Feature Modeling and Development with FeatureIDE

Thomas Thüm, Thomas Leich, Sebastian Krieter

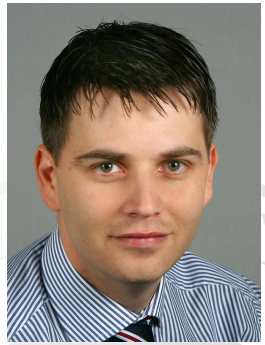
February 21, 2018 — Modellierung'18 in Braunschweig, Germany

Part I

Introduction round

Who we are

Dr. Thomas Thüm @ TU Braunschweig
... started 2006 as developer
... coordinates FeatureIDE development + research



Prof. Dr. Thomas Leich @ Metop GmbH
... initiated FeatureIDE project in 2004
... coordinates FeatureIDE consulting

M.Sc. Sebastian Krieter @ University of Magdeburg
... started 2012 as developer
... an expert on FeatureIDE's architecture

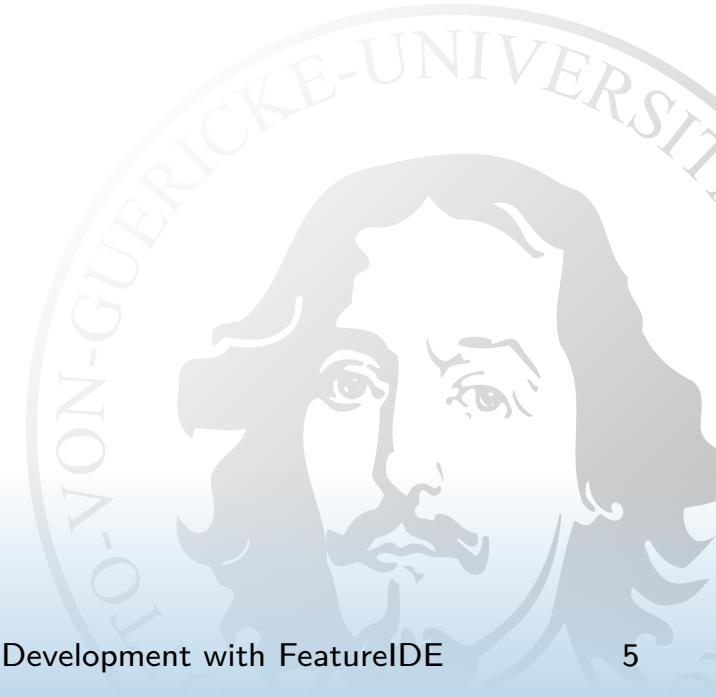
Introduce yourself

Keep it short (3 sentences):

1. What is your name and affiliation?
2. What is your expertise?
3. What are you expecting to learn in this tutorial?

What is your background?

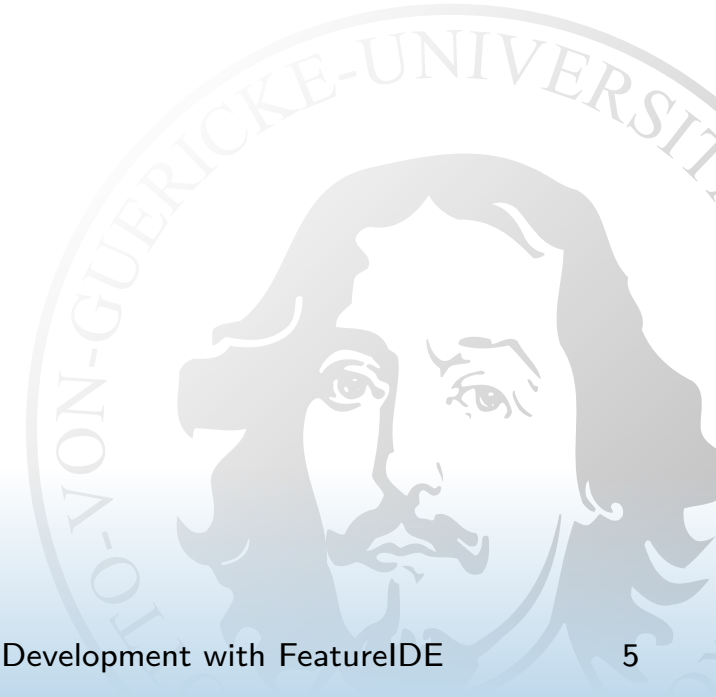
Are you involved in professional software development?



What is your background?

Are you involved in professional software development?

Are you performing research?

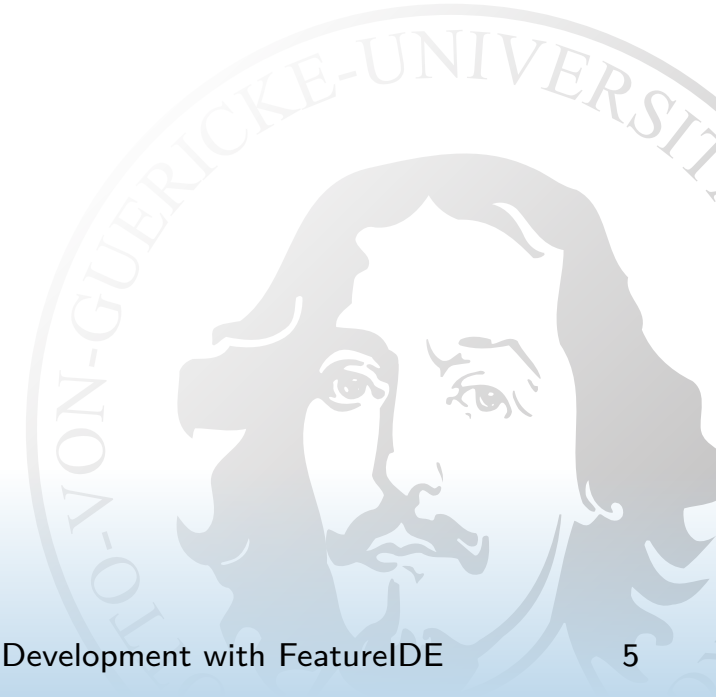


What is your background?

Are you involved in professional software development?

Are you performing research?

Are you teaching?



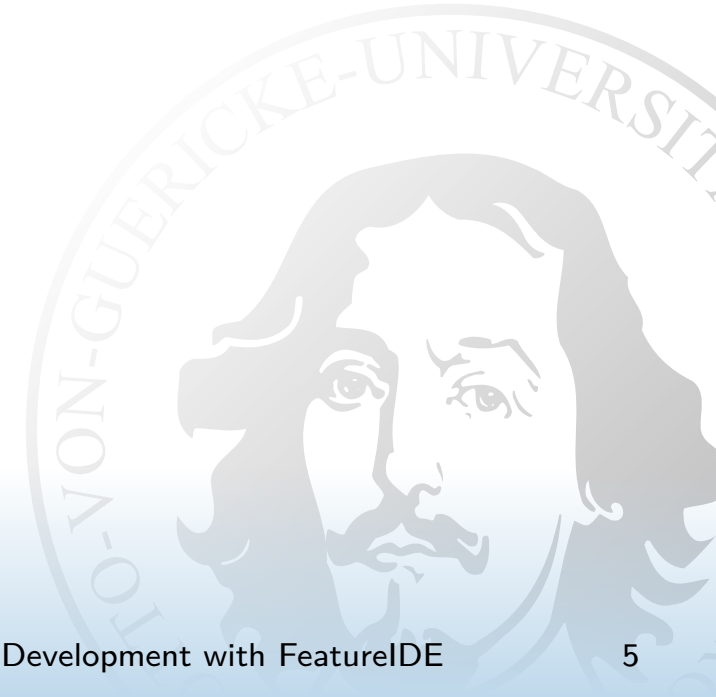
What is your background?

Are you involved in professional software development?

Are you performing research?

Are you teaching?

Do you know feature models?



What is your background?

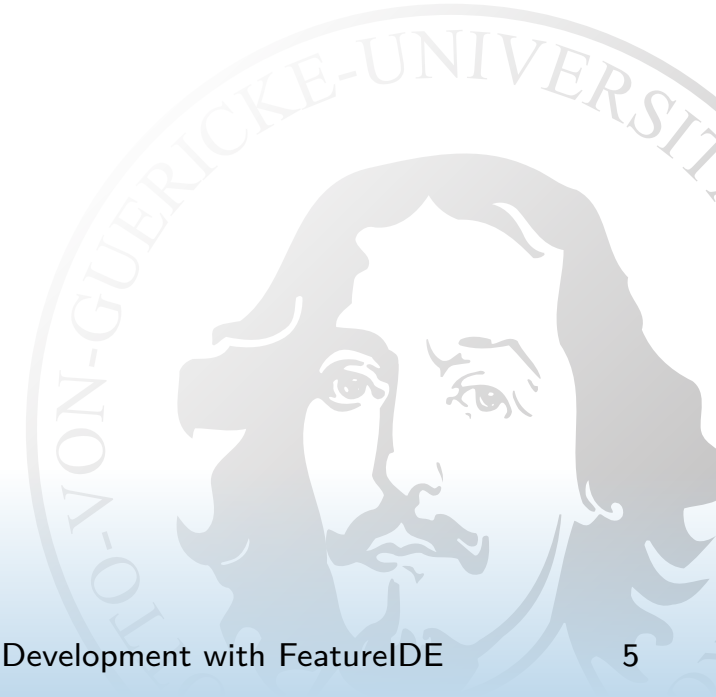
Are you involved in professional software development?

Are you performing research?

Are you teaching?

Do you know feature models?

Do you know preprocessors?



What is your background?

Are you involved in professional software development?

Are you performing research?

Are you teaching?

Do you know feature models?

Do you know preprocessors?

Have you experienced problems with preprocessors?

Disclaimer: This is not a lecture!

- Combination of presentations and hands-on sessions
- Notebook with Windows, Linux, or MacOS available?
- Feel free to ask at any time

Part II

Product-line maintenance is difficult

Preprocessing principle is easy

BestLap

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();


   this.sounds[soundIndex].deallocate();


   this.sounds[soundIndex].setMediaTime(0);


}
```

Preprocessing principle is easy

BestLap

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   //#ifdef (mmapi)
   //#ifdef (sound_prefetch)
   this.sounds[soundIndex].deallocate();
   //#endif
   //#ifdef (!media_time)
   this.sounds[soundIndex].setMediaTime(0);
   //#endif
   //#endif
}
```

Preprocessing principle is easy

BestLap

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   //#ifdef (mmapi)
   //#ifdef (sound_prefetch)
   this.sounds[soundIndex].deallocate();
   //#endif
   //#ifdef (!media_time)
   this.sounds[soundIndex].setMediaTime(0);
   //#endif
   //#endif
}
```

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();


}
```

Preprocessing principle is easy

BestLap

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   //#ifdef (mmapi)
   //#ifdef (sound_prefetch)
   this.sounds[soundIndex].deallocate();
   //#endif
   //#ifdef (!media_time)
   this.sounds[soundIndex].setMediaTime(0);
   //#endif
   //#endif
}
```

```
private vo
   this.sou


}
```

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   this.sounds[soundIndex].deallocate();

}
```

Preprocessing principle is easy

BestLap

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   //#ifdef (mmapi)
   //#ifdef (sound_prefetch)
   this.sounds[soundIndex].deallocate();
   //#endif
   //#ifdef (!media_time)
   this.sounds[soundIndex].setMediaTime(0);
   //#endif
   //#endif
}
```

```
private vo
   this.sou

}
```

```
private vo
   this.sou
   this.sou

}
```

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();

   this.sounds[soundIndex].setMediaTime(0);
}
```

Preprocessing principle is easy

BestLap

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   //#ifdef (mmapi)
   //#ifdef (sound_prefetch)
   this.sounds[soundIndex].deallocate();
   //#endif
   //#ifdef (!media_time)
   this.sounds[soundIndex].setMediaTime(0);
   //#endif
   //#endif
}
```

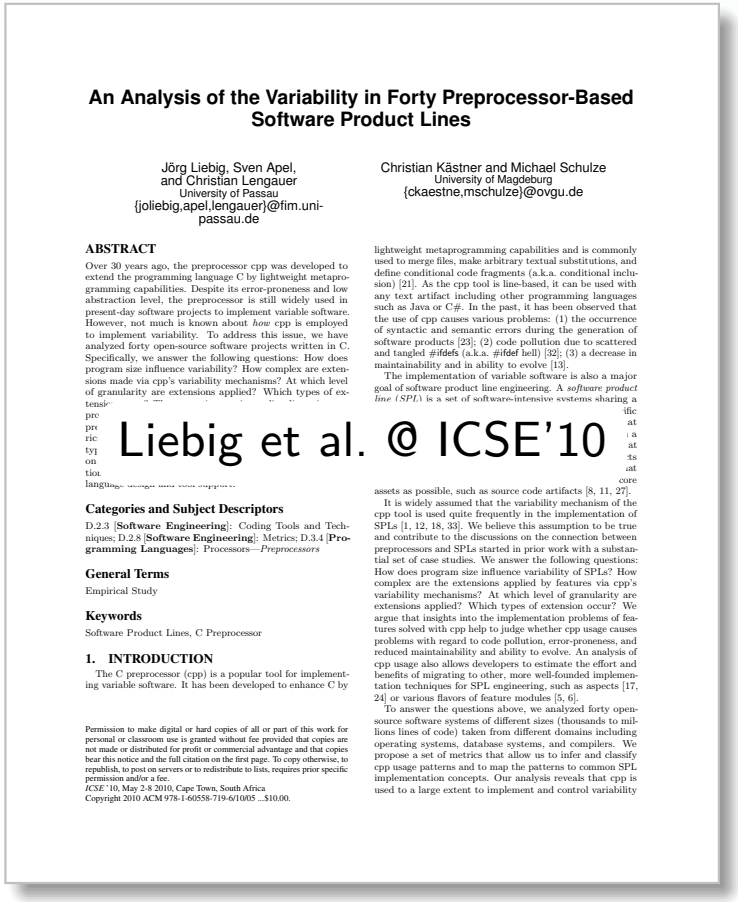
```
private vo
   this.sou
}
```

```
private vo
   this.sou
   this.sou
}
```

```
private vo
   this.sou
   this.sou
}
```

```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();
   this.sounds[soundIndex].deallocate();
   this.sounds[soundIndex].setMediaTime(0);
}
```

Preprocessing is widely adopted



An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines

Jörg Liebig, Sven Apel, and Christian Lengauer
University of Passau
{joliebig,apel,lengauer}@fim.uni-passau.de

Christian Kästner and Michael Schulze
University of Magdeburg
{ckaestne,mschulze}@ovgu.de

ABSTRACT

Over 30 years ago, the preprocessor cpp was developed to extend the programming language C by lightweight metaprogramming capabilities. Despite its error-proneness and low abstraction level, the preprocessor is still widely used in present-day software projects to implement variable software. However, not much is known about *how* cpp is employed to implement variability. To address this issue, we have analyzed forty open-source software projects written in C. Specifically, we answer the following questions: How does program size influence variability? How complex are extensions made via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of ex-

Liebig et al. @ ICSE'10

lightweight metaprogramming capabilities and is commonly used to merge files, make arbitrary textual substitutions, and define conditional code fragments (a.k.a. conditional inclusion) [21]. As the cpp tool is line-based, it can be used with any text artifact including other programming languages such as Java or C#. In the past, it has been observed that the use of cpp causes various problems: (1) the occurrence of syntactic and semantic errors during the generation of software products [23]; (2) code pollution due to scattered and tangled #ifdefs (a.k.a. #ifdef hell) [32]; (3) a decrease in maintainability and in ability to evolve [13].

The implementation of variable software is also a major goal of software product line engineering. A *software product line (SPL)* is a set of software-intensive systems sharing a

assets as possible, such as source code artifacts [8, 11, 27].

It is widely assumed that the variability mechanism of the cpp tool is used quite frequently in the implementation of SPLs [1, 12, 18, 33]. We believe this assumption to be true and contribute to the discussions on the connection between preprocessors and SPLs started in prior work with a substantial set of case studies. We answer the following questions: How does program size influence variability of SPLs? How complex are the extensions applied by features via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? We argue that insights into the implementation problems of features solved with cpp help to judge whether cpp usage causes problems with regard to code pollution, error-proneness, and reduced maintainability and ability to evolve. An analysis of cpp usage also allows developers to estimate the effort and benefits of migrating to other, more well-founded implementation techniques for SPL engineering, such as aspects [17, 24] or various flavors of feature modules [5, 6].

To answer the questions above, we analyzed forty open-source software systems of different sizes (thousands to millions lines of code) taken from different domains including operating systems, database systems, and compilers. We propose a set of metrics that allow us to infer and classify cpp usage patterns and to map the patterns to common SPL implementation concepts. Our analysis reveals that cpp is used to a large extent to implement and control variability

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Empirical Study

Keywords

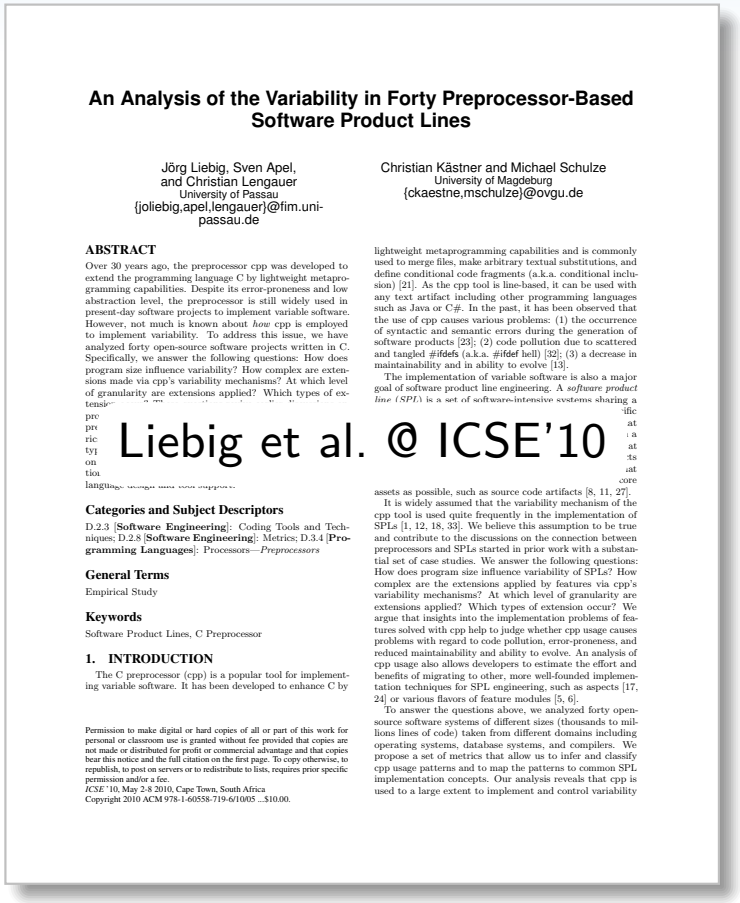
Software Product Lines, C Preprocessor

1. INTRODUCTION

The C preprocessor (cpp) is a popular tool for implementing variable software. It has been developed to enhance C by

ICSE'10, May 2-8 2010, Cape Town, South Africa

Preprocessor code is complex



An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines

Jörg Liebig, Sven Apel, and Christian Lengauer
University of Passau
{joliebig,apel,lengauer}@fim.uni-passau.de

Christian Kästner and Michael Schulze
University of Magdeburg
{ckaestne,mschulze}@ovgu.de

ABSTRACT

Over 30 years ago, the preprocessor cpp was developed to extend the programming language C by lightweight metaprogramming capabilities. Despite its error-proneness and low abstraction level, the preprocessor is still widely used in present-day software projects to implement variable software. However, not much is known about *how* cpp is employed to implement variability. To address this issue, we have analyzed forty open-source software projects written in C. Specifically, we answer the following questions: How does program size influence variability? How complex are extensions made via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of ex-

Liebig et al. @ ICSE'10

language design and tool support.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Empirical Study

Keywords

Software Product Lines, C Preprocessor

1. INTRODUCTION

The C preprocessor (cpp) is a popular tool for implementing variable software. It has been developed to enhance C by lightweight metaprogramming capabilities and is commonly used to merge files, make arbitrary textual substitutions, and define conditional code fragments (a.k.a. conditional inclusion) [21]. As the cpp tool is line-based, it can be used with any text artifact including other programming languages such as Java or C#. In the past, it has been observed that the use of cpp causes various problems: (1) the occurrence of syntactic and semantic errors during the generation of software products [23]; (2) code pollution due to scattered and tangled #ifdefs (a.k.a. #ifdef hell) [32]; (3) a decrease in maintainability and in ability to evolve [13].

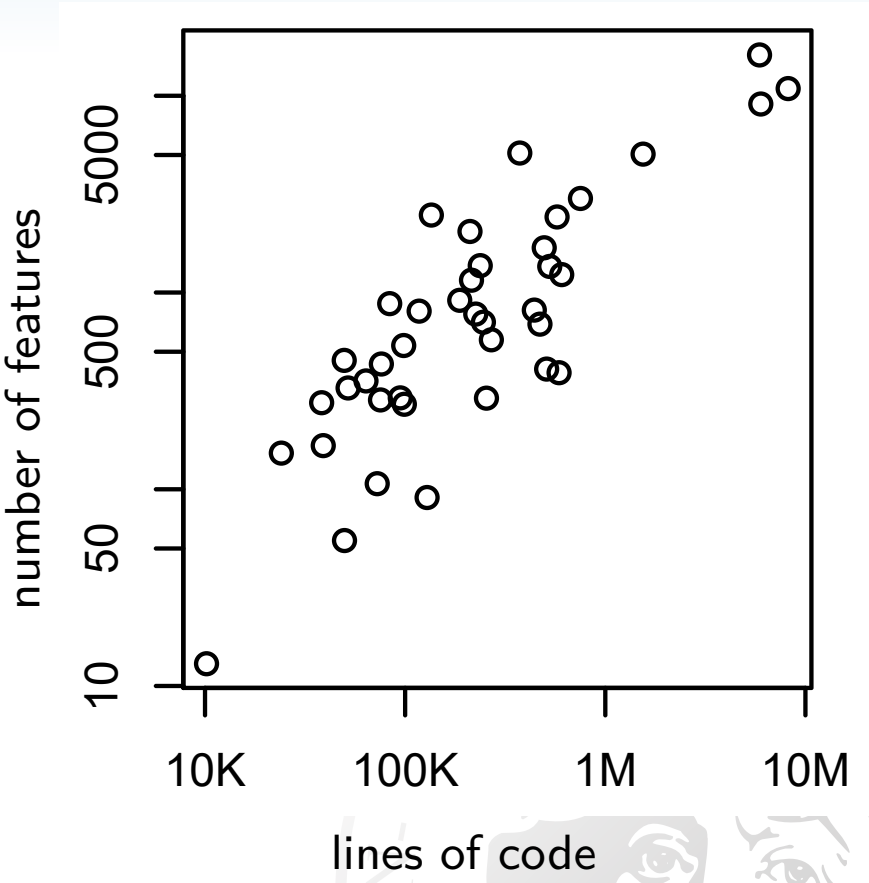
The implementation of variable software is also a major goal of software product line engineering. A *software product line (SPL)* is a set of software-intensive systems sharing a

assets as possible, such as source code artifacts [8, 11, 27].

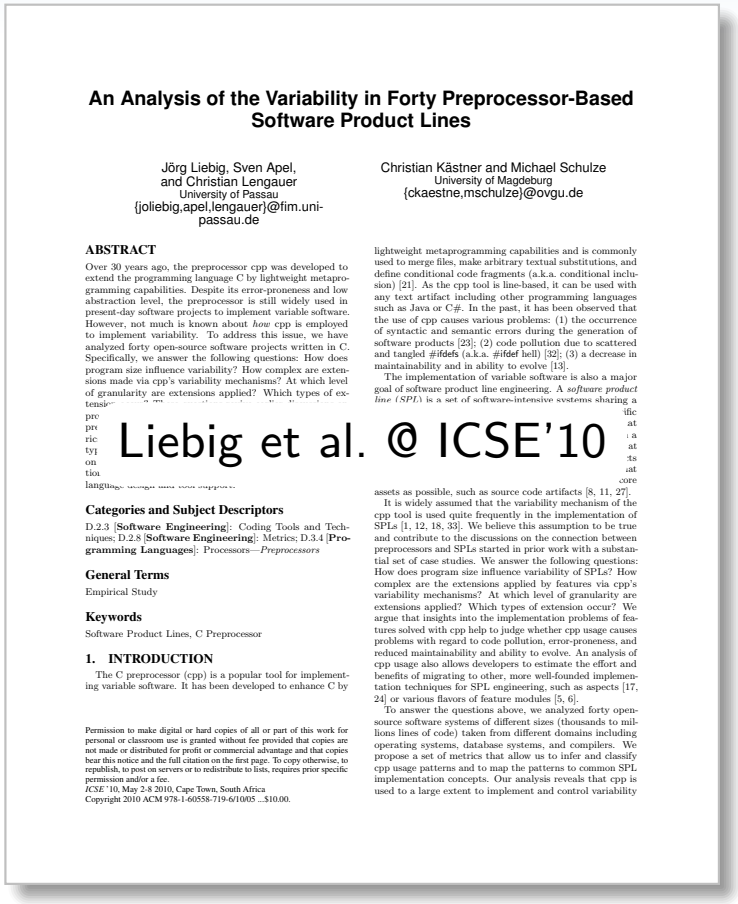
It is widely assumed that the variability mechanism of the cpp tool is used quite frequently in the implementation of SPLs [1, 12, 18, 33]. We believe this assumption to be true and contribute to the discussions on the connection between preprocessors and SPLs started in prior work with a substantial set of case studies. We answer the following questions: How does program size influence variability of SPLs? How complex are the extensions applied by features via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? We argue that insights into the implementation problems of features solved with cpp help to judge whether cpp usage causes problems with regard to code pollution, error-proneness, and reduced maintainability and ability to evolve. An analysis of cpp usage also allows developers to estimate the effort and benefits of migrating to other, more well-founded implementation techniques for SPL engineering, such as aspects [17, 24] or various flavors of feature modules [5, 6].

To answer the questions above, we analyzed forty open-source software systems of different sizes (thousands to millions lines of code) taken from different domains including operating systems, database systems, and compilers. We propose a set of metrics that allow us to infer and classify cpp usage patterns and to map the patterns to common SPL implementation concepts. Our analysis reveals that cpp is used to a large extent to implement and control variability

ICSE'10, May 2-8 2010, Cape Town, South Africa



Preprocessor code is complex



An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines

Jörg Liebig, Sven Apel, and Christian Lengauer
University of Passau
{joliebig,apel,lengauer}@fim.uni-passau.de

Christian Kästner and Michael Schulze
University of Magdeburg
{ckaestne,mschulze}@ovgu.de

ABSTRACT

Over 30 years ago, the preprocessor cpp was developed to extend the programming language C by lightweight metaprogramming capabilities. Despite its error-proneness and low abstraction level, the preprocessor is still widely used in present-day software projects to implement variable software. However, not much is known about *how* cpp is employed to implement variability. To address this issue, we have analyzed forty open-source software projects written in C. Specifically, we answer the following questions: How does program size influence variability? How complex are extensions made via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? We argue that insights into the implementation problems of features solved with cpp help to judge whether cpp usage causes problems with regard to code pollution, error-proneness, and reduced maintainability and ability to evolve. An analysis of cpp usage also allows developers to estimate the effort and benefits of migrating to other, more well-founded implementation techniques for SPL engineering, such as aspects [17, 24] or various flavors of feature modules [5, 6].

To answer the questions above, we analyzed forty open-source software systems of different sizes (thousands to millions lines of code) taken from different domains including operating systems, database systems, and compilers. We propose a set of metrics that allow us to infer and classify cpp usage patterns and to map the patterns to common SPL implementation concepts. Our analysis reveals that cpp is used to a large extent to implement and control variability

Liebig et al. @ ICSE'10

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.8 [**Software Engineering**]: Metrics; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Empirical Study

Keywords

Software Product Lines, C Preprocessor

1. INTRODUCTION

The C preprocessor (cpp) is a popular tool for implementing variable software. It has been developed to enhance C by lightweight metaprogramming capabilities and is commonly used to merge files, make arbitrary textual substitutions, and define conditional code fragments (a.k.a. conditional inclusion) [21]. As the cpp tool is line-based, it can be used with any text artifact including other programming languages such as Java or C#. In the past, it has been observed that the use of cpp causes various problems: (1) the occurrence of syntactic and semantic errors during the generation of software products [23]; (2) code pollution due to scattered and tangled #ifdefs (a.k.a. #ifdef hell) [32]; (3) a decrease in maintainability and in ability to evolve [13].

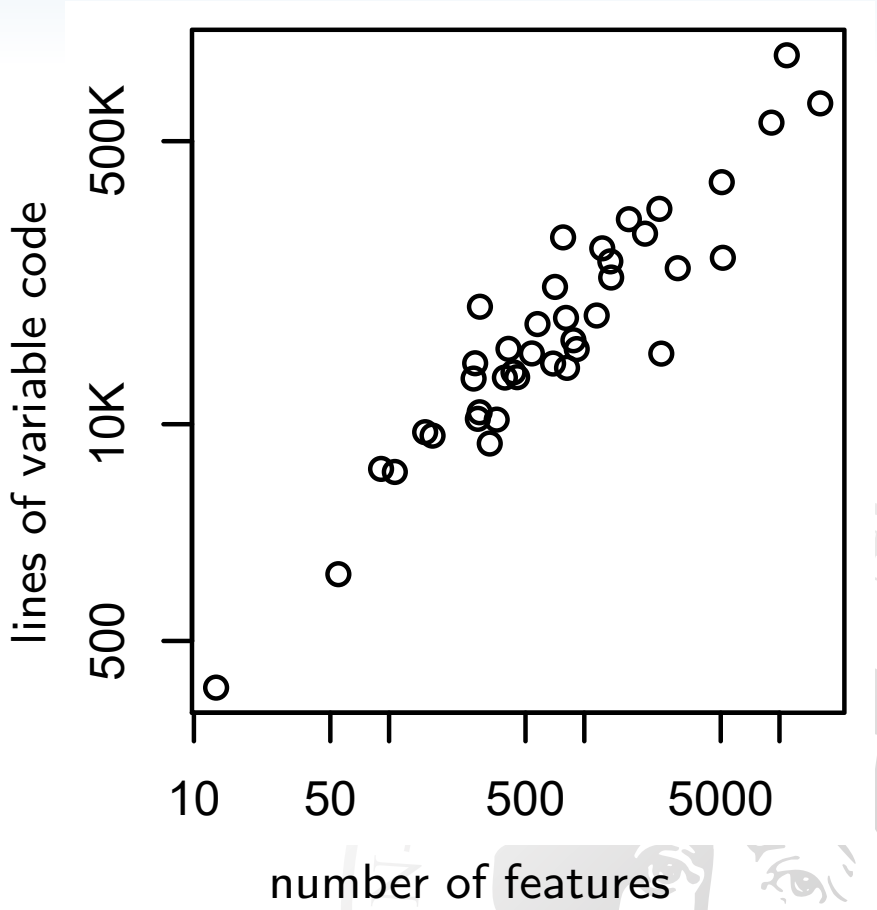
The implementation of variable software is also a major goal of software product line engineering. A *software product line (SPL)* is a set of software-intensive systems sharing a

assets as possible, such as source code artifacts [8, 11, 27].

It is widely assumed that the variability mechanism of the cpp tool is used quite frequently in the implementation of SPLs [1, 12, 18, 33]. We believe this assumption to be true and contribute to the discussions on the connection between preprocessors and SPLs started in prior work with a substantial set of case studies. We answer the following questions: How does program size influence variability of SPLs? How complex are the extensions applied by features via cpp's variability mechanisms? At which level of granularity are extensions applied? Which types of extension occur? We argue that insights into the implementation problems of features solved with cpp help to judge whether cpp usage causes problems with regard to code pollution, error-proneness, and reduced maintainability and ability to evolve. An analysis of cpp usage also allows developers to estimate the effort and benefits of migrating to other, more well-founded implementation techniques for SPL engineering, such as aspects [17, 24] or various flavors of feature modules [5, 6].

To answer the questions above, we analyzed forty open-source software systems of different sizes (thousands to millions lines of code) taken from different domains including operating systems, database systems, and compilers. We propose a set of metrics that allow us to infer and classify cpp usage patterns and to map the patterns to common SPL implementation concepts. Our analysis reveals that cpp is used to a large extent to implement and control variability

ICSE'10, May 2-8 2010, Cape Town, South Africa

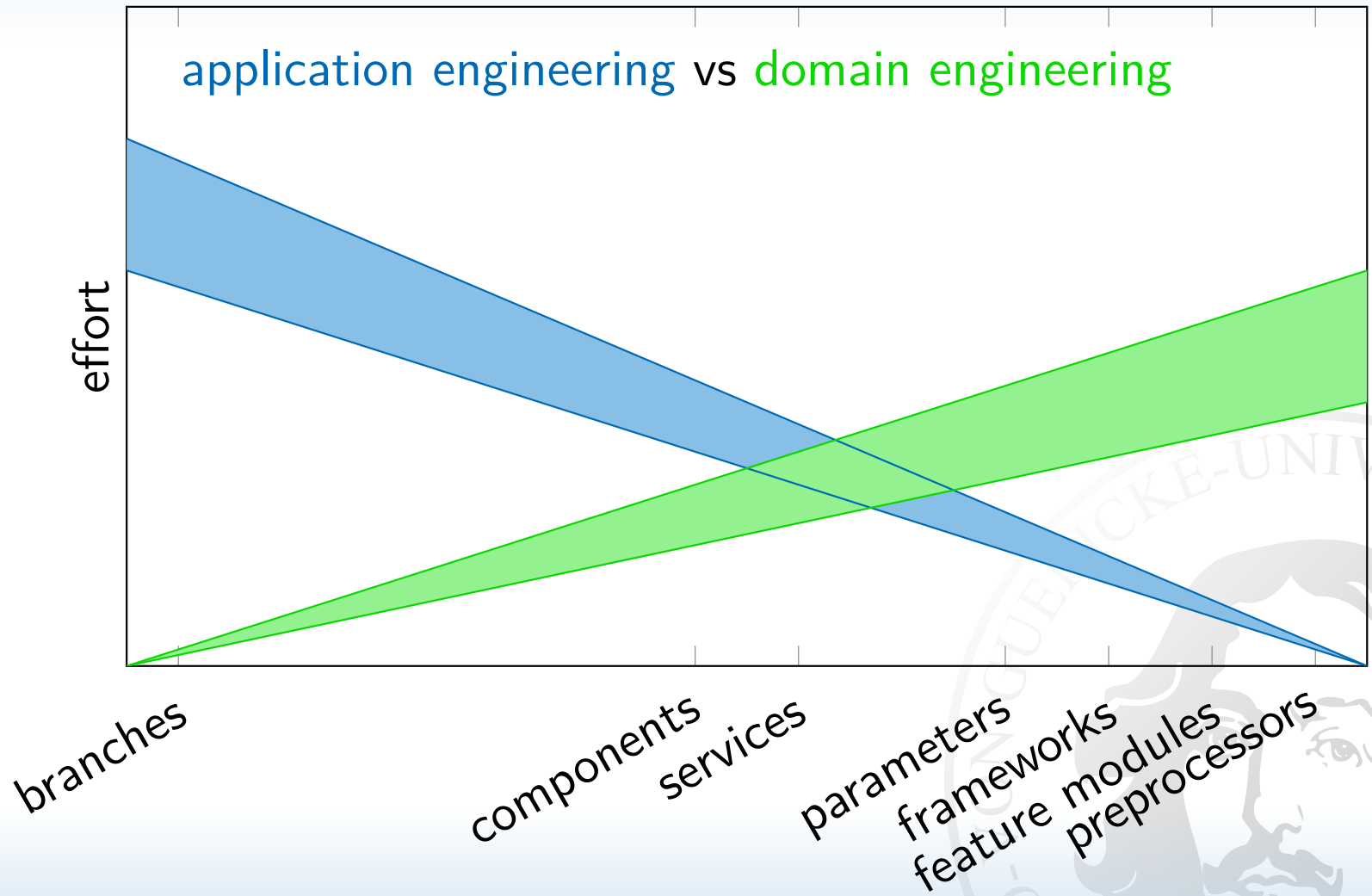


The #ifdef hell

Part III

Feature-oriented software development

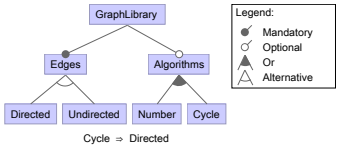
Application engineering vs domain engineering



Heterogeneous software artifacts

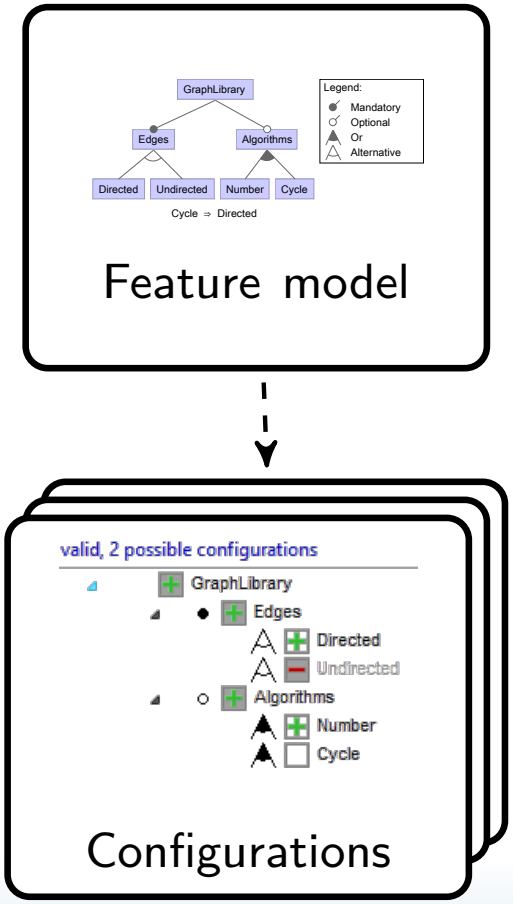
- models (in this tutorial: feature models)
- source code (in this tutorial: Java)
- binaries (in this tutorial: Java Bytecode)
- tests (in this tutorial: JUnit)
- specification
- documentation

Feature-oriented software development

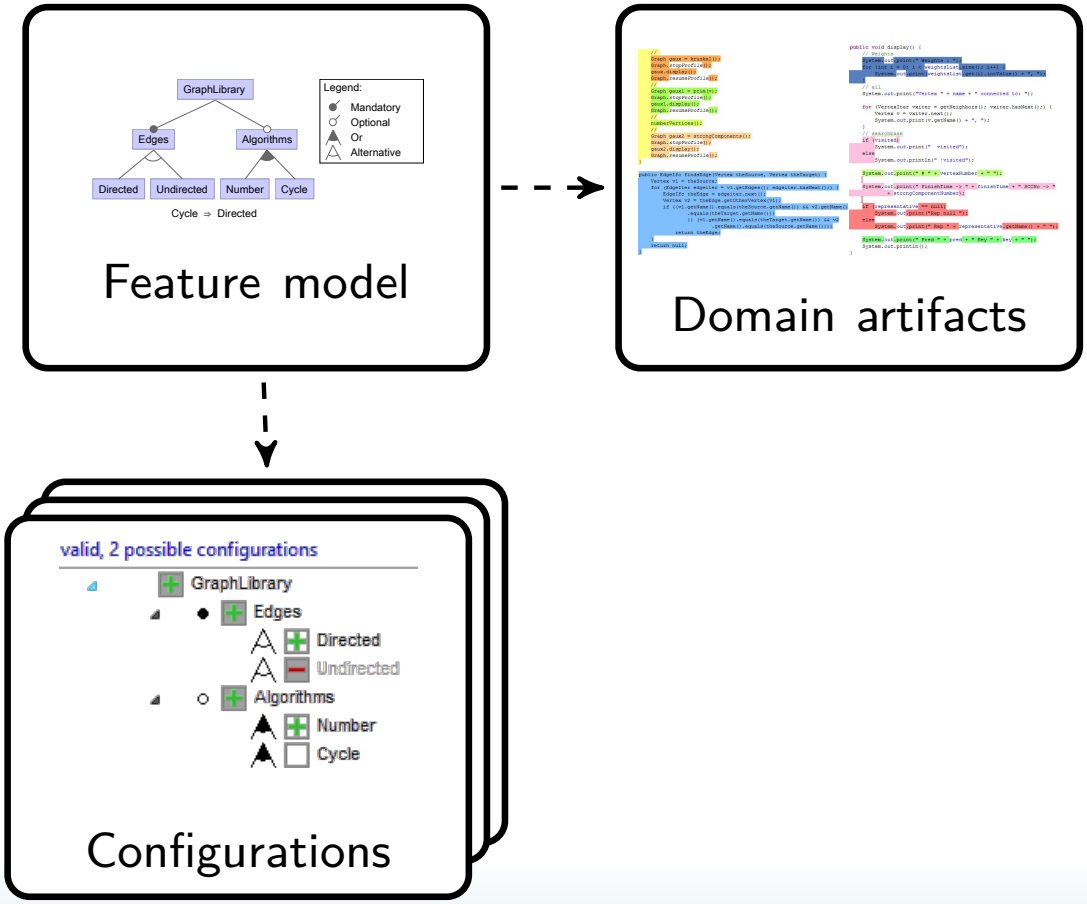


Feature model

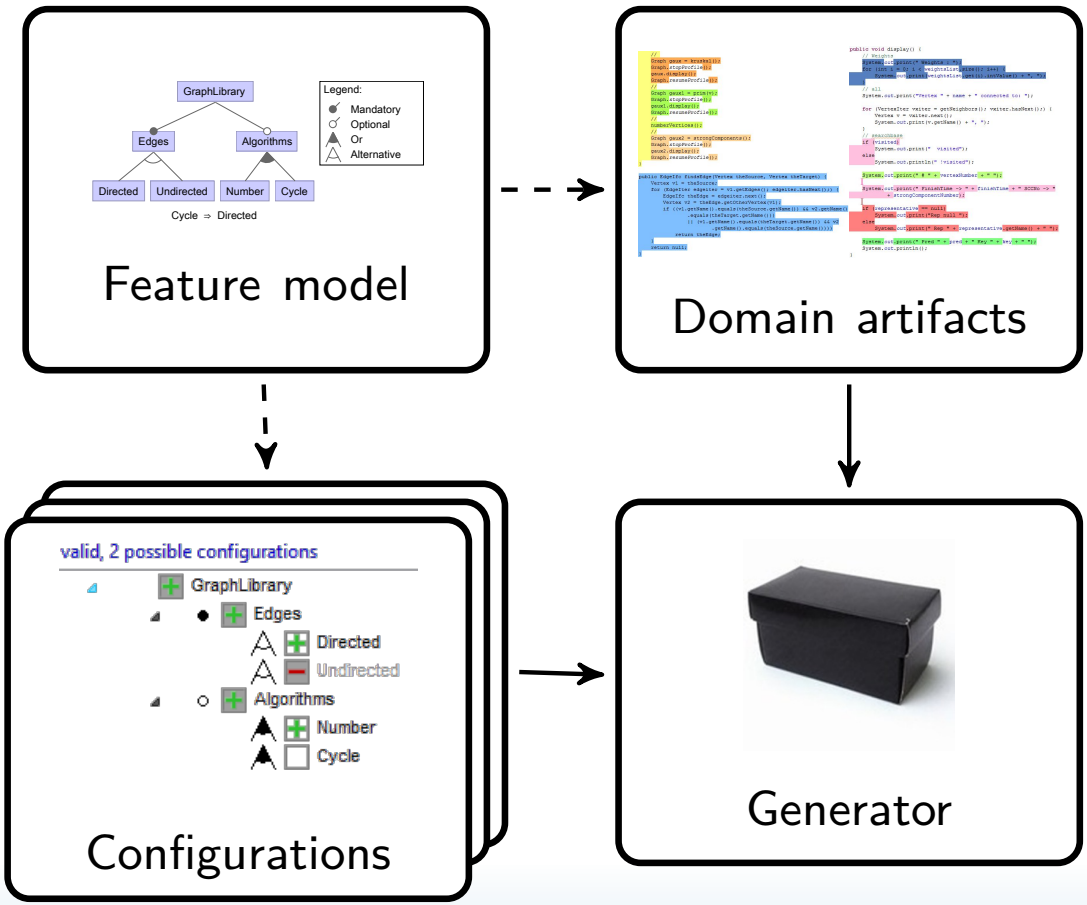
Feature-oriented software development



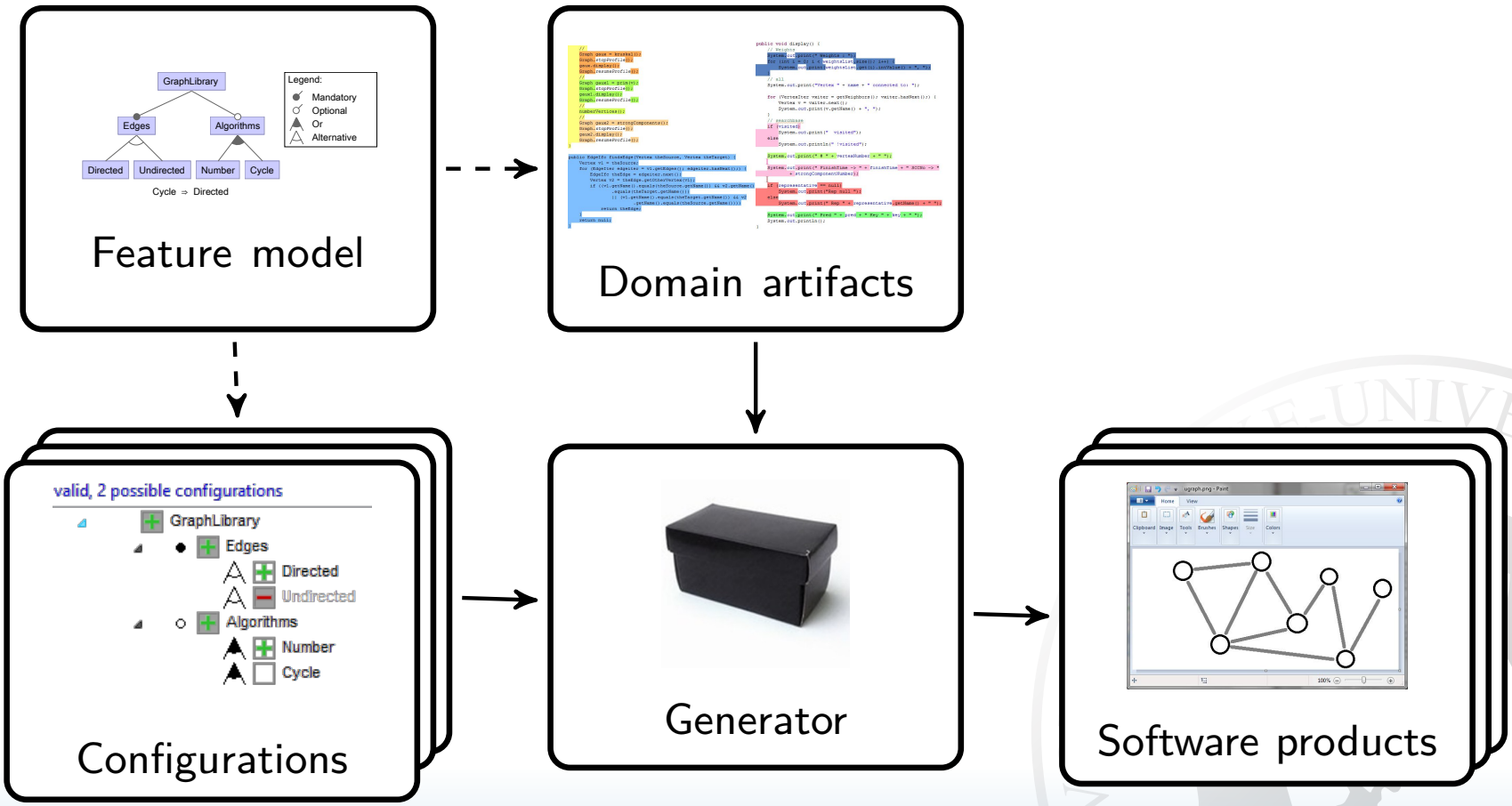
Feature-oriented software development



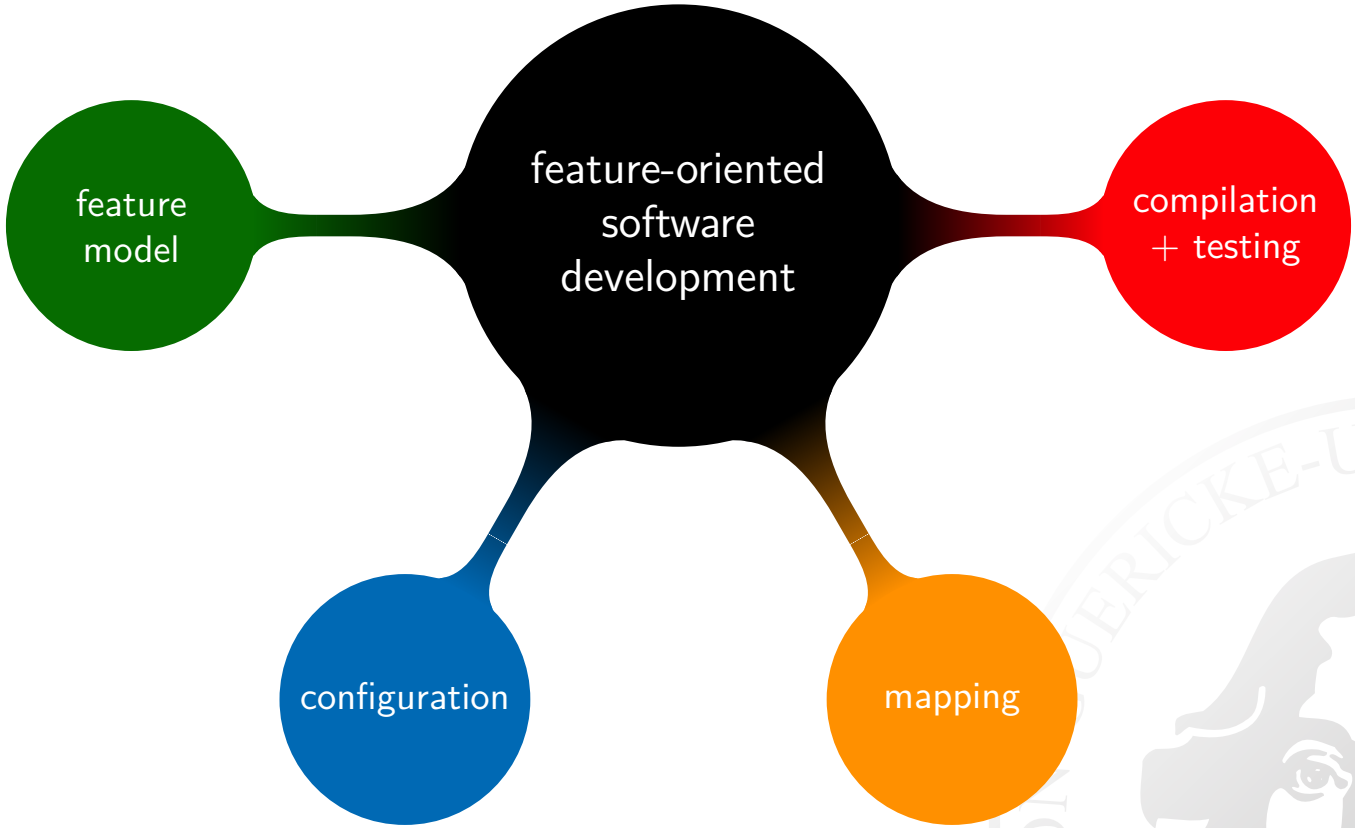
Feature-oriented software development



Feature-oriented software development



Tutorial overview

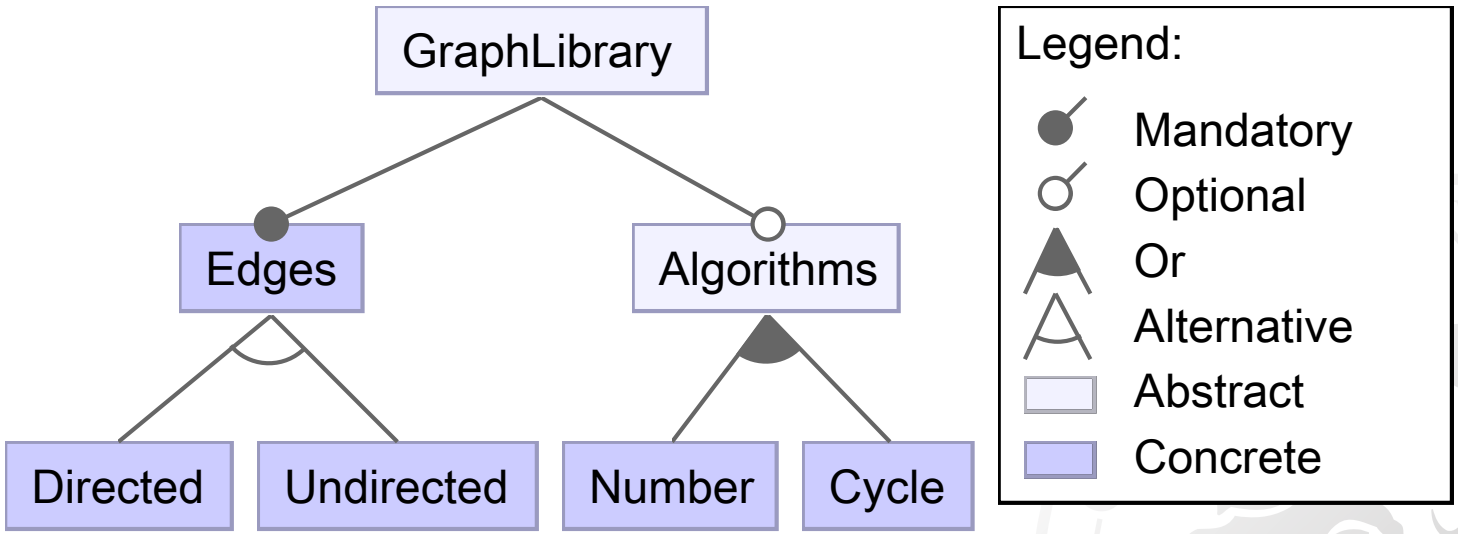


Hands-on I: run different elevator configurations

1. Install Java 1.7 or higher (if not yet installed)
2. Copy/download and unzip FeatureIDE for your OS
3. Start FeatureIDE by running Eclipse
4. Open default workspace (i.e., `../workspace`)
5. Import example `Elevator-Antenna-v1.0` using the menu `File` > `New` > `Example` > `FeatureIDE Examples`
6. Run and try the elevator: right click on folder `src` > Run As > Java Application
7. Change configuration: right click on `*.xml` in folder `configs` > `FeatureIDE` > `Set as current configuration`
8. Repeat steps 6 and 7 to experience differences of elevator configurations

Part IV

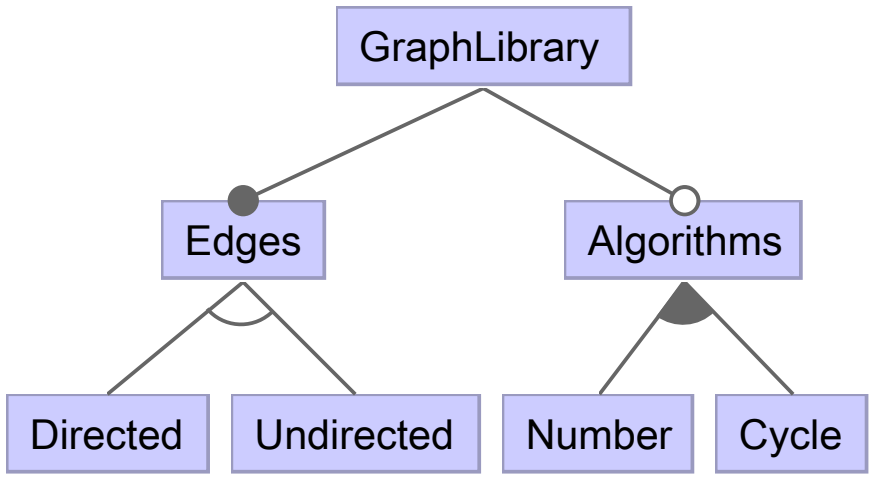
Smells in feature modeling



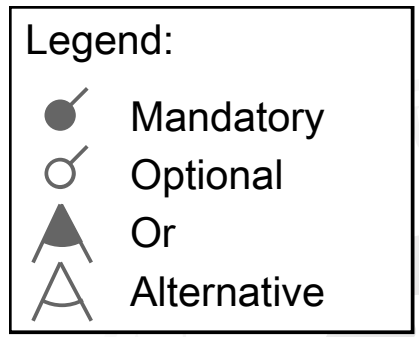
Cycle $\Rightarrow$ Directed

Feature model

- Specifies valid combinations of features
- Graphically represented by a feature diagram
- Defines the scope/domain of a software product line

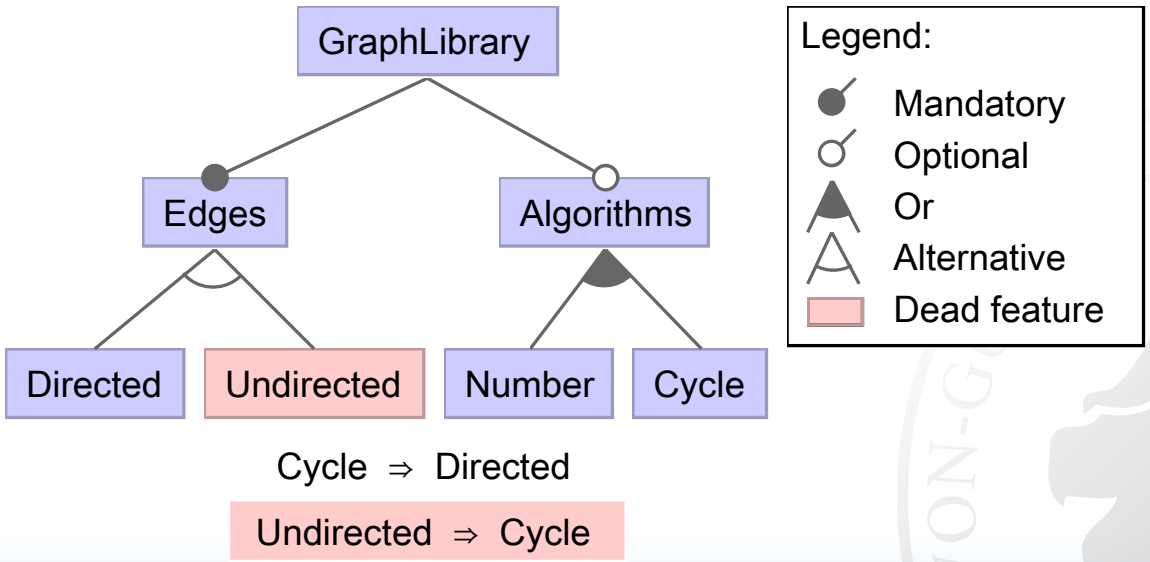


Cycle ⇒ Directed



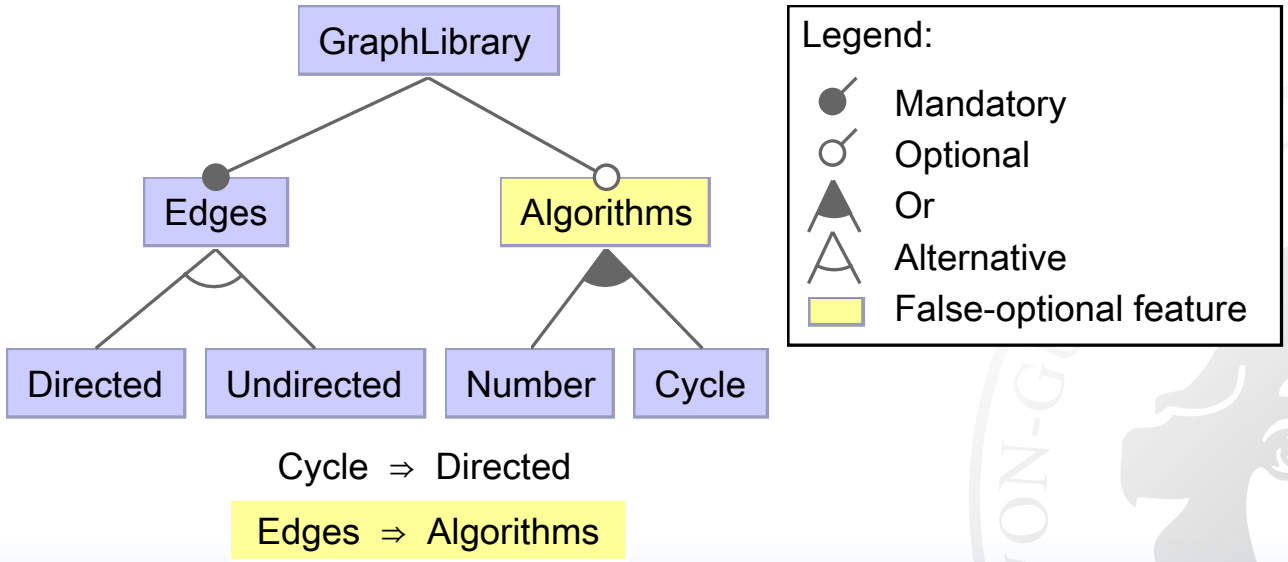
Dead features

- There is no valid configuration containing that feature
- Frustrating during configuration, may result in dead code
- Fix: remove feature (refactoring) or remove/edit constraints



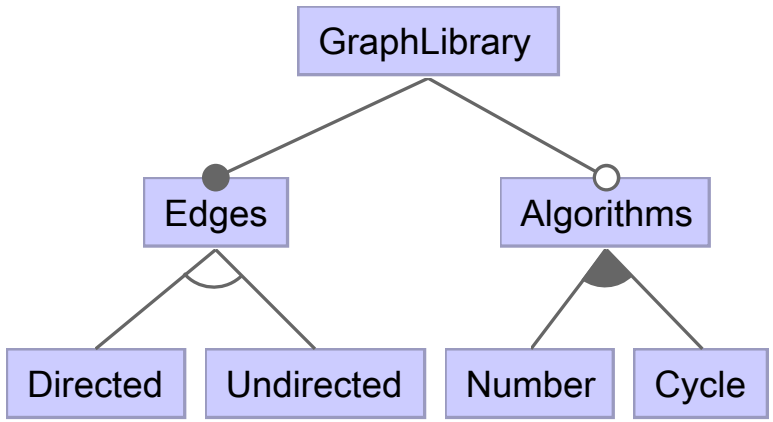
False-optional features

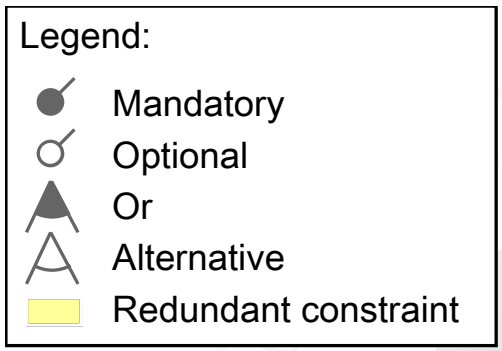
- Feature is modeled as optional, but is effectively mandatory
- Suggest more variability than actually available
- Fix: make the feature mandatory to avoid confusion



Redundant constraints

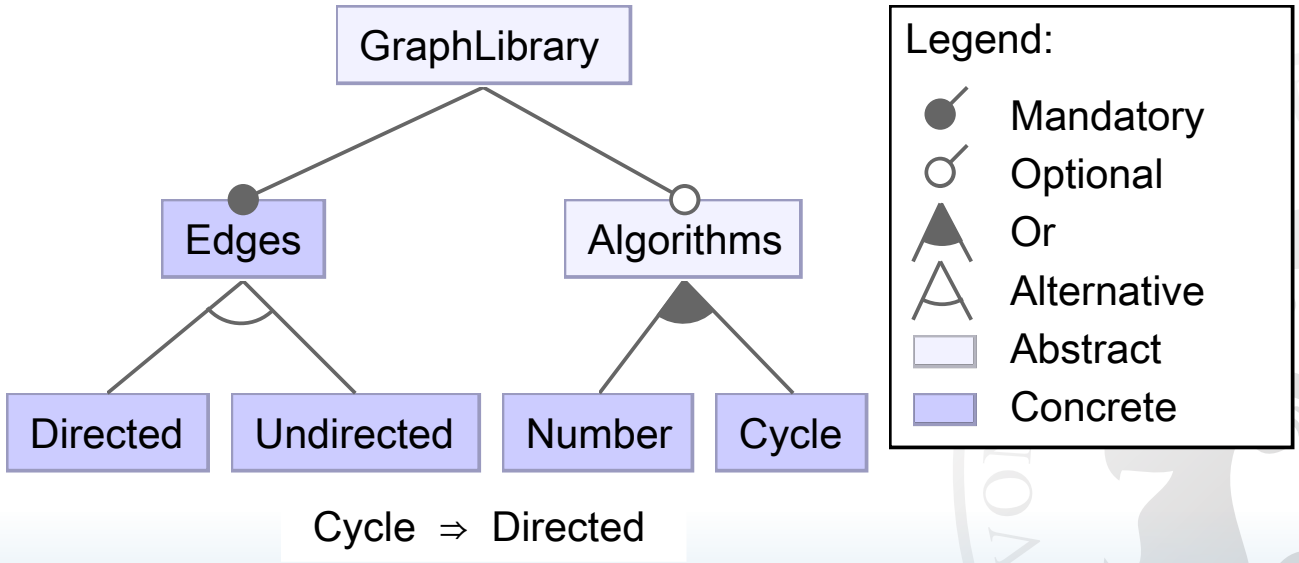
- Constraint does not influence valid configurations
- Model is more verbose than necessary
- Fix: remove it (refactoring) or edit/remove other constraints



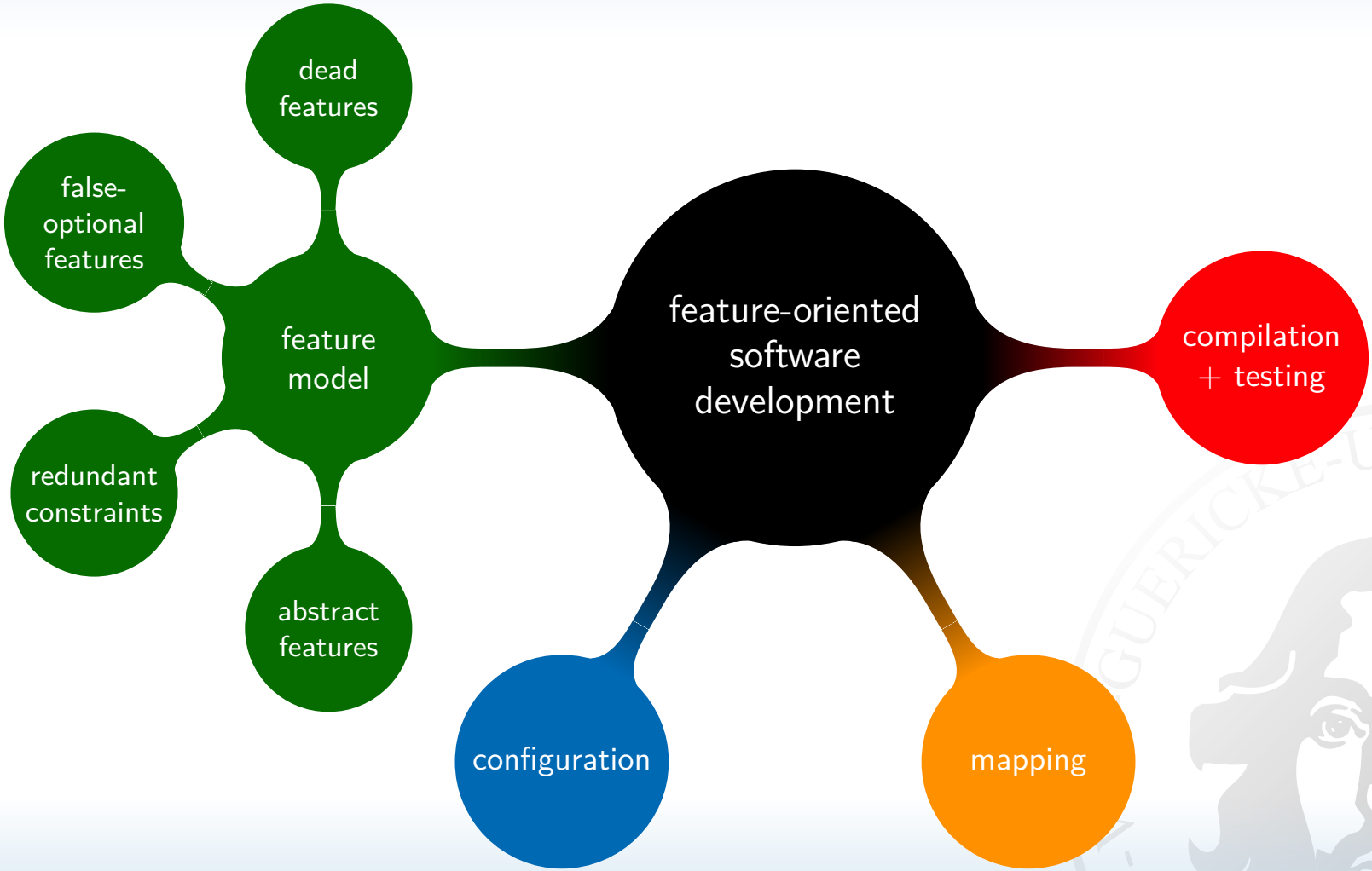


Abstract and concrete features

- Feature is called concrete if it is mapped to domain artifacts and abstract otherwise
- Helps to identify mismatch between modeled and actually implemented variability, used for some analyses (sampling)

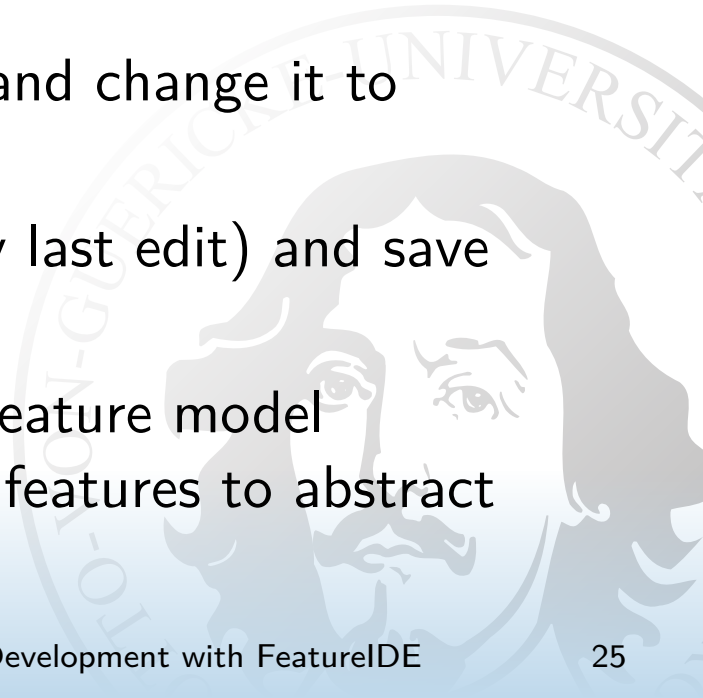


Tutorial overview



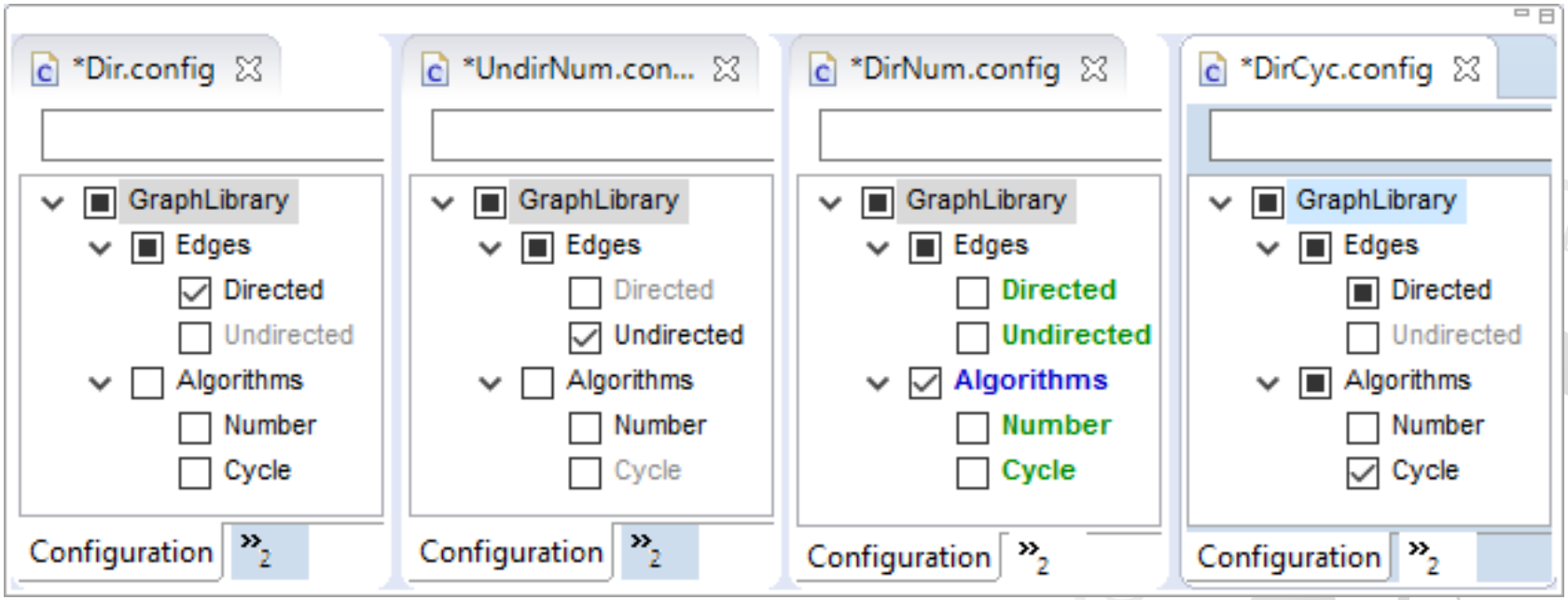
Hands-on II: fix smells in the feature model

1. (Continue with previous project `Elevator-Antenna-v1.0`)
2. Open the feature model (i.e., file `model.xml`)
3. Select a feature to see constraints making the feature model void and remove the last constraint
4. Select dead feature *Overloaded* and remove the highlighted constraint
5. Select false-optional feature *Permission* and change it to mandatory
6. Remove redundant constraint (caused by last edit) and save the feature model
7. Fix warning in problems view regarding feature model (model.xml) by changing the mentioned features to abstract



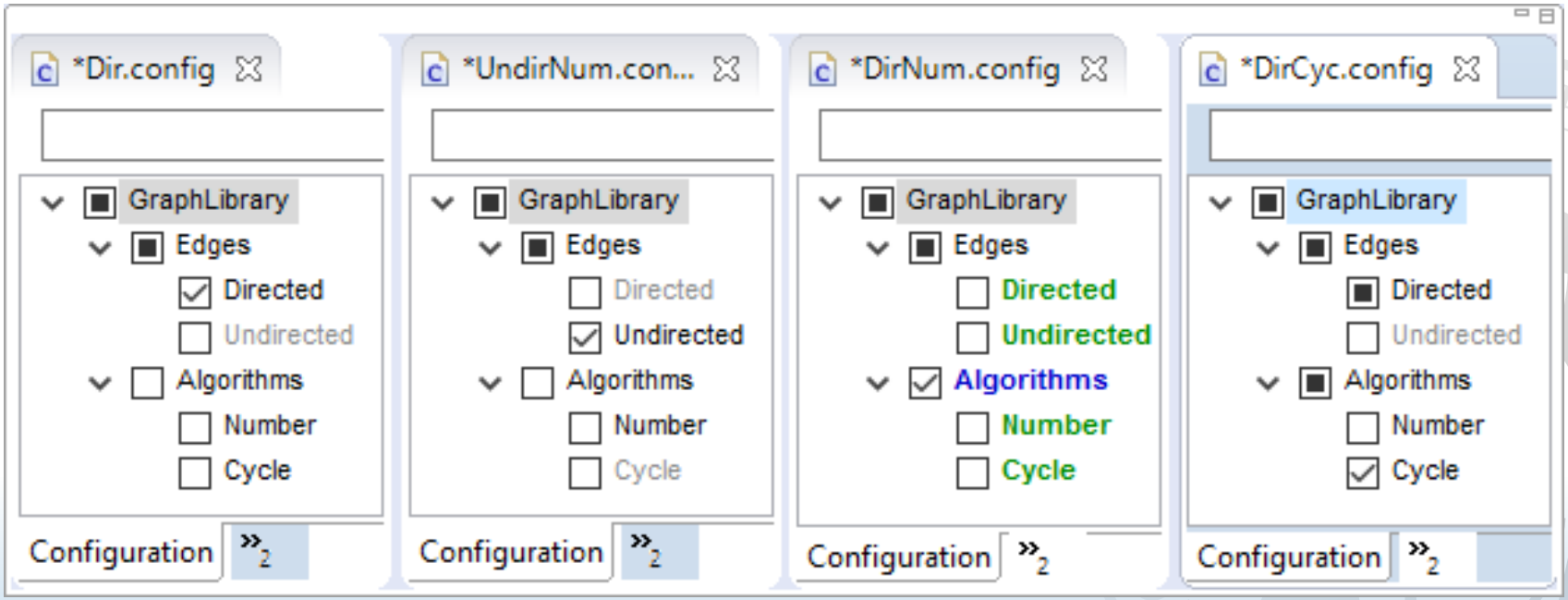
Part V

Smells in configurations



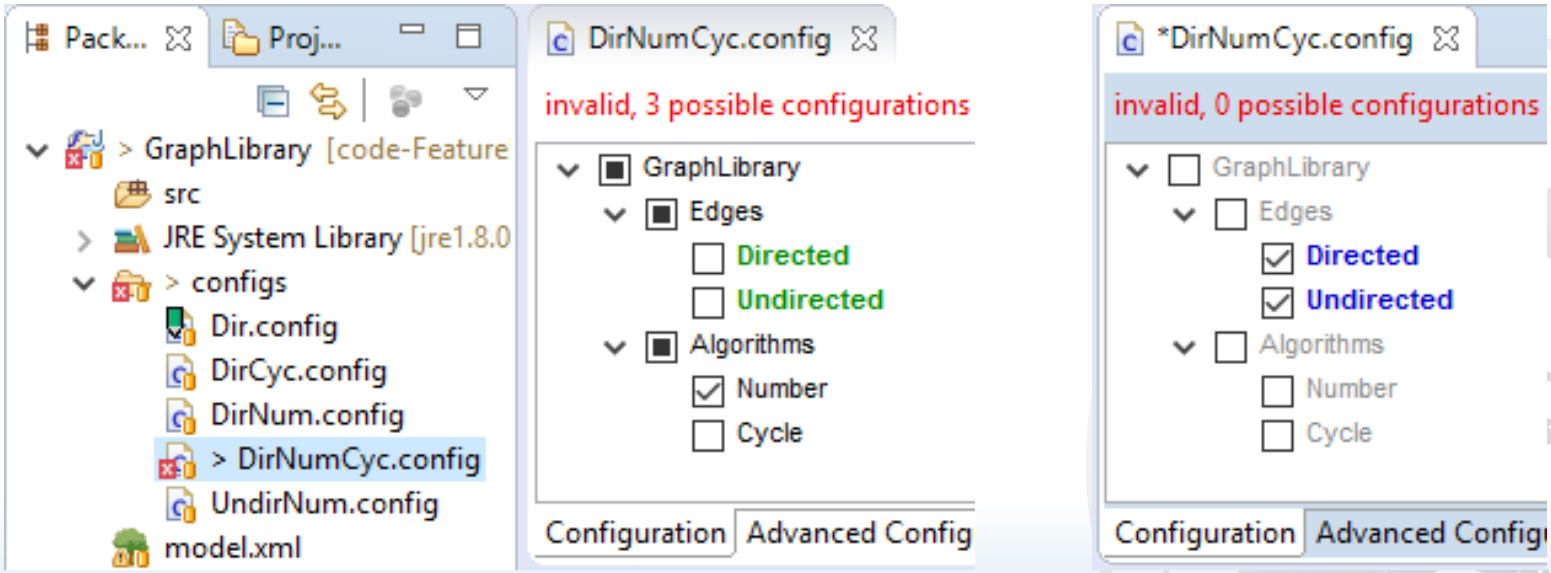
Creating configurations

- Possible to manually select features
- Automatic selections and deselections are computed
- Validity with respect to feature model is computed
- Green/blue indicate that selection/deselection will lead to valid configuration



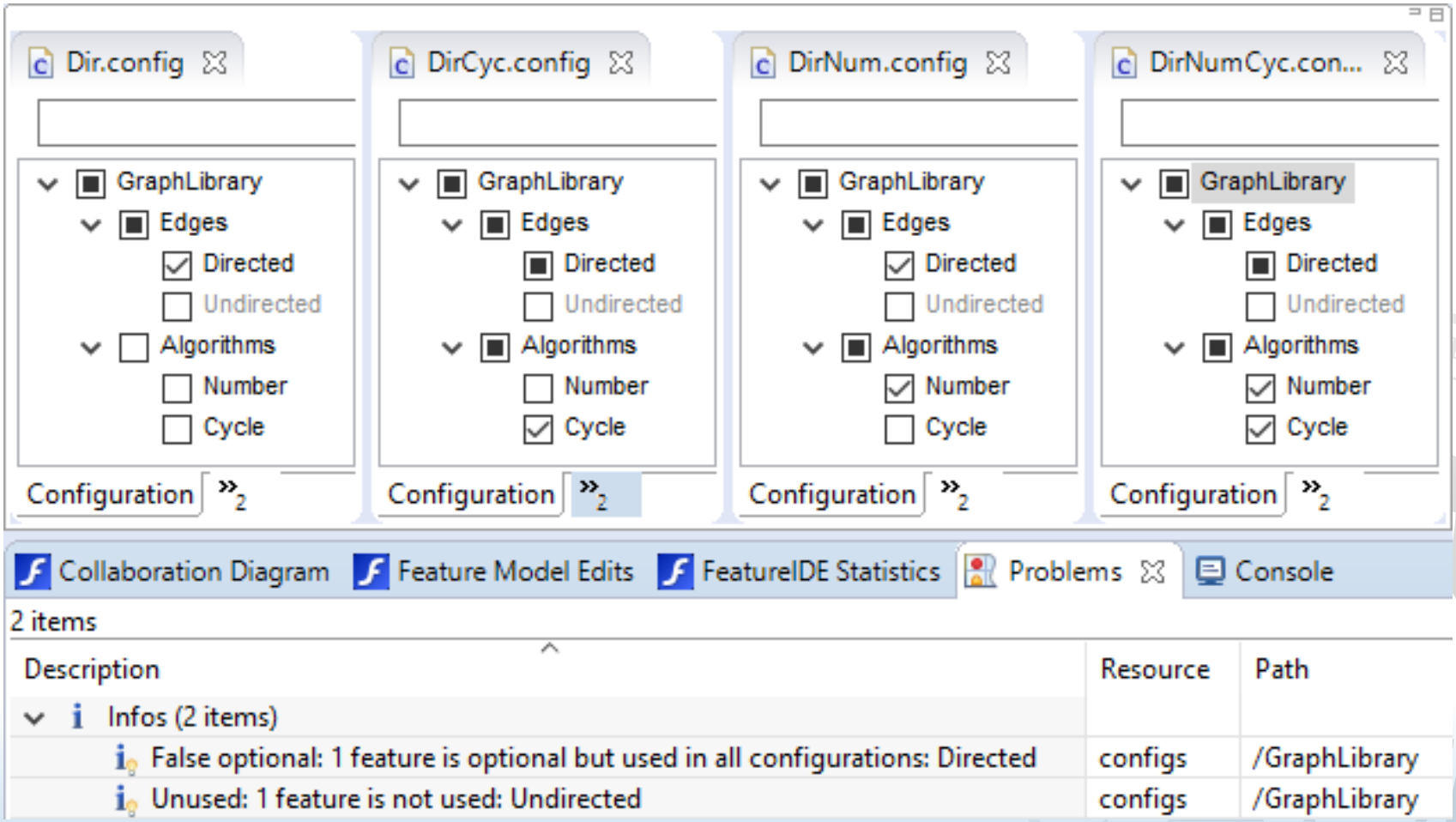
Invalid configurations

- Selection of features not allowed in feature model
- Saved invalid or invalid by change in feature model
- Computed whenever feature model or configurations change
- Missing selections (green), conflicting selections (blue)

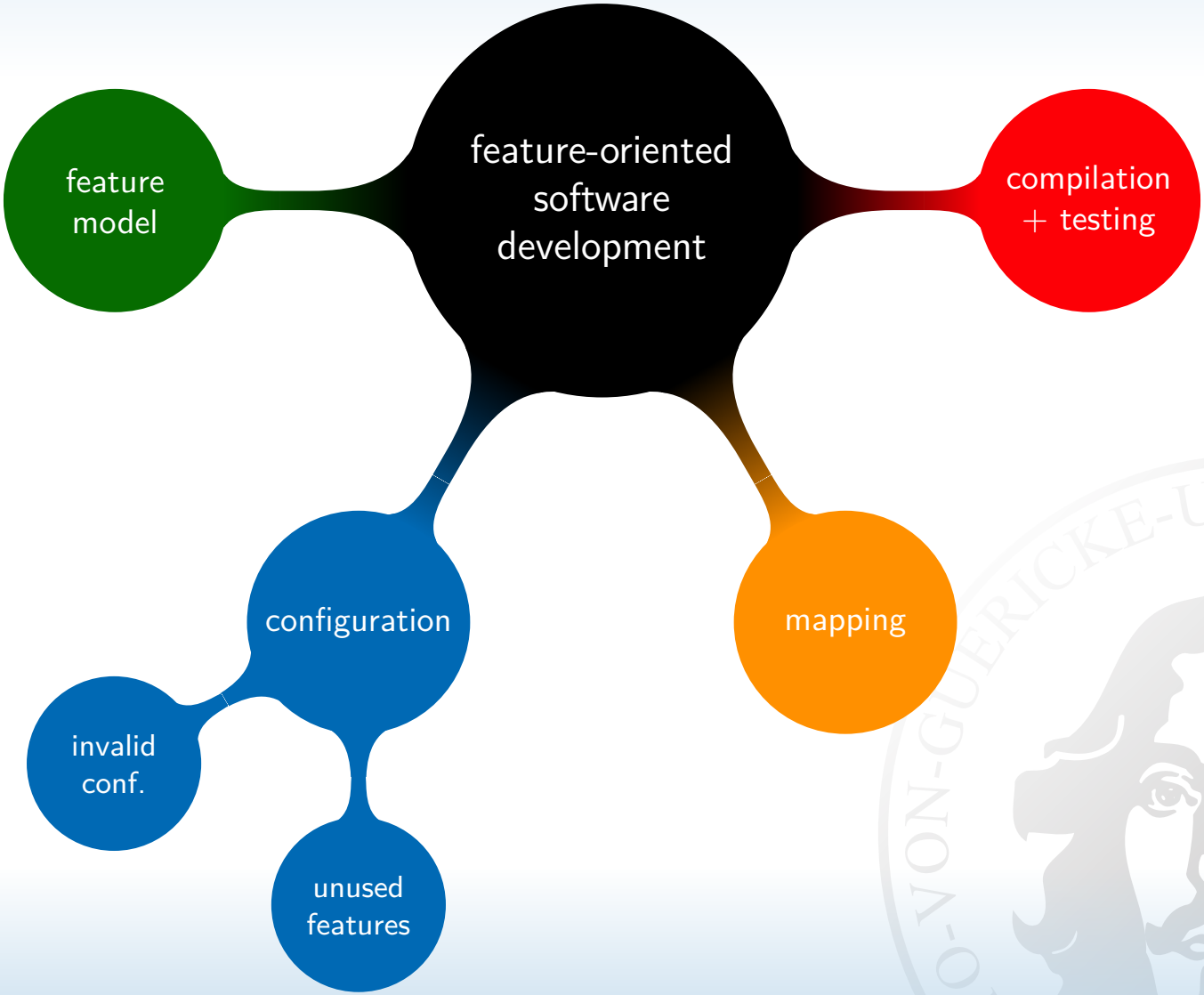


Unused features

- Unused feature is not selected in any configuration
- Could indicate unnecessary or untested code



Tutorial overview

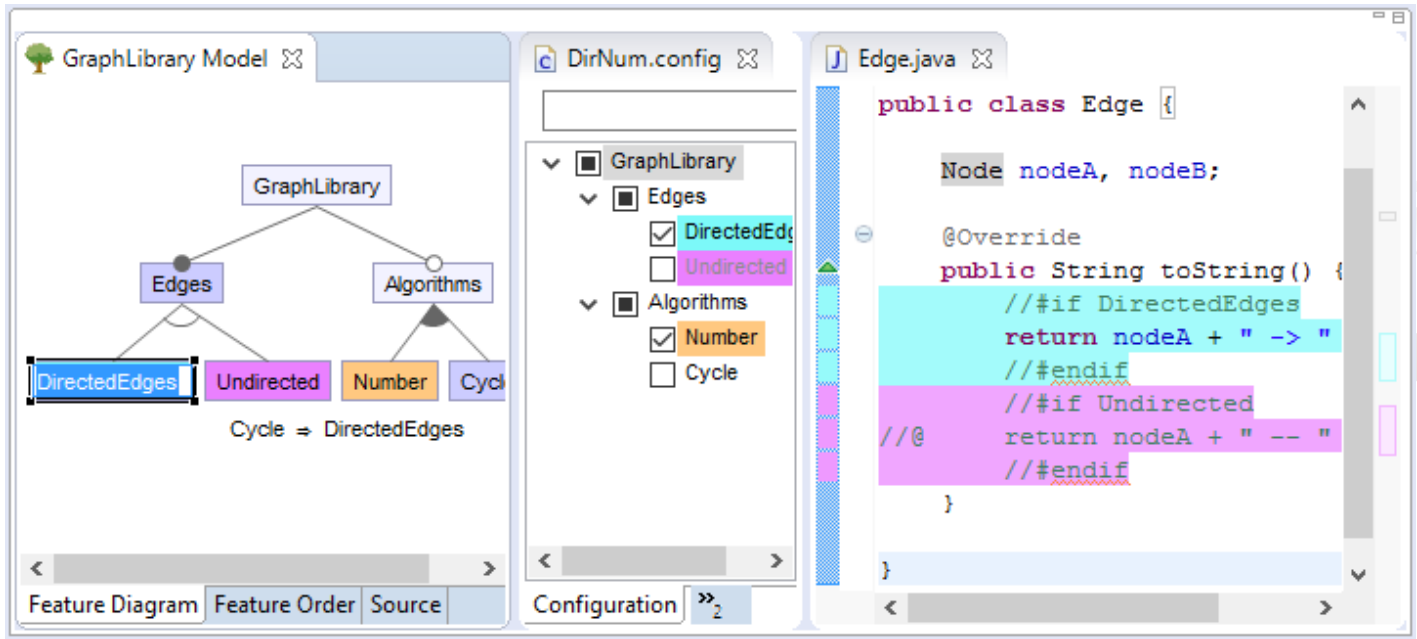


Hands-on III: fix smells of configurations

1. (Continue with previous project or import `Elevator-Antenna-v1.1`)
2. Create a new configuration called `Professional` with menu `File` > `New` > `Configuration File`
3. Select the features `FIFO`, `DirectedCall`, `FloorPermission`, save and close the editor
4. Open feature model, use context menu to change group type below feature `Modes` to an alternative, and save the model
5. Open the invalid configuration `Professional`, deselect feature `FIFO`, and save it
6. Problem view indicates unused features, use `Quick Fix` in context menu to create a configuration with feature `Sabbath`, rename configuration to `Starter` and open it

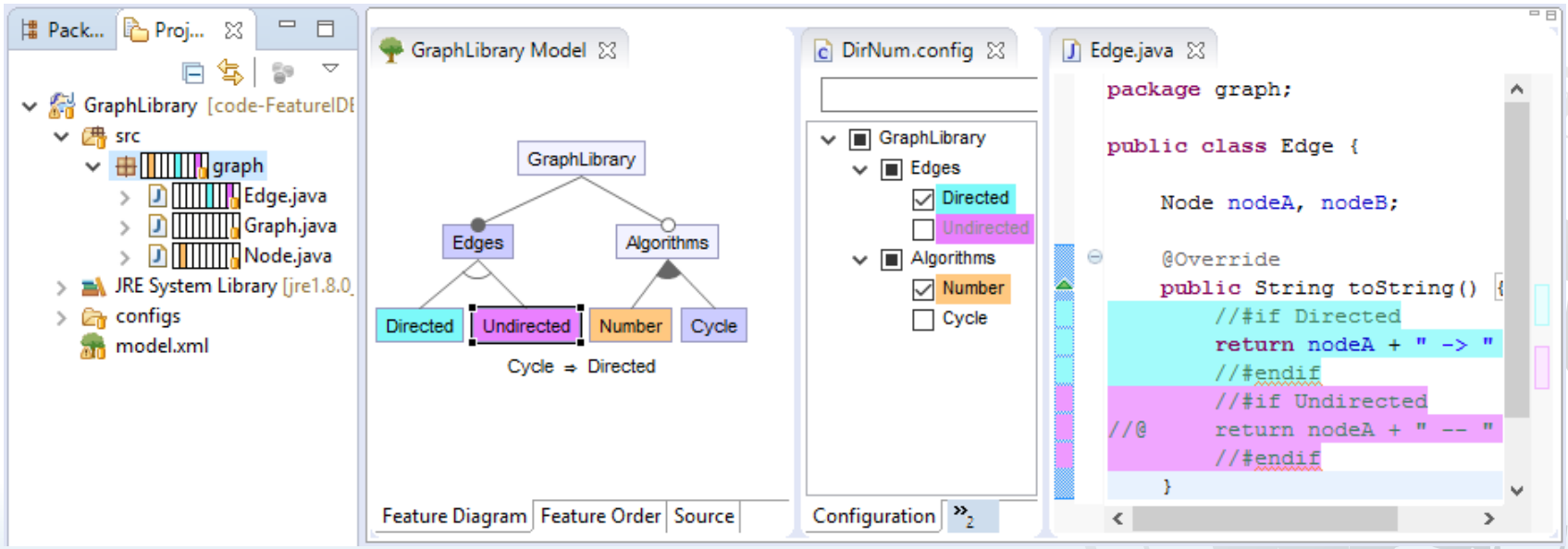
Part VI

Smells in feature-to-code mapping



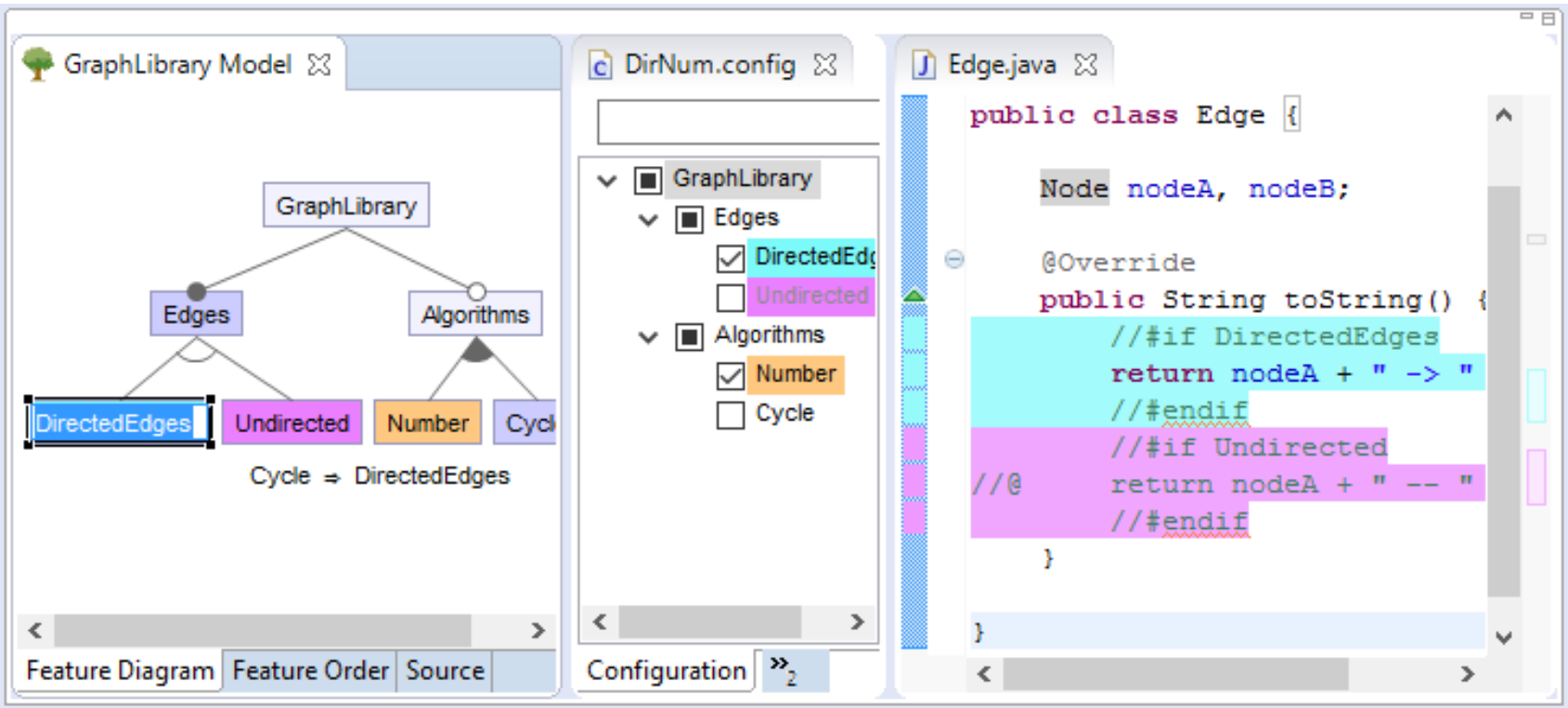
Feature traceability problem

- Feature traceability is hard to establish for large projects
- Product-line maintenance typically requires to trace features
- Manually assigned colors help to locate features in feature model, configurations, packages, and source code



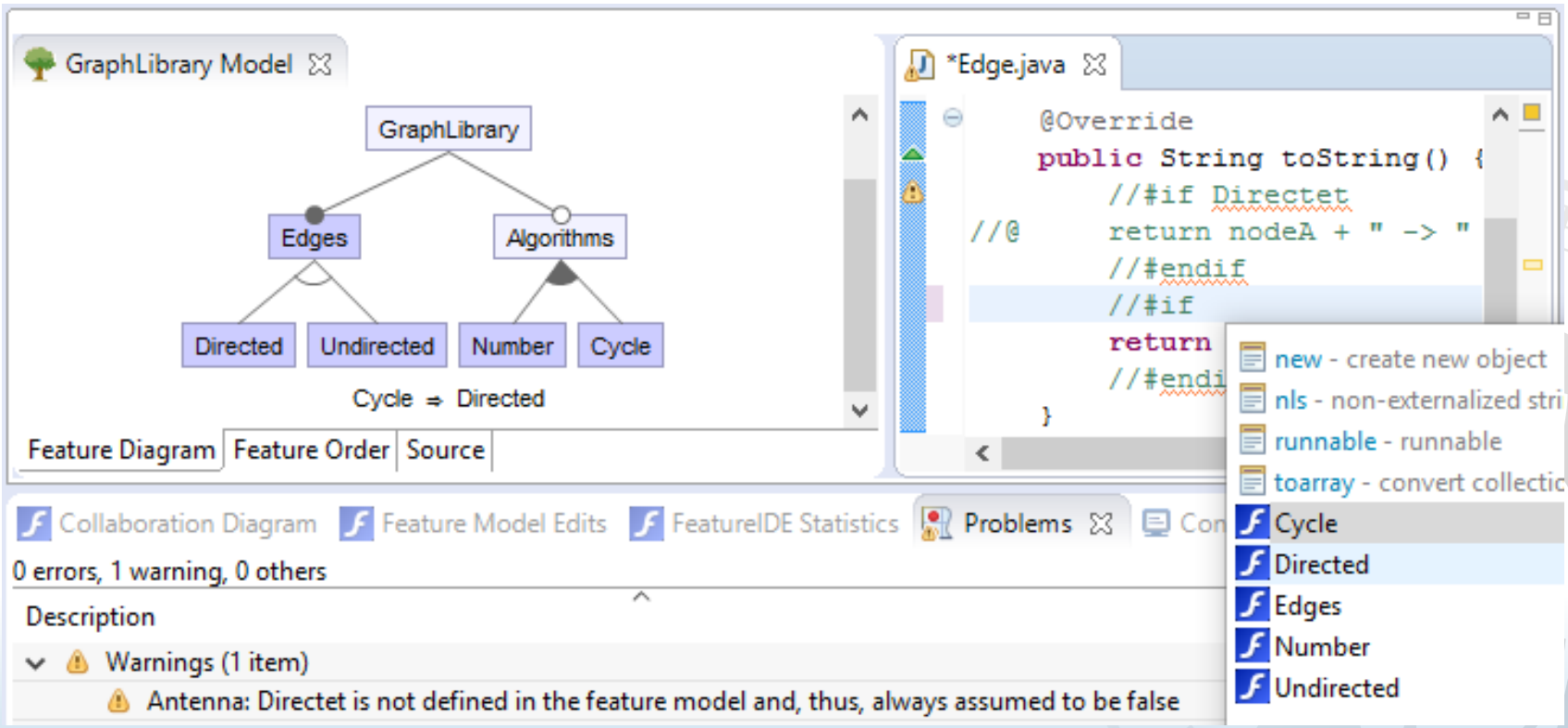
Misleading feature names

- During evolution, features change what they used to be
- Misleading names create confusion and obfuscate artifacts
- FeatureIDE renames features automatically in all artifacts



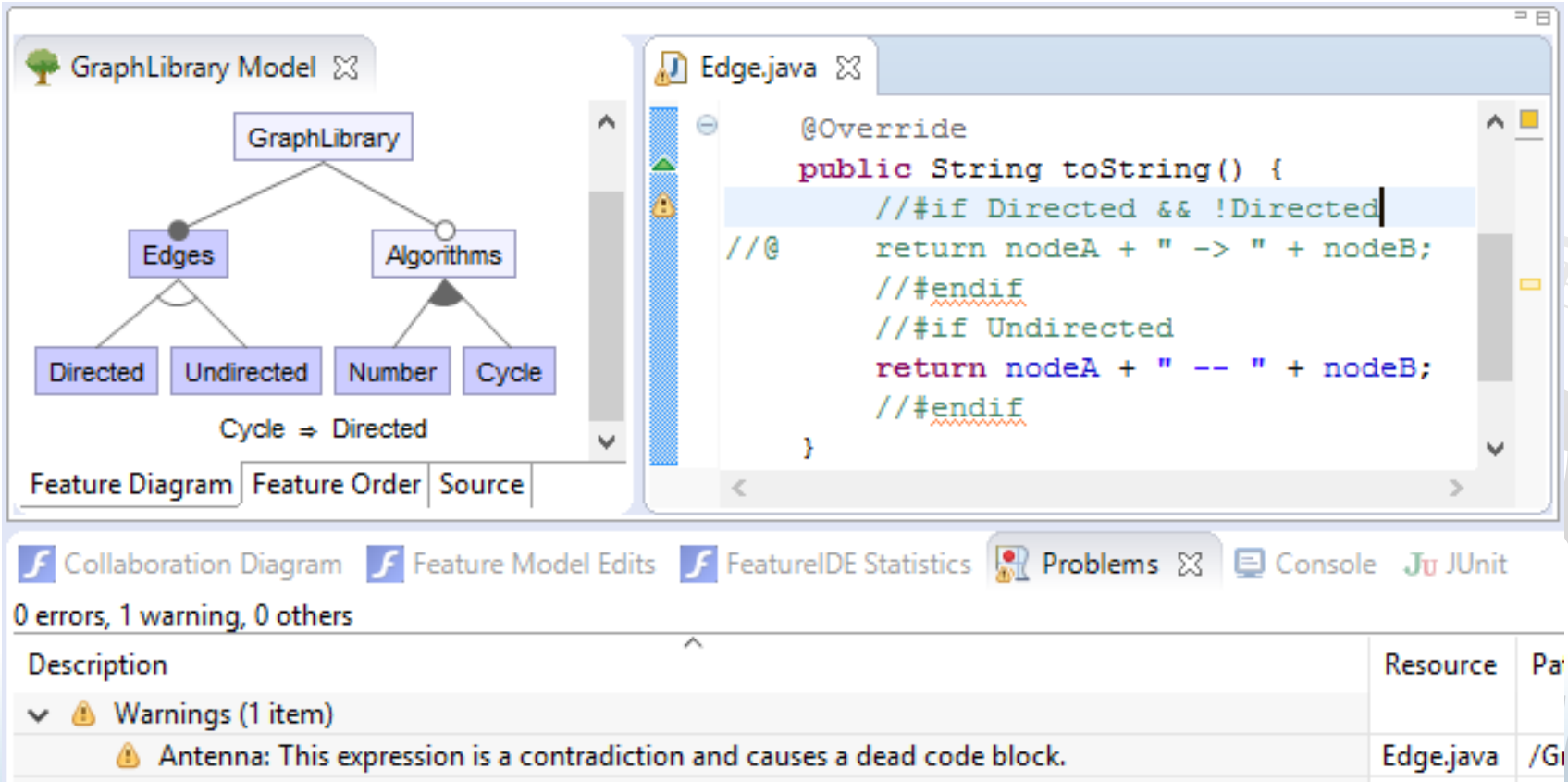
Undefined feature references

- Features referenced in source code that do not exist in the feature model are always assumed to be false
- May lead to code being maintainted but never used
- Content assist and warnings support developers to avoid them



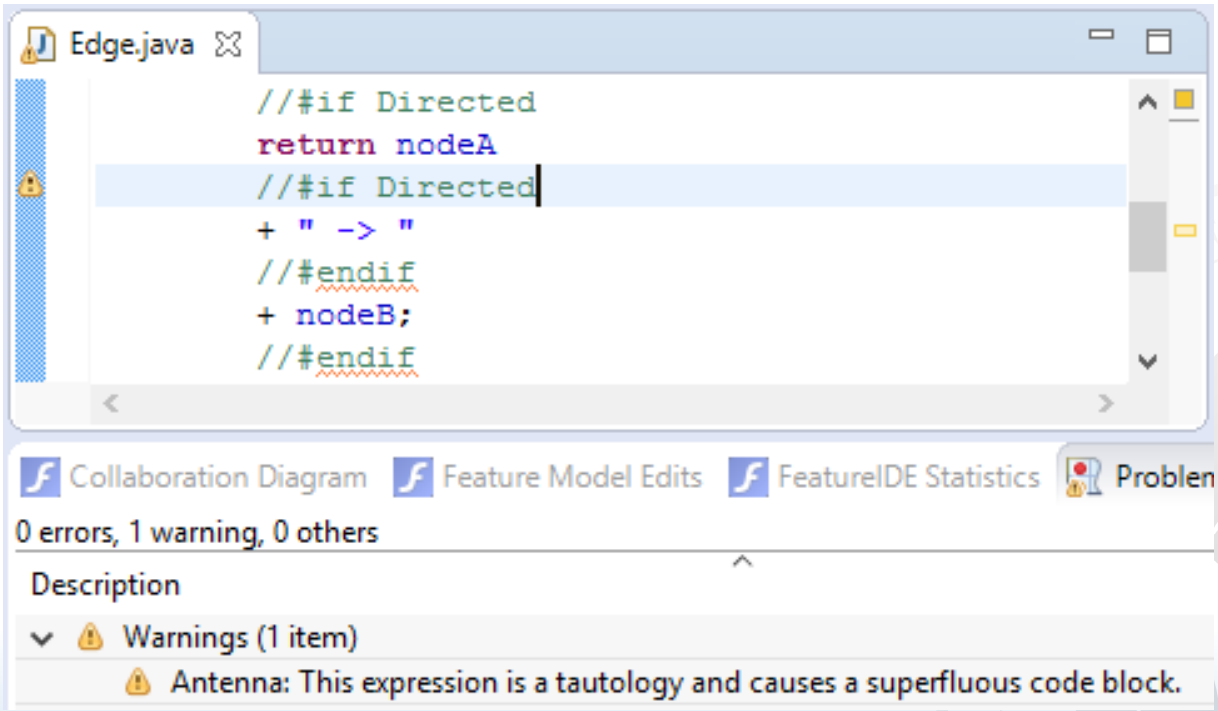
Dead code blocks

- Even if all features are defined, some blocks are never selected
- Contradictions typically arise due to nesting or feature model
- Similar to unreachable code in a programming language



Superfluous preprocessor annotations

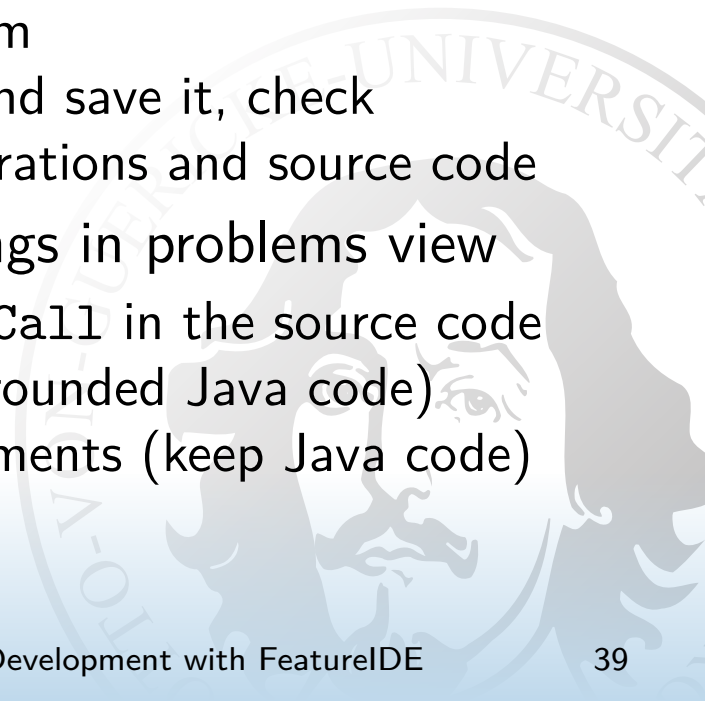
- Depending on nesting and feature model, some blocks may always be enabled making the preprocessor annotations useless
- Make source code unnecessarily complex and can be removed



Tutorial overview

Hands-on IV: fix smells in mapping

1. (Continue with previous project or import `Elevator-Antenna-v1.2`)
2. Trace the misspelled feature `ShortestPath` in source code
   - 2.1 Open feature model and use context menu to assign a color to feature `ShortestPath`
   - 2.2 Identify source files containing that feature in project (*not package*) explorer and open one of them
   - 2.3 Rename the feature in feature model and save it, check whether renaming is applied to configurations and source code
3. Locate and fix issues identified as warnings in problems view
   - 3.1 Fix misspelled feature name `DirectedCall` in the source code
   - 3.2 Remove dead code blocks (remove surrounded Java code)
   - 3.3 Remove superfluous preprocessor statements (keep Java code)

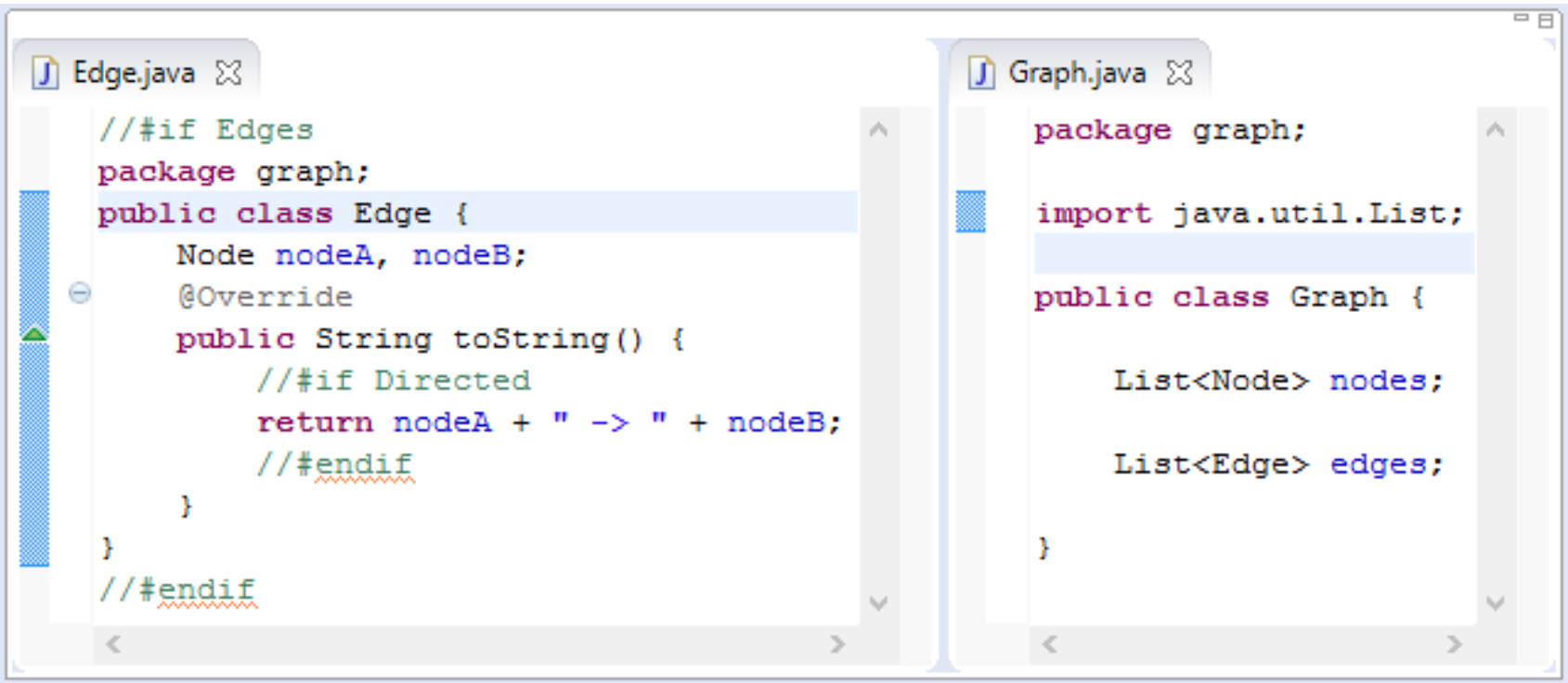


Part VII

Smells in compilation and testing

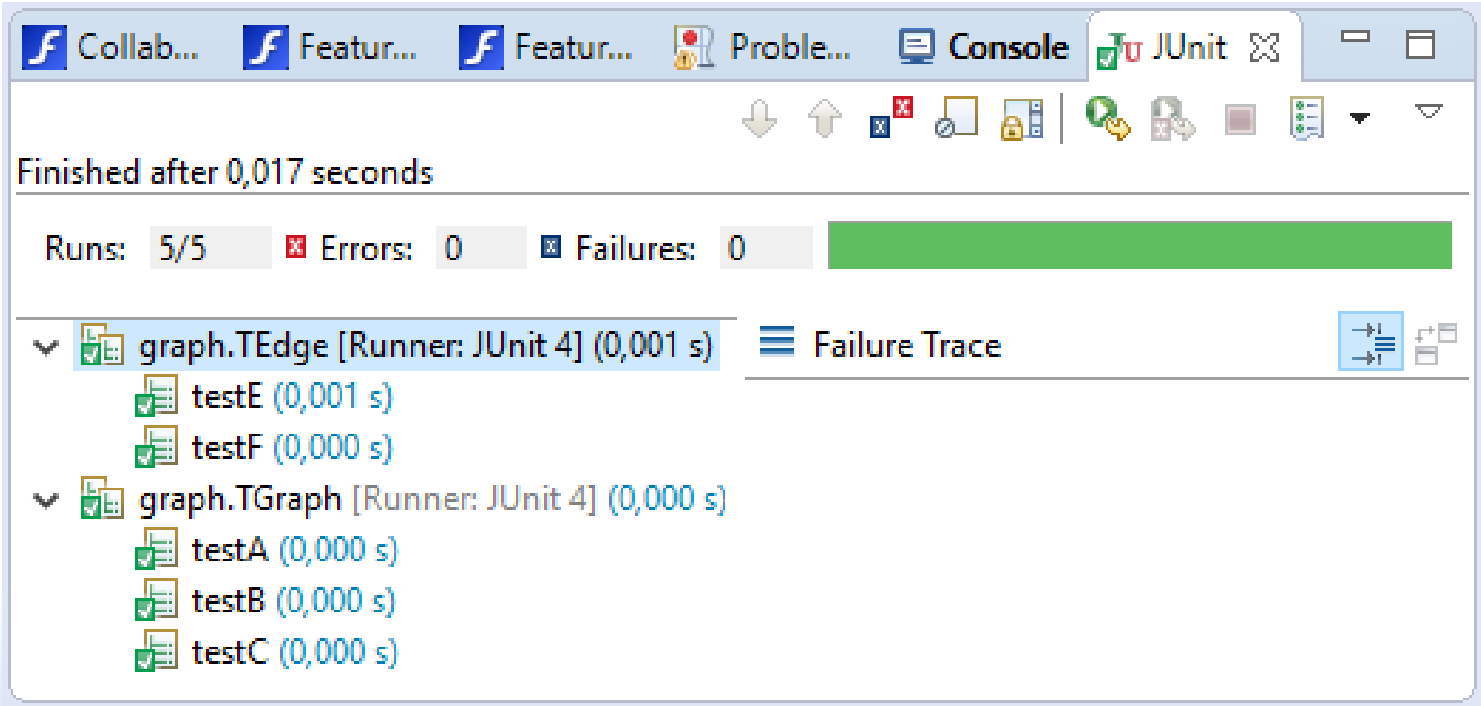
Invisible compiler errors

- Compiler errors may exist only in some configurations
- Problematic if found late in development process
- Large effort to create and generate many configurations



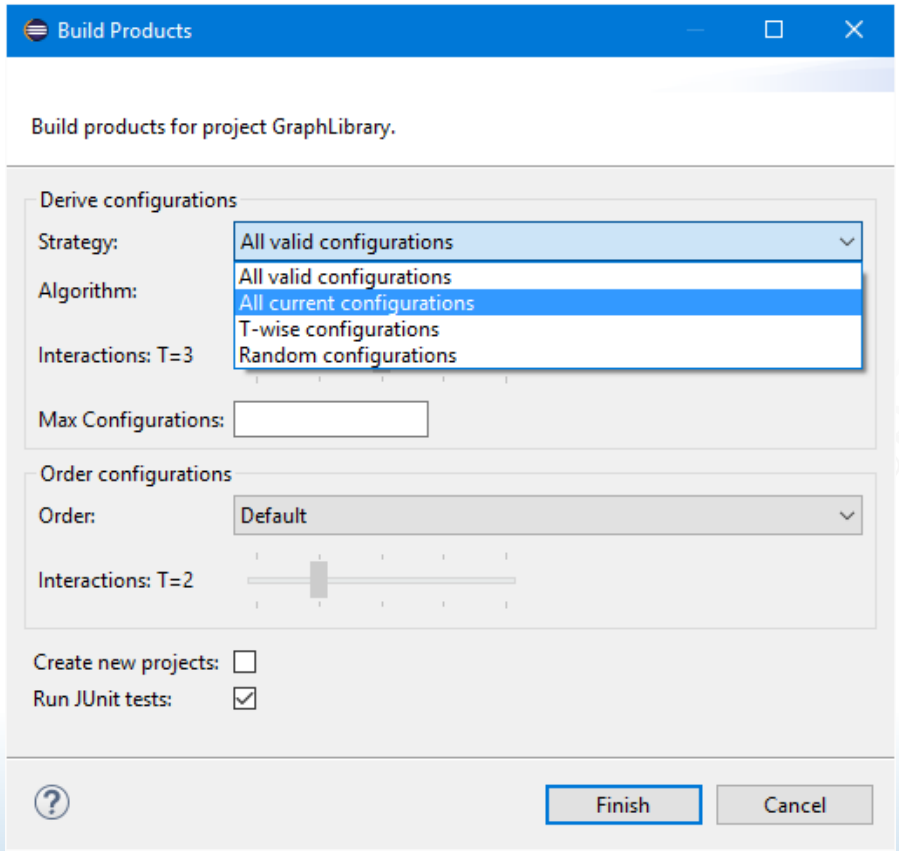
Hidden test failures

- Each unit test may or may not fail for some configurations
- It is not enough if all tests pass in one configuration
- Running unit tests manually for each valid configuration and interpreting the results is time-consuming



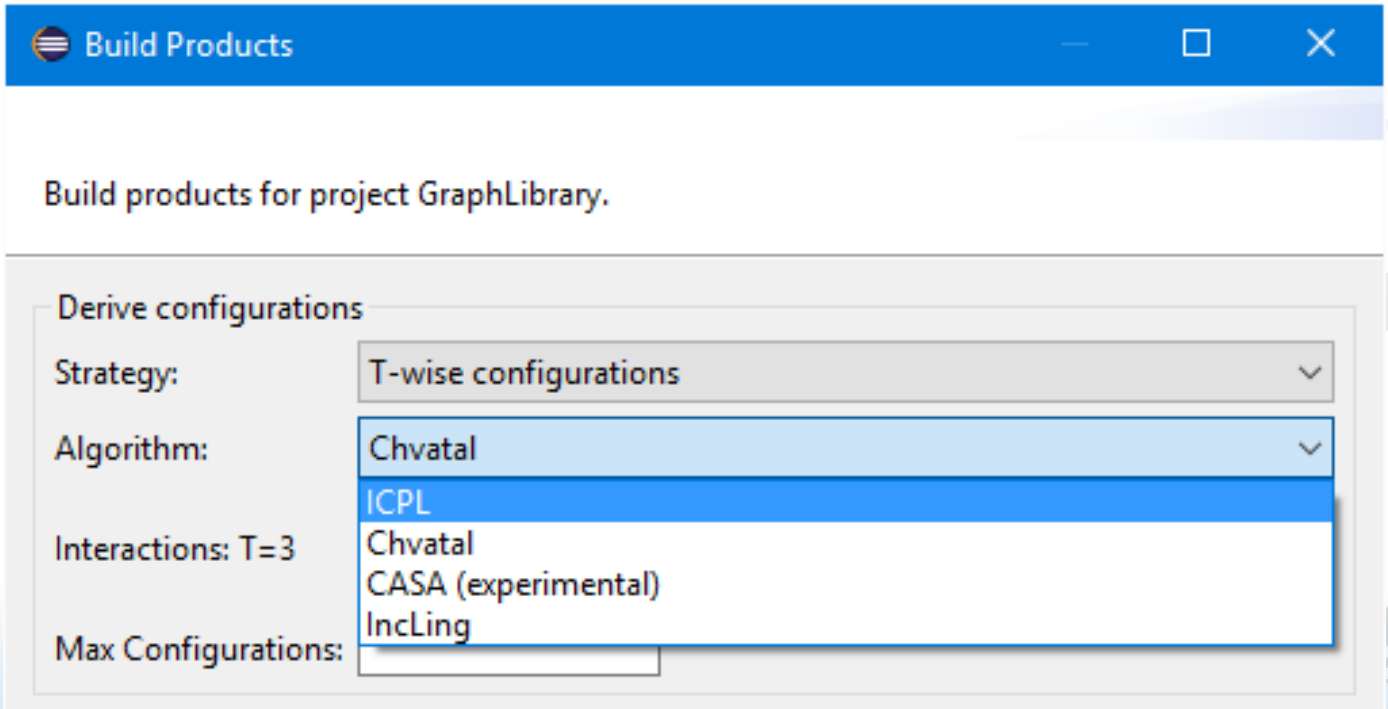
Solution: product generation in batch mode

- FeatureIDE can compile and test configurations on demand
- Compiler errors are mapped back to original source code
- All unit tests are visualized in the JUnit view



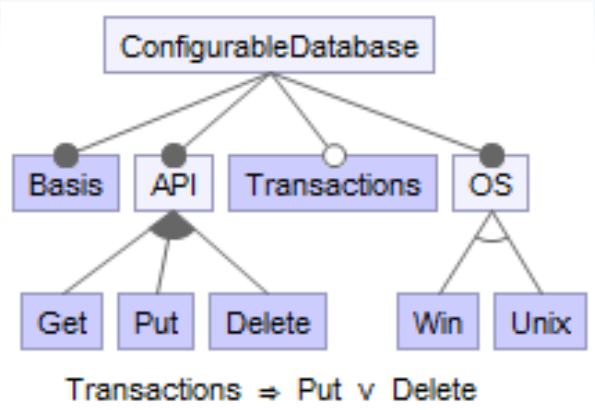
T-wise sampling

- Compiling every single product is not always feasible
- Executing all tests for all configurations does not scale
- Most defects are due to an interaction of few features
- Configuration set covers interactions between up-to T features



Example of pairwise sampling (T=2)

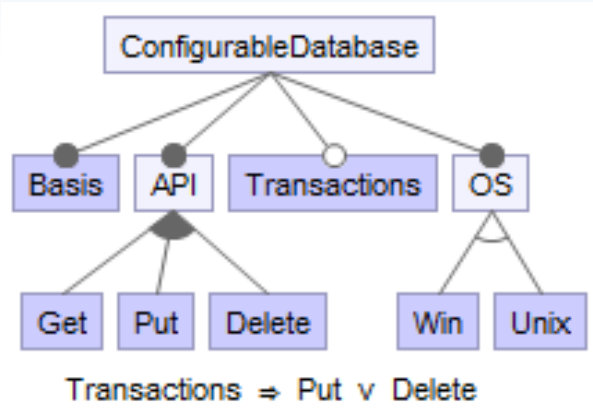
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Example of pairwise sampling (T=2)

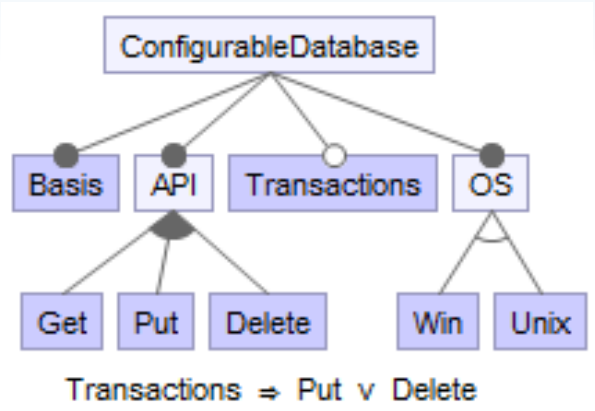
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Example of pairwise sampling (T=2)

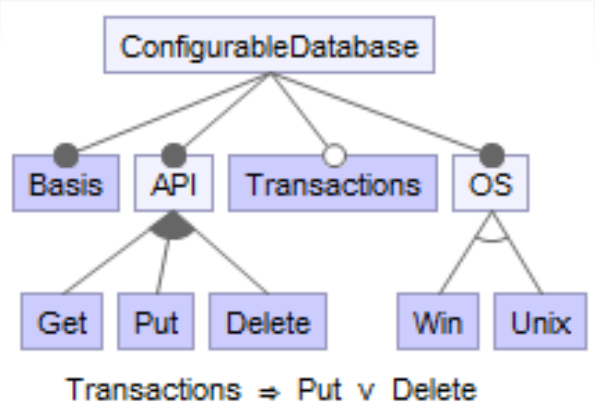
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Example of pairwise sampling (T=2)

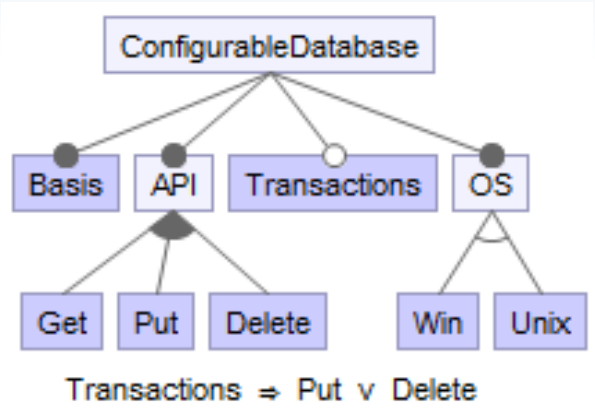
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Example of pairwise sampling (T=2)

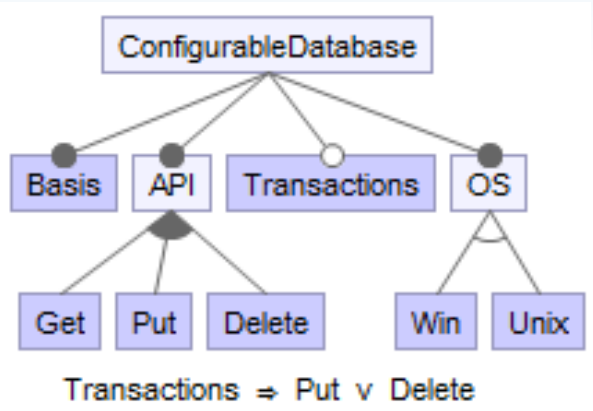
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Example of pairwise sampling (T=2)

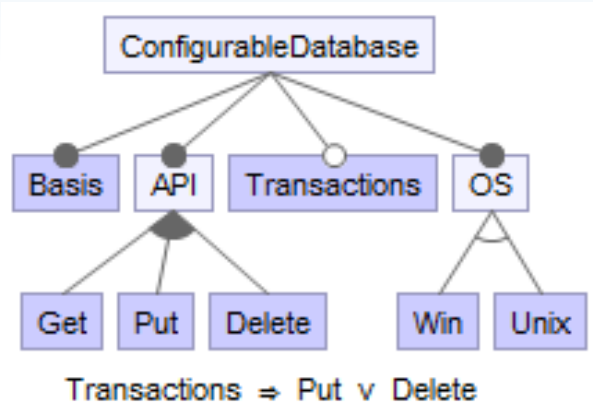
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Example of pairwise sampling (T=2)

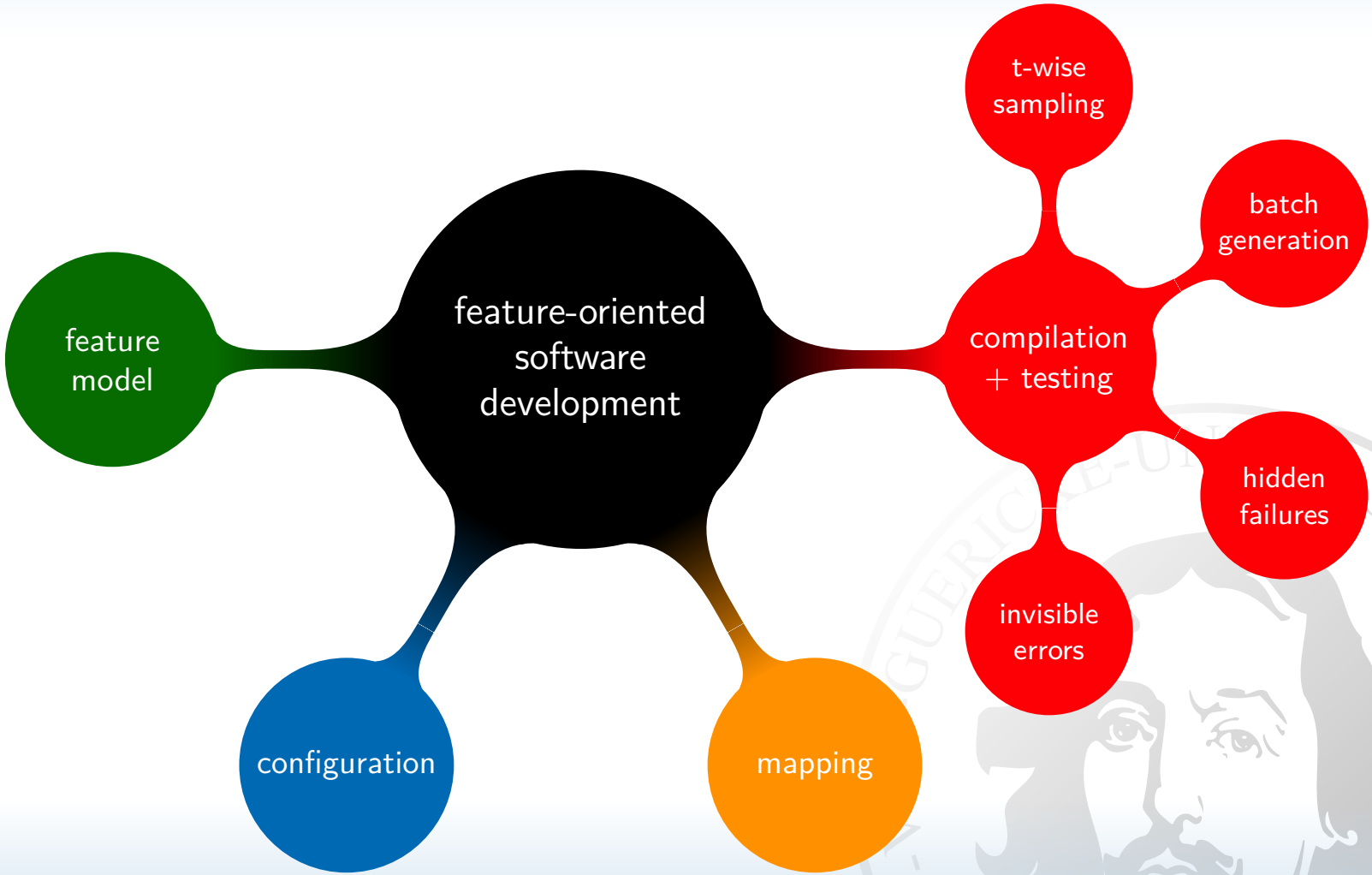
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$	$G \wedge \neg U$	$\neg G \wedge U$	$\neg G \wedge \neg U$
$P \wedge D$	$P \wedge \neg D$	$\neg P \wedge D$	$\neg P \wedge \neg D$
$P \wedge T$	$P \wedge \neg T$	$\neg P \wedge T$	$\neg P \wedge \neg T$
$P \wedge W$	$P \wedge \neg W$	$\neg P \wedge W$	$\neg P \wedge \neg W$
$P \wedge U$	$P \wedge \neg U$	$\neg P \wedge U$	$\neg P \wedge \neg U$
$D \wedge T$	$D \wedge \neg T$	$\neg D \wedge T$	$\neg D \wedge \neg T$
$D \wedge W$	$D \wedge \neg W$	$\neg D \wedge W$	$\neg D \wedge \neg W$
$D \wedge U$	$D \wedge \neg U$	$\neg D \wedge U$	$\neg D \wedge \neg U$
$T \wedge W$	$T \wedge \neg W$	$\neg T \wedge W$	$\neg T \wedge \neg W$
$T \wedge U$	$T \wedge \neg U$	$\neg T \wedge U$	$\neg T \wedge \neg U$
	$W \wedge \neg U$	$\neg W \wedge U$	



$\{B, P, D, T, W\}$
$\{B, G, D, U\}$
$\{B, G, P, T, U\}$
$\{B, G, W\}$
$\{B, P, W\}$
$\{B, D, T, U\}$

Already 6/26 configurations cover all pairwise interactions

Tutorial overview



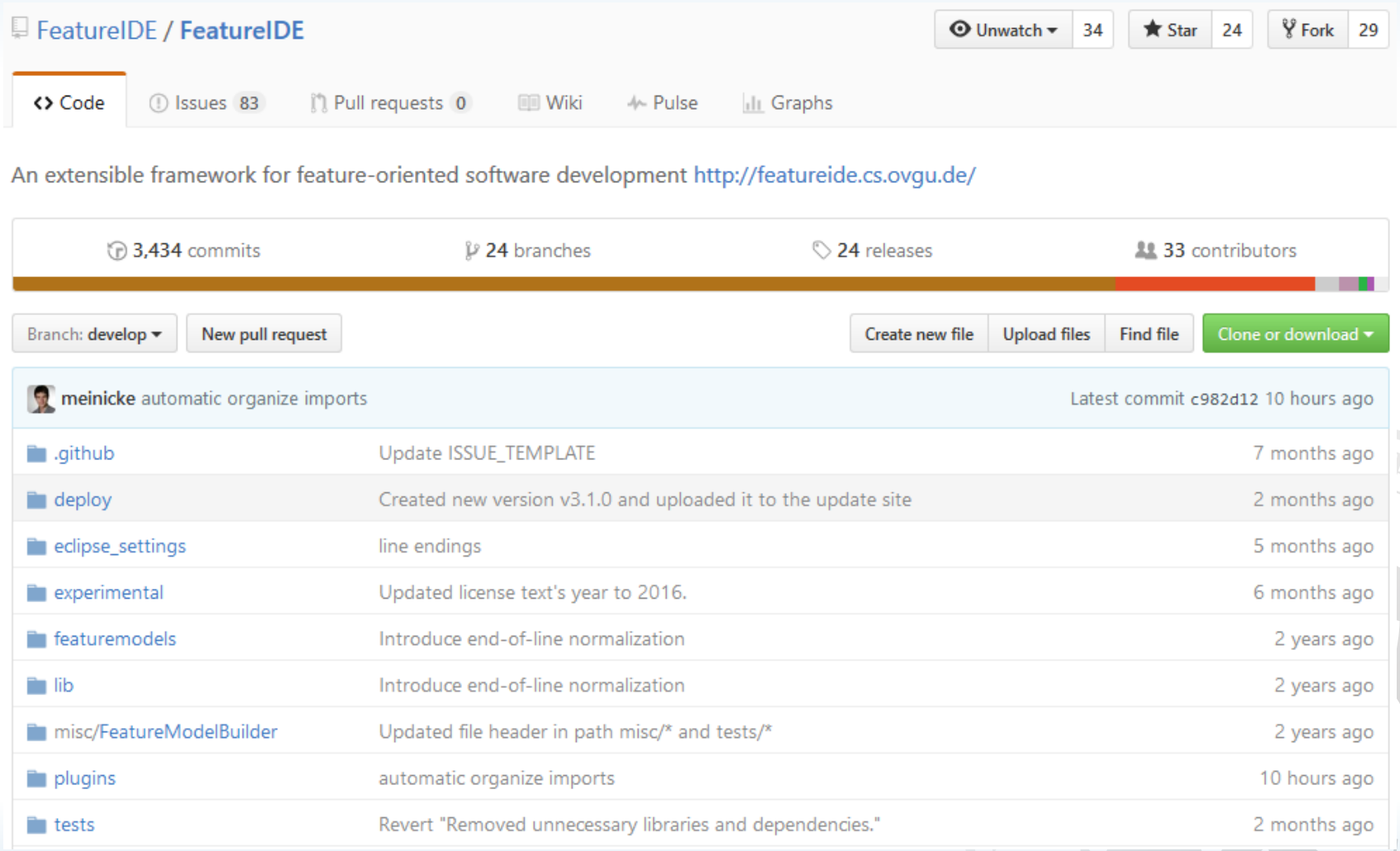
Hands-on V: fix smells with compilation and testing

1. (Continue with previous project or import `Elevator-Antenna-v1.3`)
2. Generate and compile all six configurations by right click on project > FeatureIDE > Product Generator > All current configurations, deselect JUnit tests
3. Fix compiler errors for configuration `Starter` by adding annotation `//#if CallButtons` (try autocompletion) and `//#endif` around `ITickListener.onRequestFinished()` and repeat Step 2 for validation
4. Repeat Step 2 but with selection of JUnit tests to uncover problem with processing order of feature `FIFO`
5. Assign a color to feature `FIFO` to identify and fix the problem in class `Request` and repeat Step 4 for validation
6. Repeat Step 4 but select ICPL and Chvatal with different values of T to identify three lines that should be removed

Part VIII

Closing remarks on FeatureIDE

Available under LGPLv3 on Github



FeatureIDE / FeatureIDE

Unwatch 34 | Star 24 | Fork 29

Code | Issues 83 | Pull requests 0 | Wiki | Pulse | Graphs

An extensible framework for feature-oriented software development http://featureide.cs.ovgu.de/

3,434 commits | 24 branches | 24 releases | 33 contributors

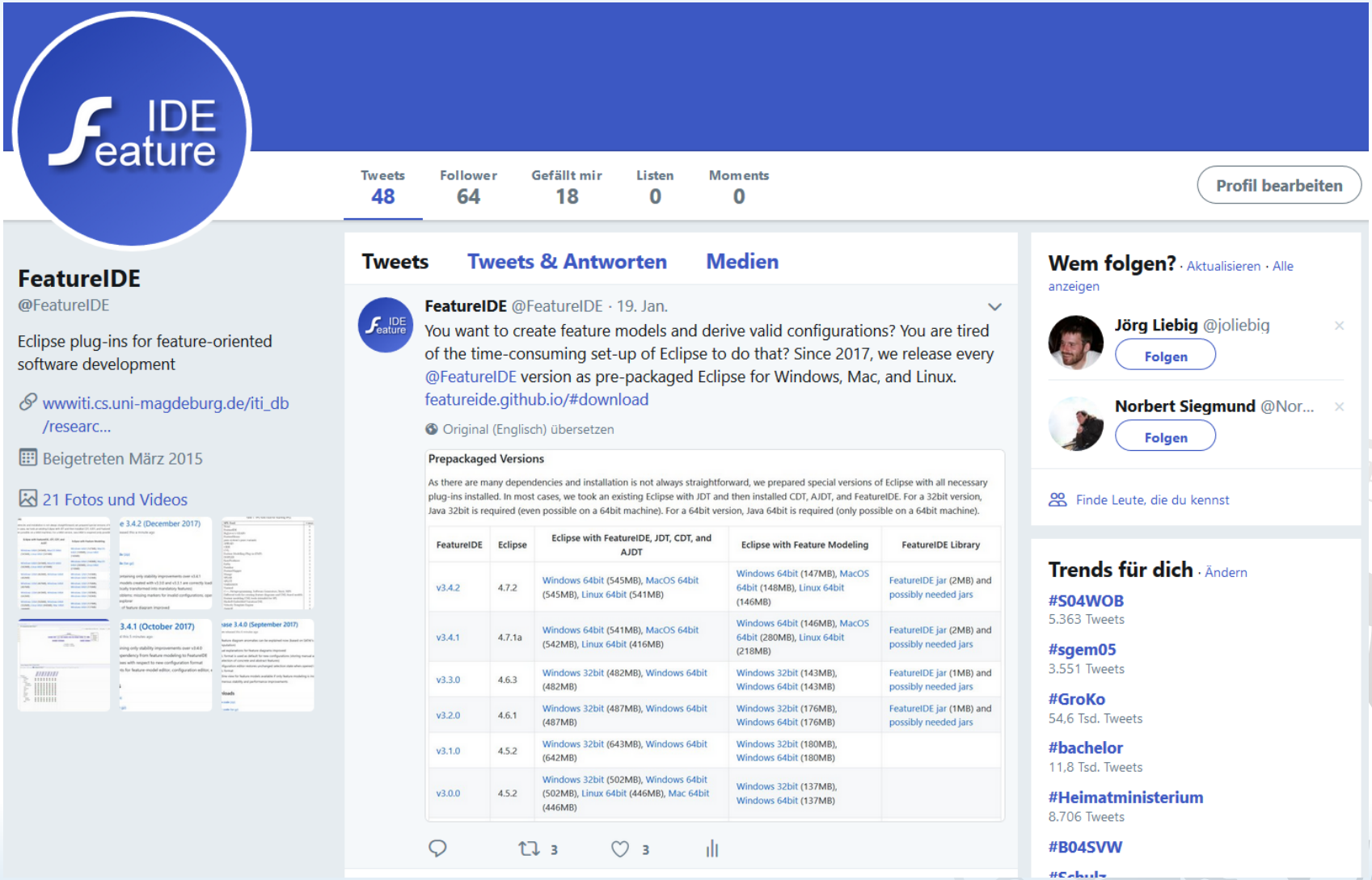
Branch: develop | New pull request | Create new file | Upload files | Find file | Clone or download

meinicke automatic organize imports — Latest commit c982d12 10 hours ago

.github	Update ISSUE_TEMPLATE	7 months ago
deploy	Created new version v3.1.0 and uploaded it to the update site	2 months ago
eclipse_settings	line endings	5 months ago
experimental	Updated license text's year to 2016.	6 months ago
featuremodels	Introduce end-of-line normalization	2 years ago
lib	Introduce end-of-line normalization	2 years ago
misc/FeatureModelBuilder	Updated file header in path misc/* and tests/*	2 years ago
plugins	automatic organize imports	10 hours ago
tests	Revert "Removed unnecessary libraries and dependencies."	2 months ago

More than 100 support requests since 2007

Follow FeatureIDE on Twitter



Tweets 48 | Follower 64 | Gefällt mir 18 | Listen 0 | Moments 0

Profil bearbeiten

FeatureIDE
@FeatureIDE

Eclipse plug-ins for feature-oriented software development

wwwiti.cs.uni-magdeburg.de/iti_db/researc...

Beigetreten März 2015

21 Fotos und Videos

Tweets | Tweets & Antworten | Medien

FeatureIDE @FeatureIDE · 19. Jan.

You want to create feature models and derive valid configurations? You are tired of the time-consuming set-up of Eclipse to do that? Since 2017, we release every @FeatureIDE version as pre-packaged Eclipse for Windows, Mac, and Linux. featureide.github.io/#download

Original (Englisch) übersetzen

Prepackaged Versions

As there are many dependencies and installation is not always straightforward, we prepared special versions of Eclipse with all necessary plug-ins installed. In most cases, we took an existing Eclipse with JDT and then installed CDT, AJDT, and FeatureIDE. For a 32bit version, Java 32bit is required (even possible on a 64bit machine). For a 64bit version, Java 64bit is required (only possible on a 64bit machine).

FeatureIDE	Eclipse	Eclipse with FeatureIDE, JDT, CDT, and AJDT	Eclipse with Feature Modeling	FeatureIDE Library
v3.4.2	4.7.2	Windows 64bit (545MB), MacOS 64bit (545MB), Linux 64bit (541MB)	Windows 64bit (147MB), MacOS 64bit (148MB), Linux 64bit (146MB)	FeatureIDE jar (2MB) and possibly needed jars
v3.4.1	4.7.1a	Windows 64bit (541MB), MacOS 64bit (542MB), Linux 64bit (416MB)	Windows 64bit (146MB), MacOS 64bit (280MB), Linux 64bit (218MB)	FeatureIDE jar (2MB) and possibly needed jars
v3.3.0	4.6.3	Windows 32bit (482MB), Windows 64bit (482MB)	Windows 32bit (143MB), Windows 64bit (143MB)	FeatureIDE jar (1MB) and possibly needed jars
v3.2.0	4.6.1	Windows 32bit (487MB), Windows 64bit (487MB)	Windows 32bit (176MB), Windows 64bit (176MB)	FeatureIDE jar (1MB) and possibly needed jars
v3.1.0	4.5.2	Windows 32bit (643MB), Windows 64bit (642MB)	Windows 32bit (180MB), Windows 64bit (180MB)	
v3.0.0	4.5.2	Windows 32bit (502MB), Windows 64bit (502MB), Linux 64bit (446MB), Mac 64bit (446MB)	Windows 32bit (137MB), Windows 64bit (137MB)	

3 3

Wem folgen? · Aktualisieren · Alle anzeigen

Jörg Liebig @joliebig — Folgen

Norbert Siegmund @Nor... — Folgen

Finde Leute, die du kennst

Trends für dich · Ändern

#S04WOB
5.363 Tweets

#sgem05
3.551 Tweets

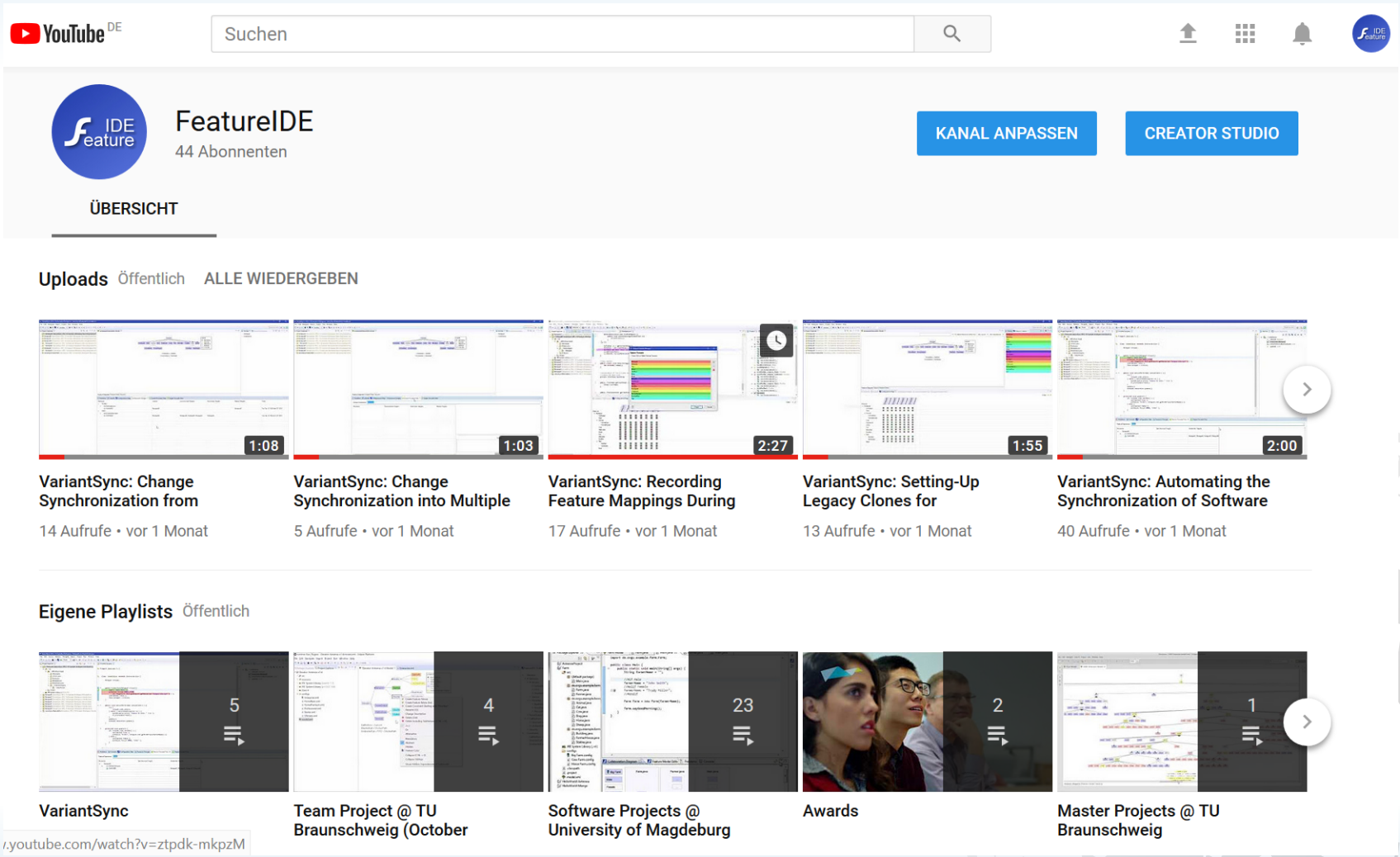
#GroKo
54,6 Tsd. Tweets

#bachelor
11,8 Tsd. Tweets

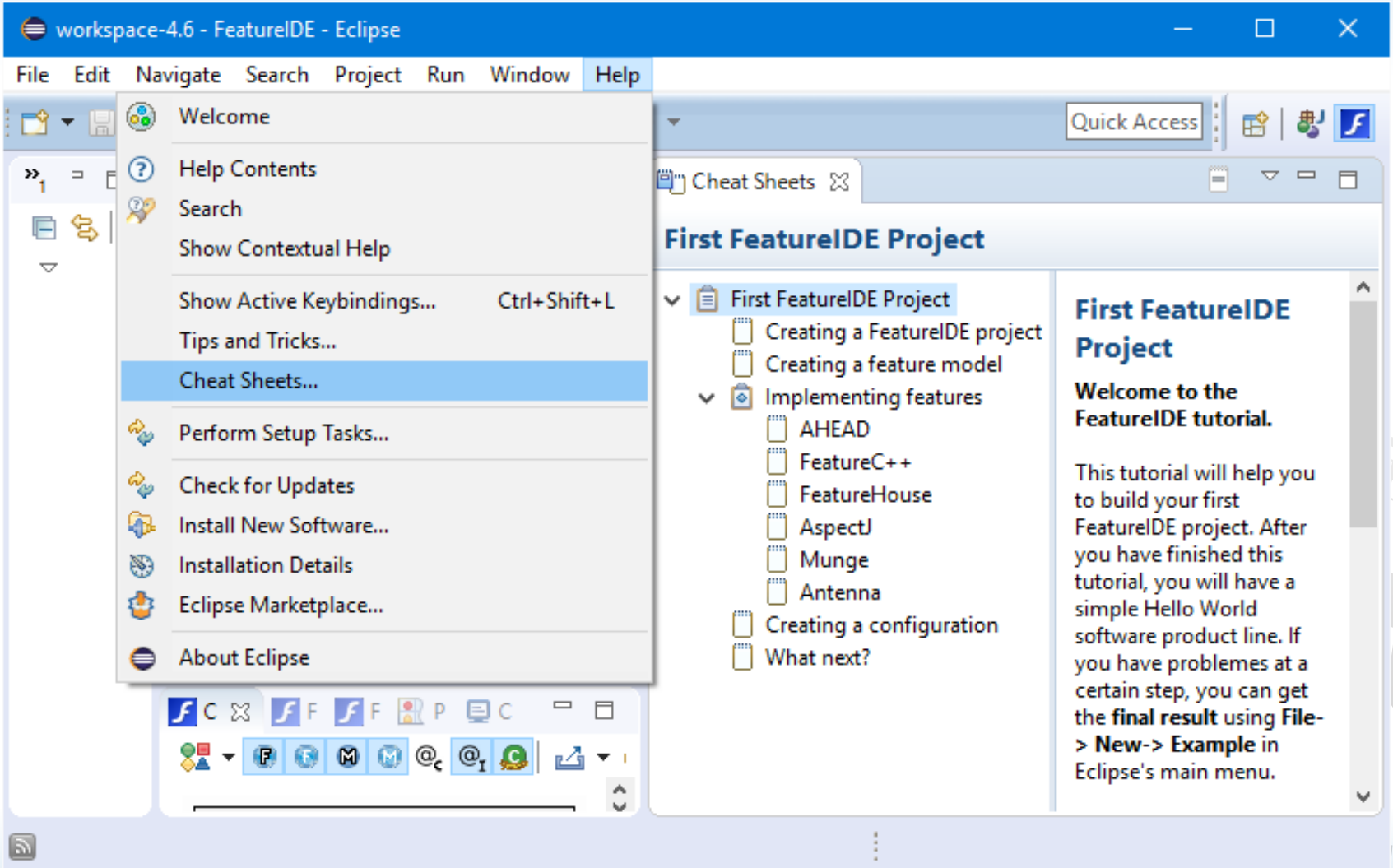
#Heimatministerium
8.706 Tweets

#B04SVW

Enjoy FeatureIDE on Youtube



Learn more with FeatureIDE cheat sheet



Feedback and discussion

1. What did you learn?
2. What did you miss?

Keep it short (2-3 sentences)

FeatureIDE Tutorial