# 5. Conditional Compilation – Handout

Software Product Lines | Thomas Thüm, Elias Kuiter, Timo Kehrer | April 23, 2023

universität ulm

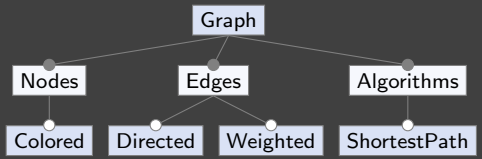
UNIVERSITÄT BERN

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

# 5. Conditional Compilation

# How to Implement Features?

## Given a feature model for graphs . . .

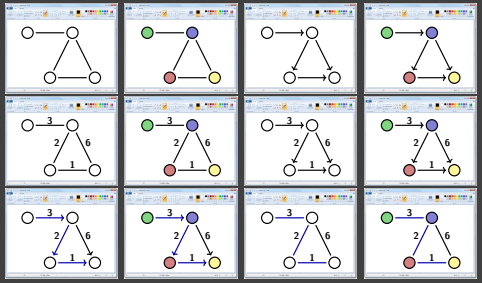



*ShortestPath* $\rightarrow$ *Weighted*

## . . . we can derive a valid configuration

| | | |
|---|---|---|
| $\{G\}$ | $\{G, W\}$ | $\{G, W, S\}$ |
| $\{G, C\}$ | $\{G, C, W\}$ | $\{G, C, W, S\}$ |
| $\{G, D\}$ | $\{G, D, W\}$ | $\{G, D, W, S\}$ |
| $\{G, C, D\}$ | $\{G, C, D, W\}$ | $\{G, C, D, W, S\}$ |

## How to Generate Products Automatically?



## Goals

- descriptive specification of a product (i.e., a configuration, a selection of features)
- automated generation of a product with compile-time variability

Focus of the next three lectures . . .

# Problems of Ad-Hoc Approaches for Variability – Features with Runtime Variability?

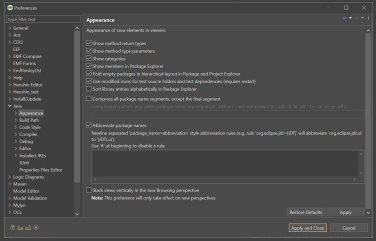
```
public class Config {
  public static boolean COLORED = true;
  public static boolean WEIGHTED = false;
}
```

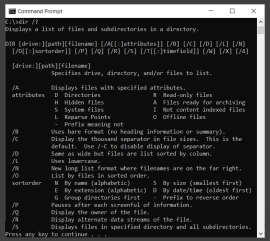
```
class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    if (Config.WEIGHTED) { e.weight = new Weight(); }
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Config.WEIGHTED) {
      throw new RuntimeException();
    }
    Edge e = new Edge(n, m);
    e.weight = w;
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
}
```

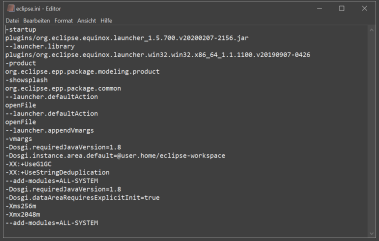
```
class Node {
  Color color; ...
  Node() {
    if (Config.COLORED) { color = new Color(); }
  }
  void print() {
    if (Config.COLORED) {
      Color.setDisplayColor(color);
    }
    System.out.print(id);
  }
}
```

```
class Edge {
  Weight weight; ...
  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    if (Config.WEIGHTED) { weight.print(); }
  }
}
```

# Problems of Ad-Hoc Approaches for Variability – Features with Runtime Variability?




## How to? – Preference Dialog

- implement runtime variability
- compile the program
- run the program
- **manually adjust preferences based on configuration**

## How to? – Command-Line Options / Configuration Files

- implement runtime variability
- compile the program
- **automatically generate command-line options / configuration files based on configuration**
- run the program

# Problems of Ad-Hoc Approaches for Variability – Features with Runtime Variability?

```
public class Config {
  public final static boolean COLORED = true;
  public final static boolean WEIGHTED = false;
}
```

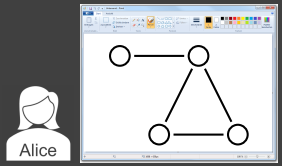
**How to? – Immutable Global Variables**

- implement runtime variability
- automatically generate class with global variables based on configuration
- compile and run the program

**What is missing?**

- automated generation: for preference dialogs
- no compile-time variability / same large binary: for all except immutable global variables
- very limited compile-time variability: for immutable global variables

# Problems of Ad-Hoc Approaches for Variability – Features with Clone-and-Own?




**How to?**

- implement separate project for each product (i.e., branch with version control)
- download project / checkout branch based on configuration
- run build script, if existent
- compile and run the program

**What is missing?**

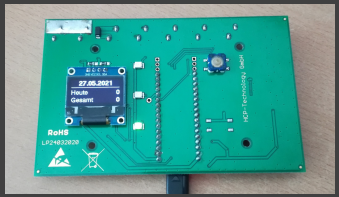
- compile-time variability only for implemented products
- no automated generation: for clone-and-own (with version control systems)
- automated generation based on build script and extra files: for clone-and-own with build systems
- no free feature selection (i.e., configuration)

# Recap: Clone-and-Own with Build Systems [Kuiter et al. 2021]

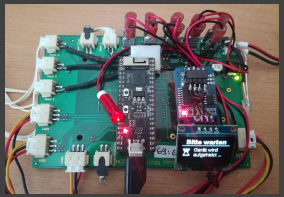
## Case Study: Anesthesia Device

- C application
- targets embedded devices (ESP32)
- configurations are hard-coded as build scripts

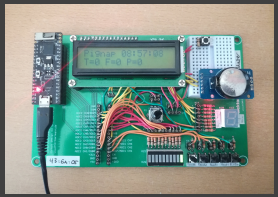
## Production Device: OLED, Clock




## Prototype: OLED Display




## Prototype: LCD, Real-Time Clock



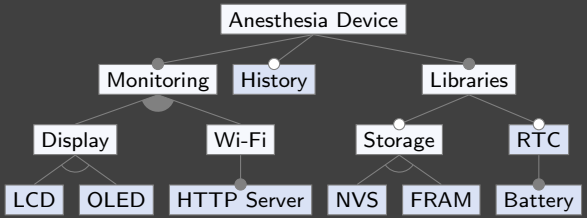
```
main-variants
  config
    cfg_production_oled.mk
    cfg_prototype_lcd_rtc.mk
    cfg_prototype_oled.mk
  lib
    battery.c
    ds3231.c
    fram.c
    i2cdev.c
    rtc.c
  monitor
    display.c
    http.c
    lcd.c
    oled.c
    wifi.c
    build.mk
    history.c
    main.c
```

# Introducing Features to Build Systems [Kuiter et al. 2021]

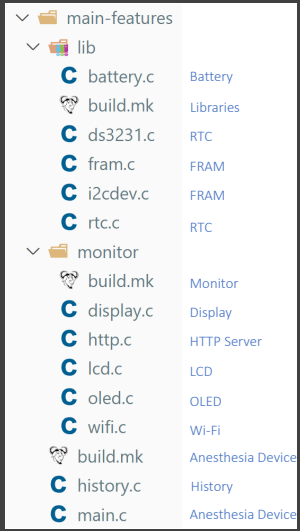
**How to Implement Features with Build Systems?**

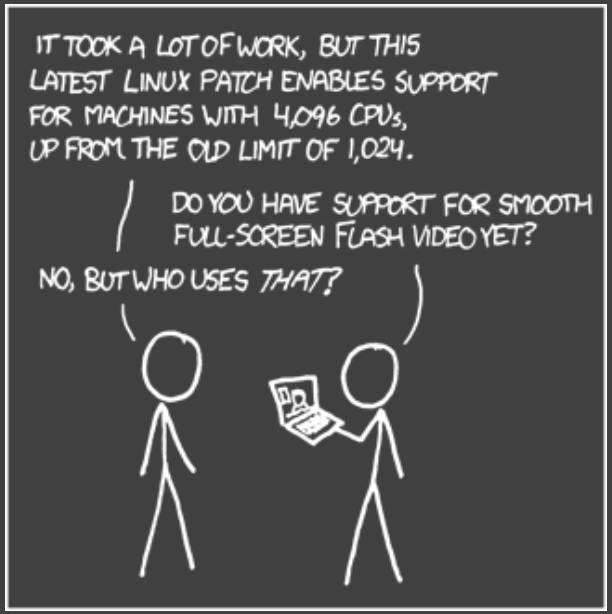
- step 1: model variability in a feature model
- step 2: in build scripts, in- and exclude files based on feature selection
- step 3: pass a feature selection at build time

$\Rightarrow$ one build script per group of related features




$History \rightarrow Storage \wedge RTC$




IT TOOK A LOT OF WORK, BUT THIS LATEST LINUX PATCH ENABLES SUPPORT FOR MACHINES WITH 4,096 CPUs, UP FROM THE OLD LIMIT OF 1,024.
DO YOU HAVE SUPPORT FOR SMOOTH FULL-SCREEN FLASH VIDEO YET?
NO, BUT WHO USES THAT?

# The Linux Kernel – KConfig for Feature Modeling

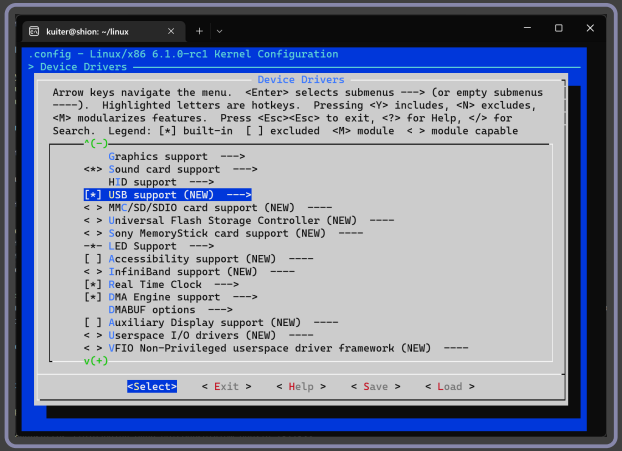
**Part of the x86 Architecture** [linux/arch/x86/Kconfig]

```
config 64BIT
  bool "64-bit kernel" if "$(ARCH)" = "x86"
  default "$(ARCH)" != "i386"
  help
    Say yes to build a 64-bit kernel (x86_64)
    Say no to build a 32-bit kernel (i386)

config X86_32
  def_bool y
  depends on !64BIT
  # Options that are inherently 32-bit kernel only:
  select GENERIC_VDSO_32
  select ARCH_SPLIT_ARG64

config X86_64
  def_bool y
  depends on 64BIT
  # Options that are inherently 64-bit kernel only:
  select ARCH_HAS_GIGANTIC_PAGE
  select ARCH_SUPPORTS_INT128 if CC_HAS_INT128
```

**KConfig Language** [kernel.org]

- configuration language used in embedded/OS development (e.g., Linux, Zephyr, ESP32)
- similar to UVL, but has many quirks (e.g., tristate features, `select`)
- transformation into formula or feature model possible, but not trivial [Oh et al. 2021]

# The Linux Kernel – MenuConfig for Configuration




**make menuconfig**

- configures KConfig models
- generates a `.config` file
- widely used to configure Linux
- still: it is possible to create invalid configurations and products

# The Linux Kernel – KBuild as Build System

## Feature Model with KConfig

[linux/arch/x86/Kconfig]

```
config X86_32 ...
config X86_64 ...

config IA32_EMULATION
  bool "IA32 Emulation"
  depends on X86_64
  help Include code to run legacy 32-bit programs under a 64-bit
       kernel. You should likely enable this, unless you're 100% sure
       that you don't have any 32-bit programs left.
```

## KBuild

[kernel.org]

- a style for writing Makefiles in Linux
- defines goals with Make variables
  - `obj-y`: static linkage (= include feature)
  - `obj-m`: dynamic linkage (= as module)
  - `obj-`: no linkage (= exclude feature)
- full power of Make ⇒ hard to comprehend

## Feature Mapping with KBuild

[linux/arch/x86/Kbuild]

```
# link these subdirectories statically:
obj-y += entry/ # entry routines
obj-y += realmode/ # 16-bit support
obj-y += kernel/ # x86 kernel
obj-y += mm/ # memory management

# link these depending on a configuration option:
obj-$(CONFIG_IA32_EMULATION) += ia32/
obj-$(CONFIG_XEN) += xen/ # paravirtualization

# the KConfig feature model can even be overridden:
obj-$(subst m,y,$(CONFIG_HYPERV)) += hyperv/
```

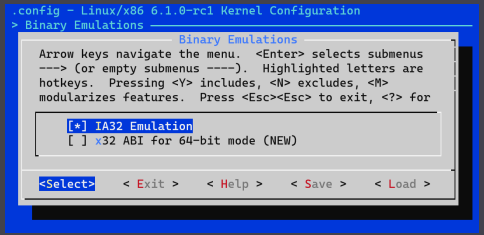
## Recurse into Subsystems

[linux/arch/x86/ia32/Makefile]

```
# ia32 kernel emulation subsystem
obj-$(CONFIG_IA32_EMULATION) := ia32_signal.o
audit-class-$(CONFIG_AUDIT) := audit.o

# IA32_EMULATION and AUDIT required for audit.o:
obj-$(CONFIG_IA32_EMULATION) += $(audit-class-y)
```

# The Linux Kernel – KBuild as Build System

## Interactive Linux Kernel Configurator




## Feature Model and Example Configuration

```
config AUDIT ... # configured as NO
config IA32_EMULATION ... # configured as YES
config HYPERV ... # configured as MODULE
config XEN ... # configured as NO
```

## Feature Mapping

```
obj-y += entry/ realmode/ kernel/ mm/
obj-$(CONFIG_IA32_EMULATION) += ia32/
obj-$(CONFIG_XEN) += xen/
obj-$(subst m,y,$(CONFIG_HYPERV)) += hyperv/
obj-$(CONFIG_IA32_EMULATION) := ia32_signal.o
audit-class-$(CONFIG_AUDIT) := audit.o
obj-$(CONFIG_IA32_EMULATION) += $(audit-class-y)
```

## Feature Mapping for Example Configuration

```
obj-y += entry/ realmode/ kernel/ mm/
obj-y += ia32/
obj- += xen/
obj-y += hyperv/
obj-y := ia32_signal.o
audit-class- := audit.o
obj-y +=
```

i.e., `entry, realmode, kernel, mm, ia32, hyperv, ia32_signal.o` are compiled

# Discussion of Features with Build Systems

**Advantages**

- compile-time variability
  ⇒ **fast, small binaries** with smaller attack surface and without disclosing secrets
- automated generation of arbitrary products
  ⇒ **free feature selection**
- allows in- and exclusion of individual files or even entire subsystems
  ⇒ high-level, **modular variability**

**Challenges**

- not easily reconfigurable at run- or load-time
- build scripts may become complex, there is no limit to what can be done (e.g., you can run arbitrary shell commands on files)
  ⇒ **hard to understand and analyze**
- no simple in- and exclusion of individual lines or chunks of code
  ⇒ high-level use **only**!

# Features with Build Systems – Summary

**Lessons Learned**

- ad-hoc variability is lacking
- features with build systems allow for automated generation of products and free feature selection
- build systems include entire files, not lines or chunks

**Further Reading**

- Apel et al. 2013, Chapter 5.2.1, pp. 105–106 — brief introduction to variability in build scripts
- Apel et al. 2013, Chapter 5.2.3, pp. 107–108 — variability in build scripts of the Linux kernel

**Practice**

1. How are features implemented with build systems different from clone-and-own with build systems?
2. What is the respective granularity?

# 5. Conditional Compilation

## 5a. Features with Build Systems

## 5b. Features with Preprocessors

- Granularity of Variability
- What is a Preprocessor?
- CPP – The C Preprocessor
- Preprocessors for Java
- Preprocessors in FeatureIDE
- Discussion of Preprocessors
- Preprocessor-Based Product Lines in the Wild
- Summary

## 5c. Feature Traceability

# Granularity of Variability

**Granularity of Variability**

- Depending on the implementation technique, variability can be introduced at different levels of granularity.
- A level of granularity refers to
  - the hierarchical organization of implementation artifacts (e.g., through the file system),
  - the hierarchical structure of an implementation artifact (e.g., given by its syntax)

**Granularity Levels in Java**

modules > libraries > packages > classes > members > statements > parameters

**What we have seen so far?**

- Coarse-grained: Clone-and-own with version control (entire variants)
- Medium-grained: Clone-and-own with build systems (file level)
- Medium-grained: Features with build systems (file level)
- Medium-grained: Design patterns for variability (class or member level)
- Fine-grained: Runtime parameters (statement level)

**What is missing?**

Yet no approach supporting fine-grained compile-time variability!

# What is a Preprocessor? [Apel et al. 2013, pp. 110–111]

**Preprocessor**

- tool manipulating source code before compilation (i.e., at compile time)
- preprocessors are used:
  - to inline files (e.g., header files)
  - to define and expand macros (cf. metaprogramming)
  - for **conditional compilation** (e.g., remove debug code for release)

**Preprocessor**

- the C Preprocessor (CPP) is used in almost every C/C++ project
- preprocessors are typically oblivious to the target language as they operate on text files (e.g., the C Preprocessor can also used for Fortran or Java)
- conditional compilation is a very common technique to implement product lines

# CPP – The C Preprocessor

## CPP Directives [cppreference.com]

file inclusion

- `#include`

text replacement

- `#define`
- `#undef`

conditional compilation

- `#if`, `#endif`
- `#else`, `#elif`
- `#ifdef`, `#ifndef`
- new: `#elifdef`, `#elifndef`

## Example Input

```
1  #include <iostream>
2
3  #define Hello true
4  #define Beautiful true
5  #define Wonderful false
6  #define World true
7
8  int main() {
9    ::std::cout
10   #if Hello
11     << "Hello "
12   #endif
13   #if Beautiful
14     << "beautiful "
15   #endif
16   #if Wonderful
17     << "wonderful ";
18   #endif
19   #if World
20     << "world!"
21   #endif
22     << std::endl;
23 }
```

## Example Output (Simplified)

```
1 int main() {
2   ::std::cout
3     << "Hello "
4     << "Beautiful "
5     << "World!"
6     << std::endl;
7 }
```

## Why simplified?

- preprocessed file can get very long due to to included header files
- preprocessors typically do not remove line breaks to not influence line numbers reported by compilers

# Munge – A Simple Preprocessor for Java [Meinicke et al. 2017]

## Example Input and Output

```
 1 public class Main {                          public class Main {
 2   public static void main(String[]             public static void main(String[]
         args) {                                      args) {
 3     /*if[Hello]*/
 4     System.out.print("Hello");                   System.out.print("Hello");
 5     /*end[Hello]*/
 6     /*if[Beautiful]*/
 7     System.out.print(" beautiful");              System.out.print(" beautiful");
 8     /*end[Beautiful]*/
 9     /*if[Wonderful]*/
10     System.out.print(" wonderful");
11     /*end[Wonderful]*/
12     /*if[World]*/
13     System.out.print(" world!");                 System.out.print(" world!");
14     /*end[World]*/
15   }                                            }
16 }                                            }
```

## Munge

- preprocessor for Java
- written in Java
- about 300 LOC

## Call of Munge on Command Line

```
Munge -DHello -DBeautiful -DWorld Main.java targetDir
```

# Antenna – An In-Place Preprocessor for Java [Meinicke et al. 2017]

## Example Input and Output

```
1  public class Main {
2    public static void main(String[]
         args) {
3      //#if Hello
4      System.out.print("Hello");
5      //#endif
6      //#if Beautiful
7      System.out.print(" beautiful");
8      //#endif
9      //#if Wonderful
10 //@ System.out.print(" wonderful");
11     //#endif
12     //#if World
13     System.out.print(" world!");
14     //#endif
15   }
16 }
```

```
public class Main {
  public static void main(String[]
       args) {
    //#if Hello
    System.out.print("Hello");
    //#endif
    //#if Beautiful
//@ System.out.print(" beautiful");
    //#endif
    //#if Wonderful
    System.out.print(" wonderful");
    //#endif
    //#if World
    System.out.print(" world!");
    //#endif
  }
}
```

## Antenna

- preprocessor for Java
- written in Java
- has been used for Java ME (micro edition) projects

## Call of Munge on Command Line

```
java Antenna Main.java Hello,World,Beautiful
```

# In-Place and Out-of-Place Preprocessors [Meinicke et al. 2017]

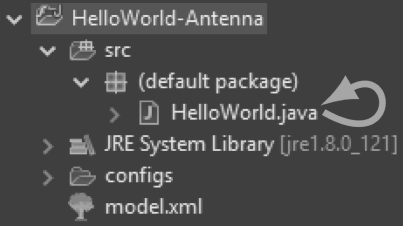
**In-Place Preprocessor**

- input file manipulated
- lines commented out where necessary
- often: better support in IDEs

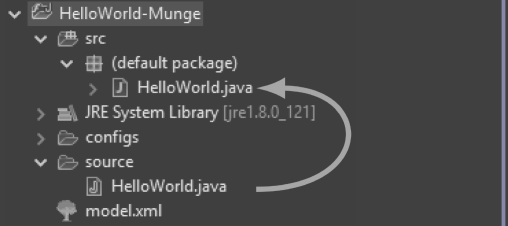
**Out-of-Place Preprocessor**

- separate output file generated
- lines deleted where necessary
- often: worse support in IDEs

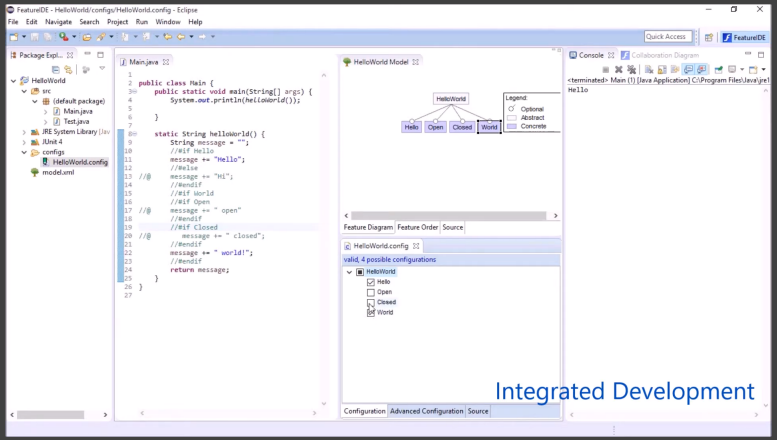
**Antenna, . . .**




**CPP, Munge, . . .**



# Preprocessors in FeatureIDE [Meinicke et al. 2017]




## Demo Video

- preprocessing with Antenna on command line
- feature modeling
- warnings for unreferenced features
- content assist proposing feature names
- configuration and automated regeneration
- (first 2 min relevant here)

# Discussion of Preprocessors

**Powerful Preprocessors**

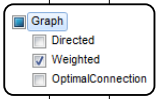
we can annotate

- complete files (i.e., Java classes)
- members (e.g., fields, methods)
- (parts of) statements
- parameters
- ...
- single characters

and automatically generate variants

everyone happy?

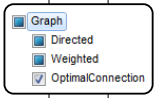
```
class Edge {
  Node first, second;
//#ifdef Weighted
  int weight;
//#endif
  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first
//#ifndef Directed
      || first == e.second
      && second == e.first
      ) && weight == e.weight;
//#endif
  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```




```
class Edge {
  Node first, second;

  int weight;

  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first

      || first == e.second
      && second == e.first
      ) && weight == e.weight;

  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

# Discussion of Preprocessors

```
class Edge {
  Node first, second;
//#ifdef Weighted
  int weight;
//#endif
  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first
//#ifndef Directed
      || first == e.second
      && second == e.first
      ) && weight == e.weight;
//#endif
  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```




```
class Edge {
  Node first, second;

  int weight;

  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first

  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

**Syntax Error**

missing ) and semicolon!

**Powerful Preprocessors**

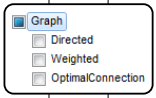
can produce syntactically ill-formed programs:

- errors may appear only for certain configurations,
- are hard to find, and
- are more likely than for other techniques

[more in Lecture 5c]

# Discussion of Preprocessors

```
class Edge {
  Node first, second;
//#ifdef Weighted
  int weight;
//#endif
  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first
//#ifndef Directed
      || first == e.second
      && second == e.first
      ) && weight == e.weight;
//#endif
  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```




```
class Edge {
  Node first, second;


  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first

      || first == e.second
      && second == e.first
      ) && weight == e.weight  ;

  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

**Type Error**

`weight` undefined!

**Powerful Preprocessors**

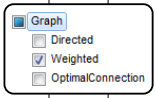
can produce ill-typed programs:

- errors may appear only for certain configurations,
- are hard to find, and
- are more likely than for other techniques

[more in Lecture 10]

# Discussion of Preprocessors

```
class Edge {
  Node first, second;
//#ifdef Weighted
  int weight;
//#endif
  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first
//#ifndef Directed
      || first == e.second
      && second == e.first
      ) && weight == e.weight;
//#endif
  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```




```
class Edge {
  Node first, second;

  int weight;

  Edge(Node first, Node second) {...}
  boolean equals(Edge e) {
    return (first == e.first
      && second == e.first

      || first == e.second
      && second == e.first
      ) && weight == e.weight;

  }
  void testEquality() {
    Node a = new Node();
    Node b = new Node();
    Edge e = new Edge(a, b);
    Assert.assertTrue(e.equals(e));
} }
```

**Runtime Error**

assertion failed!

**Powerful Preprocessors**

can produce programs with unwanted behavior:

- errors may appear only for certain configurations,
- are hard to find, and
- are more likely than for other techniques

[more in Lecture 11]

# Problem: Source-Code "Obfuscation"

## Observations on Readability

- Mixing **two languages** (C and #ifdefs, or Java and Munge, ...)
- Control flow difficult to understand
- Long annotations hard to find
- Extra line breaks destroy layout

## Problem: Undisciplined Annotations

- the preprocessor language (e.g., #ifdefs) does not care about the preprocessed language (e.g., C)
- allows for "undisciplined" preprocessor usage (precise definition later)
- considerably worsens readability

## Can you read this source code? [Liebig et al. 2011; xterm]

```
#if defined(__GLIBC__)
  // additional lines of code
#elif defined(__MVS__)
  result = pty_search(pty);
#else
#ifdef USE_ISPTS_FLAG
  if (result) {
#endif
    result = ((*pty=open("/dev/ptmx", O_RDWR))<0);
#endif
#if defined(SVR4) || defined(__SCO__) || \
    defined(USE_ISPTS_FLAG)
    if (!result)
      strcpy(ttydev, ptsname(*pty));
#ifdef USE_ISPTS_FLAG
    IsPts = !result;
  }
#endif
#endif
```

# Discussion of Features with Preprocessors

## Advantages

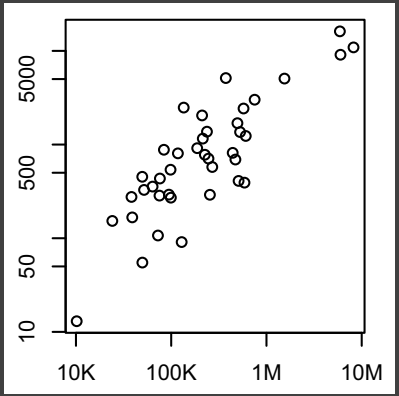
- Well-known and mature tools, readily available
- Easy to use
  $\Rightarrow$ just annotate and remove
- Supports **compile-time variability**
- Flexible, arbitrary levels of **granularity**
- Can handle code and non-code artifacts (**uniformity**)
- Little **preplanning** required
  $\Rightarrow$ variability can be added to an existing project

## Challenges

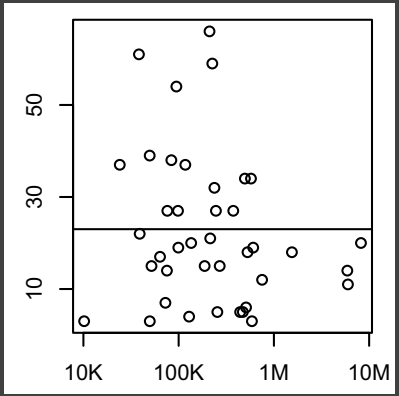
- **Scattering** and **tangling**
  $\Rightarrow$ separation of concerns?
- Mixes multiple languages in the same development artifact
- May **obfuscate** source code and severely impact its readability
- Hard to analyze and process for existing IDEs
- Often used in an ad-hoc or **undisciplined** fashion
- Prone to subtle syntax, type, or runtime errors which are hard to detect [Lecture 9–Lecture 11]

# Preprocessor-Based Product Lines in the Wild [Liebig et al. 2010]

**Number of Features**

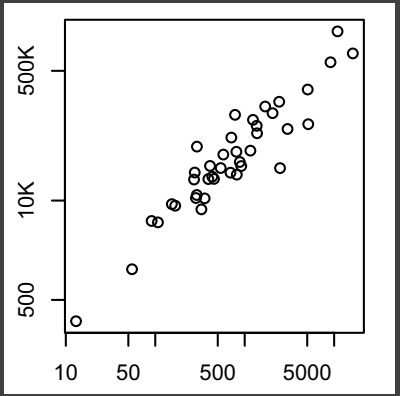



**Percentage of Variable Code**

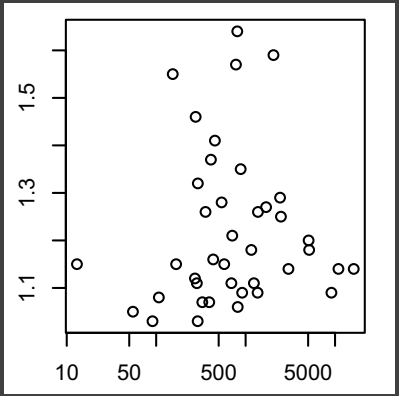


# Preprocessor-Based Product Lines in the Wild [Liebig et al. 2010]

## Lines of Variable Code

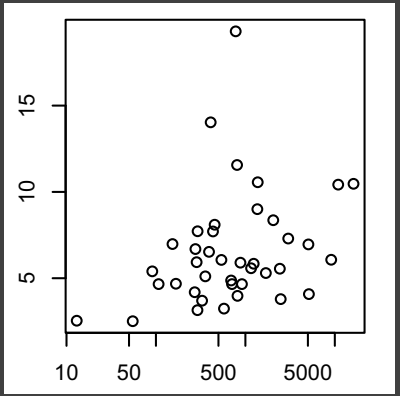



## Average Nesting Depth

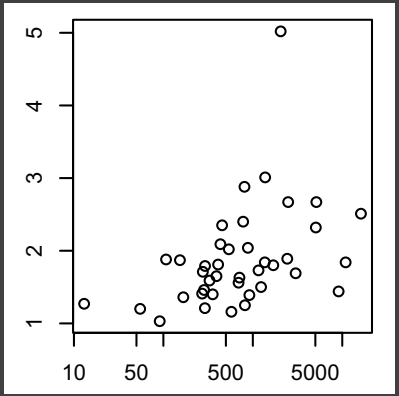


# Preprocessor-Based Product Lines in the Wild [Liebig et al. 2010]

**Average Number of Feature References**




**Average Number of Features per Annotation**



# Features with Preprocessors – Summary

**Lessons Learned**

- granularity of variability at file level is not sufficient
- preprocessors facilitate fine-grained variability within files
- a widely applied preprocessor is the C Preprocessor
- industrial systems often combine preprocessors and build systems for features (e.g., Linux kernel)

**Further Reading**

- Apel et al. 2013, Section 5.3 Preprocessors

**Practice**

1. Antenna performs an in-place transformation on implementation artifacts. What might be the benefits of using an in-place approach? Do you see any drawbacks?
2. The preprocessors we have seen so far are also called lexical preprocessors. What is emphasized by the notion of lexical and can you think of other preprocessing approaches?
3. The literature on software product lines has coined the term "#ifdef hell". What could be meant with this?

# 5. Conditional Compilation

## 5a. Features with Build Systems

## 5b. Features with Preprocessors

## 5c. Feature Traceability

# Recap: Code Scattering and Tangling

```
public class Graph {
  List nodes = new ArrayList();
  List edges = new ArrayList();

  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    e.weight = w;
    return e;
  }
  void print() {
    for (int i = 0; i < edges.size(); i++) {
      ((Edge) edges.get(i)).print();
    }
  }
}
```

```
public class Node {
  int id = 0;
  Color color = new Color();

  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
public class Edge {
  Node a, b;
  Weight weight = new Weight();

  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    weight.print();
  }
}
```

```
public class Color {
  static void
        setDisplayColor(
        Color c) {...}
}
```

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

**What is . . .**

- code scattering?
- code tangling?
- feature traceability?

# Recap: Code Scattering and Tangling

```
public class Graph {
  List nodes = new ArrayList();
  List edges = new ArrayList();

  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    e.weight = new Weight();
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    Edge e = new Edge(n, m);
    nodes.add(n); nodes.add(m); edges.add(e);
    e.weight = w;
    return e;
  }
  void print() {
    for (int i = 0; i < edges.size(); i++) {
      ((Edge) edges.get(i)).print();
    }
  }
}
```

```
public class Node {
  int id = 0;
  Color color = new Color();

  void print() {
    Color.setDisplayColor(color);
    System.out.print(id);
  }
}
```

```
public class Edge {
  Node a, b;
  Weight weight = new Weight();

  Edge(Node a, Node b) {
    this.a = a; this.b = b;
  }
  void print() {
    a.print(); b.print();
    weight.print();
  }
}
```

```
public class Color {
  static void
        setDisplayColor(
        Color c) {...}
}
```

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

**Is it a problem for . . .**

(a) single systems?

(b) runtime variability? [Lecture 2]

(c) clone-and-own? [Lecture 3]

(d) conditional compilation? [Lecture 5]

# Feature Traceability Problem

**Recap: Feature Traceability**

Feature traceability is the ability to trace a feature throughout the software life cycle (i.e., from requirements to source code).

**Recap: Intuition on Feature Traceability**

find feature  in product 

**Feature Traceability (revisited)** [Apel et al. 2013, p. 54]

"Feature traceability is the ability to trace a feature from the **problem space** (for example, the feature model) to the **solution space** (that is, its manifestation in design and code artifacts)."

**Feature Traceability Problem**

The feature traceability problem is the challenge to trace a feature in software artifacts (i.e., all or some locations).

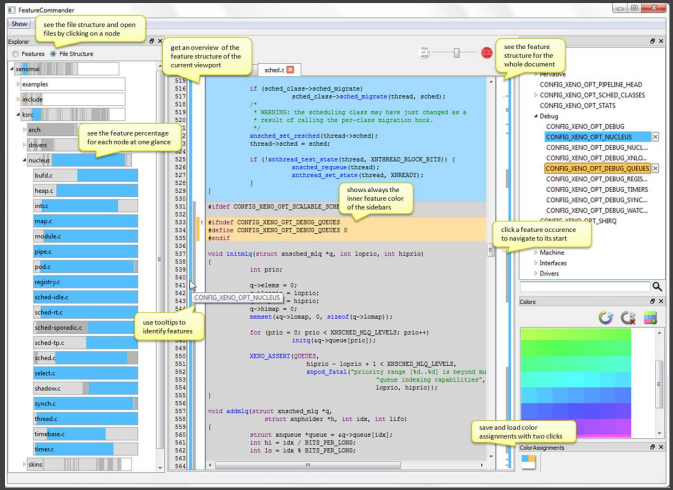
# Feature Traceability with Colors – Feature Commander

## Feature Commander

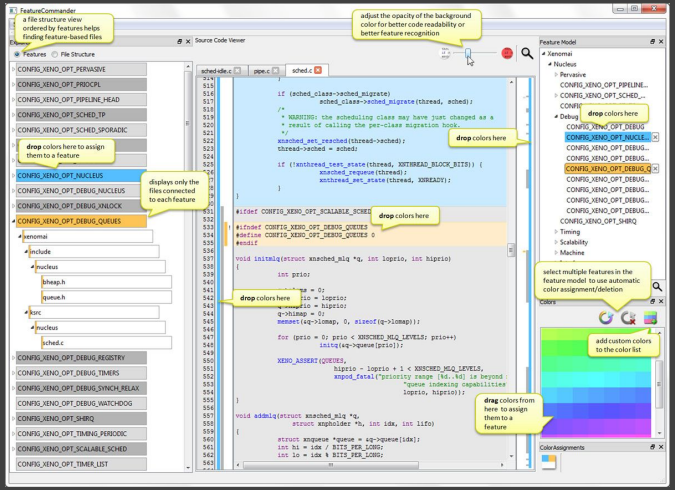
- each feature can be assigned to a color
- color used to support feature traceability
- features not assigned to a color shown in a shade of grade
- visualizations based on preprocessor directives

## Demo Video

(there is no sound)



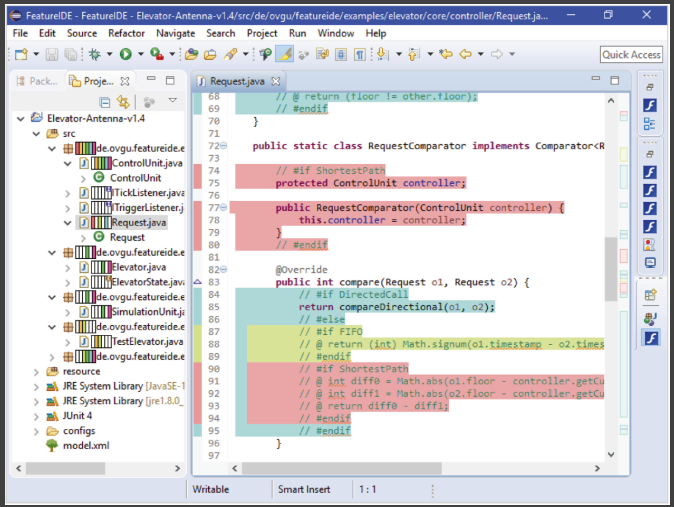
# Feature Traceability with Colors – Feature Commander




## Feature Commander

- research prototype (last update August 2010)
- only static view on the source code
- only works for Xenomai (a real-time core for Linux)
- further reading on experiments with developers: Feigenspan et al. 2013

# Feature Traceability with Colors – FeatureIDE




**FeatureIDE** [Meinicke et al. 2017]

- tool support for feature traceability
- inspired by Feature Commander
- color can be assigned to features
- colors used in feature model, configurations, package explorer, and source code

**Demo Video (last minute only)**

- collaboration view
- support for colors

# Virtual Separation of Concerns [Kästner 2010]

**Virtual Separation of Concerns**

- annotations of code based on the underlying structure (i.e., abstract syntax)
- **disciplined annotations**: only optional nodes in the abstract syntax tree can be annotated
- tool support used to provide views and navigate in source code
- **syntactic correctness** guaranteed for all generated program variants

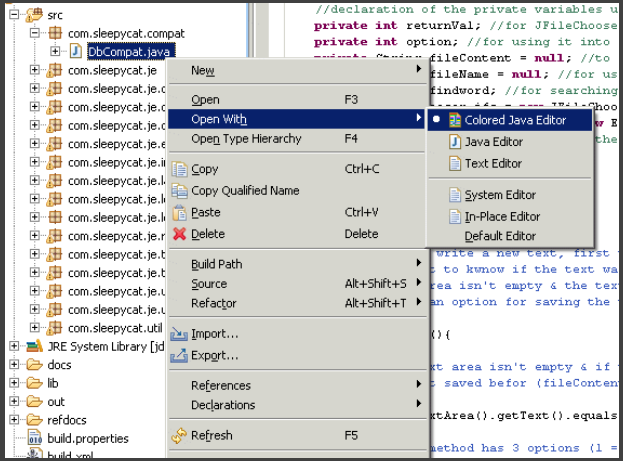
**What is different with preprocessors?**

- annotation of characters in plain text
- undisciplined annotations possible
- can lead to generation of syntactically invalid program variants

**What is different with physical separation?**

- features physically separated from each other
- dedicated components, services, plug-ins [Lecture 6]
- dedicated modules, folders, files [Lecture 7]

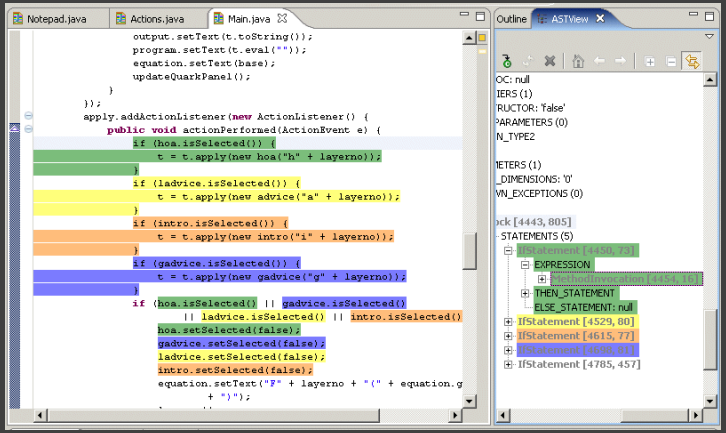
# Virtual Separation of Concerns – CIDE [CIDE]




## What is CIDE?

- stands for **C**olored **I**ntegrated **D**evelopment **E**nvironment
- colors used to mark features
- based on Eclipse 3.5 and FeatureIDE
- research prototype (last update in May 2012)
- special editors available for several languages: ANTLR, Bali, C, C++, C#, JavaScript, Featherweight Java, **Java 1.5**, gCIDE, Haskell, HTML, JavaCC, OSGi Manifest, Properties, Python, and XHTML

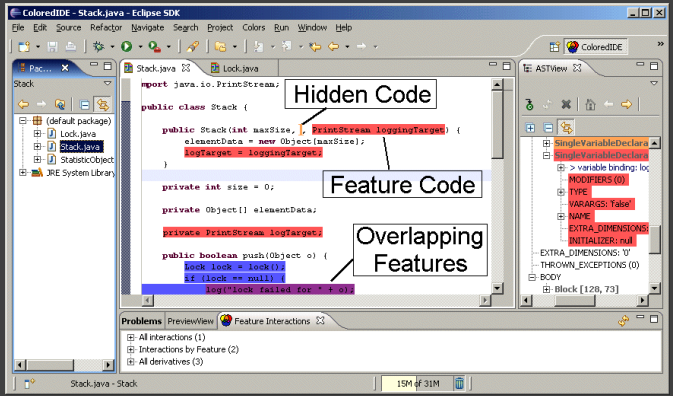
# Virtual Separation of Concerns – CIDE [CIDE]




## Why colors?

- colors replace preprocessor directives
- features relevant for development task are assigned a color
- code annotated to a feature by selection and context menu
- features visualized by background colors
- annotations stored externally (no changes outside the special editor feasible)

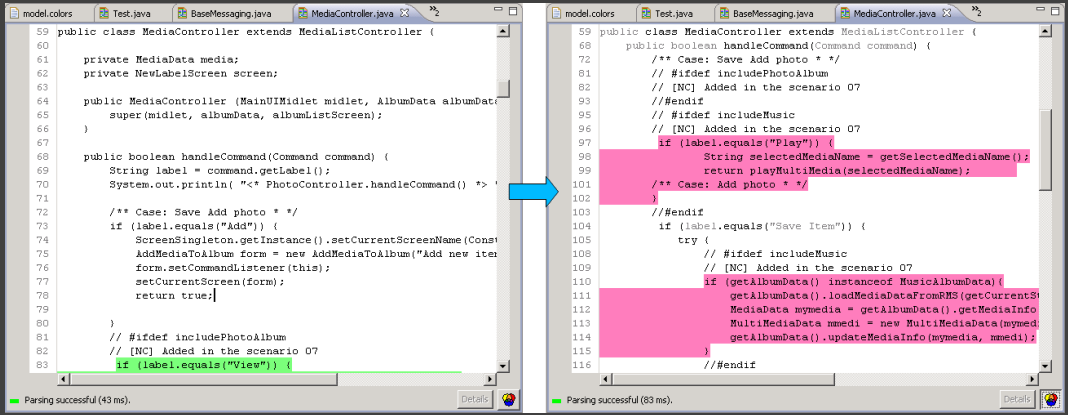
# Virtual Separation of Concerns – CIDE [CIDE]




**Why virtual separation?**

- source code is a view on the abstract syntax tree (AST)
- possible to hide irrelevant features
- possible to show overlapping features
- supporting development despite scattering and tangling
- no need to handle separators and logical connectors: “,”, “||”
- efficient detection of type errors [Lecture 10]

# Virtual Separation of Concerns – CIDE [CIDE]

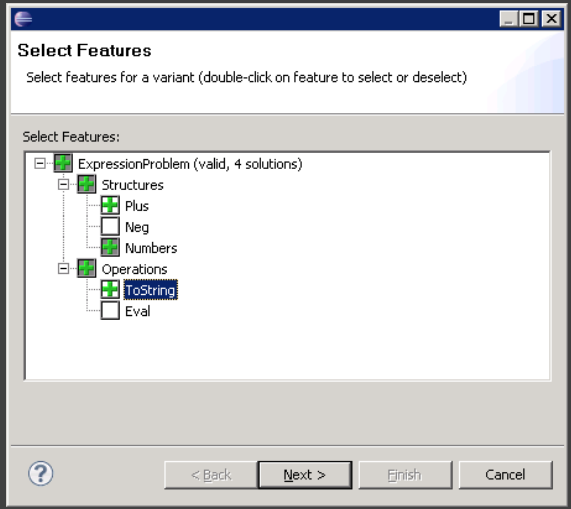



**view on a feature**: possible to only show a single feature – in its surrounded code

# Virtual Separation of Concerns – CIDE [CIDE]

**Why configuration?**

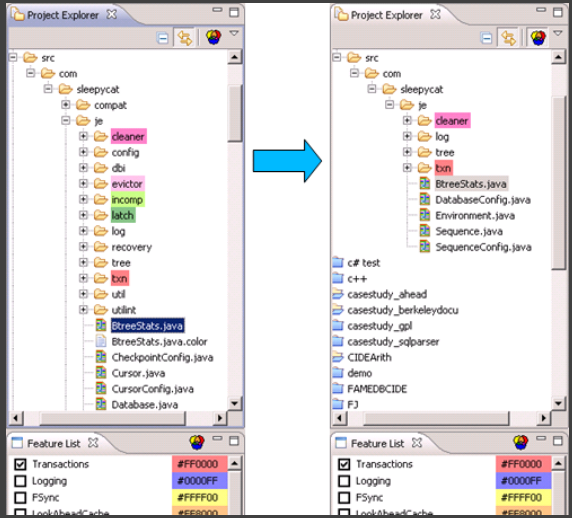
- features specified in FeatureIDE feature model
- configuration created in FeatureIDE configuration editor
- configuration used to generate and visualize variant
- ...



# Virtual Separation of Concerns – CIDE [CIDE]

**Why configuration?**

- ...
- **view on a variant**: variant visualized in source code and project explorer
- only necessary to press CIDE button in project explorer
- pressing it again returns to the view of the product line



# Feature Traceability – Summary

**Lessons Learned**

- preprocessor variability suffers from scattering and tangling
- feature traceability can be established with tool support (e.g., Feature Commander)
- virtual separation of concerns is an alternative to preprocessors (e.g., CIDE)

**Further Reading**

- Apel et al. 2013, Section 3.2.2 Feature Traceability
- Apel et al. 2013, Chapter 7 Advanced, Tool-Driven Variability Mechanisms

**Practice**

1. Why is it beneficial to have feature traceability even for single systems?
2. Using disciplined annotations, only optional nodes in the abstract syntax tree can be annotated (i.e., assigned to features); if we remove them, no syntax error occurs. What are examples of optional and non-optional (i.e., mandatory) types of nodes in Java?

# FAQ – 5. Conditional Compilation

## Lecture 5a

- What are problems of ad-hoc approaches for variability?
- How to implement features with build systems?
- How is it different from clone-and-own with build systems?
- How is the Linux kernel developed in terms of feature model, configuration, and feature mapping?
- What are KConfig, MenuConfig, KBuild (used for)?
- What are tristate features in Linux?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of conditional compilation with build systems?
- When (not) to implement features with build systems?

## Lecture 5b

- What are different levels of granularity for variability?
- Which granularity level supported by each techniques?
- What is a preprocessor and how does it work?
- What are examples for preprocessors and what are their differences?
- What is better in-place or out-of-place preprocessing?
- What is the problem with fine-grained variability or undisciplined annotations?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of cond. comp. with preprocessors?
- When (not) to implement features with preprocessors?

## Lecture 5c

- Which implementation techniques suffer from code scattering, code tangling, missing traceability?
- What is feature traceability? What is the feature traceability problem?
- What is the difference between problem and solution space?
- How can feature traceability be achieved for preprocessor-based product lines?
- Can feature traceability be automated for every potential feature of a domain?
- What is virtual separation of concerns? How is it different from preprocessors or physical separation?
- What is the principle of conditional compilation? How can it be implemented?