Clean Your Variable Code with FeatureIDE

Thomas Thüm, Sebastian Krieter, Thomas Leich

September 11, 2018 — SPLC'18 in Gothenburg, Schweden

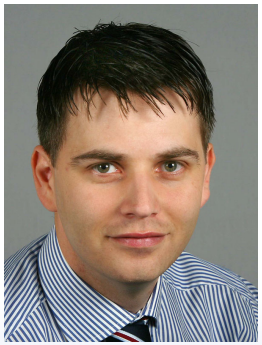
Part I

Introduction round

Who we are

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

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



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

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

}
```

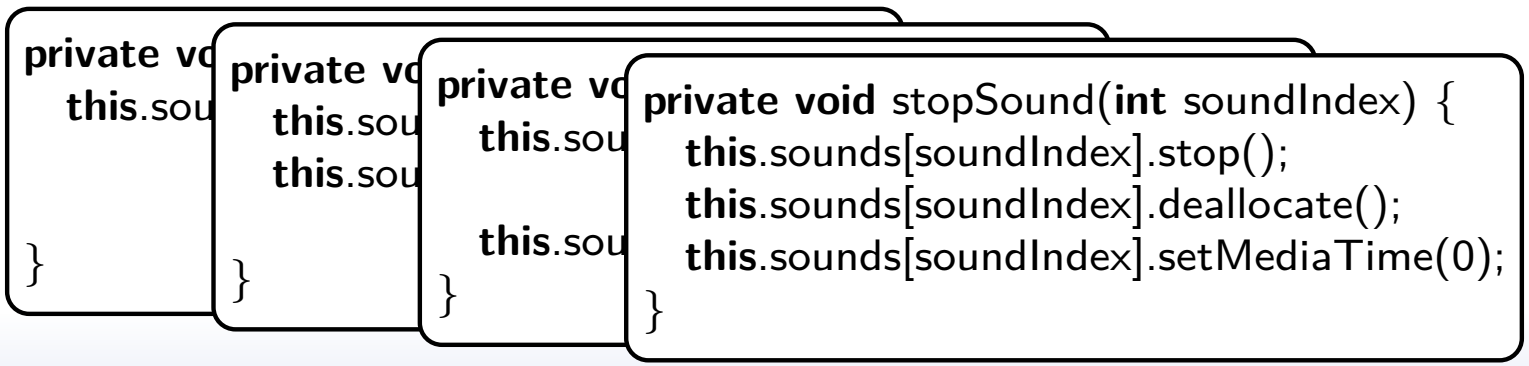
```
private void stopSound(int soundIndex) {
   this.sounds[soundIndex].stop();

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

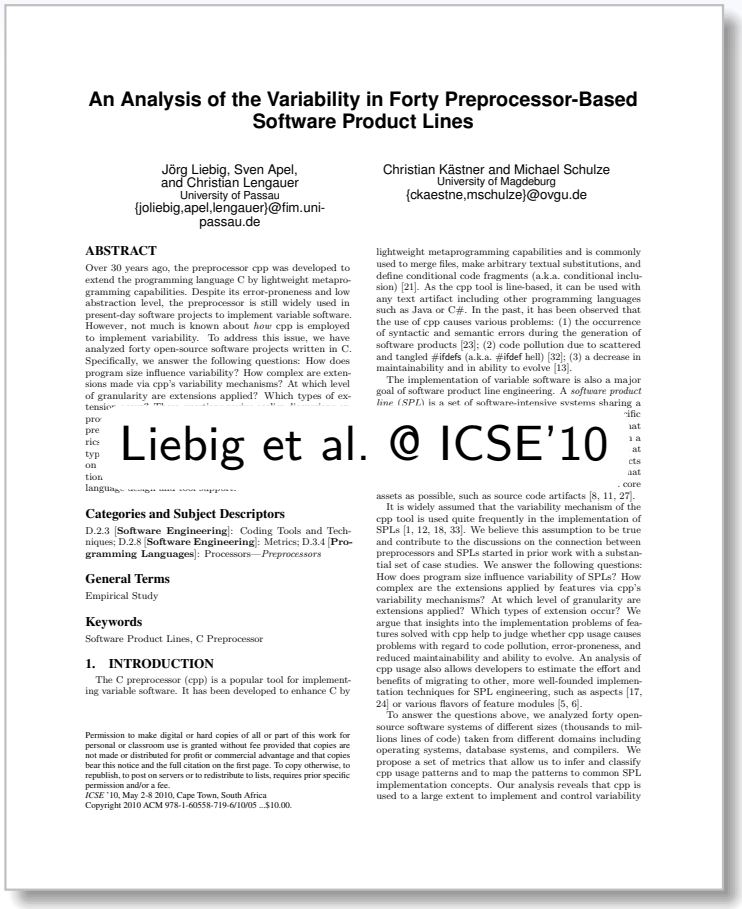
Preprocessing principle is easy

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
}
```



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 extensi[illegible] pro[illegible] pre[illegible] ries typ[illegible] on[illegible] tion[illegible] langua[illegible]

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 [illegible]ific [illegible]at [illegible]a a [illegible]at [illegible]cts [illegible]aat [illegible], core 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

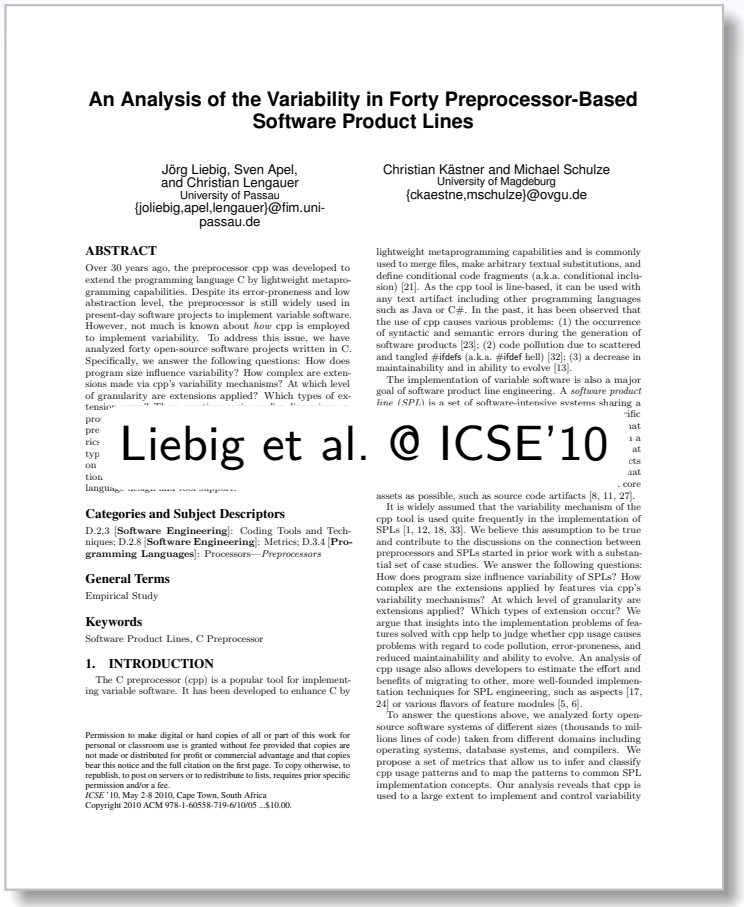
Software Product Lines, C Preprocessor

1. INTRODUCTION

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

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

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].

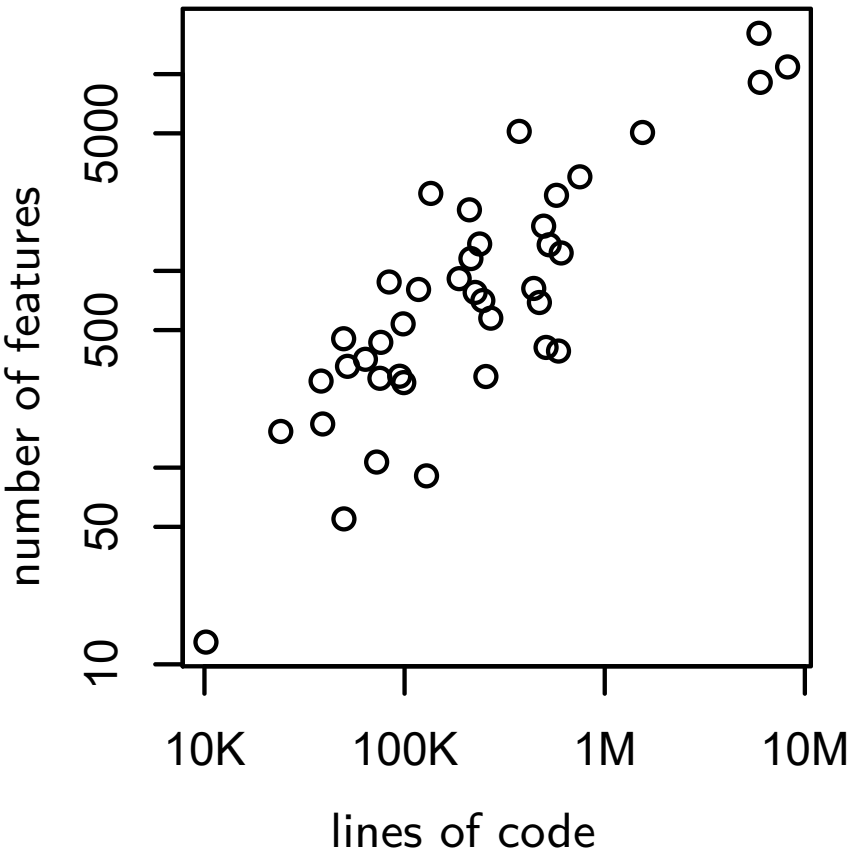
The implementation of variable software is also a major goal of software product line engineering. A *software product line (SPL)* is a set of software-intensive systems sharing a

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

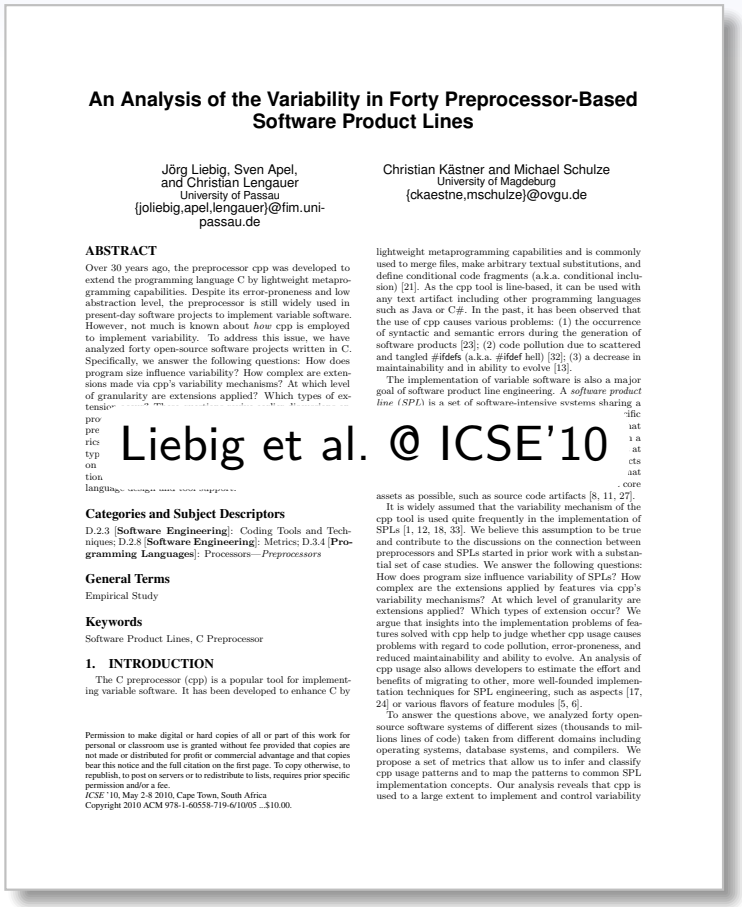
It is widely assumed that the variability mechanism of the cpp tool is used quite frequently in the implementation of SPLs [1, 12, 18, 33]. We believe this assumption to be true and contribute to the discussions on the connection between preprocessors and SPLs started in prior work with a substantial 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ..$10.00.



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 extensio… pro… pre… ries typ… on… tion… langua…

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

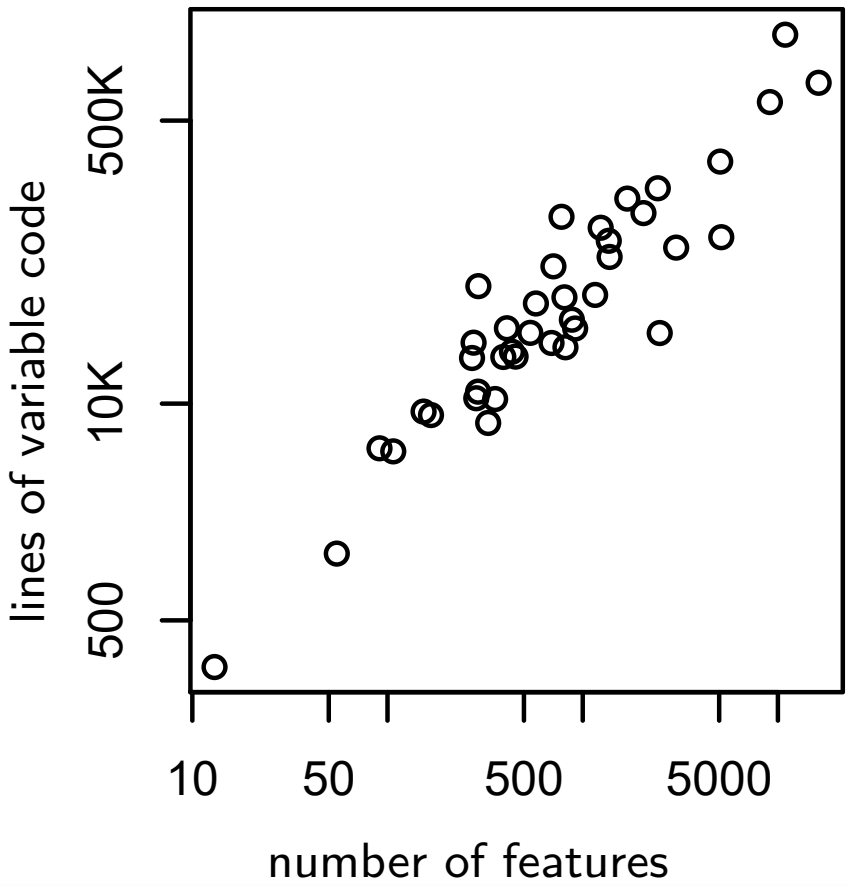
The C preprocessor (cpp) is a popular tool for implementing variable software. It has been developed to enhance C by lightweight metaprogramming capabilities and is commonly used to merge files, make arbitrary textual substitutions, and define conditional code fragments (a.k.a. conditional inclusion) [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 …ific …at …a …at …cts …at …core 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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICSE '10, May 2-8 2010, Cape Town, South Africa
Copyright 2010 ACM 978-1-60558-719-6/10/05 ...$10.00.

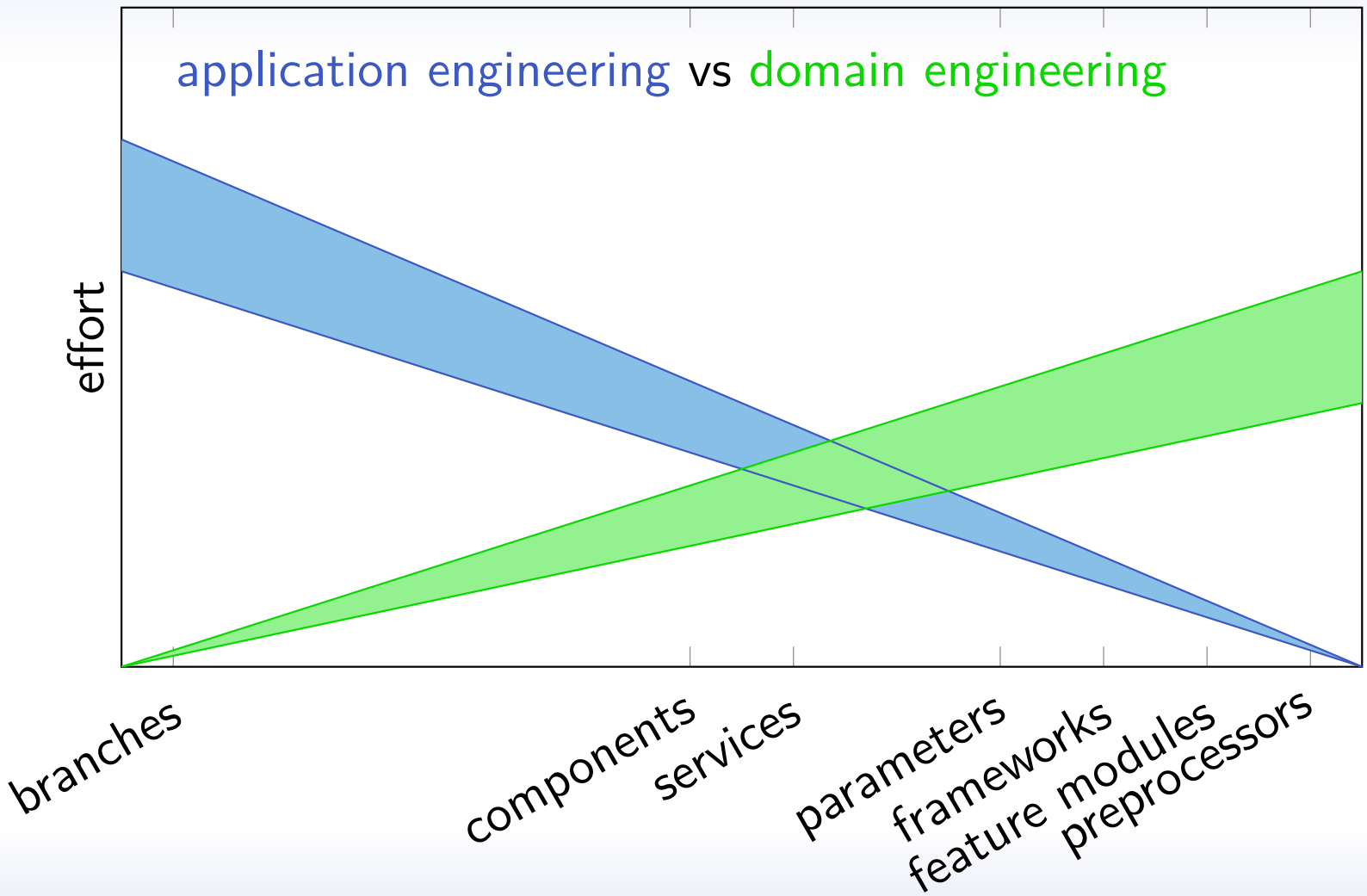


The #ifdef hell

Part III

Feature-oriented software development

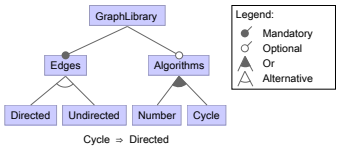
Application engineering vs domain engineering



Heterogeneous software artifacts

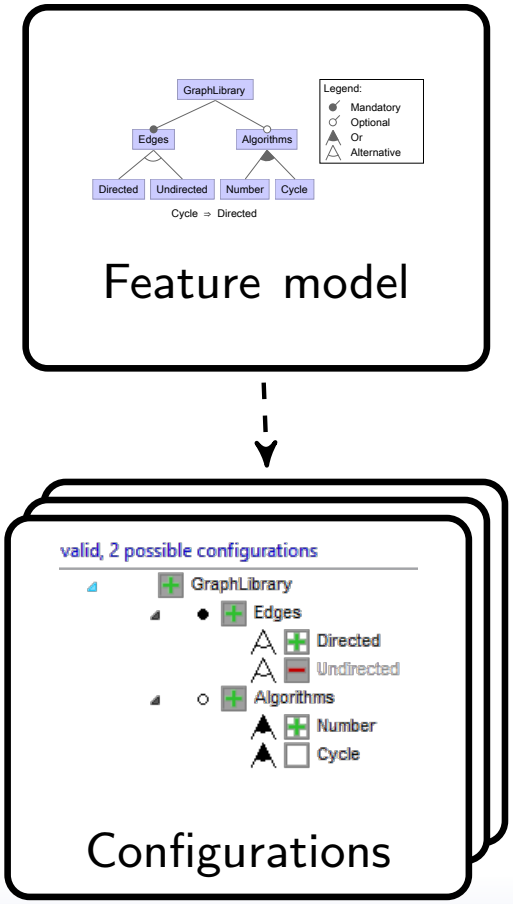
- models (in this tutorial: *feature models*)
- source code (in this tutorial: *Java*)
- binaries (in this tutorial: *Java Bytecode*)
- tests (in this tutorial: *JUnit*)
- specification
- documentation

Feature-oriented software development

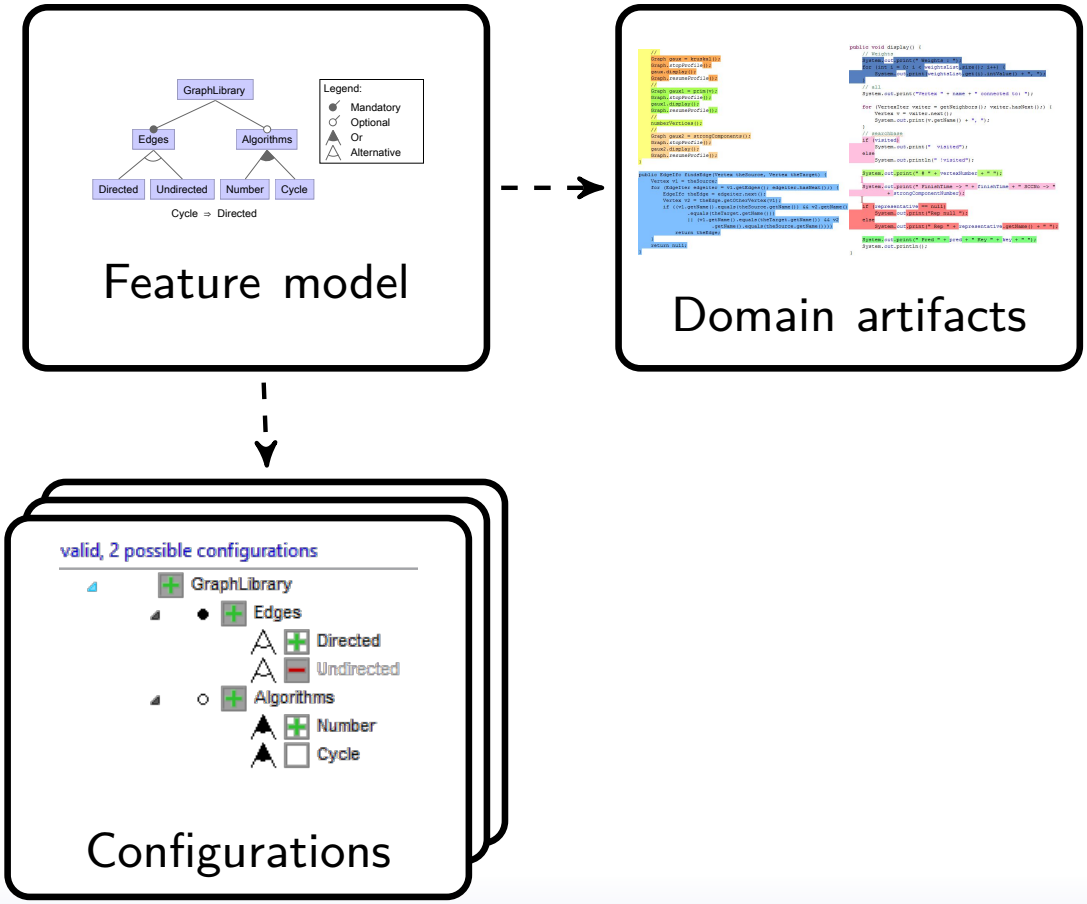


Feature model

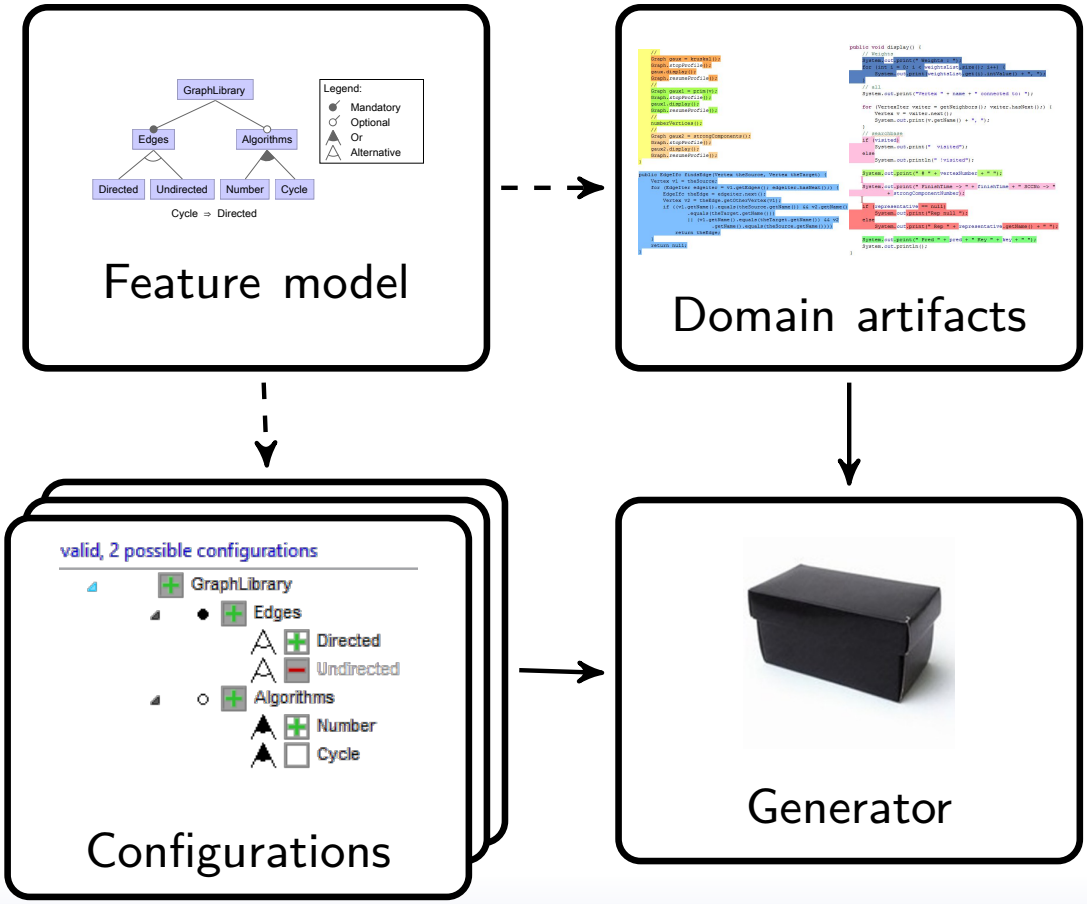
Feature-oriented software development



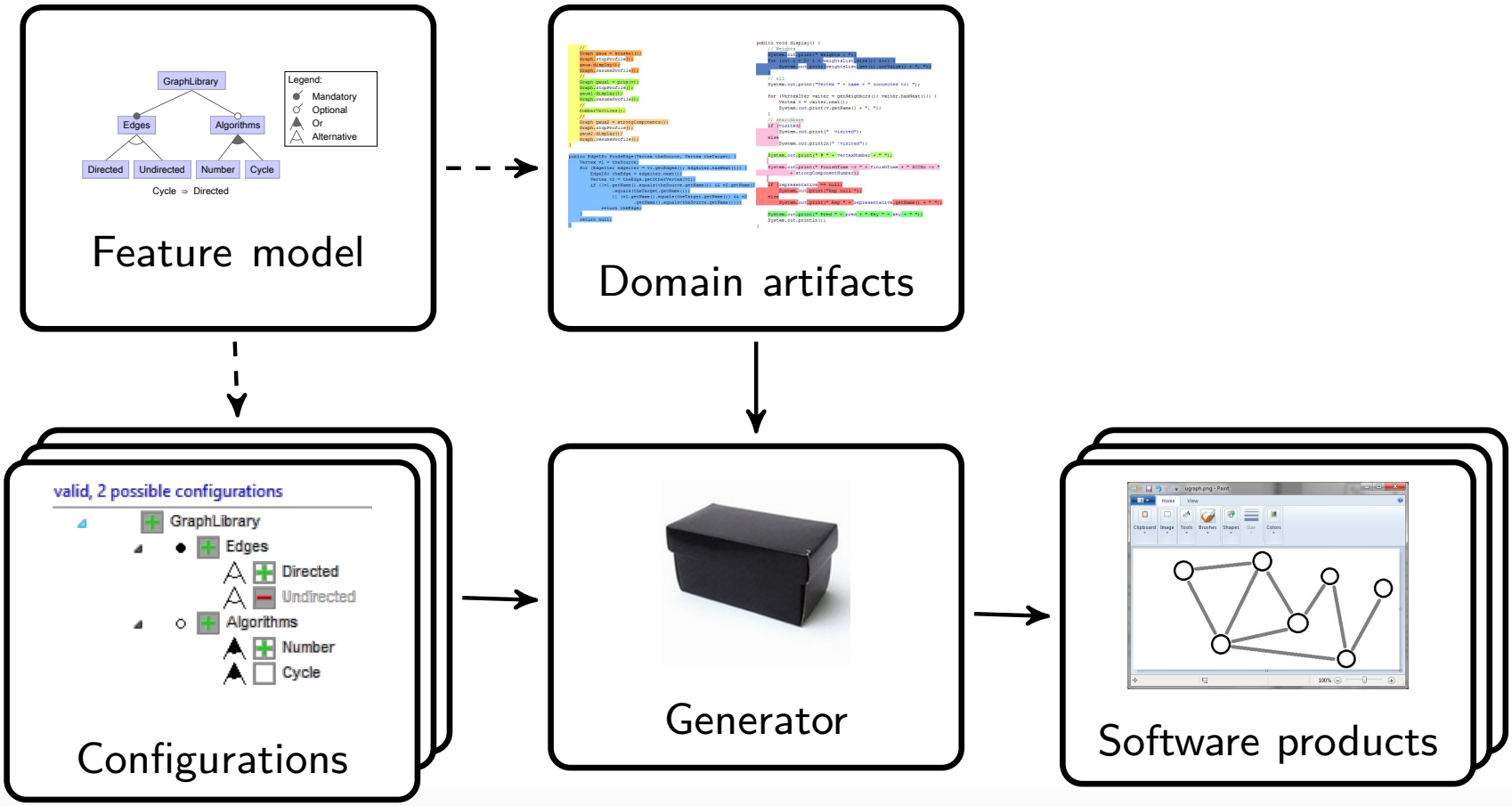
Feature-oriented software development



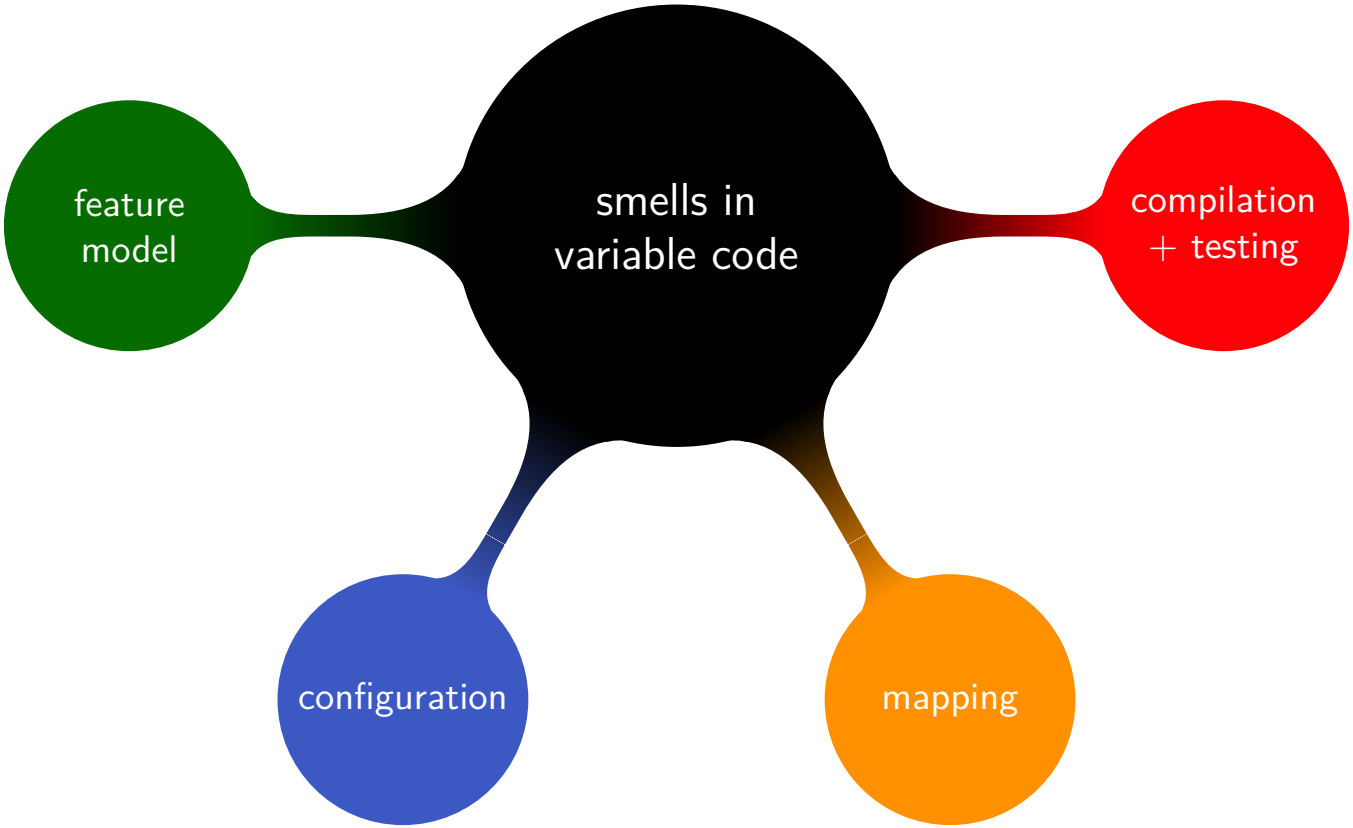
Feature-oriented software development



Feature-oriented software development



Tutorial overview

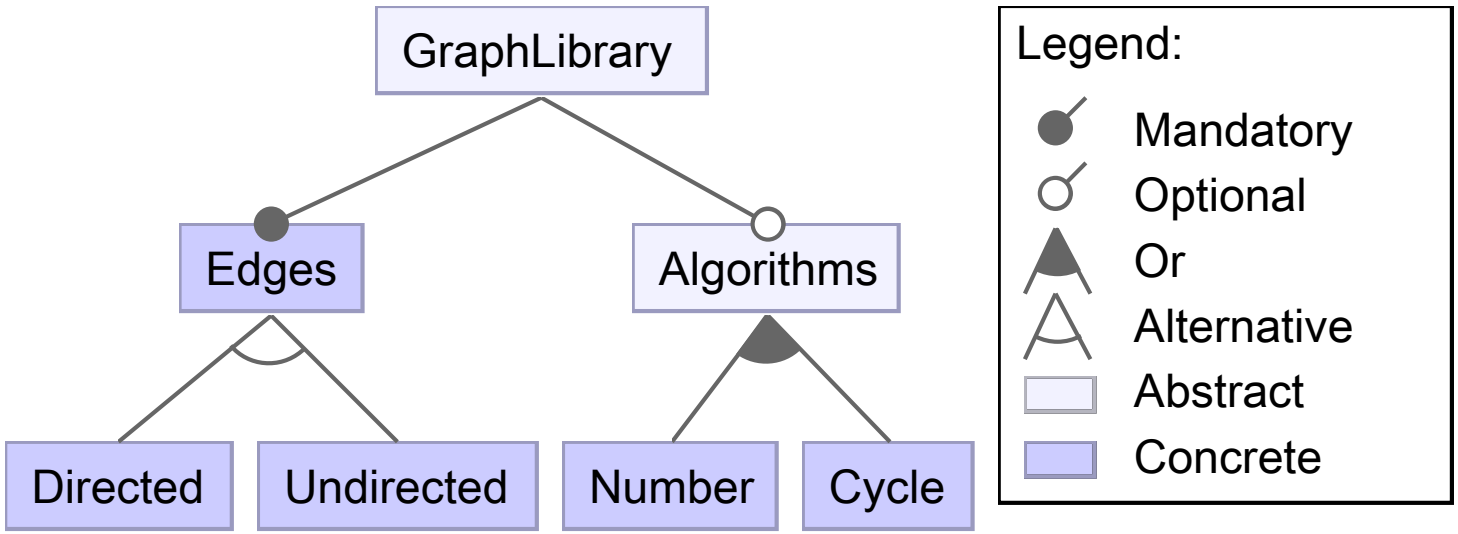


Hands-on I: run different elevator configurations

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

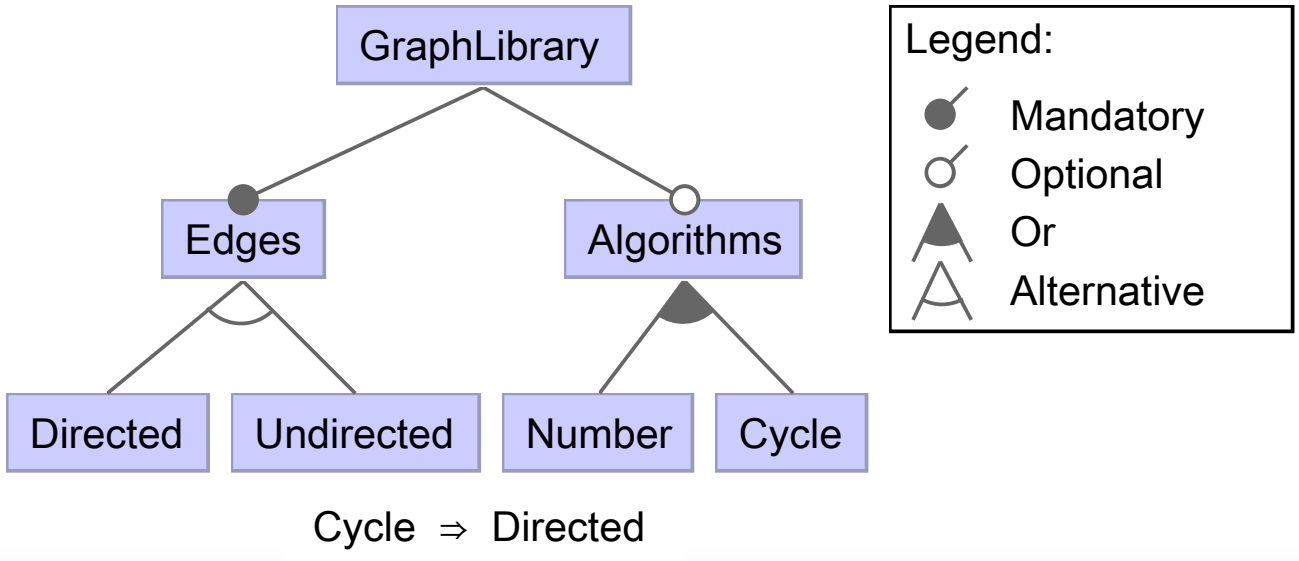
Smells in feature modeling



Cycle $\Rightarrow$ Directed

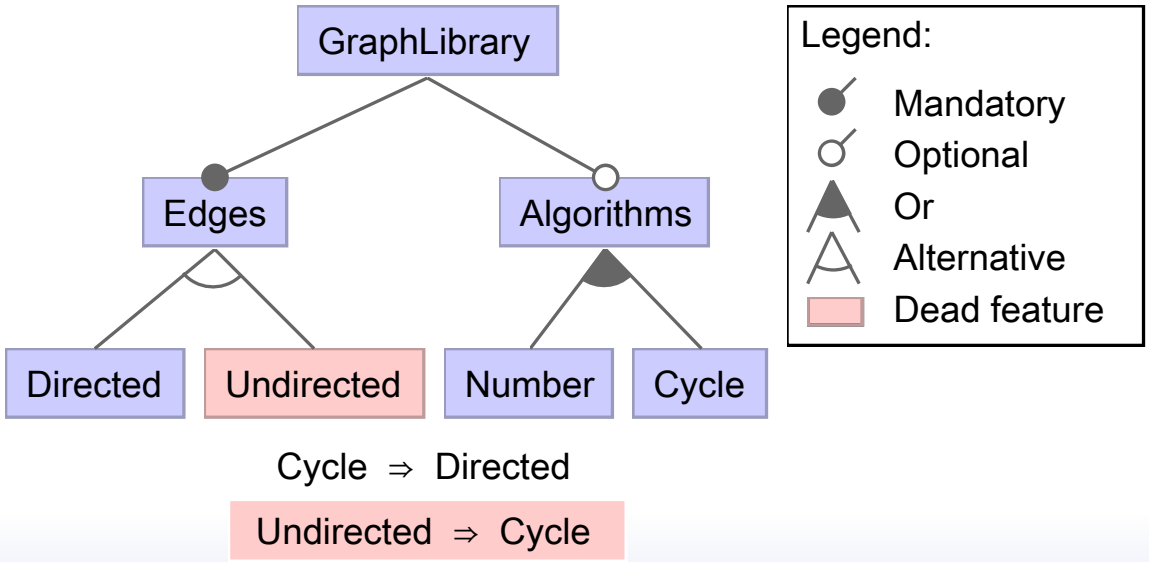
Feature model

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



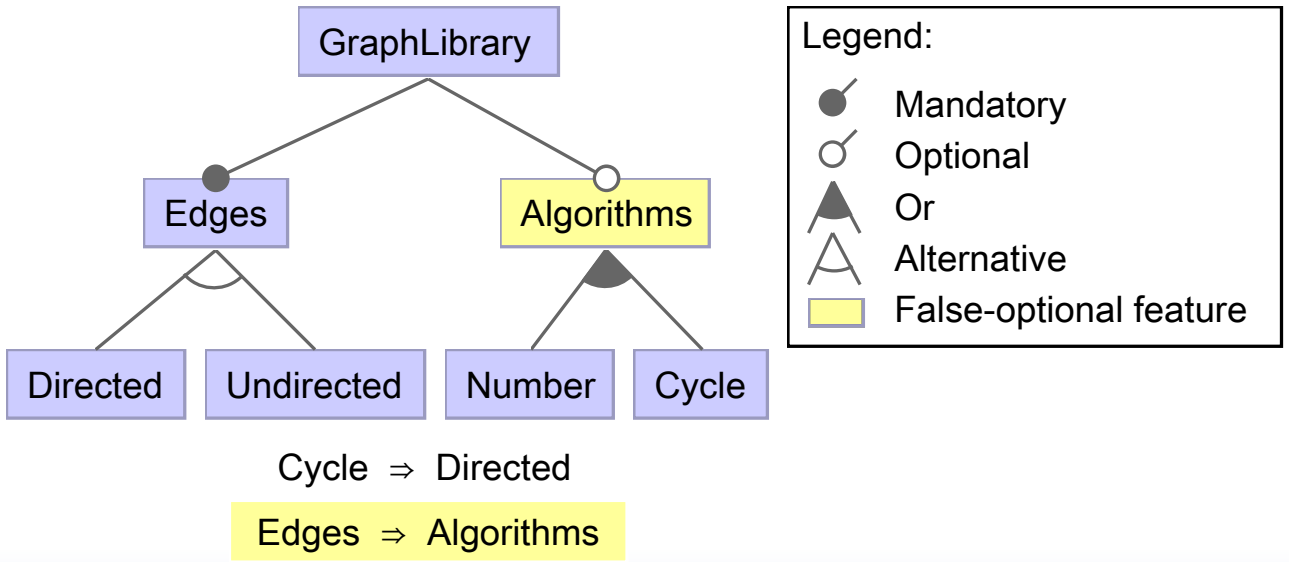
Dead features

- There is no valid configuration containing that feature
- Frustrating during configuration, may result in dead code
- **Fix:** remove feature (refactoring) or remove/edit constraints
- **Hint:** a feature model is void if all features are dead



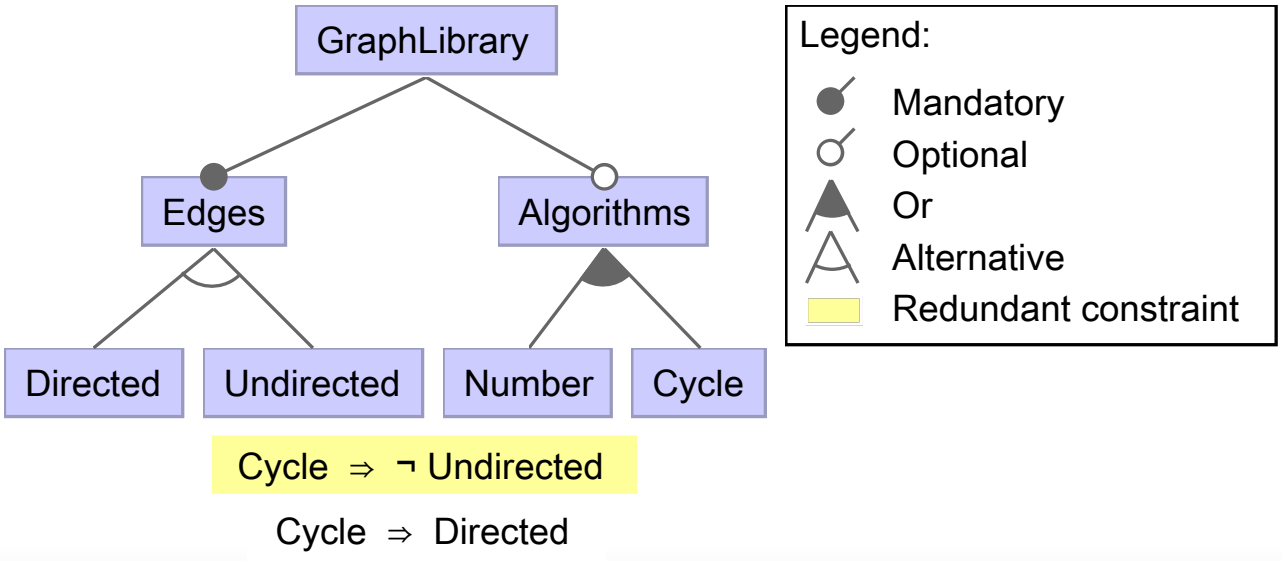
False-optional features

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



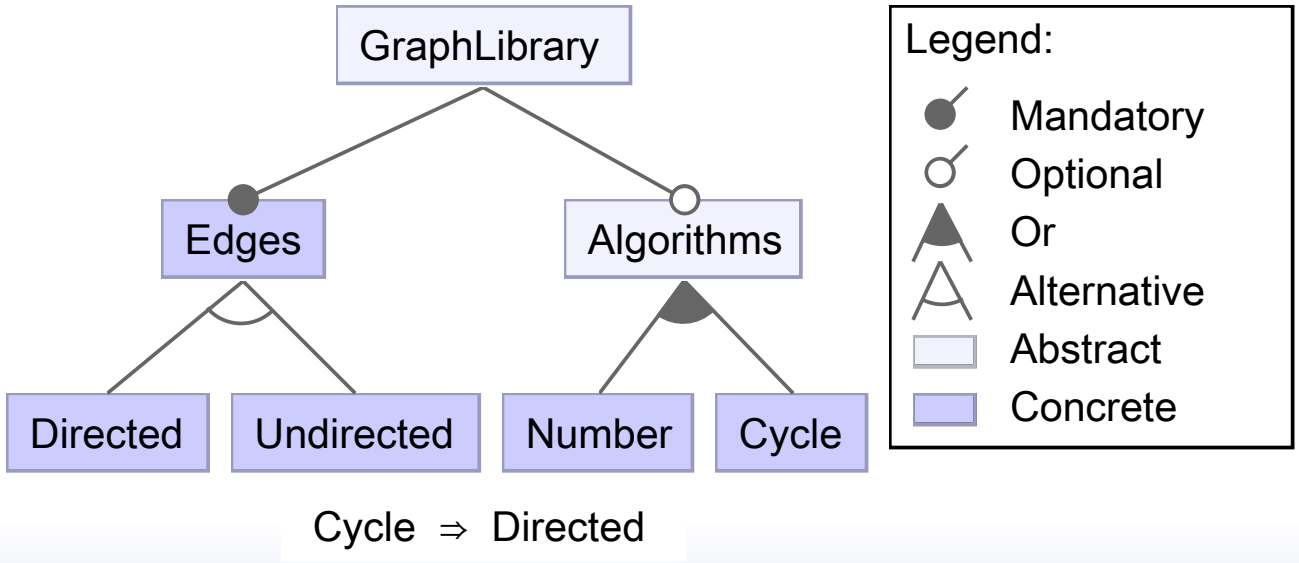
Redundant constraints

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

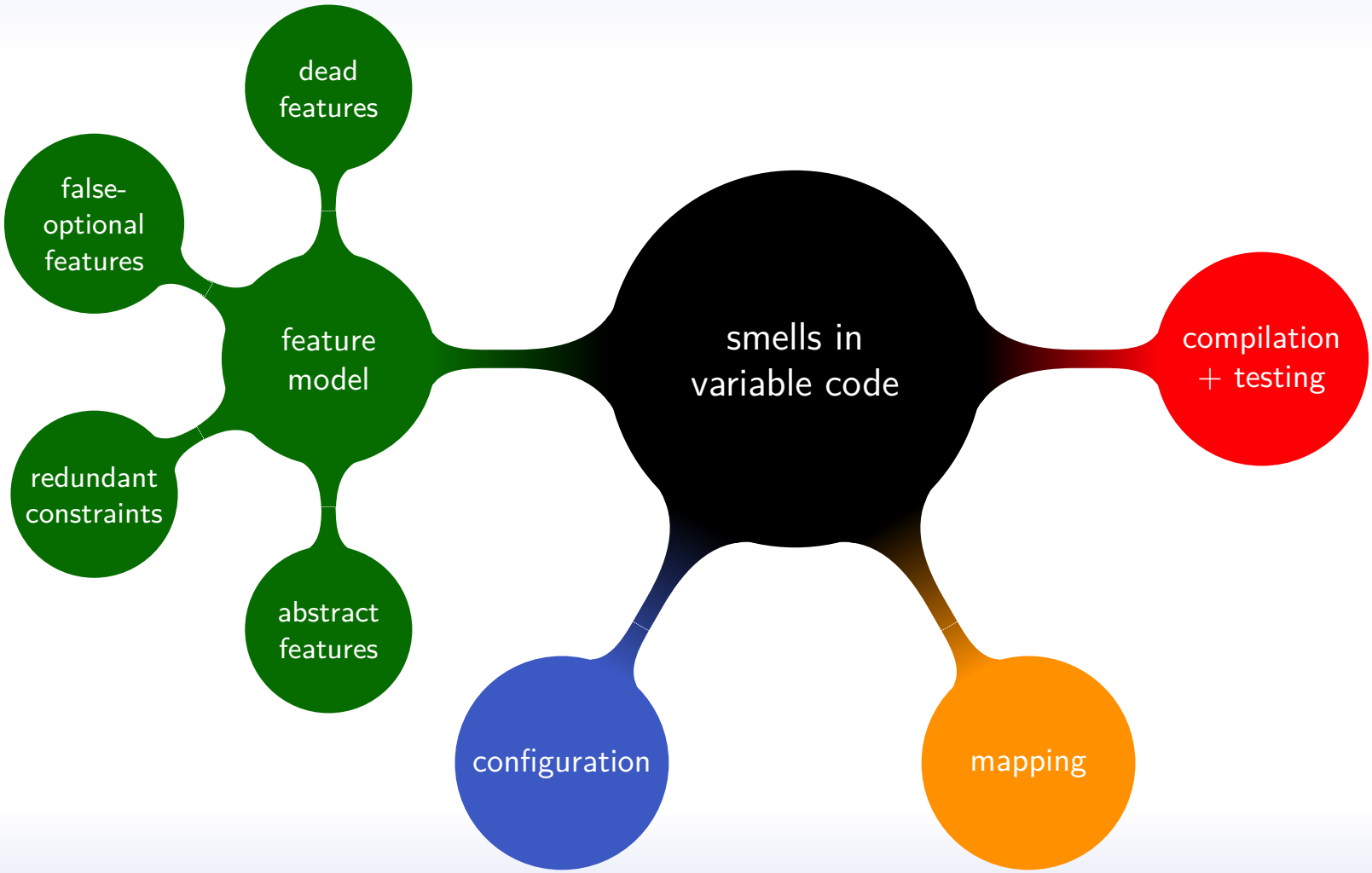


Abstract and concrete features

- 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

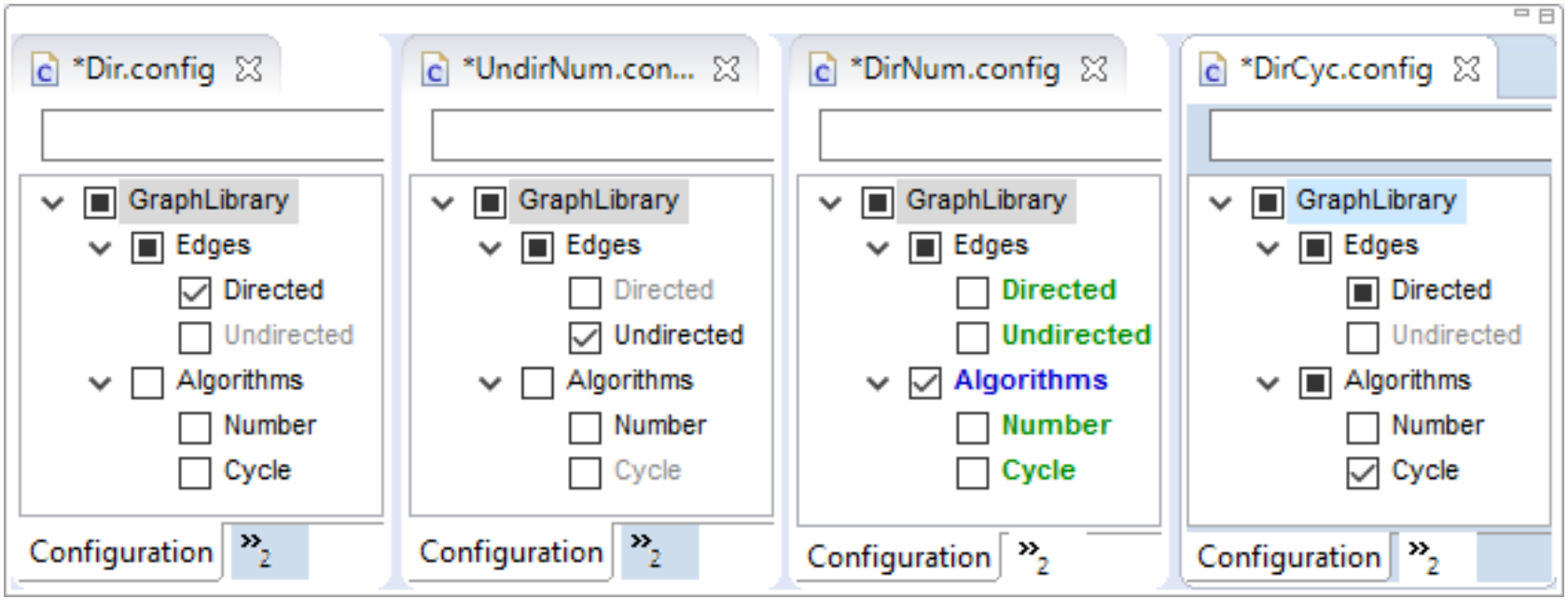


Hands-on II: fix smells in the feature model

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 in the list
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 Step 5) and save the feature model
7. Fix warning in problems view regarding feature model (model.xml) by changing the mentioned features to abstract

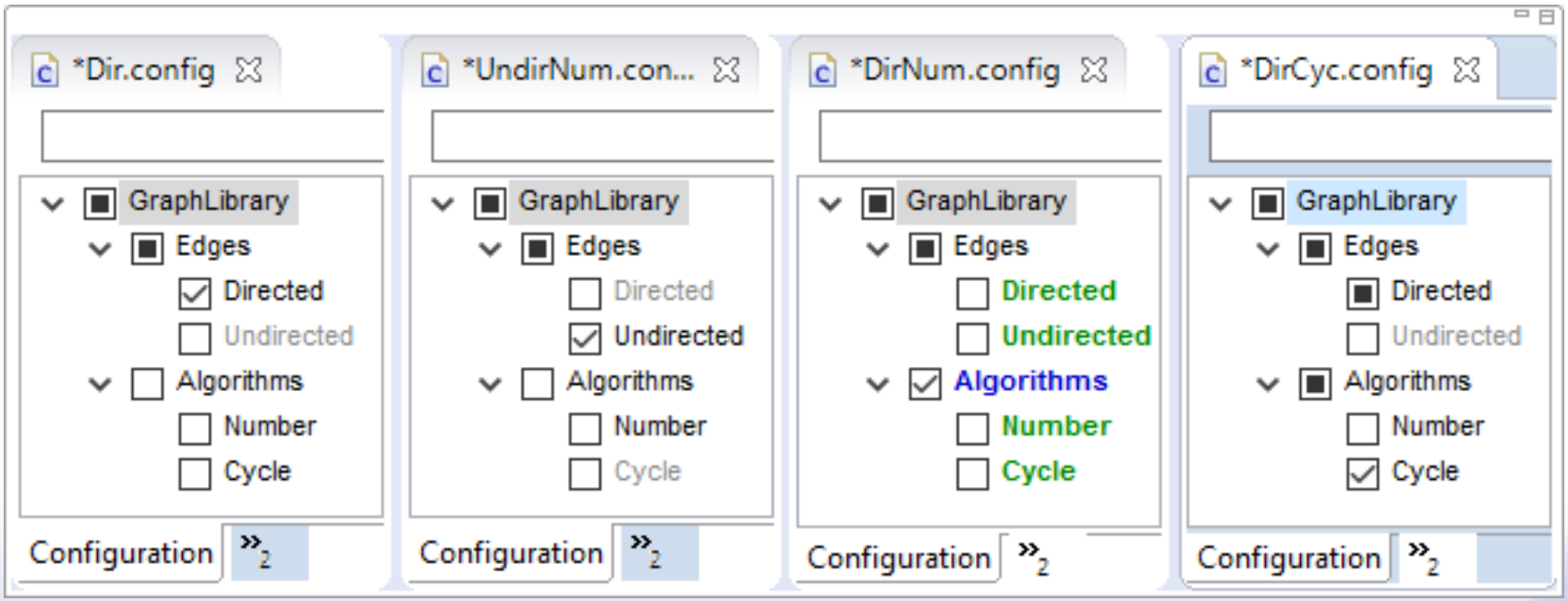
Part V

Smells in configurations



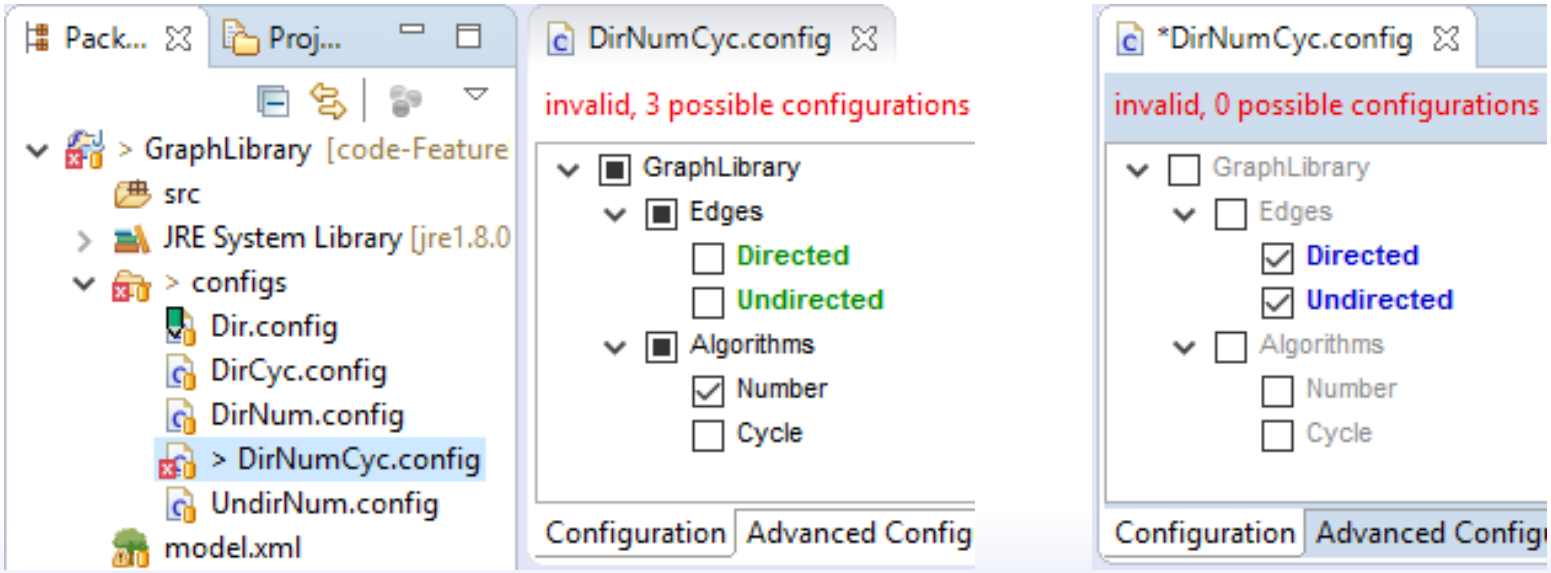
Creating configurations

- 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



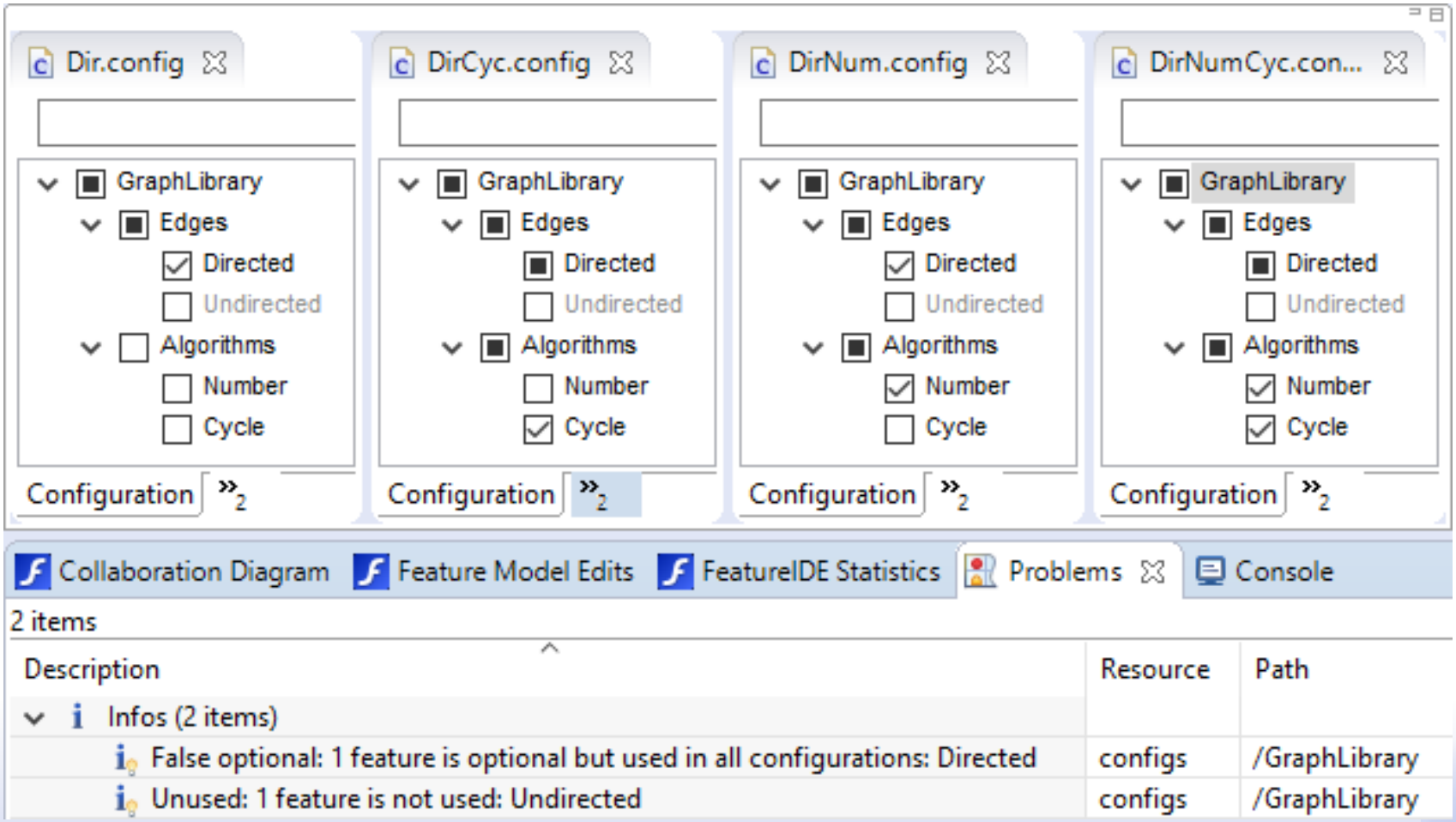
Invalid configurations

- Selection of features contradicts feature model
- Saved invalid or invalid by change in feature model
- Computed whenever feature model or configurations change
- Missing selections (green), conflicting selections (blue)

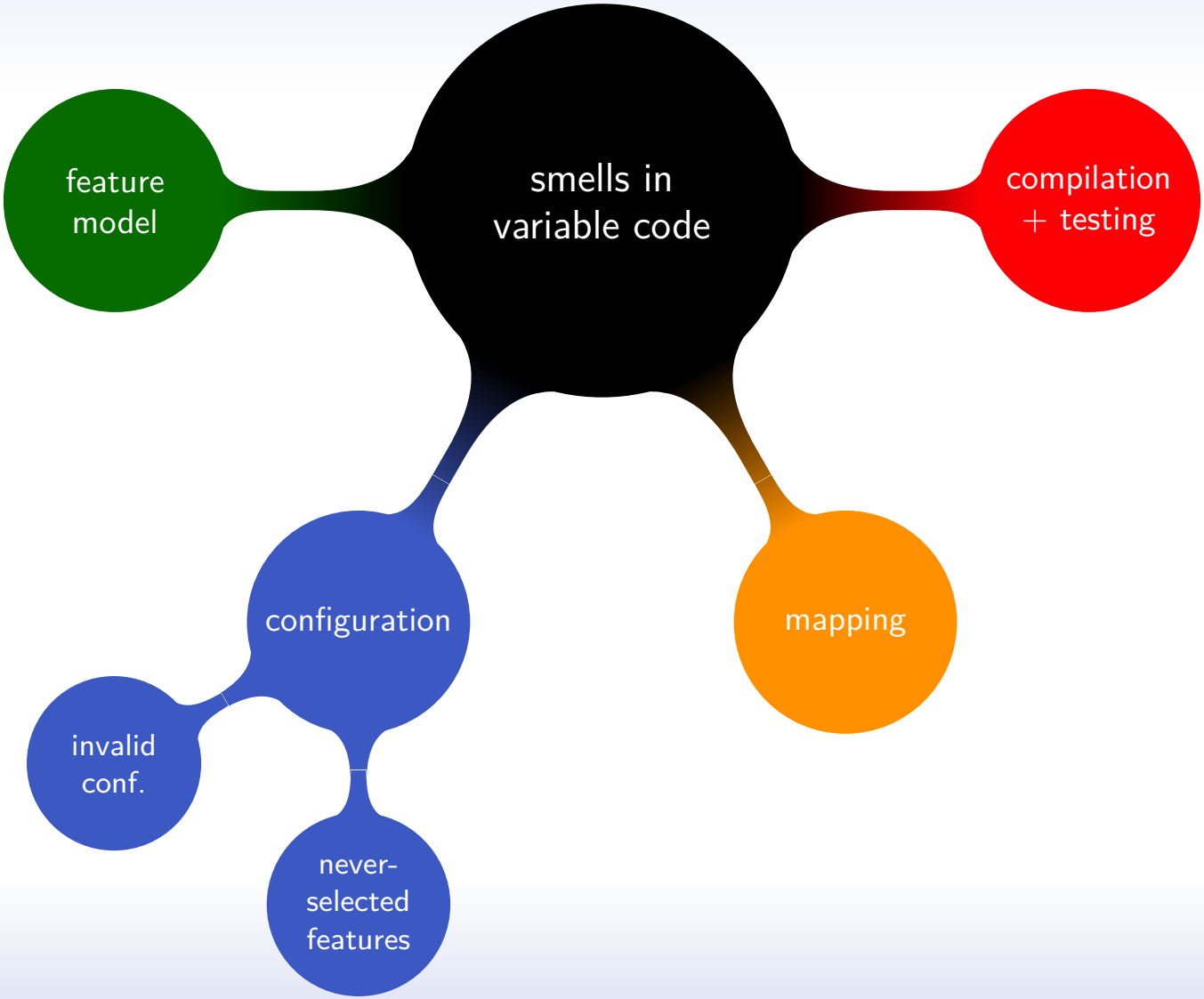


Never-selected features

- Feature is not selected in any configuration (aka. unused)
- Could indicate unnecessary or untested code



Tutorial overview

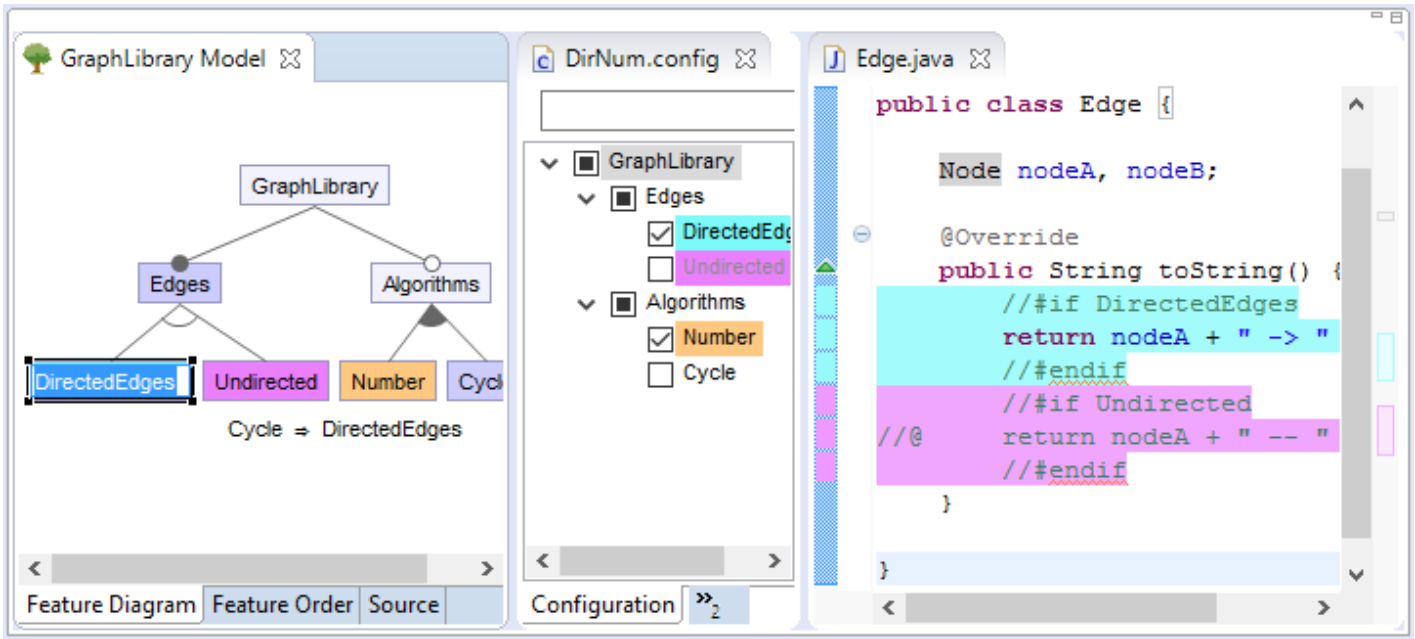


Hands-on III: fix smells of configurations

1. Continue with previous project (or import `Elevator-Antenna-v1.1` and close other projects)
2. Create a new configuration called `Professional` via menu `File` > `New` > `Configuration File`
3. Select the features `FIFO`, `DirectedCall`, and `FloorPermission`; save and close the editor
4. Open feature model; use context menu to change group type below feature `Modes` to an alternative; save the model
5. Open the invalid configuration `Professional`; deselect feature `FIFO`; save the configuration
6. Problem view indicates never-selected features; use `Quick Fix` in context menu to create a configuration with feature `Sabbath`; rename configuration to `Starter` and open it

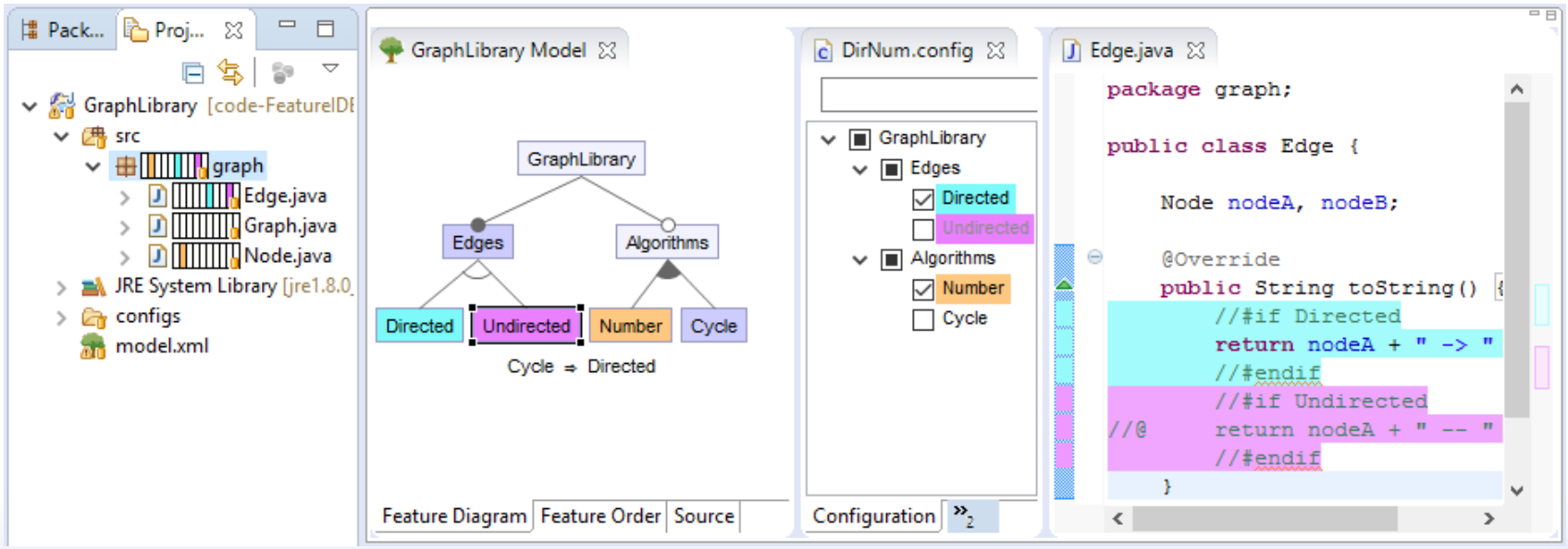
Part VI

Smells in feature-to-code mapping



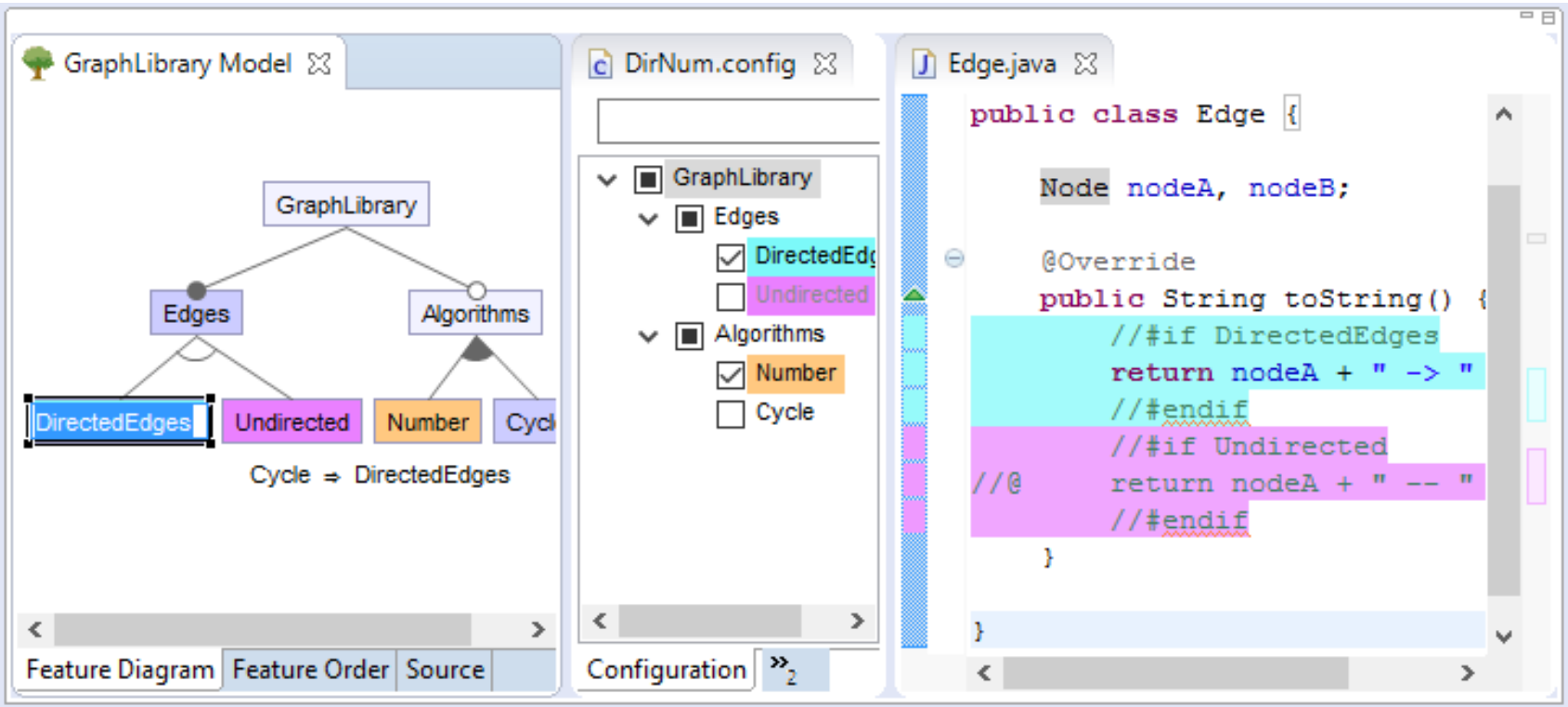
Feature traceability problem

- 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



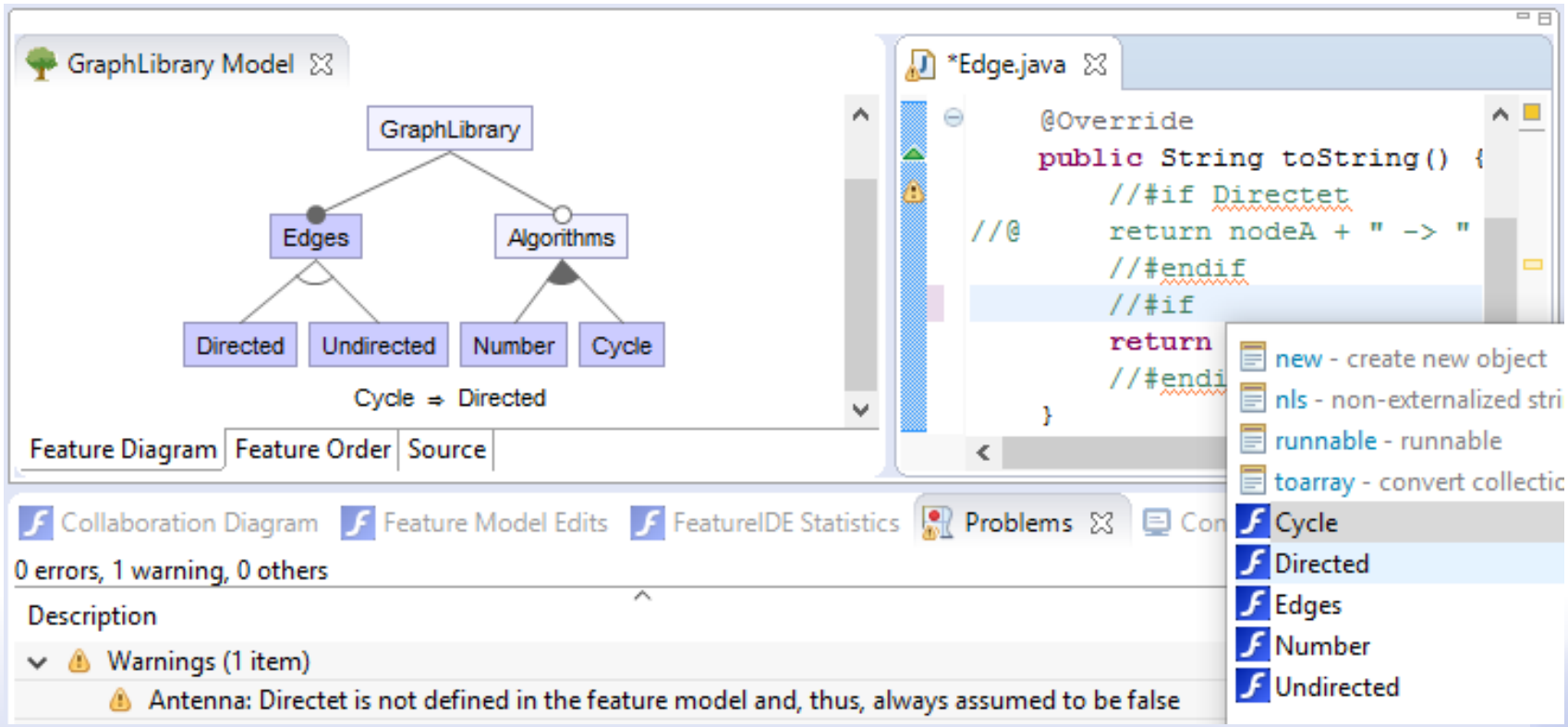
Misleading feature names

- During evolution, meaning of a feature may change
- Misleading names create confusion and obfuscate artifacts
- FeatureIDE renames features automatically in all artifacts



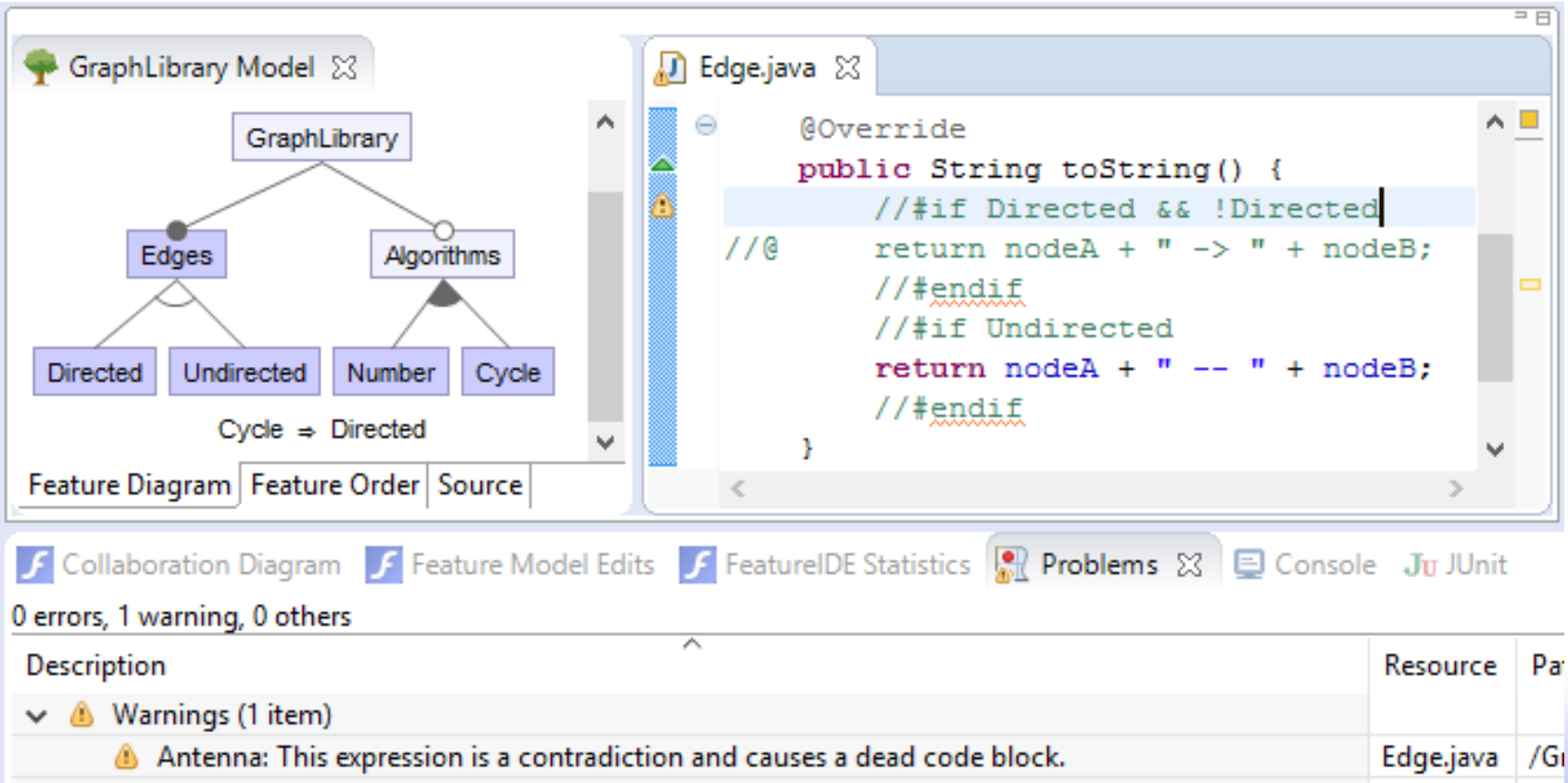
Undefined feature references

- 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



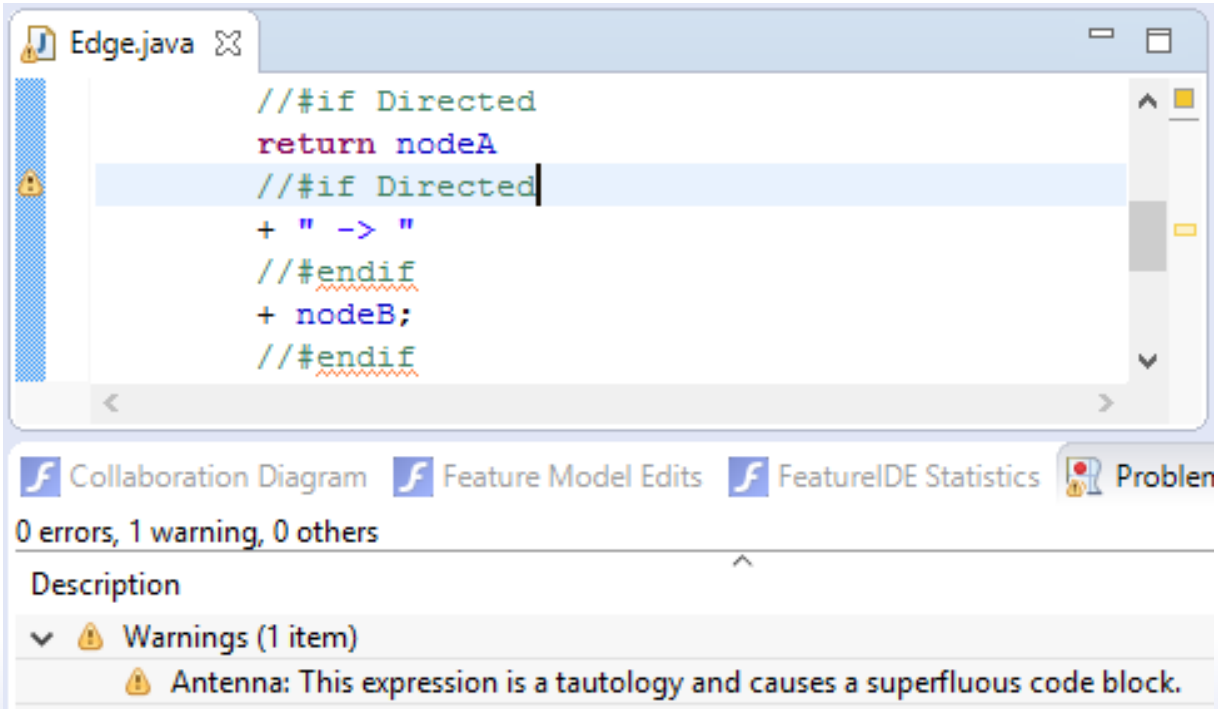
Dead code blocks

- 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

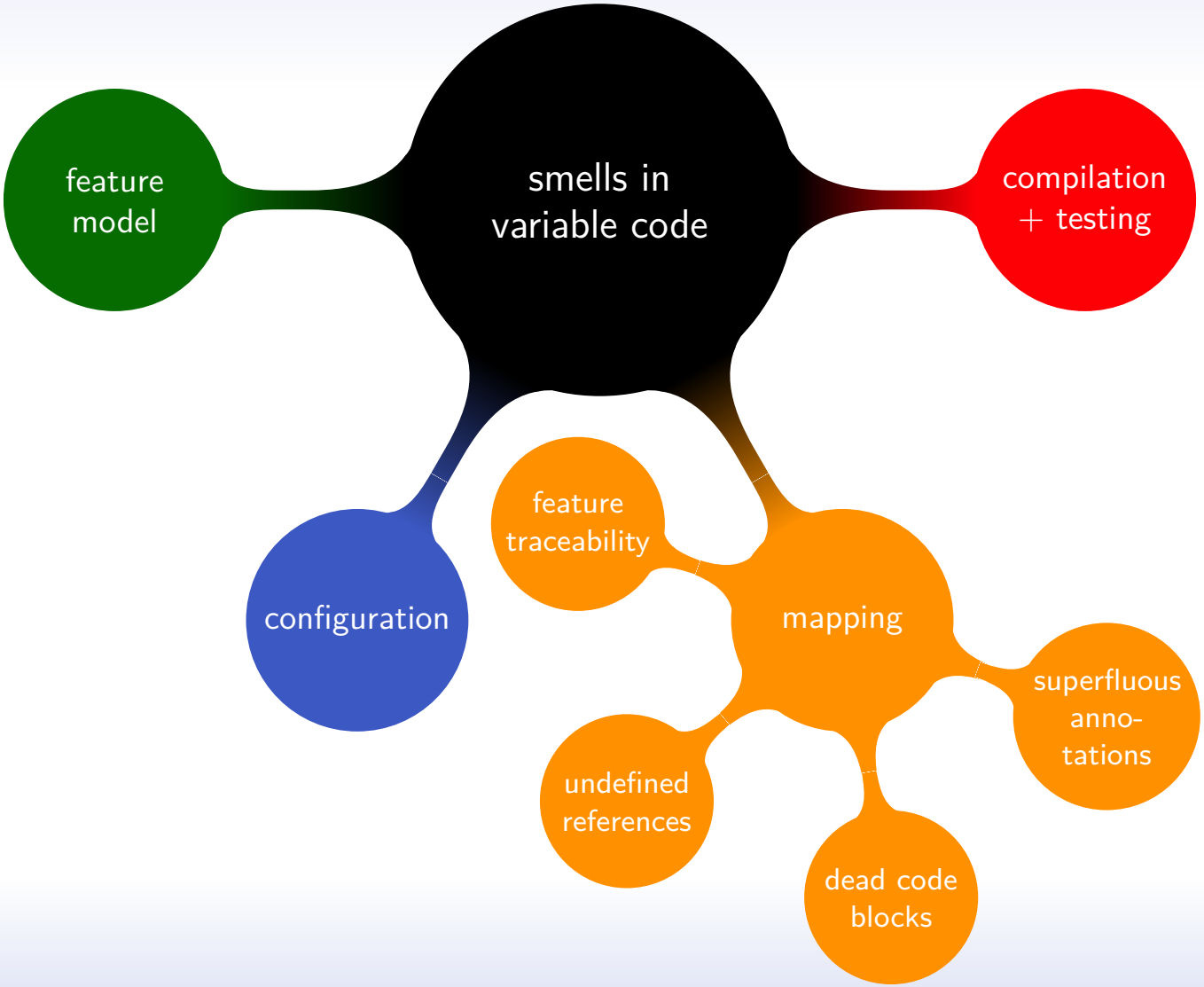


Superfluous preprocessor annotations

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



Tutorial overview



Hands-on IV: fix smells in mapping

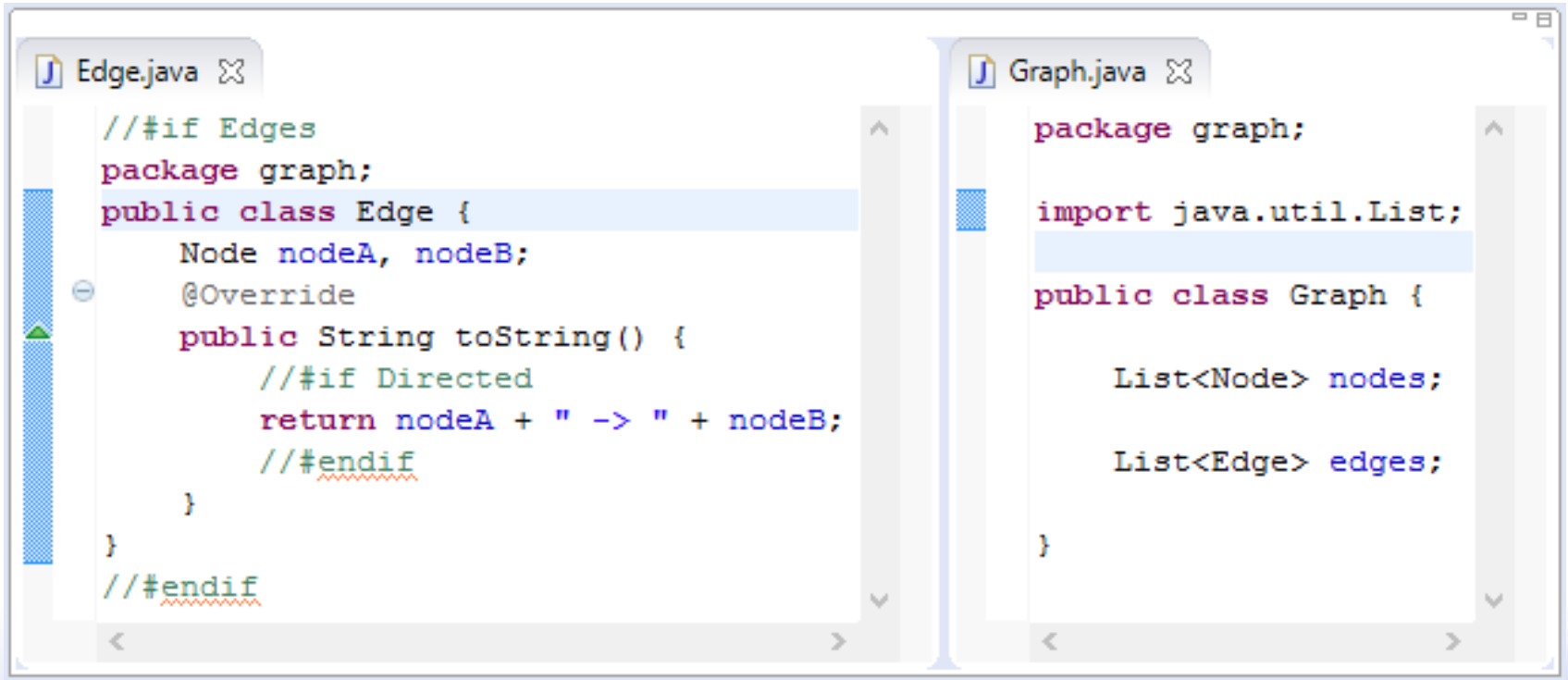
1. Continue with previous project (or import `Elevator-Antenna-v1.2` and close other projects)
2. Trace the misspelled feature `ShortestPath` in source code
   - 2.1 Open feature model and use context menu to assign a color to feature `ShortestPath`
   - 2.2 Identify source files containing this feature using the project explorer (*not package*) 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 enclosed Java code)
   - 3.3 Remove superfluous preprocessor statements (keep Java code)

Part VII

Smells in compilation and testing

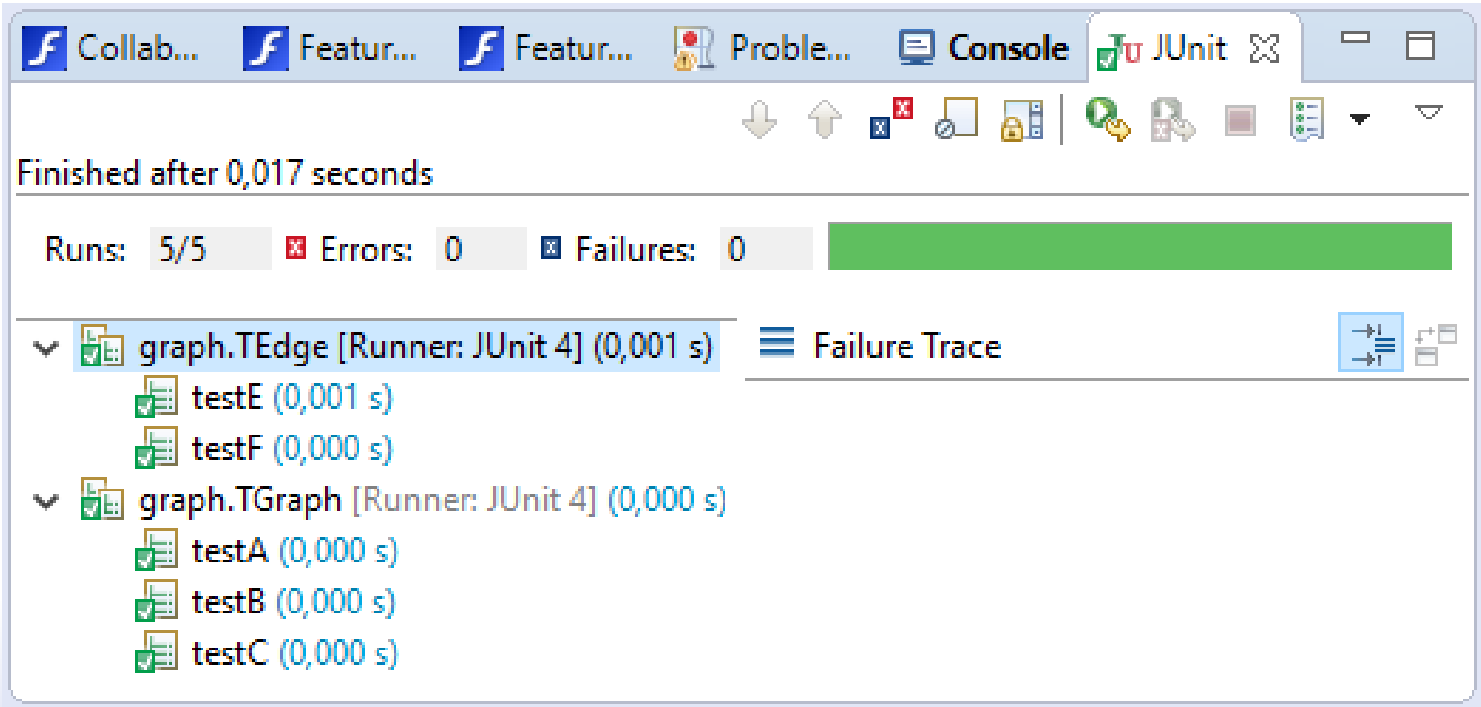
Invisible compiler errors

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



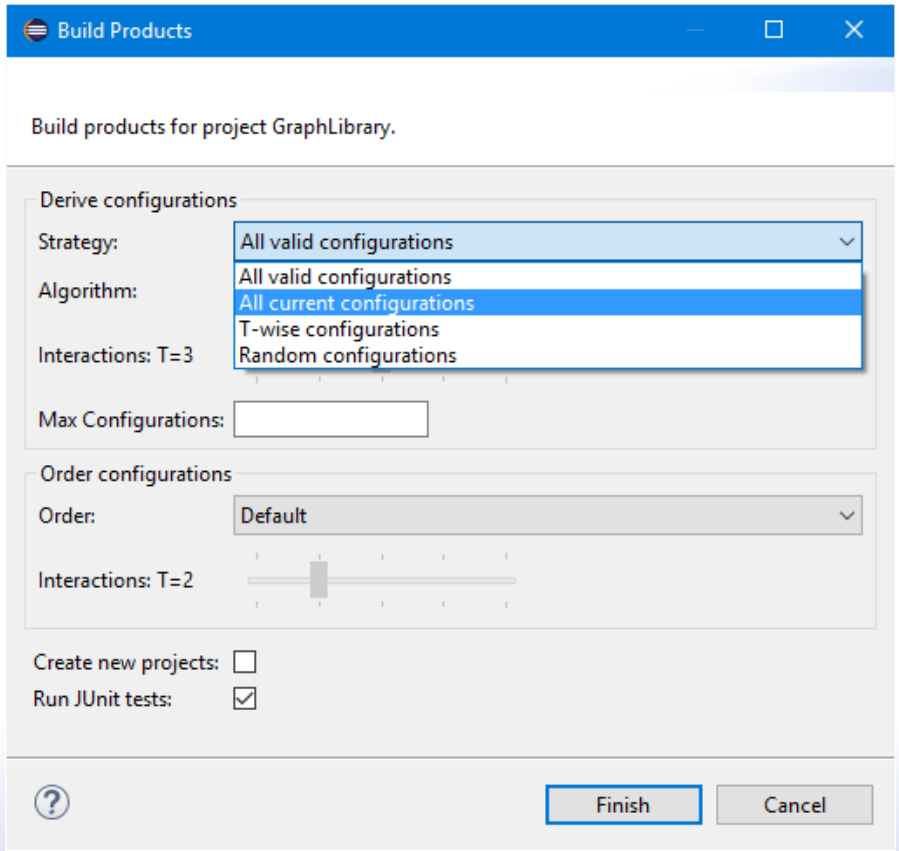
Hidden test failures

- Each unit test may or may not fail for some configurations
- Testing only one configuration is insufficient
- Running unit tests manually for each valid configuration and interpreting the results is time-consuming



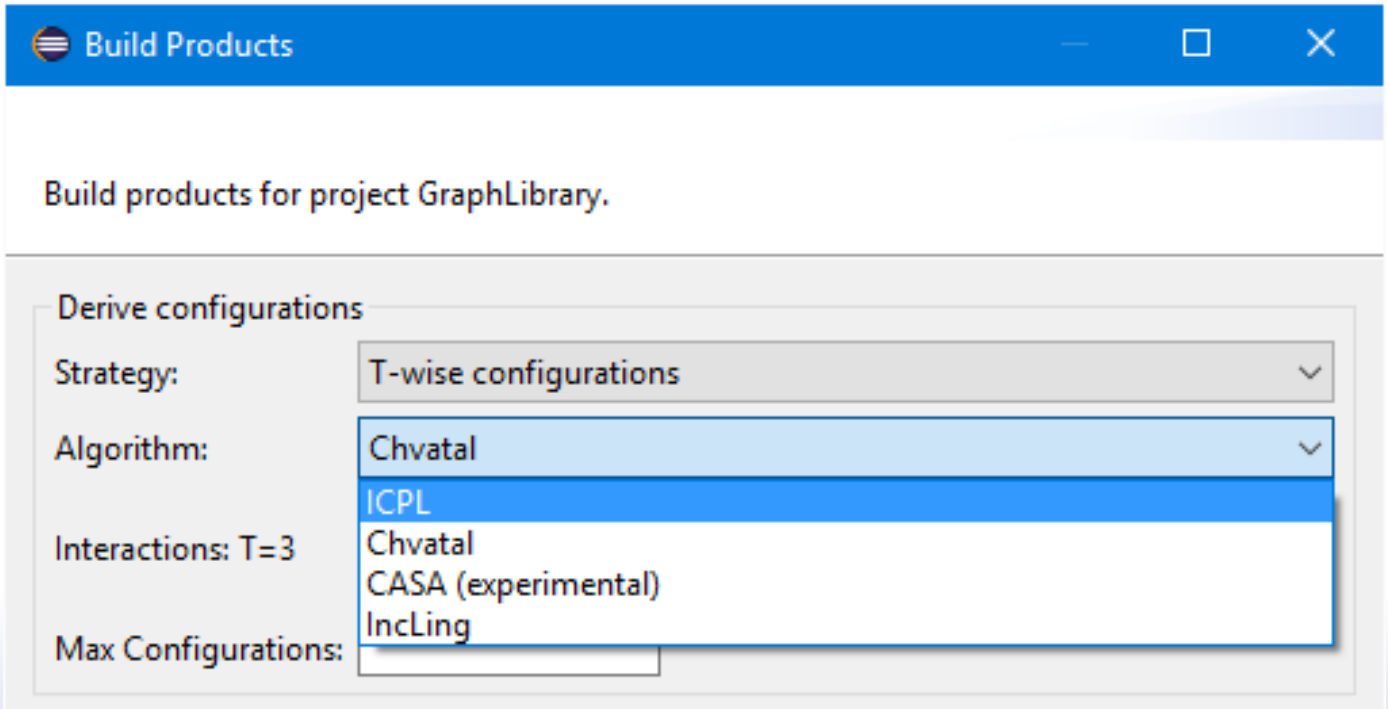
Solution: product generation in batch mode

- 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



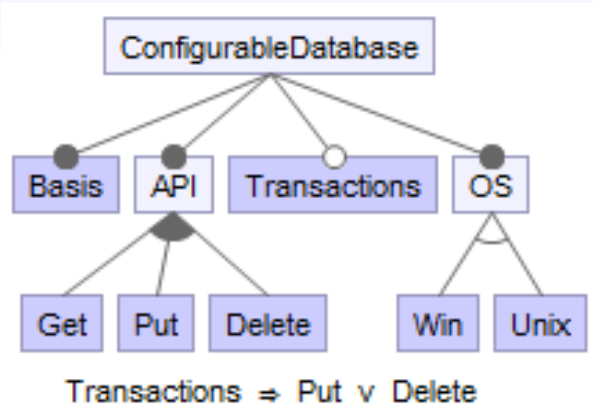
T-wise sampling

- 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)

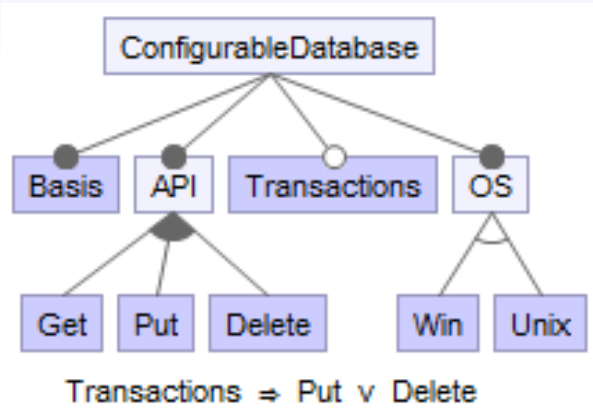
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Example of pairwise sampling (T=2)

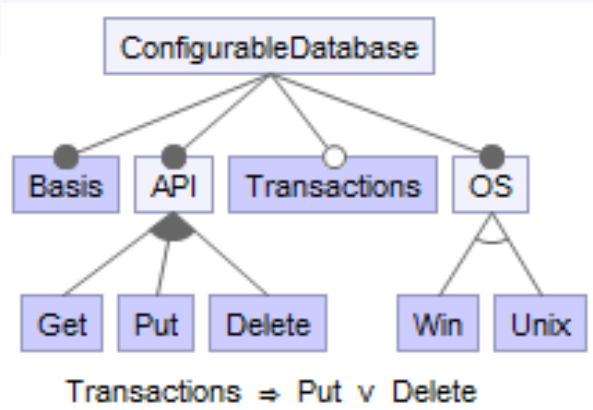
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Example of pairwise sampling (T=2)

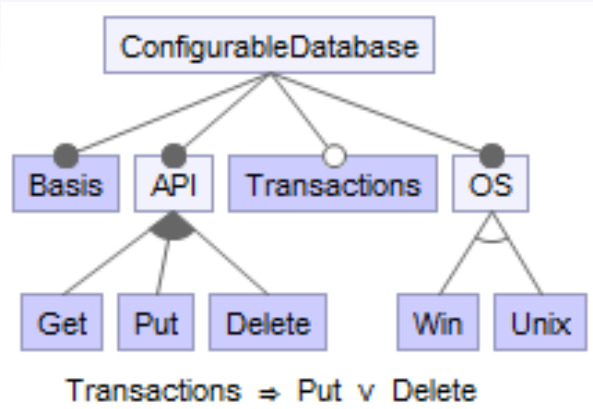
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Example of pairwise sampling (T=2)

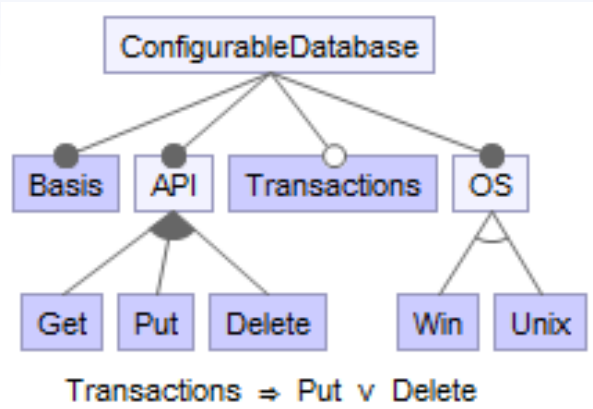
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Example of pairwise sampling (T=2)

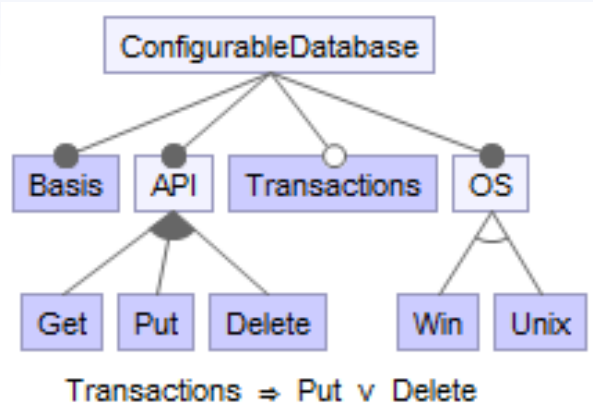
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Example of pairwise sampling (T=2)

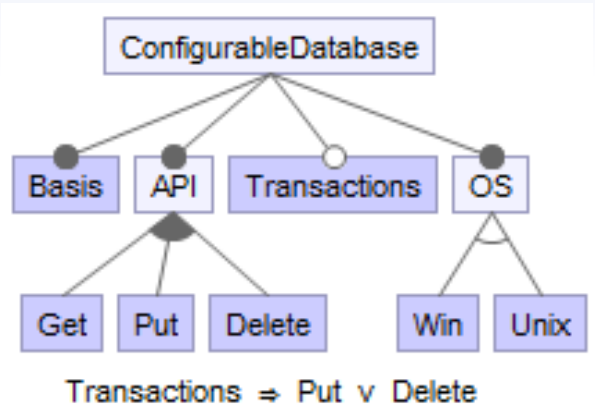
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Example of pairwise sampling (T=2)

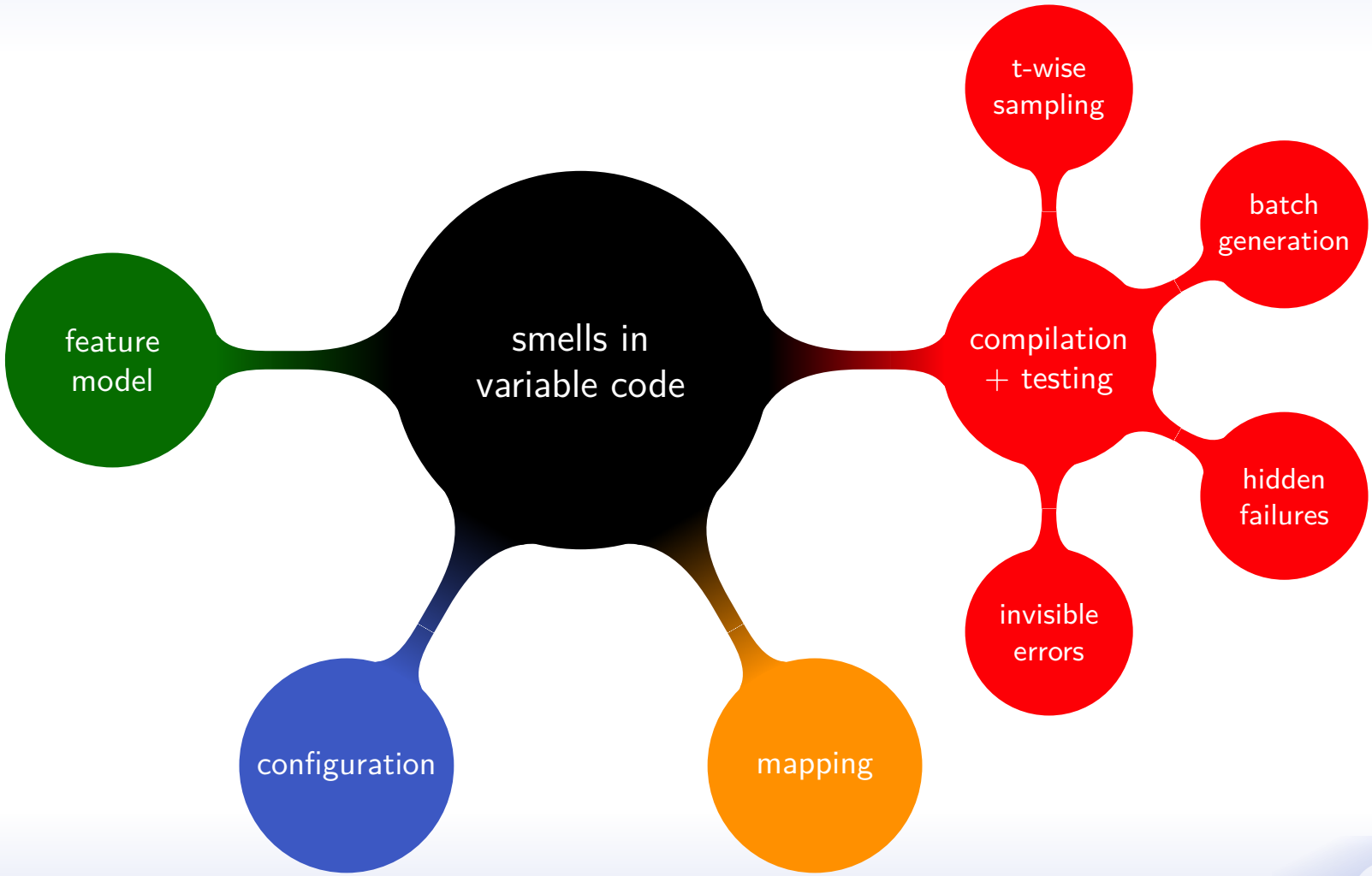
$G \wedge P$	$G \wedge \neg P$	$\neg G \wedge P$	$\neg G \wedge \neg P$
$G \wedge D$	$G \wedge \neg D$	$\neg G \wedge D$	$\neg G \wedge \neg D$
$G \wedge T$	$G \wedge \neg T$	$\neg G \wedge T$	$\neg G \wedge \neg T$
$G \wedge W$	$G \wedge \neg W$	$\neg G \wedge W$	$\neg G \wedge \neg W$
$G \wedge U$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



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

Already 6/26 configurations cover all pairwise interactions

Tutorial overview



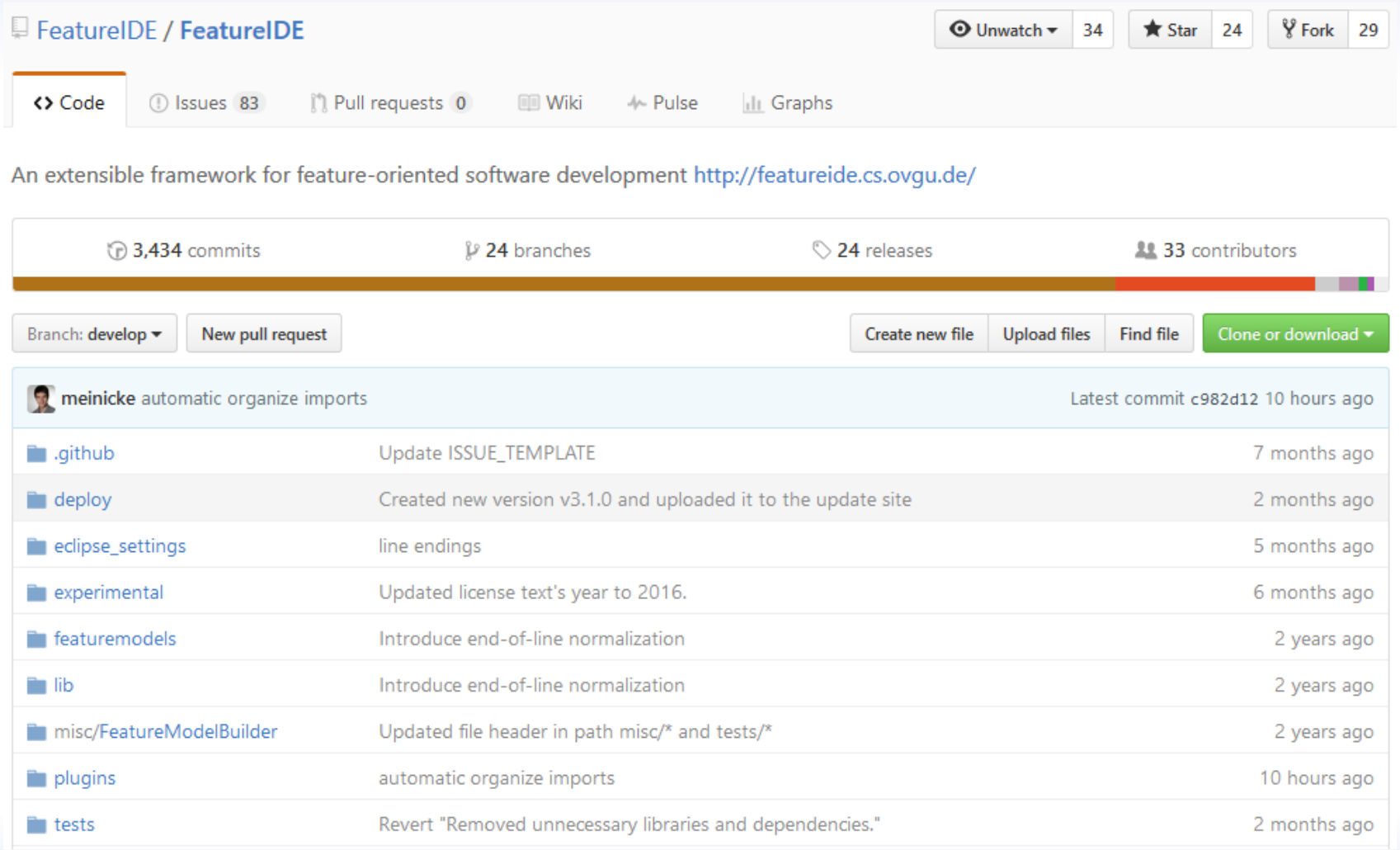
Hands-on V: fix smells with compilation and testing

1. Continue with previous project (or import `Elevator-Antenna-v1.3` and close other projects)
2. Generate and compile all six configurations by right clicking on `project` > `FeatureIDE` > `Product Generator` > `All current configurations`, deselect JUnit tests
3. Fix compiler errors for configuration `Starter` by adding annotations `//#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

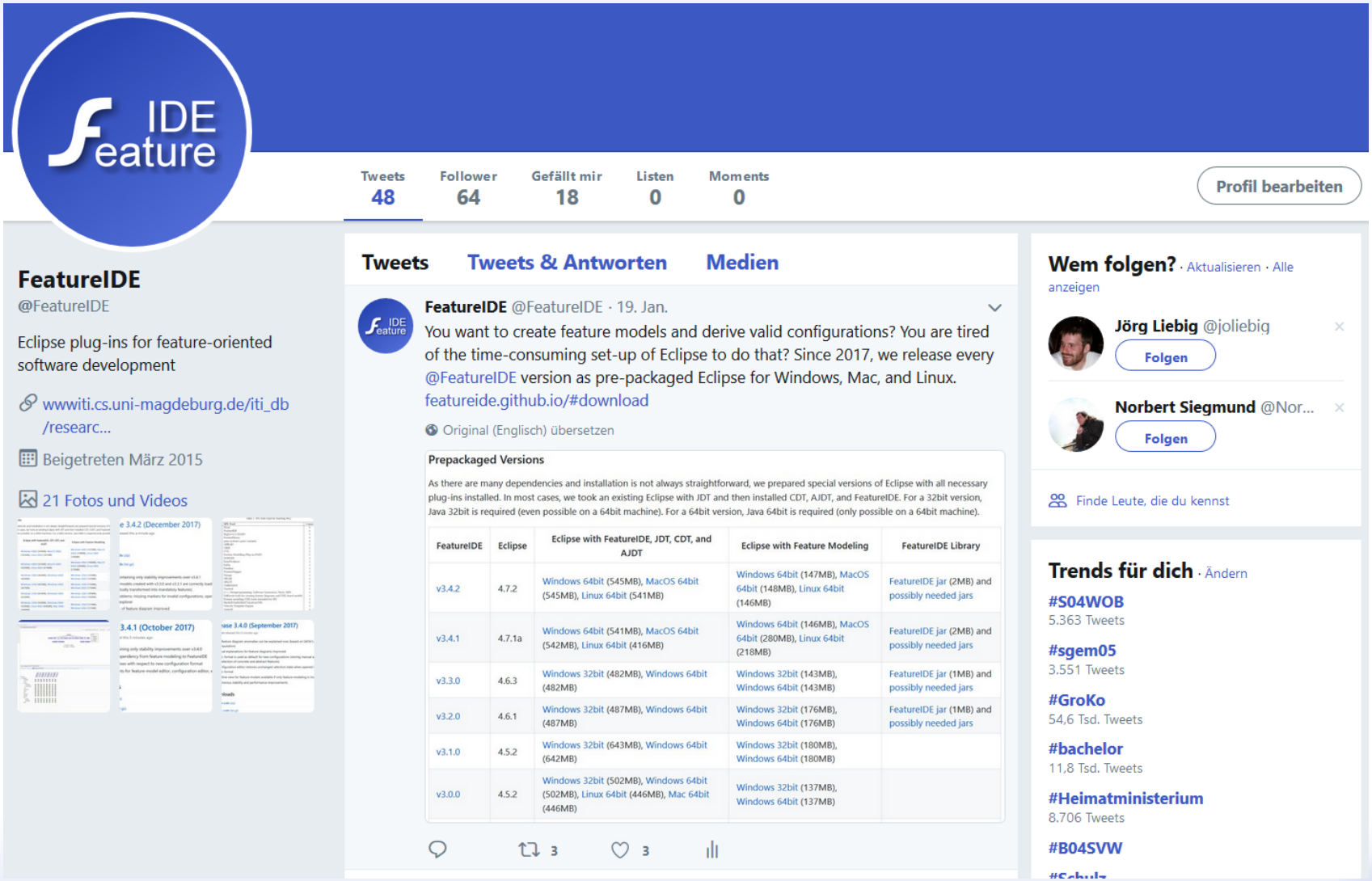
Closing remarks on FeatureIDE

Available under LGPLv3 on Github

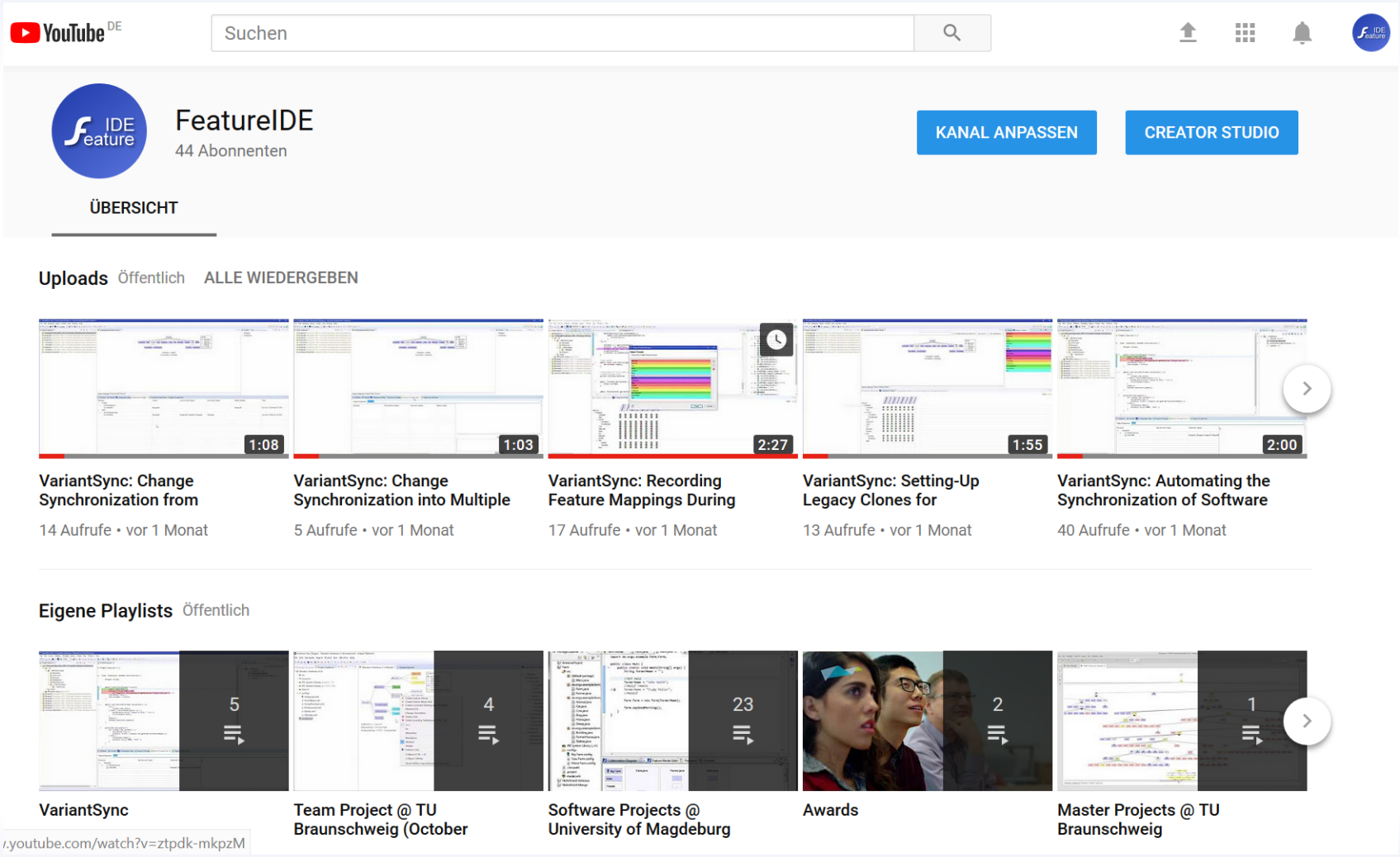


More than 100 support requests since 2007

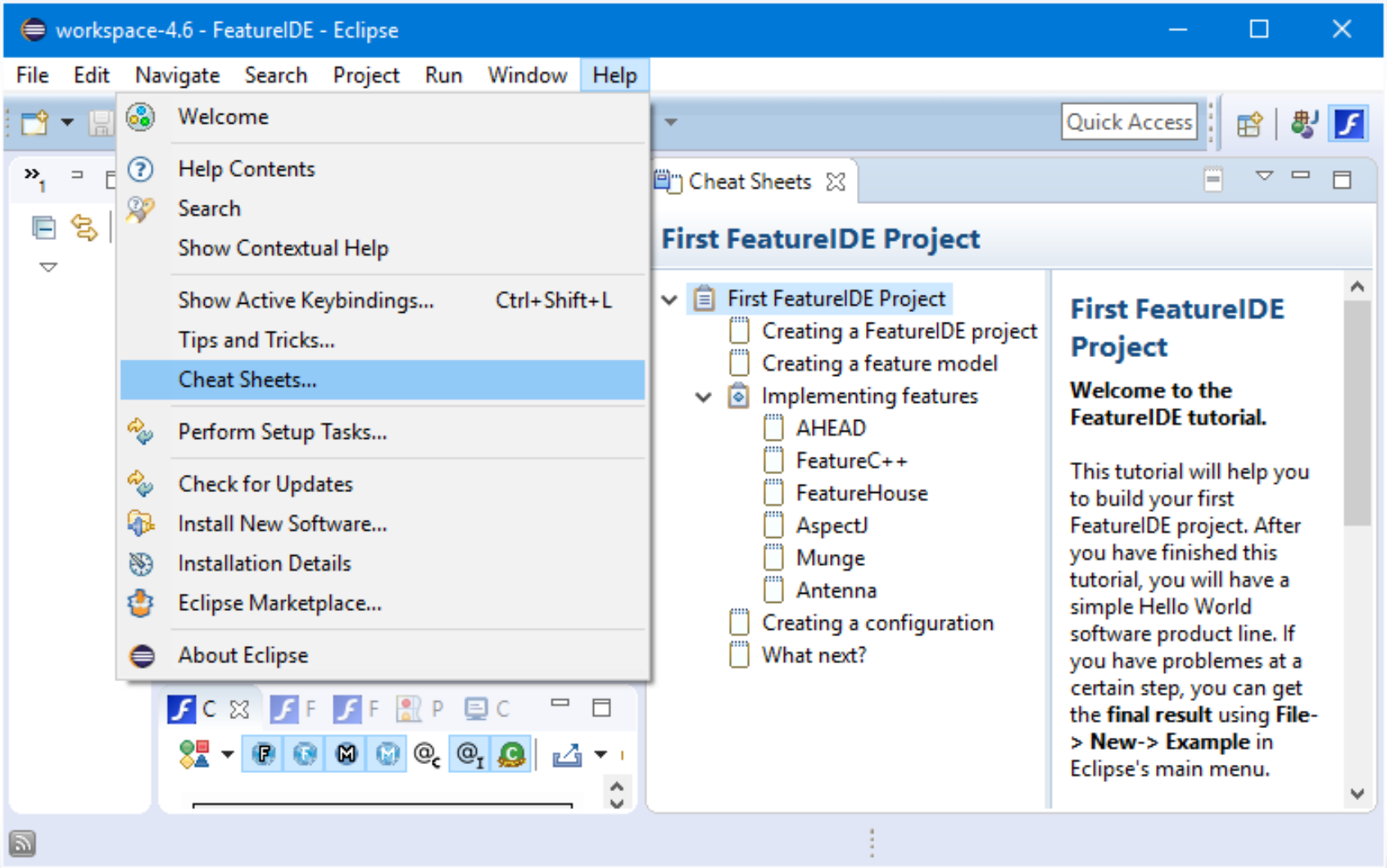
Follow FeatureIDE on Twitter



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