Favalli, Kühn, and Cazzola 2020 [3]



Language Product Lines

The order in which decisions are made

Favalli, Kühn, and Cazzola 2020 [3]



We need a toolchain to support all phases of this process.

Neverlang + AiDE

- Neverlang is a language workbench for the development of programming languages and their ecosystems in a modular way

  Vacchi, and Cazzola 2015 [7]

- AiDE is a variability management framework for the modeling and configuration of language families

  Kühn, Cazzola, and Olivares 2015 [4]

Neverlang:

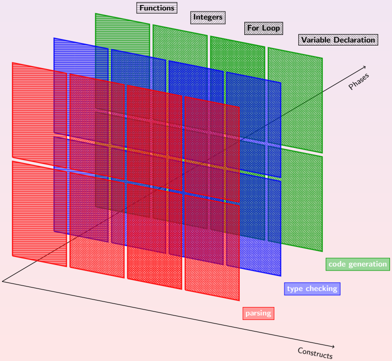
- Language decomposition
- Feature implementation
- IDE generation
- Language execution

AiDE:

- Dependency management
- Feature model generation
- Feature composition
- Product configuration
- Dependency resolution
- Product deployment

Modularization in Neverlang:

- modules are the basic units
- the reference syntax represent part of the language grammar
  - it is comprised of a set of production rules
- each role represents a phase of the compilation process
  - it represents the language semantics
- a slice regards a particular language feature (construct)
  - it is the composition of a reference syntax with their roles



Grammar Centric Approach!

Grammar Centric Approach!

Syntax is used for selecting insertion points, where semantic actions are plugged in to form slices:

- nonterminals correspond to insertion points
- semantic actions at nonterminals correspond to code to be executed during the AST visit

A syntax-driven translation mechanism.

Aho, Lam, Sethi and Ullman 1986 [1]

A module is composed of:

- A syntax block;
  - each feature needs one;
- Zero or more roles / semantic actions.

```
module AdditionExpression {
    reference syntax {
        //$0                $1                  $2
        AddExpression ← AddExpression "+" Term;
        //$3                $4
        AddExpression ← Term;
    }
    role(evaluation) {
        0 .{
            $0.value = $1.value + $2.value;
        }.
        3 .{
            $3.value = $4.value;
        }.
    }
}
```

A module is composed of:

- A syntax block;
  - each feature needs one;
- Zero or more roles / semantic actions.

```
module AdditionExpression {
    reference syntax {
        //$0                $1                  $2
        AddExpression ← AddExpression "+" Term;
        //$3                $4
        AddExpression ← Term;
    }
    role(evaluation) {
        0 .{
            $0.value = $1.value + $2.value;
        }.
        3 .{
            $3.value = $4.value;
        }.
    }
}
```

Can we do better?

One *reference* syntax module, . . .

```
module BinaryOperationAbstractSyntax {
    reference syntax {
        //$0                   $1       $2
        BinaryOperation ← Operand Operand;
        //$3                   $4
        BinaryOperation ← Operand;
    }
}
```

...several semantics hooking to the same reference syntax...

```
module AdditionSemantics {
    reference syntax from BinaryOperationAbstractSyntax
    role (evaluation) {
        0 .{
            $0.value = $1.value + $2.value;
        }.
        3 .{
            $3.value = $4.value;
        }
    }
}
```

```
module MultiplicationSemantics {
    reference syntax from BinaryOperationAbstractSyntax
    role (evaluation) {
        0 .{
            $0.value = $1.value * $2.value;
        }.
        3 .{
            $3.value = $4.value;
        }.
    }
}
```

Neverlang
Language Decomposition

. . . and several concrete syntax implementations.

```
module InfixAdditionSyntax {
    reference syntax {
        AddExpression ← AddExpression "+" Term;
        AddExpression ← Term;
    }
}
```

```
module InfixMultiplicationSyntax {
    reference syntax {
        MulExpression ← MulExpression "*" Factor;
        MulExpression ← Factor;
    }
}
```

Language Features are implemented through composition.

Composition is twofold:

1. Between roles, which yields slices (language features)

```
slice InfixAddition {
    concrete syntax from InfixAdditionSyntax
    module AdditionSemantics with role evaluation
}
```

2. Between slices, which yields languages (compilers/interpreters)

```
language InfixLang {
    slices
        InfixAddition
        InfixMultiplication
        ...
    roles syntax < evaluation
}
```

Different compositions yield different languages

Different compositions yield different languages

```
module PolishAdditionSyntax {
    reference syntax {
        AddExpression ← "+" AddExpression Term;
        AddExpression ← Term;
    }
}
```

Different compositions yield different languages

```
module PolishAdditionSyntax {
    reference syntax {
        AddExpression ← "+" AddExpression Term;
        AddExpression ← Term;
    }
}
```

```
slice PolishAddition {
    concrete syntax from PolishAddSyntax
    module AddSemantics with role evaluation
}
```

Different compositions yield different languages

```
module PolishAdditionSyntax {
    reference syntax {
        AddExpression ← "+" AddExpression Term;
        AddExpression ← Term;
    }
}
```

```
slice PolishAddition {
    concrete syntax from PolishAddSyntax
    module AddSemantics with role evaluation
}
```

```
language PolishLang {
    slices
        PolishAddition
        PolishMultiplication
        ...
    roles syntax < evaluation
}
```

Different compositions yield different languages

```
module PolishAdditionSyntax {
    reference syntax {
        AddExpression ← "+" AddExpression Term;
        AddExpression ← Term;
    }
}
```

```
slice PolishAddition {
    concrete syntax from PolishAddSyntax
    module AddSemantics with role evaluation
}
```

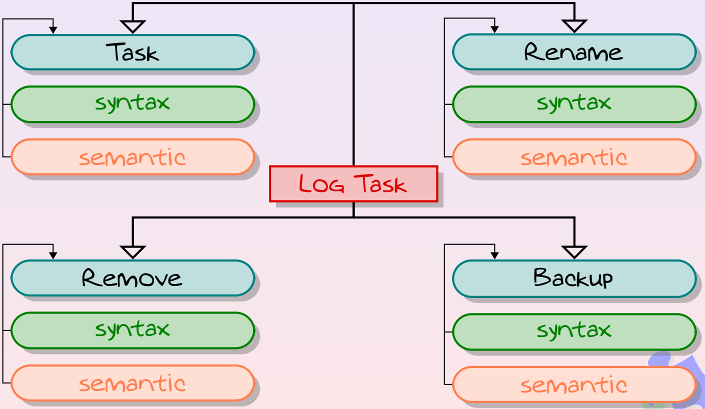
```
language PolishLang {
    slices
        PolishAddition
        PolishMultiplication
        ...
    roles syntax < evaluation
}
```

Maximizes code reuse and minimizes clone-and-own.

LogLang, DSL for a log rotating tool tasks

```
task SomeTask {
    backup "/foo/bar.txt" "/backup/bar.bak"
    rename "/foo/bar.txt" "/foo/bar.txt.old"
    merge  "/baz/qux1.txt" "/baz/qux2.txt"
    remove "/faz.dat"
}
```

Language decomposition



Feature implementation

slice

Remove

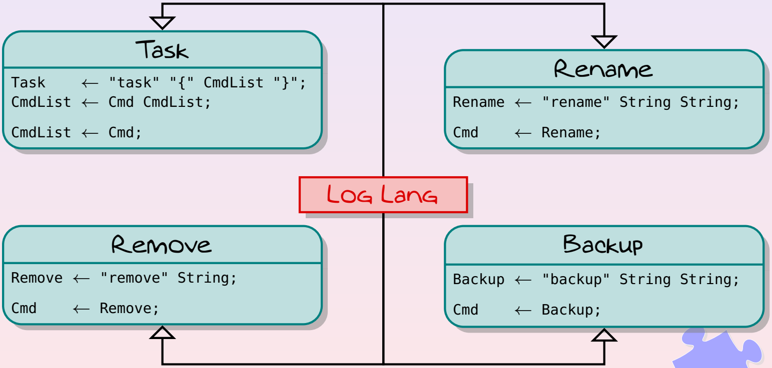
syntax

```
Remove ← "remove" String;
Cmd    ← Remove;
```

evaluation role

```
0 .{ String fileName = $1.string; }.
```

Feature composition



The semantic actions could require some supporting code:

- ancillary structures are defined in the endemic slices;
- fields and methods defined in an endemic slice are accessible by all the others modules.

```
endemic slice FileOpEndemic {
    declare {
        FileOp : neverlang.examples.loglang.utils.FileOp;
    }
}
```

```
role(execution) {
    bkp: .{
        String src  = $bkp[1].string;
        String dest = $bkp[2].string;
        $$FileOp.backup(src, dest);
    }.
}
```

To add a Merge operation to the language:

- a new slice for the operation should be created;
- one of its nonterminals must be present in the rest of the grammar definition (a sort of anchor)

slice

Merge

syntax

```
Merge ← "merge" String String;

Cmd   ← Merge;
```

evaluation role

```
0 .{  String fn1 = $1.string;
      String fn2 = $2.string;

      $$FileOp.merge(fn1, fn2); }.
```

Task

```
Task    ← "task" "{" CmdList "}";
CmdList ← Cmd CmdList;

CmdList ← Cmd;
```

To add an additional permission check:

- a new phase in the interpretation process should be defined
- to enrich each slice with a module to be used in the new phase.

Rename

syntax

```
Rename ← "rename" String String;
Cmd    ← Rename;
```

perm check role

```
0 .{
    if(!$$PermCk.check.("rnm",$1.value))
        System.err.println("Perm err")  }.
```

evaluation role

```
0 .{  String oldF = $1.string;
      String newF = $2.string;
      $$FileOp.move(oldF, newF);  }.
```

The interpreter is generated from the slices

- the files defining the slices are used to feed the generator (nlgc)
  - the generator creates the sources implementing the interpreter
- nlg runs the interpreter

```
$> nlgc -s out  BackUp.nl  FileSystemOp.nl  Identifier.nl  LogLang.nl
     Logger.nl  Main.nl  Merge.nl Remove.nl  Rename.nl  Task.nl
$> javac out/**/*.java
$> nlg LogLang TaskList.txt
Processing TaskList.txt
        ...
Task Executed
```

DSLs and languages in general need Integrated Development Environment (IDE) support.

- syntax highlighting;
- auto-completion;
- debugger.

Working with LPLs means that an IDE must be developed from scratch for each new language variant.

In Neverlang, IDE specifications are bundled with module definitions.

- better encapsulation;
- improved module reusability;
- the IDE for a language variant is automatically generated.

Kühn, Cazzola, Pirritano and Poggi 2019 [5]

Syntax highlighting using **categories**.

```
module neverlang.examples.loglang.Backup {
  reference syntax {
    /* ... */
    bkp:
      Backup ← "backup" String String;
      Cmd    ← Backup;
    categories :
      Keyword = { "backup" };
  }
  /* ... */
}
```

Auto-completion using Buckets.

```
module neverlang.commons.LogLangTypes {
  reference syntax {
    /* ... */
    Identifier ← /[A-Za-z_][0-9A-Za-z_]*/{identifier}[identifier];
    String     ← /"([^"\\]*(\\.[^"\\]*)*)"/{string}[string];
    categories :
      Identifier = { identifier },
      String = { string };
    in-buckets:
      $1#0 ← { Files };
    out-buckets:
      $1#0 → { Files };
  }
  /* ... */
}
```

Debugging through semantic roles.

```
/* ... */
role (debug) {
  pause: @{
    $pause.isExecutionStep = true;
  }.
  pause_timed: @{
    $pause_timed.isExecutionStep = true;
  }.
}
/* ... */
```

AiDE

AiDE is the variability management framework built on top of Neverlang

- has FeatureIDE support

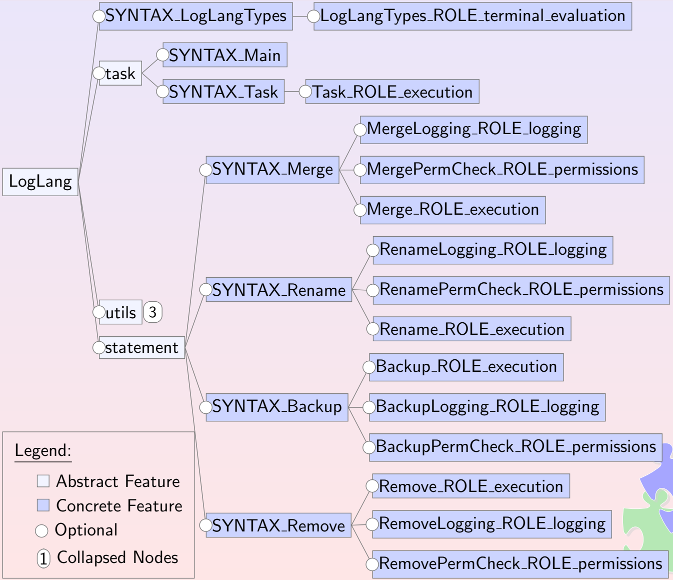
The feature model is built from bottom-up

Tag-based feature model building algorithm

```
module neverlang.examples.loglang.Backup {
  reference syntax {
    provides {
      Backup : backup, statement;
      Cmd    :         statement;
    }
    requires {
      String;
    }
    bkp:   Backup ← "backup" String String;
           Cmd    ← Backup;
    categories :
      Keyword = { "backup" } with style "loglangstyle.json";
    in-buckets:
      $bkp[1] <-- { Files },
      $bkp[2] <-- { Files };
  }
  /* ... */
}
```

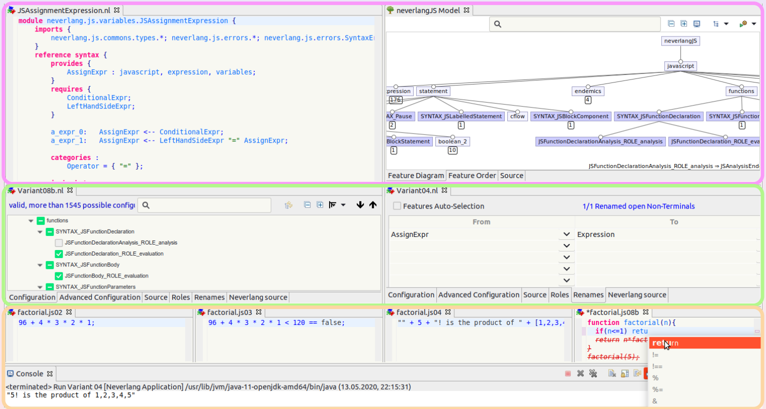
AiDE

Bottom-up language product line engineering support



Neverlang modules and the AiDE algorithm are synchronized with FeatureIDE

Neverlang modules and the AiDE algorithm are synchronized with FeatureIDE



Favalli, Kühn, and Cazzola 2020 [3]

AiDE assesses the design quality of Neverlang language components with regards to several metrics

Project	CC	LoC	V	D	E	T	B	MI	VS
JS+Slicing	0.00	5.00	5.00	4.00	172.08	9.56	0.01	125.37	73.31
Java+SM	0.00	5.00	5.00	4.00	172.08	9.56	0.01	125.37	73.31
Compilation Unit	1.00	12.00	12.00	7.00	863.91	47.99	0.03	104.45	61.08
Types	0.98	15.59	15.59	4.66	1090.56	60.59	0.03	98.59	57.66
LogLang	1.80	18.93	18.93	6.53	1925.15	106.95	0.05	93.64	54.76
Desk	1.00	21.00	21.00	8.33	2782.56	154.59	0.06	90.92	53.17
Statements	1.67	24.83	24.83	8.17	3334.29	185.24	0.07	87.96	51.44
Lambda	1.75	26.75	26.75	9.75	7093.58	394.09	0.11	84.10	49.18
Variables	2.44	27.39	27.39	8.61	8589.59	477.20	0.11	83.61	48.89
Control Flow	2.50	32.08	32.08	6.92	2954.60	164.14	0.06	83.01	48.54
Expressions	3.27	29.42	29.42	14.27	24029.29	1334.96	0.21	79.94	46.75
Neverlang	3.85	41.35	41.35	9.72	12058.23	669.90	0.15	74.35	43.48
Arrays	3.00	32.58	32.58	8.92	8136.50	452.03	0.12	79.77	46.65
State Machines	3.96	39.50	39.50	11.33	10387.41	577.08	0.13	76.64	44.82
Javascript	4.69	38.88	38.88	14.41	21270.55	1181.70	0.20	75.05	43.89
Java Role Extension	5.20	40.40	40.40	17.60	20477.80	1137.66	0.24	73.75	43.13
Functions	3.29	45.86	45.86	11.14	11165.46	620.30	0.15	72.57	42.44
PowerJava	4.57	42.86	42.86	14.00	21421.32	1190.07	0.23	72.09	42.16
Java	7.36	48.57	48.57	15.06	35645.70	1980.32	0.26	69.22	40.48
Rumer	7.80	54.33	54.33	21.57	49047.38	2724.85	0.37	66.17	38.70
Tyl$_{refactored}$	4.00	60.25	60.25	10.25	21061.02	1170.06	0.22	65.48	38.29
Rava	5.40	61.80	61.80	20.80	48978.01	2721.00	0.41	63.50	37.13
Java Relations	7.65	60.35	60.35	21.94	57099.55	3172.20	0.43	63.50	37.13
Tyl$_{legacy}$	8.45	63.86	63.86	17.73	50014.63	2778.59	0.36	62.74	36.69
Object Teams	9.38	64.75	64.75	21.00	54806.41	3044.80	0.42	61.72	36.09

Cazzola and Favalli 2021 [2]

The End?

The End?

Don't bet on it!

Don't bet on it!

Demo time!

The End

Questions & Maybe Answers

References

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman.
  Compilers: Principles, Techniques, and Tools.
  Addison-Wesley, Boston, MA, USA, second edition, 2006.
- Walter Cazzola and Luca Favalli.
  Towards a Recipe for Language Decomposition: Quality Assessment of Language Product Lines.
  Empirical Software Engineering, 27, April 2022.
- Luca Favalli, Thomas Kühn, and Walter Cazzola.
  Neverlang and FeatureIDE Just Married: Integrated Language Product Line Development Environment.
  In Philippe Collet and Sarah Nadi, editors, Proceedings of the 24th International Software Product Line Conference (SPLC'20), pages 285–295, Montréal, Canada, 19th–23rd of October 2020. ACM.
- Thomas Kühn, Walter Cazzola, and Diego Mathias Olivares.
  Choosy and Picky: Configuration of Language Product Lines.
  In Goetz Botterweck and Jules White, editors, Proceedings of the 19th International Software Product Line Conference (SPLC'15), pages 71–80, Nashville, TN, USA, 20th–24th of July 2015. ACM.

References 2

- Thomas Kühn, Walter Cazzola, Nicola Pirritano Giampietro, and Massimiliano Poggi.

  Piggyback IDE Support for Language Product Lines.

  In Thomas Thüm and Laurence Duchien, editors, Proceedings of the 23rd International Software Product Line Conference (SPLC'19), pages 131–142, Paris, France, 9th–13th of September 2019. ACM.

- Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake.

  Mastering Software Variability with FeatureIDE.

  Springer, 2017.

- Edoardo Vacchi and Walter Cazzola.

  Neverlang: A Framework for Feature-Oriented Language Development.

  Computer Languages, Systems & Structures, 43(3):1–40, October 2015.