
INF FAKULTÄT FÜR INFORMATIK

# A Taxonomy of Software Product Line Reengineering

**Wolfram Fenske,** Thomas Thüm, Gunter Saake

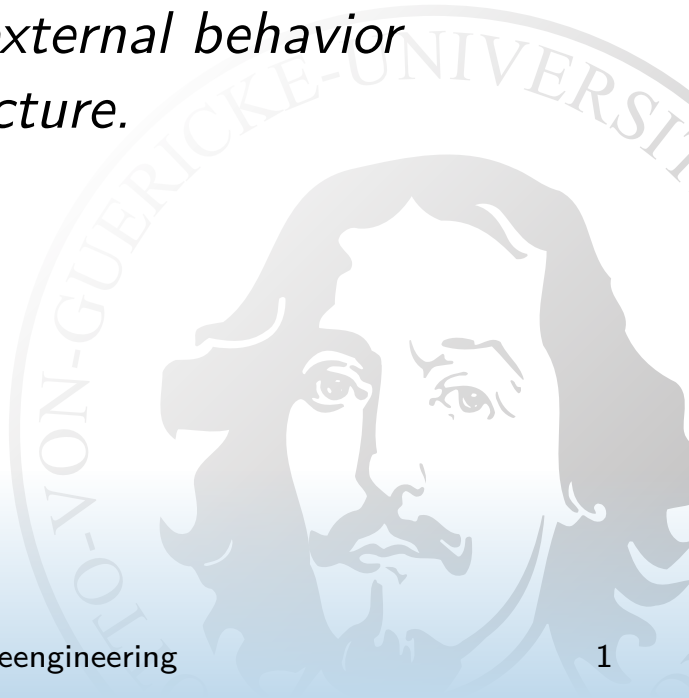
January 22th, 2014

University of Magdeburg, Germany

# Motivation (1)

Widely accepted definition of *refactoring* in single-system engineering (M. Fowler et al., 1999):

> *Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.*

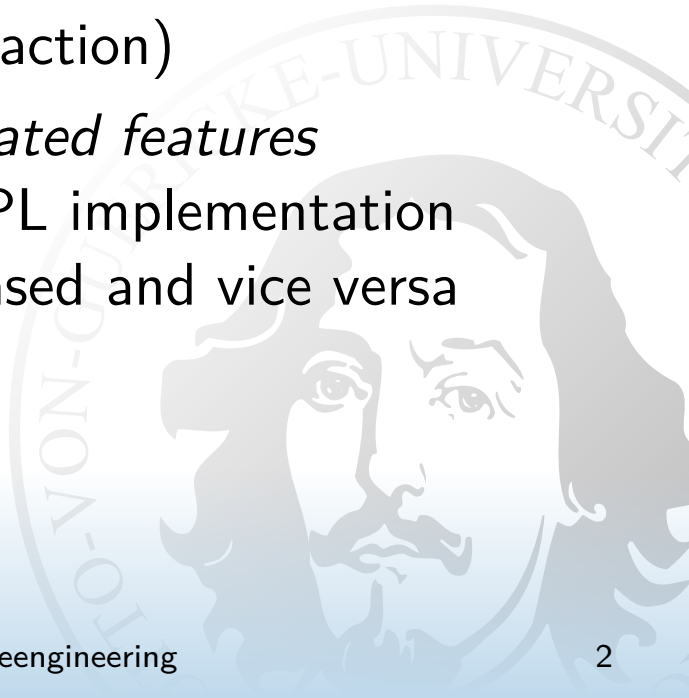


# Motivation (2)

Some notions of refactoring in SPLE:

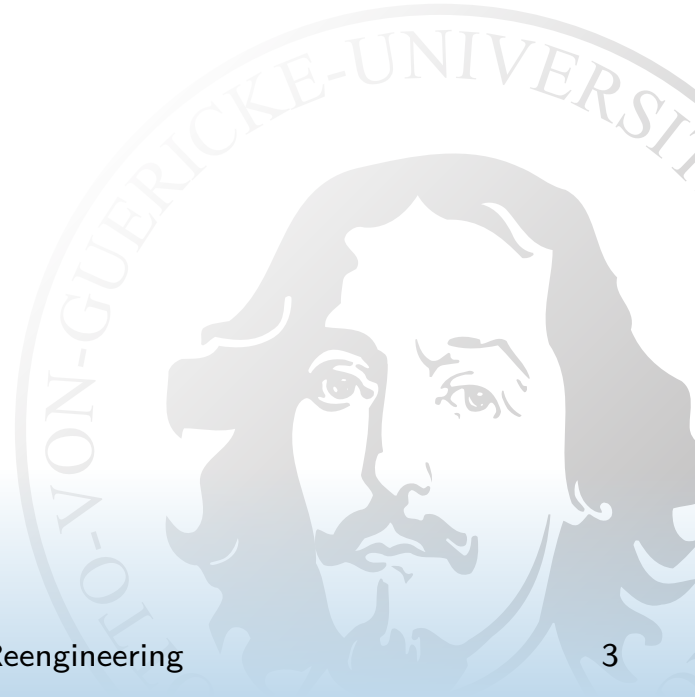
- *Feature oriented refactoring of legacy applications* (J. Liu et al., ICSE'06): Decompose a legacy application into feature modules of a feature-oriented SPL
- *Refactoring delta-oriented software product lines* (S. Schulze et al., AOSD'13): Restructure modules of a delta-oriented SPL (e. g., rename feature, extract delta action)
- *Refactoring physically and virtually separated features* (C. Kästner et al., GPCE'09): Change SPL implementation from annotation-based to composition-based and vice versa

Confused?



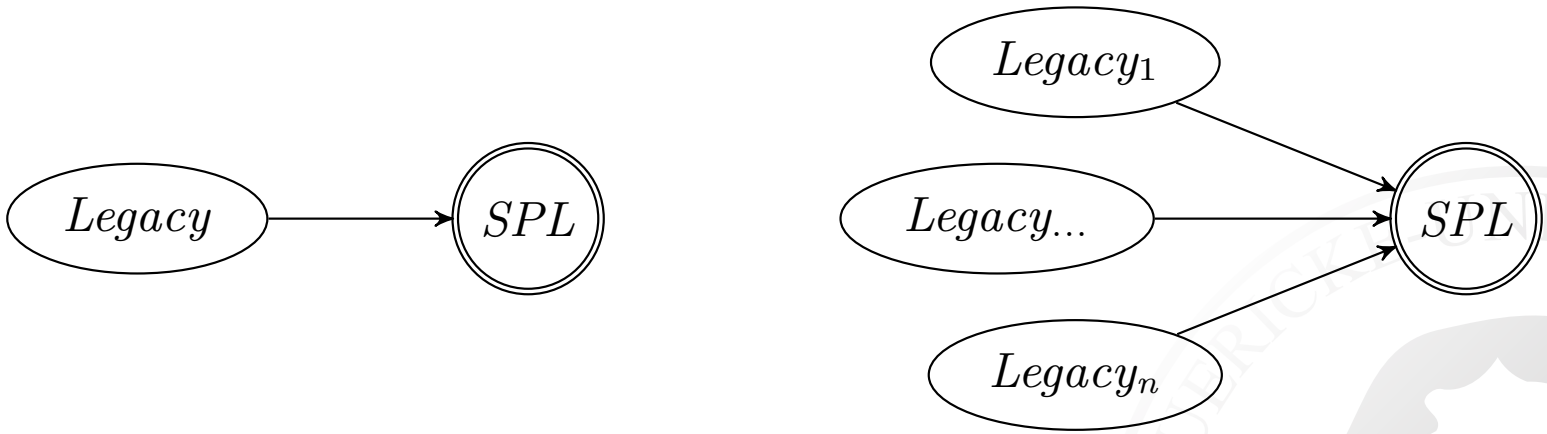
# Motivation (3): Our Contribution

1. Identification of three dimensions of SPL reengineering
2. Taxonomy of SPL reengineering activities
3. Classification of existing work



# Dimensions (1)

*Legacy→SPL:* One or several legacy software product(s) are transformed into an SPL.



# Dimensions (2)

*Legacy→SPL:* 1→SPL

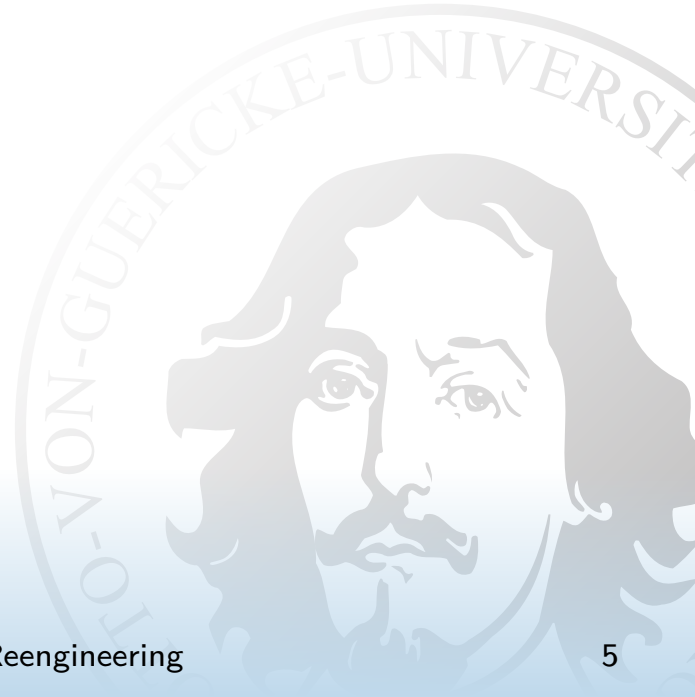
```
class Adder {
  int add(int a, int b) {

    if (isOverflow(a, b))
      throw new IntOverflow();

    return a+b;
  }

  boolean isOverflow(/*...*/)

}
```



# Dimensions (3)

*Legacy→SPL:* 1→SPL

```
class Adder {
  int add(int a, int b) {

    if (isOverflow(a, b))
      throw new IntOverflow();

    return a+b;
  }

  boolean isOverflow(/*...*/)

}
```

```
class Adder {
  int add(int a, int b) {
#ifdef SafeMath
    if (isOverflow(a, b))
      throw new IntOverflow();
#endif
    return a+b;
  }
#ifdef SafeMath
  boolean isOverflow(/*...*/)
#endif
}
```

# Dimensions (4)

*Legacy→SPL:* Many→SPL (1)

```
class Adder {
  int add(int a, int b) {
    if (isOverflow(a, b))
      throw new IntOverflow();
    return a+b;
  }

  boolean isOverflow(/*...*/)
}
```

$Legacy_1$

```
class Adder {
  int add(int a, int b) {


    return a+b;
  }


}
```

$Legacy_2$

# Dimensions (5)

## *Legacy→SPL:* Many→SPL (2)

```
class Adder {
  int add(int a, int b) {
#ifdef SafeMath
    if (isOverflow(a, b))
      throw new IntOverflow();
#endif
    return a+b;
  }
#ifdef SafeMath
  boolean isOverflow(/*...*/)
#endif
}
```

$$Legacy_1 \cup Legacy_2$$

# Dimensions (6)

*Quality:* Improve some property of the code, the feature model, or the feature-to-code mapping (e. g., readability, extensibility)

```
class Adder {
  int add(int a, int b) {
#ifdef SafeMath
    if (isOverflow(a, b))
      throw new IntOverflow();
    return a+b;
#else
    return a+b; // Repetition
#endif
  }

#ifdef SafeMath
  boolean isOverflow(/*...*/)
#endif
}
```

```
class Adder {
  int add(int a, int b) {
#ifdef SafeMath
    if (isOverflow(a, b))
      throw new IntOverflow();
#endif

    // Repetition removed
    return a+b;
  }

#ifdef SafeMath
  boolean isOverflow(/*...*/)
#endif
}
```

# Dimensions (7)

*SPL implementation technique:* Differentiates between SPL implementation techniques

```
class Adder {
  int add(int a, int b) {
#ifdef SafeMath
    if (isOverflow(a, b))
      throw new IntOverflow();
#endif

    return a+b;
  }


#ifdef SafeMath
  boolean isOverflow(/*...*/)
#endif
}
```

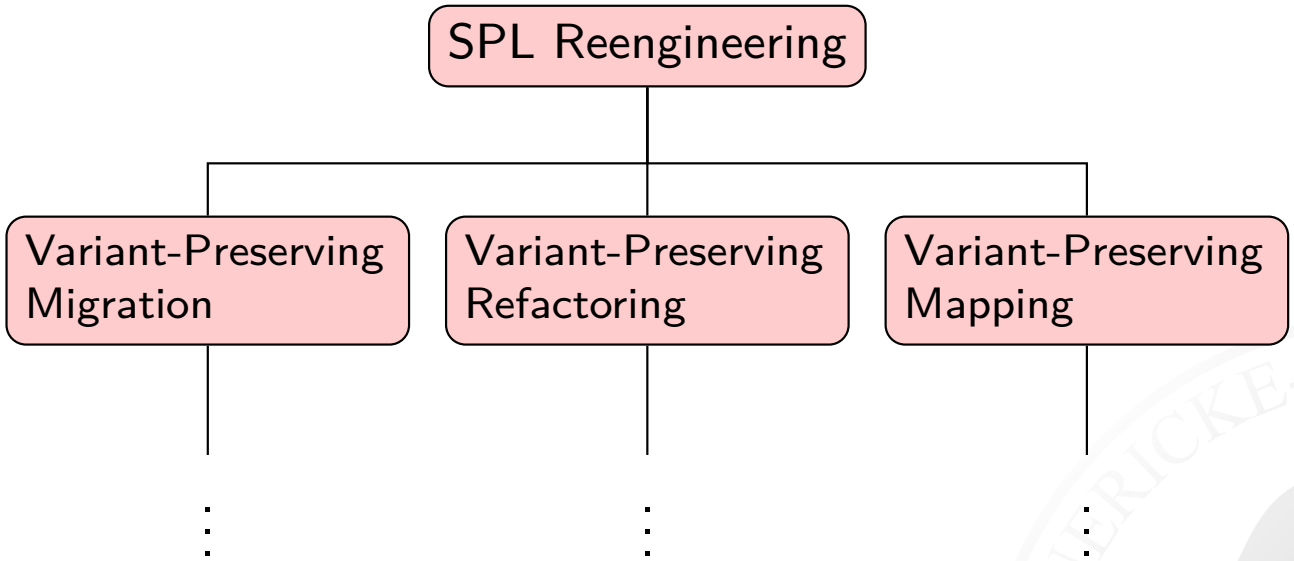
Annotation-based

```
class Adder {
  int add(int a, int b) {
    return a+b; }
}

// Feature Module 'SafeMath'
refines class Adder {
  int add(int a, int b) {
    if (isOverflow(a, b)) {
      throw new IntOverflow();
    }
    Super.add(a, b);
  }
  boolean isOverflow(/*...*/)
}
```

Composition-based

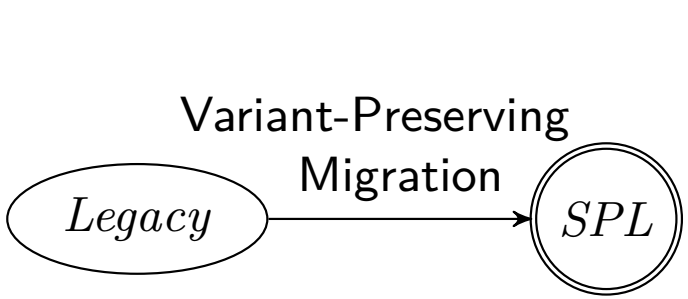
# Taxonomy (1)


SPL Reengineering
Variant-Preserving Migration
Variant-Preserving Refactoring
Variant-Preserving Mapping

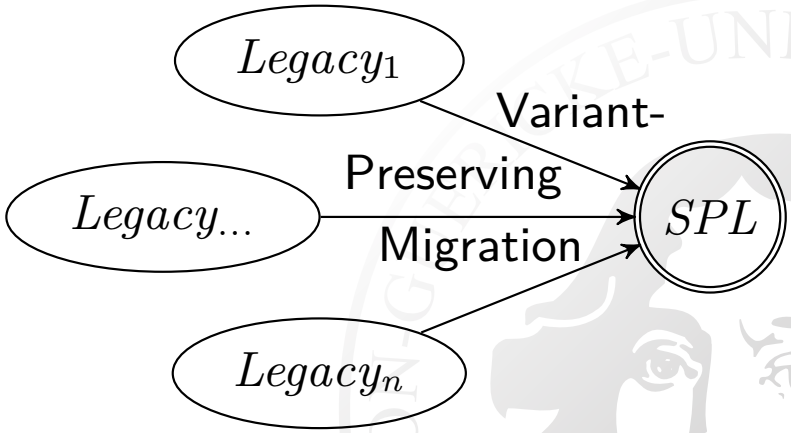
# Taxonomy (2)

Definition (Variant-Preserving Migration)

Variant-preserving migration *is the process of transforming one legacy software product or a family of related legacy software products into a software product line such that for each migrated legacy software product there is a product line instance with the same external behavior.*

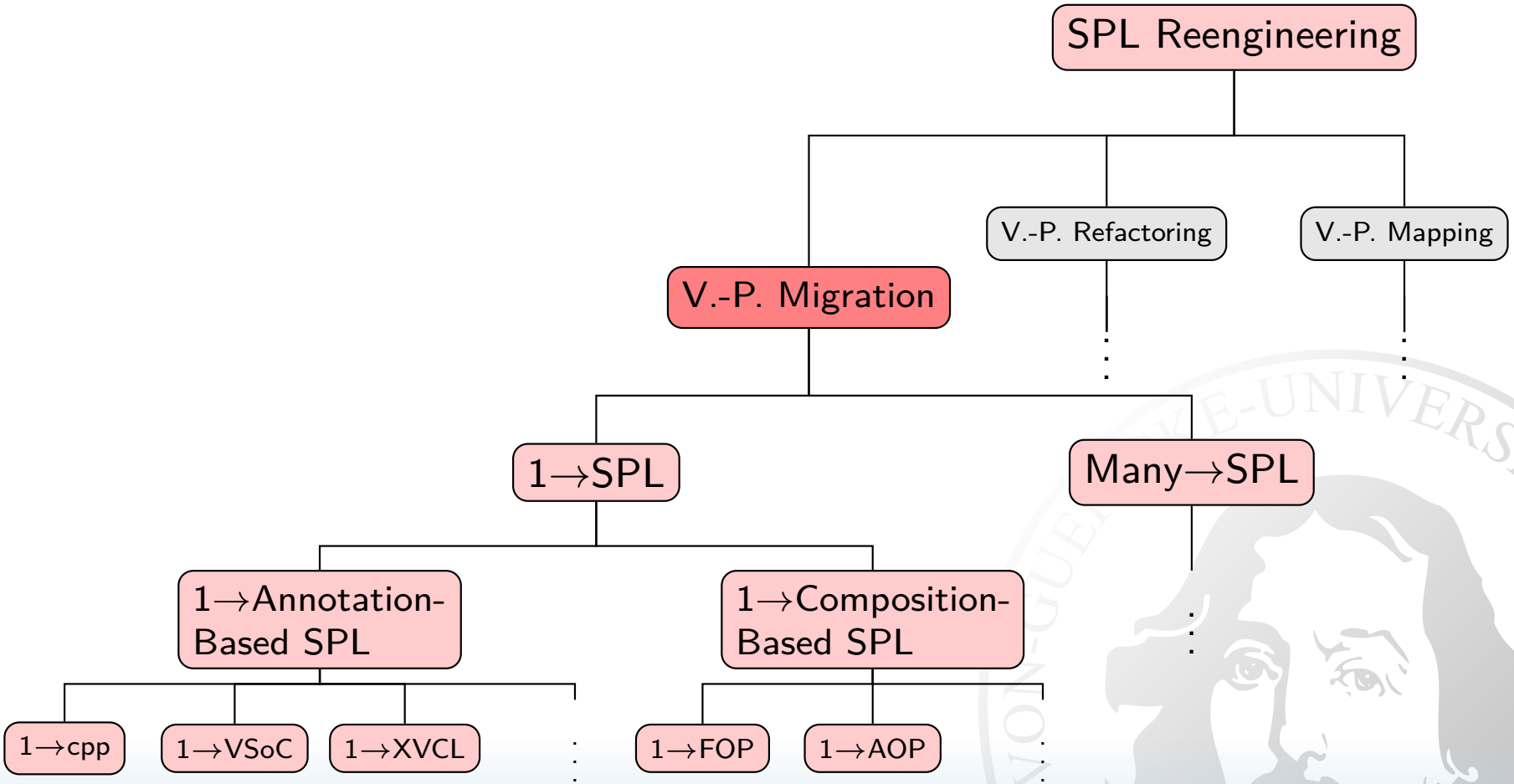



1→SPL




Many→SPL

# Taxonomy (3)

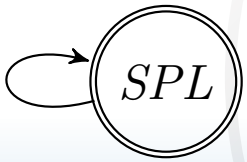

SPL Reengineering
V.-P. Refactoring
V.-P. Mapping
V.-P. Migration
1→SPL
Many→SPL
1→Annotation-Based SPL
1→Composition-Based SPL
1→cpp
1→VSoC
1→XVCL
1→FOP
1→AOP

# Taxonomy (4)

Definition (Variant-Preserving Refactoring[1])

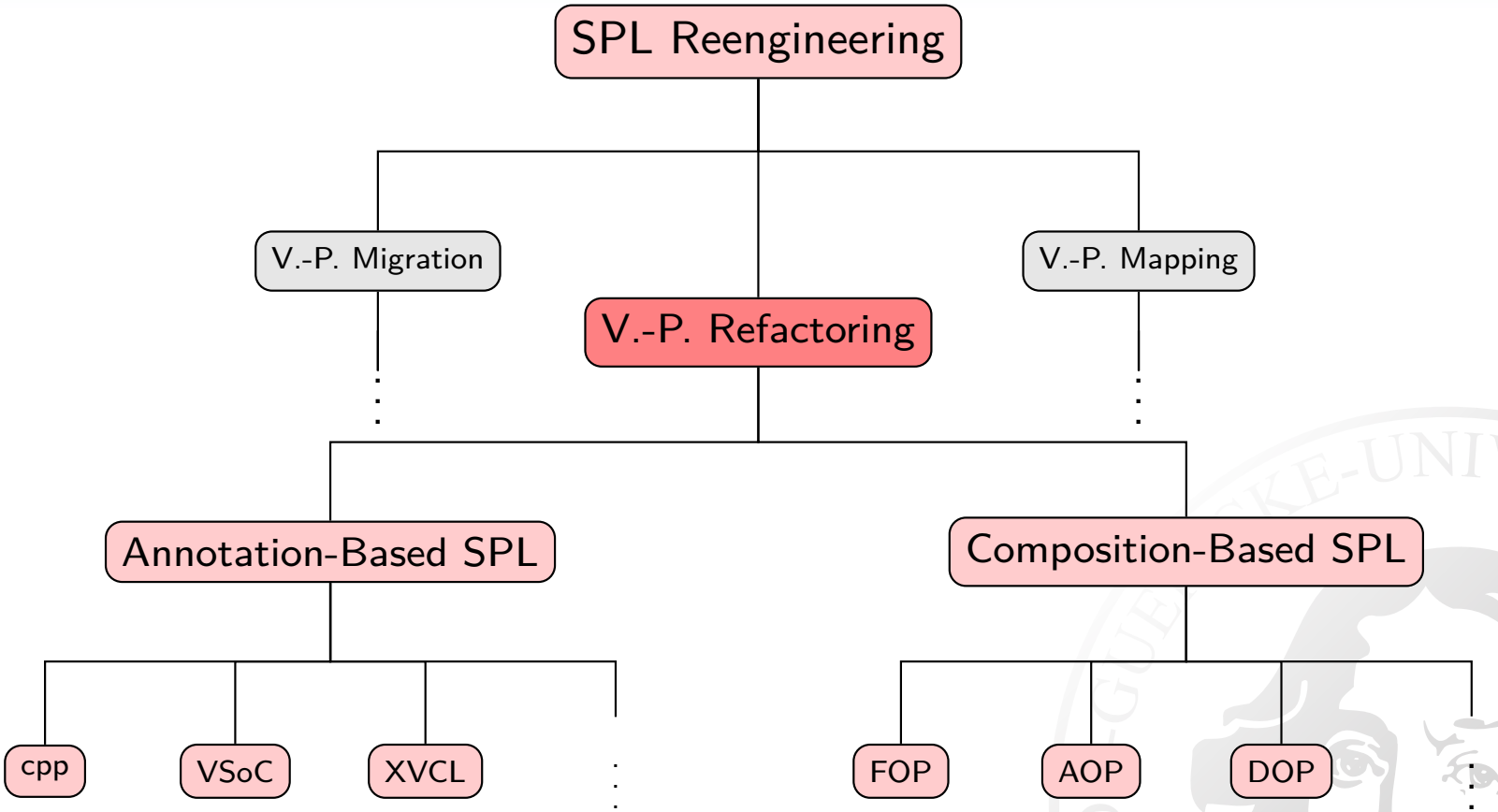
*A change to the feature model or the implementation of features or both is called* variant-preserving refactoring *if the following two conditions hold:*

1. *Each valid combination of features remains valid after the refactoring, whereas the validity is specified by the feature model.*
2. *Each valid combination of features that was compilable before can still be compiled and has the same external behavior after the refactoring.*




[1] Schulze et al. (VaMoS'12)

# Taxonomy (5)

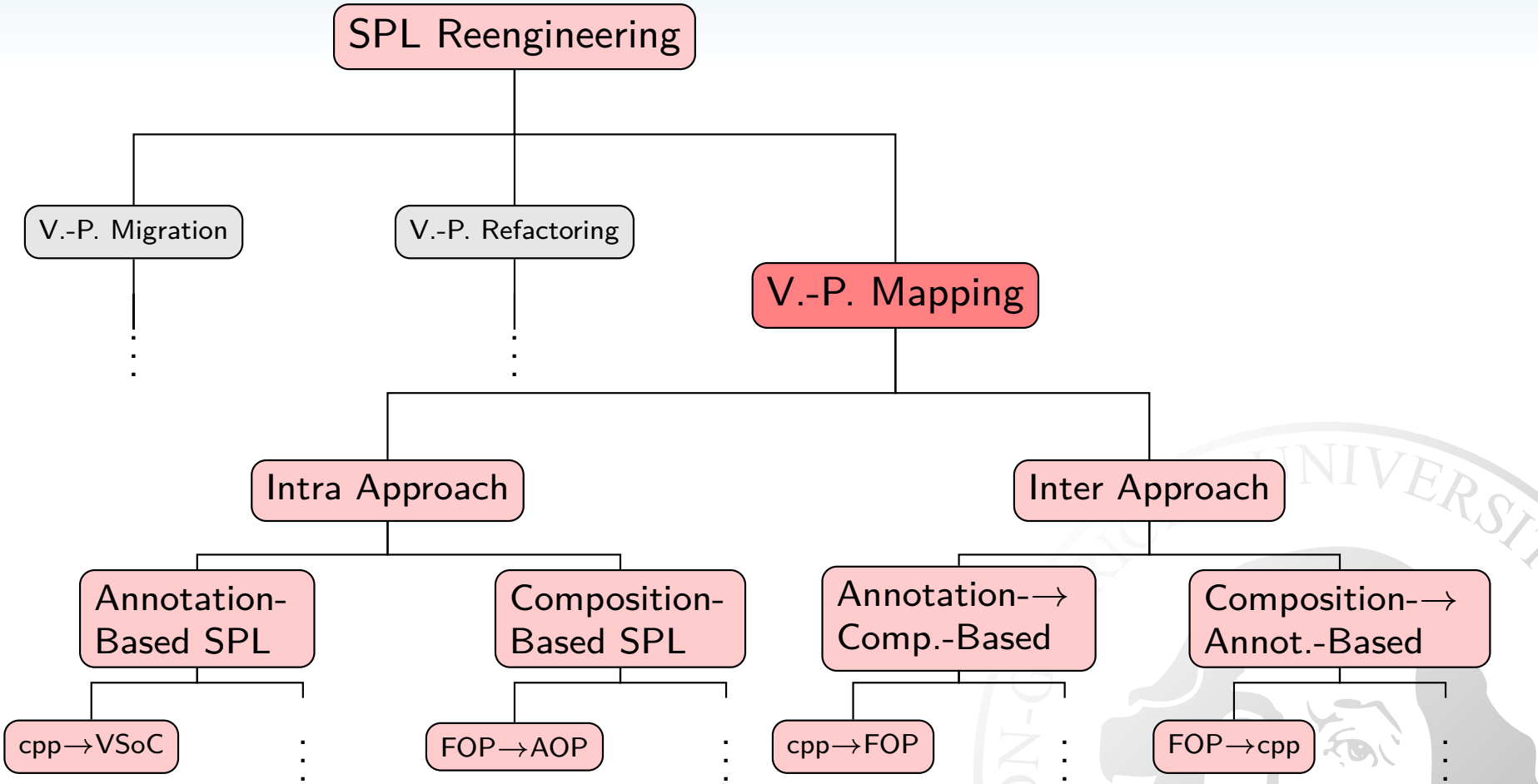

SPL Reengineering
V.-P. Migration
V.-P. Refactoring
V.-P. Mapping
Annotation-Based SPL
Composition-Based SPL
cpp
VSoC
XVCL
FOP
AOP
DOP

# Taxonomy (6)

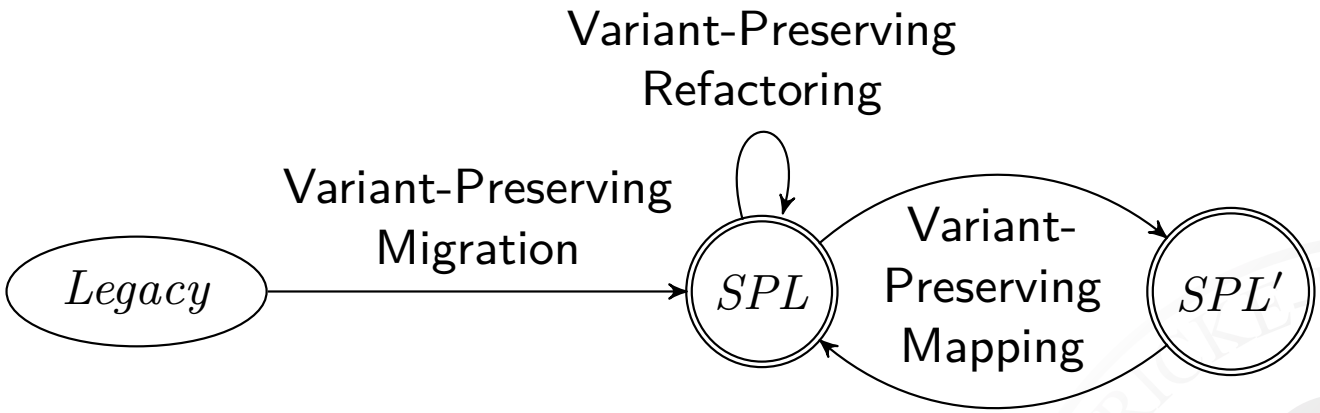
Definition (Variant-Preserving Mapping)

*A substitution of the implementation approach of a software product line is called* variant-preserving mapping *if for each instance of the original product line there is an instance of the new product line that has the same external behavior.*

# Taxonomy (7)


SPL Reengineering
V.-P. Migration
V.-P. Refactoring
V.-P. Mapping
Intra Approach
Inter Approach
Annotation-Based SPL
Composition-Based SPL
Annotation-→ Comp.-Based
Composition-→ Annot.-Based
cpp→VSoC
FOP→AOP
cpp→FOP
FOP→cpp

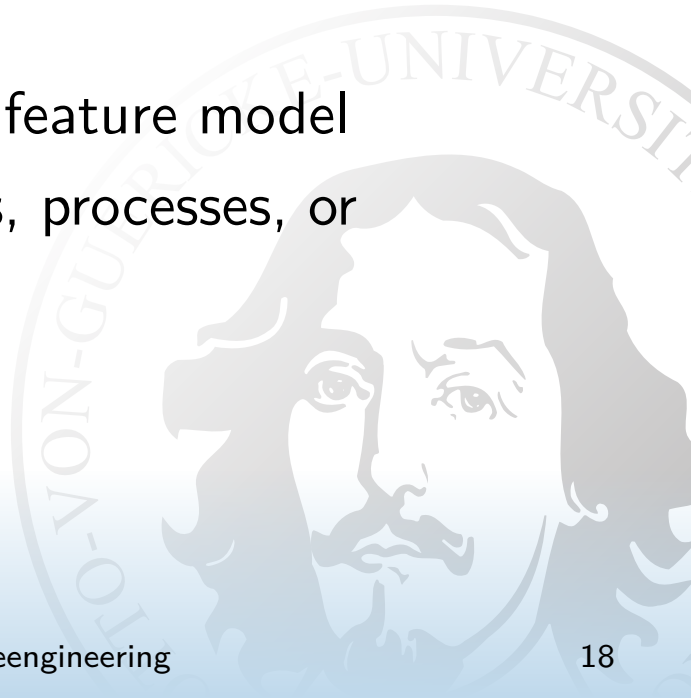
# Taxonomy (8): Relationship



# Literature Review (1): Selection

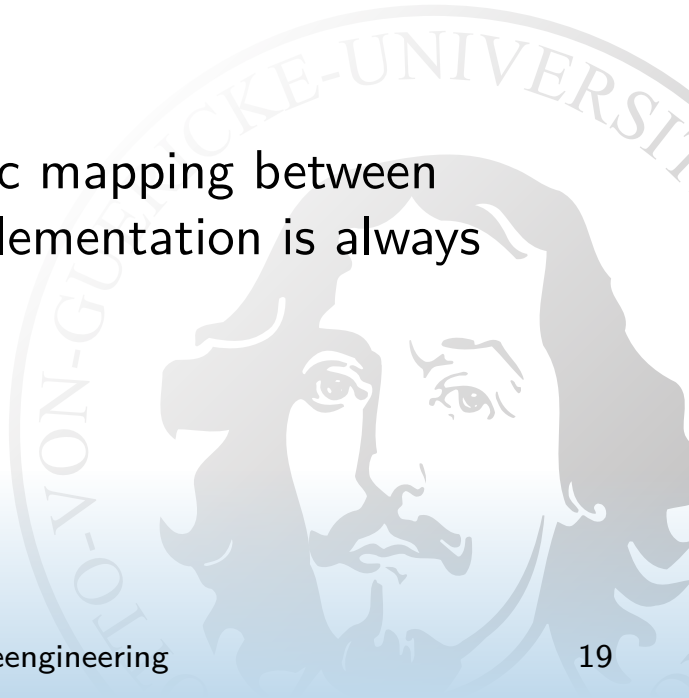
- Review of SPL reengineering literature
- Basis: M. A. Laguna and Y. Crespo. A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring (*Sci. Comput. Prog.*, 2013)
- Our focus: Changes to the code and / or feature model

➥ Exclusion of publications on just analyses, processes, or organizational issues



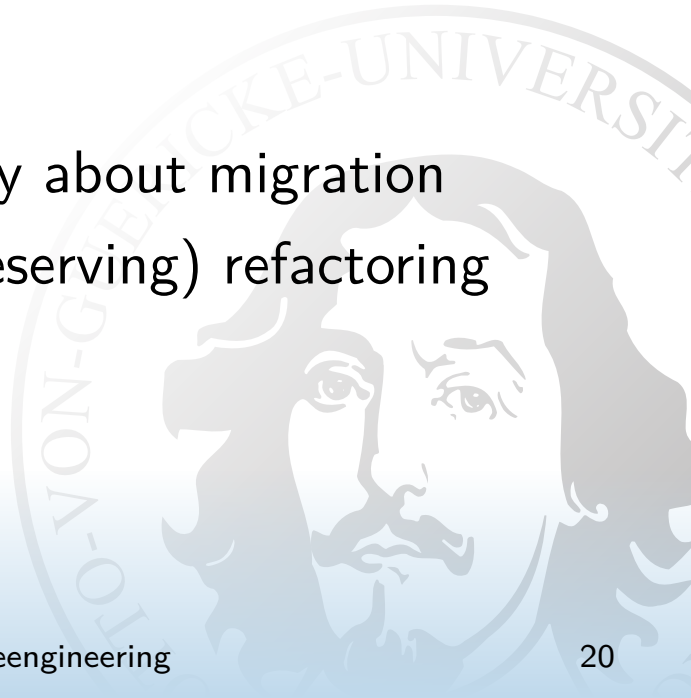
# Literature Review (2): Classification Results

- Variant-preserving migration: 56 %
  - Focus on 1→SPL
  - Target SPL most often composition-based
- Variant-preserving refactoring: 26 %
  - Focus on a small number of composition-based techniques
  - Challenges identified, but hardly tackled
- Variant-preserving mapping: 18 %
  - Some interesting results, e. g., automatic mapping between annotation- and composition-based implementation is always possible (Kästner et al., GPCE'09)
  - But mostly unexplored field



# Conclusion

- SPLE literature terms many reengineering activities as "refactoring"
- Taxonomy divides these into three categories:
  1. Variant-preserving migration,
  2. Variant-preserving refactoring,
  3. Variant-preserving mapping
- Most SPL "refactoring" literature actually about migration
- Few publications on "actual" (variant-preserving) refactoring



# Some Questions

- Are these dimensions appropriate?
  - Have you encountered approaches that do not fit?
  - Would you suggest different or new dimensions?
- What about the wide-spread use of annotation-based SPLs in practice? How do industry professionals refactor their (presumably annotation-based) SPLs?
  - What work on refactoring for preprocessor code is there?
  - Is, e. g., Alejandra Garrido's work applicable to SPLs?
- Can non-SPLE literature help, such as work on aspect-oriented refactoring?