

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 1a. Introduction to Product Lines

- Handcrafting and Customization
- Mass Production
- Mass Customization
- Recap: The Software Life Cycle
- Features and Products of a Domain
- Software Product Line
- Product-Line Engineering
- Summary

### 1b. Challenges of Product Lines

- Software Clones
- Feature Traceability
- Automated Generation
- Combinatorial Explosion
- Feature Interactions
- Continuing Change and Growth
- Summary

### 1c. Course Organization

- What You Should Know
- What You Will Learn
- What You Might Need
- Credit for the Slides
- Summary
- FAQ

# 1. Introduction – Handout

Software Product Lines | Thomas Thüm, Timo Kehrer, Elias Kuiter | April 19, 2023

# **1. Introduction**

## **1a. Introduction to Product Lines**

Handcrafting and Customization

Mass Production

Mass Customization

Recap: The Software Life Cycle

Features and Products of a Domain

Software Product Line

Product-Line Engineering

Summary

## **1b. Challenges of Product Lines**

## **1c. Course Organization**

# What do these examples have in common?



## Customization

- aka. handcrafting
- labor-intensive production
- highly individual goods

# Customization of Elevators



two buttons



one button



keyhole



floor display

# Customization of Elevators



no button to close door



two keyholes



keycard



double tap for undo

# Mass Production

## Mass Production

[Apel et al. 2013, pp. 3–4]

- consequence of industrialization
- goods are produced from standardized parts
- improved productivity wrt. handcrafting
- reduced costs, improved quality
- but: (almost) no individualism

## Principle: One Size Fits All

- t-shirts: XS, S, M, L, XL, XXL
- swiss-army knife



## Mass Production for Software?

[Apel et al. 2013, p. 7]

"The idea is to provide software that satisfies the needs of most customers, which leads almost automatically to the situation, in which customers miss desired functionality and are overwhelmed with functionality they do not need actually (just think of any contemporary office or graphics program). It is often this generality that makes software complex, slow, and buggy."

# About Every Second Car is Unique



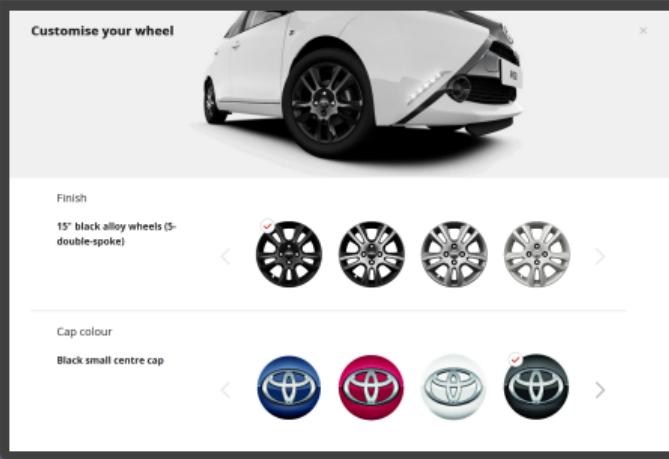
# Mass Customization

## Mass Customization

[Apel et al. 2013, p. 4]

- = mass production + customization
- customized, individual goods at costs similar to mass production

## Car Configuration



## Car Production



## Other Domains

bikes, computers, electronics, tools, medicine, clothing, food, financial services, . . . , software?

# Mass Customization for Software?

## Mass Customization for Software?

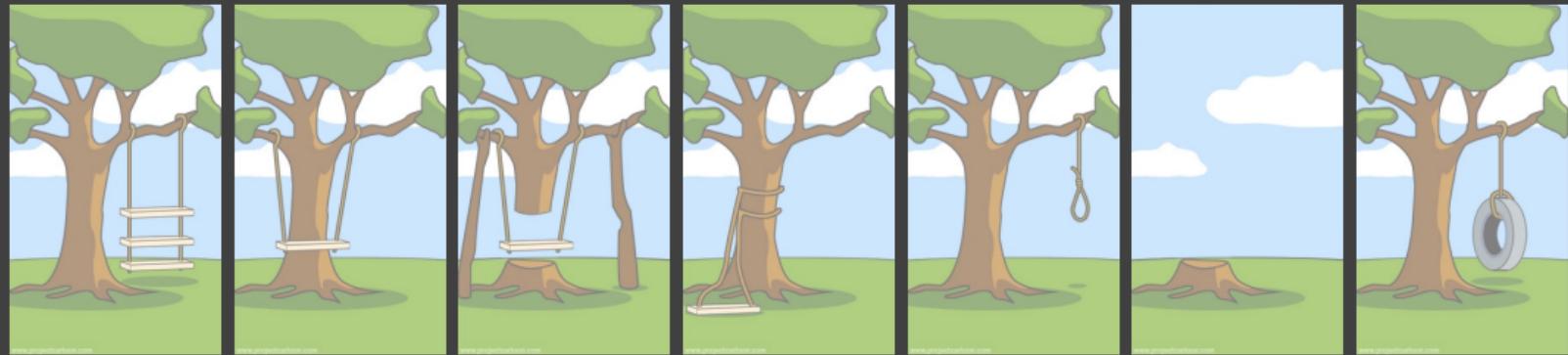
- customization: individual software developed using Waterfall model or Scrum
- mass production: standard software developed once for millions or billions of users (e.g., Whatsapp messenger)
- mass customization: software product lines

## Why Software Product Lines?

- resource limitations: memory, performance, energy
- different hardware
- different laws
- goal: avoid expensive customization
- how is software developed?



# The Project Cartoon



how the  
customer  
explained it

how the  
project leader  
understood it

how the  
analyst  
designed it

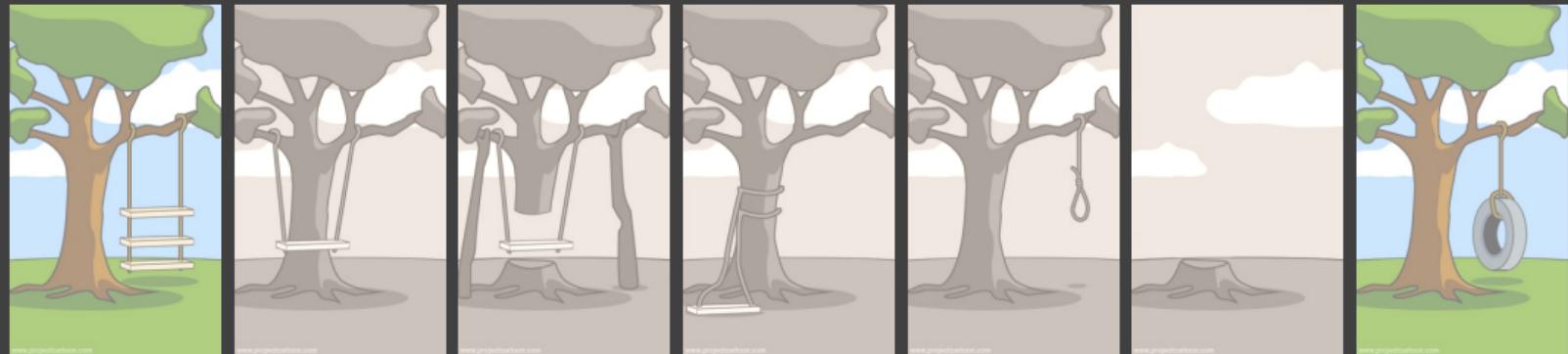
how the  
programmer  
implemented it

what the beta  
testers  
received

how it was  
supported

what the  
customer really  
needed

# The Project Cartoon



Requirements

how the  
project leader  
understood it

how the  
analyst  
designed it

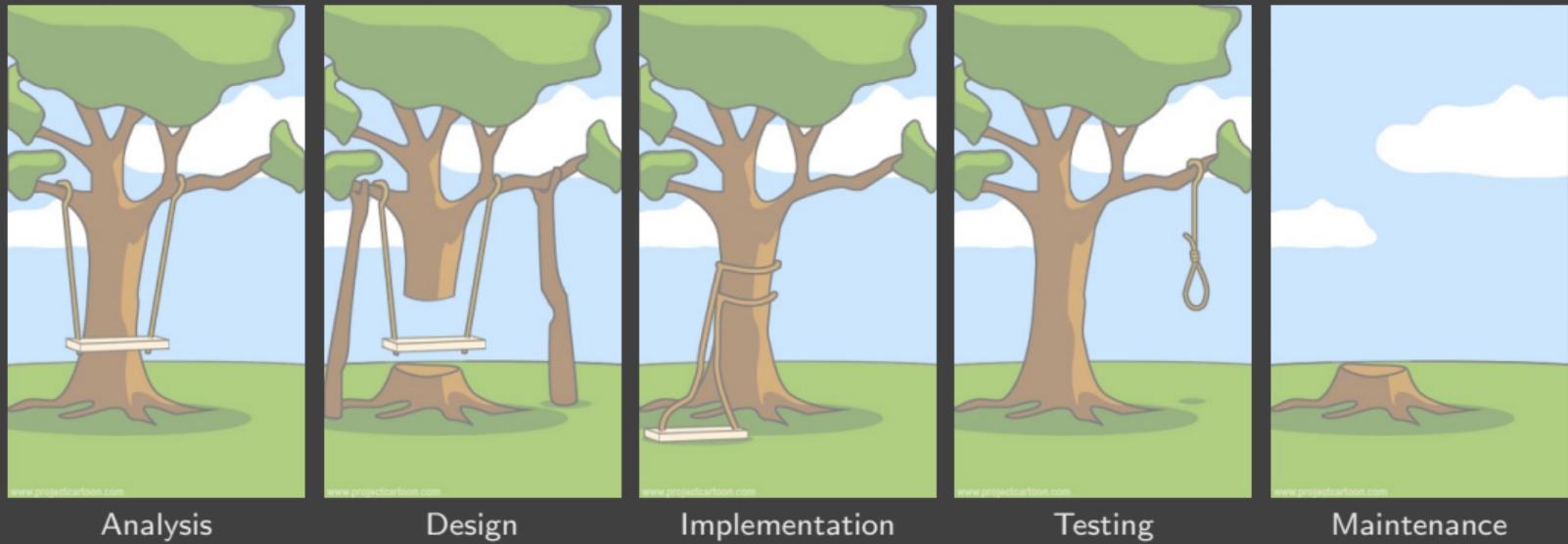
how the  
programmer  
implemented it

what the beta  
testers  
received

how it was  
supported

Product

# Recap: The Software Life Cycle

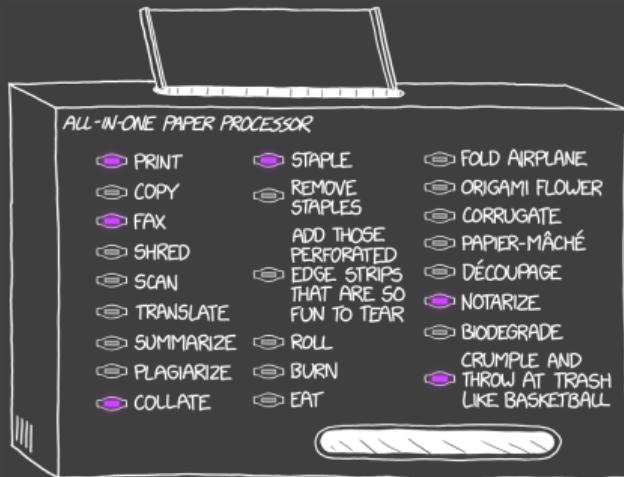


# What is a Feature?

## Feature

[Apel et al. 2013, p. 18]

"A **feature** is a characteristic or end-user-visible behavior of a software system."

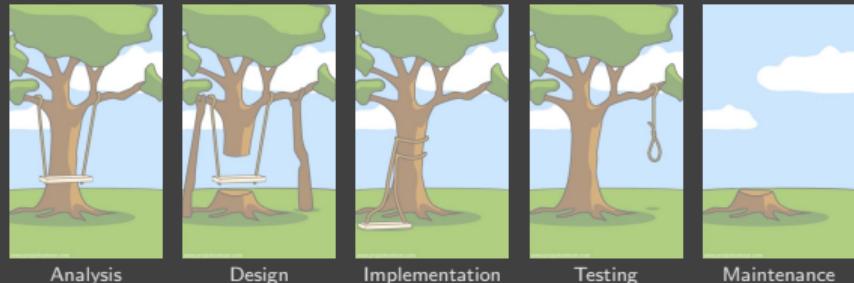


## Feature in a Product Line

[Apel et al. 2013, p. 18]

"Features are used in product-line engineering

- to specify and communicate commonalities and differences of the products between stakeholders and
- to guide structure, reuse, and variation across all phases of the software life cycle."



# What is a Product?

## Product

[Apel et al. 2013, p. 19]

"A **product of a product line** is specified by a valid feature selection (a subset of the features of the product line). A feature selection is **valid** if and only if it fulfills all feature dependencies."

## Note on Terminology

- in this course:  
product = product variant = variant
- software product: a product consisting only of software
- software is more than a program: requirements, models, source code, tests, documentation
- this course focuses on source code

## Product Map for Eclipse (excerpt)

	 Java	 Enterprise Java/Web	 C/C++	 Committees
C/C++ Development Tools				✓
Data Tools Platform			✓	
Git integration for Eclipse	✓	✓	✓	✓
Java Development Tools	✓	✓		✓
Java EE Developer Tools			✓	
JavaScript Development Tools				
Maven Integration for Eclipse	✓	✓		✓

# What is a Domain?

## Domain

[Apel et al. 2013, p. 19]

"A **domain** is an area of knowledge that:

- is scoped to maximize the satisfaction of the requirements of its stakeholders,
- includes a set of concepts and terminology understood by practitioners in that area,
- and includes the knowledge of how to build software systems (or parts of software systems) in that area."

## Features of a Domain

- a feature is a domain abstraction
- identification of features in a domain requires domain expertise
- later: select features for a product line?



# Software Product Line

## Software Product Line

[Northrop et al. 2012, p. 5]

"A **software product line** is

- a set of software-intensive systems

aka. products or variants

- that share a common, managed set of features

common set, but not all products have all features in common

- satisfying the specific needs of a particular market segment or mission

aka. domain

- and that are developed from a common set of core assets in a prescribed way."

aka. planned, structured reuse

[Software Engineering Institute, Carnegie Mellon University]

# Product-Line Engineering

## Product-Line Engineering

[Pohl et al. 2005, p. 14]

“Software product-line engineering is a paradigm to develop software applications (software-intensive systems and software products) using software platforms and mass customization.”

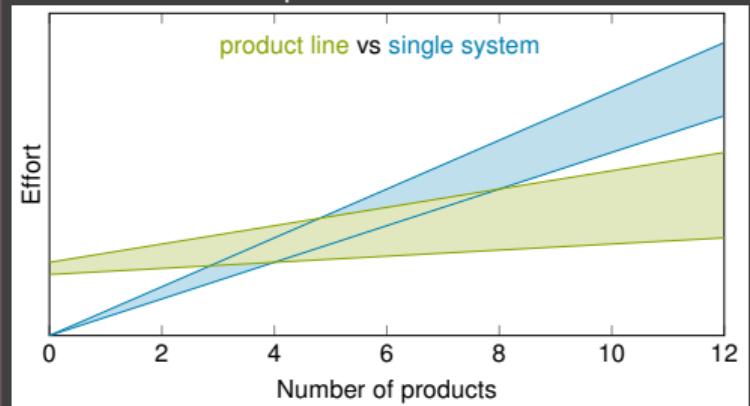
## Promises of Product Lines

[Apel et al. 2013, pp. 9–10]

- tailor-made
- reduced costs
- improved quality
- reduced time-to-market

## Idea of Product-Line Engineering

Reduce effort per product by means of an up-front investment for the product line:



## Single-System Engineering

classical software development that is not considered as product-line engineering

# Introduction to Product Lines – Summary

## Lessons Learned

- mass customization  
= mass production + customization
- features, products, domains
- software product lines
- product-line engineering

## Further Reading

- Apel et al. 2013, Chapter 1, pp. 3–15 — introduction close to this lecture, further examples
- Northrop et al. 2012, pp. 2–8 — what is not a product line?

## Practice

- What other examples of product lines do you know?
- Exemplify the differences between feature, product, domain, and product line for these examples.
- Are these product lines related to software?

# **1. Introduction**

## **1a. Introduction to Product Lines**

## **1b. Challenges of Product Lines**

Software Clones

Feature Traceability

Automated Generation

Combinatorial Explosion

Feature Interactions

Continuing Change and Growth

Summary

## **1c. Course Organization**

# Software Clones

## Software Clone

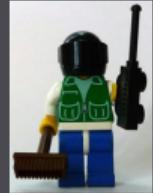
[Rattan et al. 2013, p. 1166]

- = result of copying and pasting existing fragments of the software
- code clones = copied code fragments
- replicates need to be altered consistently
- for example: bugs need to be fixed in all replicated fragments
- in practice: a common source for inconsistencies and bugs

## Cloning Parts of Software



## Cloning Whole Products (Clone-and-Own)



# Feature Traceability

## Feature Traceability

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

## Intuition on Feature Traceability



find feature



in product

## Feature Traceability with Colored Source Code

A screenshot of a Java IDE interface titled "FeatureDE - FeatureDE - Elevator-Antenna-v1.4/src/de/ovgu/featureide/examples/elevator/core/controller/Request.java...". The code editor displays a Java file named "Request.java". The code is color-coded to show feature traceability:

```
// @return (floor != other.floor)
// #endif
public static class RequestComparator implements Comparator<Request> {
    // #if ShortestPath
    protected ControlUnit controller;
    public RequestComparator(ControlUnit controller) {
        this.controller = controller;
    }
    // #endif
    @Override
    public int compare(Request o1, Request o2) {
        // #if DirectedGraph
        return compareDirectional(o1, o2);
        // #else
        // #if FIFO
        // @return (int) Math.signum(o1.timestamp - o2.timestamp)
        // #endif
        // #if ShortestPath
        // @int diff0 = Math.abs(o1.floor - controller.getControlUnit().getFloor());
        // @int diff1 = Math.abs(o2.floor - controller.getControlUnit().getFloor());
        // @return diff0 - diff1;
        // #endif
        // #endif
    }
}
```

The code editor has a toolbar at the top and a sidebar on the right containing various icons and a search bar. The status bar at the bottom shows "Writable", "Smart Insert", and "1:1".

# Automated Generation

Features



Products



# Automated Generation

## Product Line with Features



### Goal

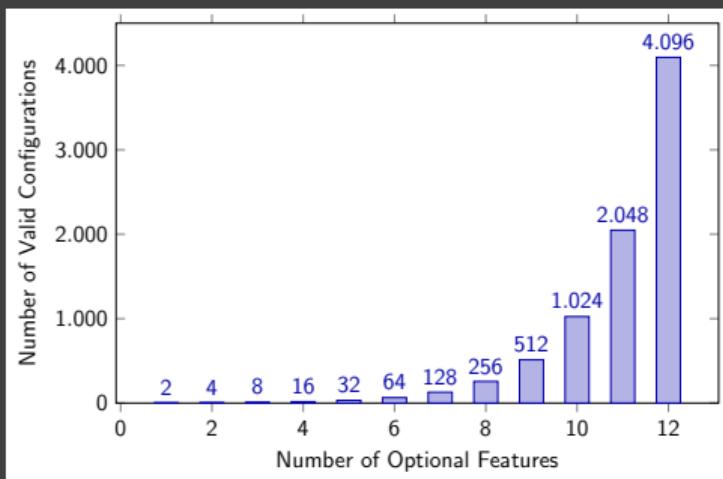
- automatic generation of products
- based on a (descriptive) selection of features

### Challenges

- how to map features to source code?
- how to combine source code of multiple features?
- how to define valid combinations of features?

# Combinatorial Explosion

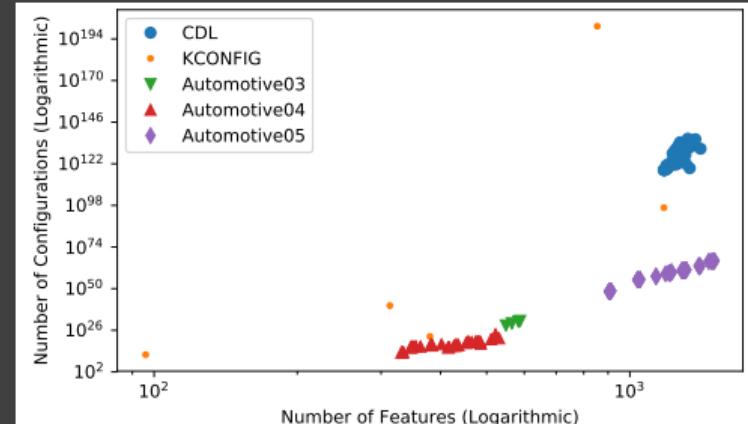
## Combinatorial Explosion



- assumption: all combinations of features are valid
- 33 features: a unique combination for every human
- 320 features: more combinations than atoms in the universe

## Industrial Configuration Spaces

[Sundermann et al. 2020]

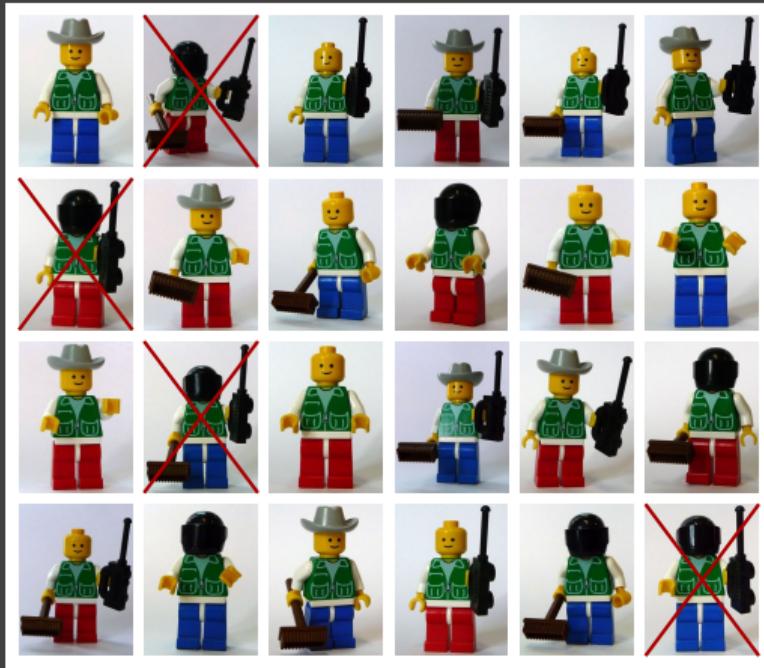


- in practice: not all combinations of features valid
- many industrial product lines too large to specify all valid combinations separately
- largest automotive product line has about  $1.7 \cdot 10^{1534}$  products

# Combinatorial Explosion



# Feature Interactions



## Example Interaction



phone  cannot be used with helmet



## Challenges

- interaction typically unknown in advance
- interactions occur in some but not all combinations
- challenge for quality assurance

# Feature Interactions

## Invalid Car Configurations

Configuration Assistant.

› Show instructions

Your most recent action requires your configuration to be adjusted.

Your choice	Price
+ Enhanced Bluetooth telephone with USB & Voice Control	+ £ 350.00
Adding	
+ BMW Navigation	£ 0.00
Removing	
- Enhanced Bluetooth with wireless charging	- £ 395.00
- Navigation system Professional	£ 0.00
- WiFi hotspot preparation	£ 0.00
- Media package - Professional	- £ 900.00
- Online Entertainment	£ 0.00
- Microsoft Office 365	- £ 150.00

# Continuing Change and Growth

## Lehman's Laws of Software Evolution (excerpt)

[Lehman et al. 1997]

- Continuing Change: systems must be continually adapted to stay satisfactory
- Increasing Complexity: complexity increases during evolution unless work is done to maintain or reduce it
- Continuing Growth: functionality must be continually increased to maintain user satisfaction
- Declining Quality: quality will decline unless rigorously maintained and adapted to operational environment changes

## Essence of the Laws

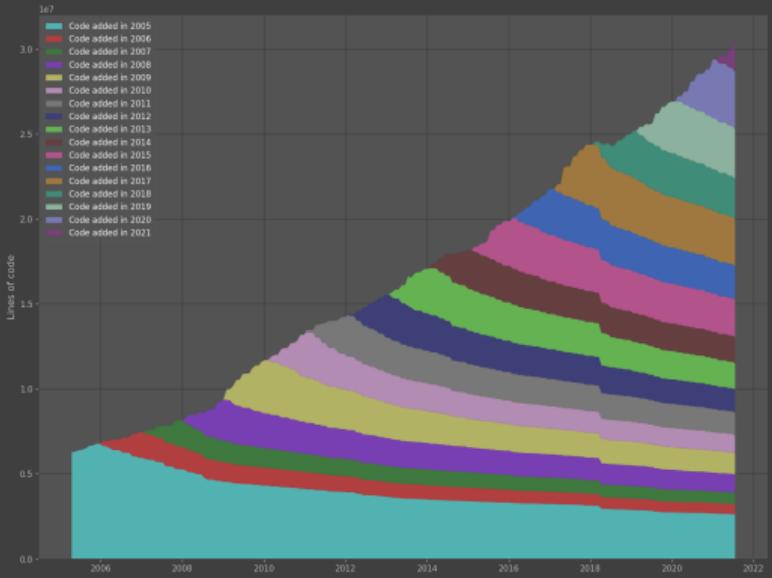
- software that is used will be modified
- when modified, its complexity will increase (unless one does actively work against it)

## Consequences for Product Lines

- number of features and size of implementation increases over time
- discussed challenges increase over time
  - more software clones
  - harder to trace features
  - automated generation more urgent
  - increasing combinatorial explosion
  - more feature interactions

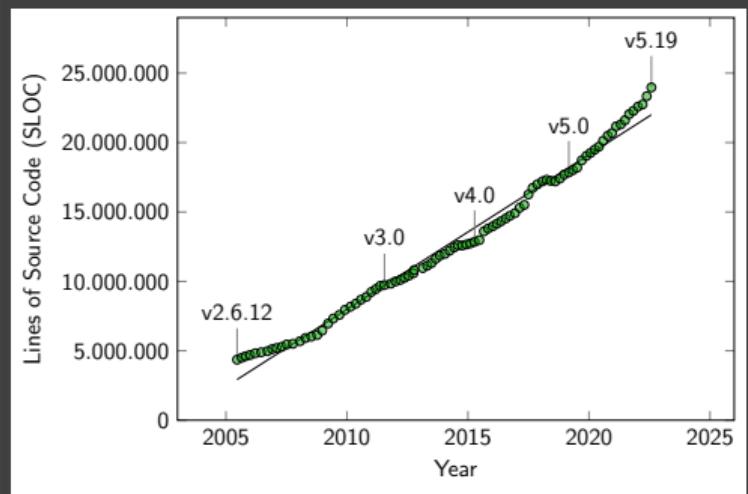
# Evolution of the Linux Kernel

- about 60,000 commits per year
- in peak weeks: new commit every 5 minutes
- in average weeks: every 9 minutes



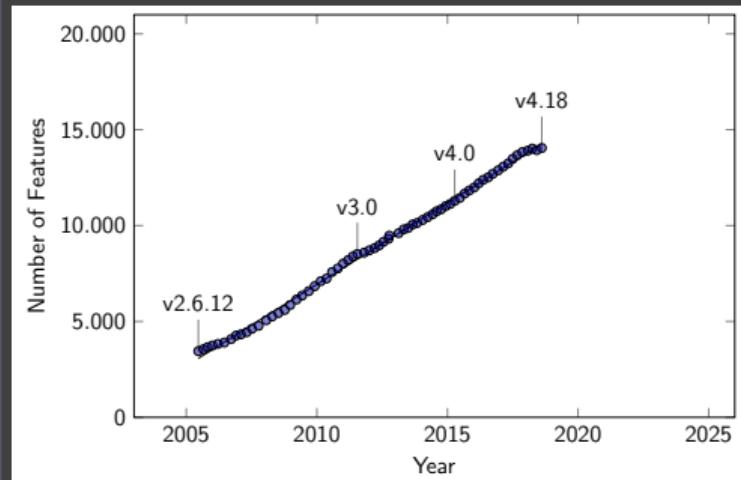
# Evolution of the Linux Kernel

Size of the Code Base



- from 4 to 24 millions in 17 years
- about one million LOC added every year
- about 3,000 LOC per day

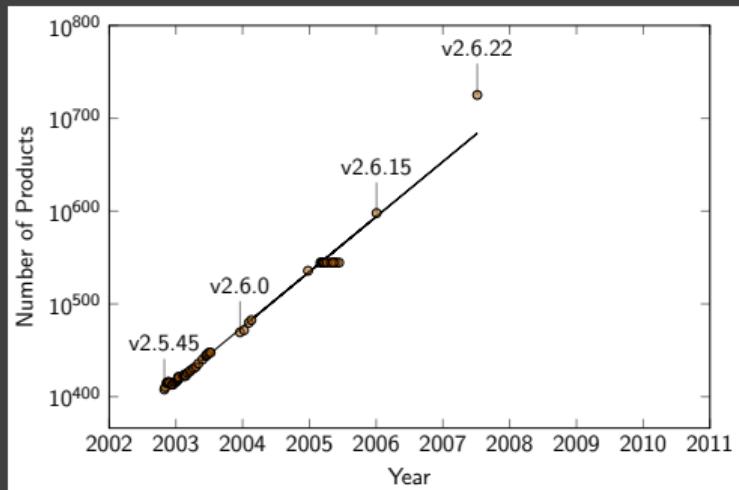
Number of Features



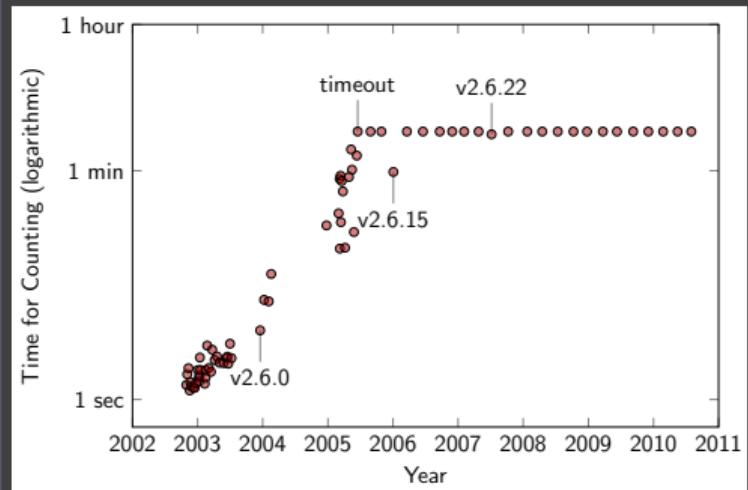
- about 800 new features every year
- about 15 new features every week
- in 2018 four times more features than in 2005

# Evolution of the Linux Kernel

Number of Products



Time to Count Products



- number of products grows by factor 100.000 each month
- the current kernel is likely to have more than  $10^{1500}$  products

- most kernel versions before 2006 can be computed within 1 minute
- most kernel versions after 2006 cannot be computed within 1 hour

# Challenges of Product Lines – Summary

## Lessons Learned

- why are product lines challenging?
- selected challenges:
  1. software clones
  2. feature traceability
  3. automated generation
  4. combinatorial explosion
  5. feature interactions
  6. continuous growth

## Further Reading

see later lectures

## Practice

- Form groups of 2–3 students
- Explain 2–3 of the six challenges to your colleagues
- Can you find own examples for these challenges?

# **1. Introduction**

## **1a. Introduction to Product Lines**

## **1b. Challenges of Product Lines**

## **1c. Course Organization**

What You Should Know

What You Will Learn

What You Might Need

Credit for the Slides

Summary

FAQ

# What You Should Know

## Fundamentals of Software Engineering

- development processes
- object-oriented programming
- design patterns
- UML class diagrams
- modularity

## Fundamentals of Theoretical Computer Science

- set theory
- propositional logic
- complexity theory

## Exercise

solid programming skills in Java

# What You Will Learn

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

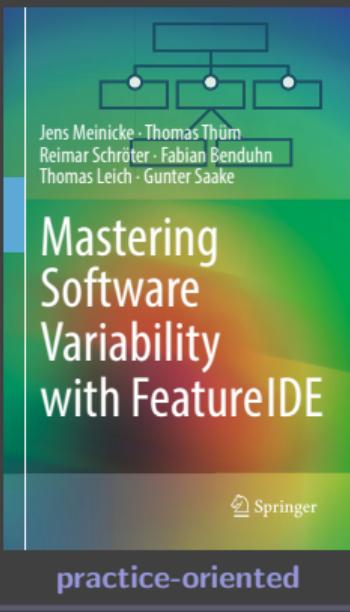
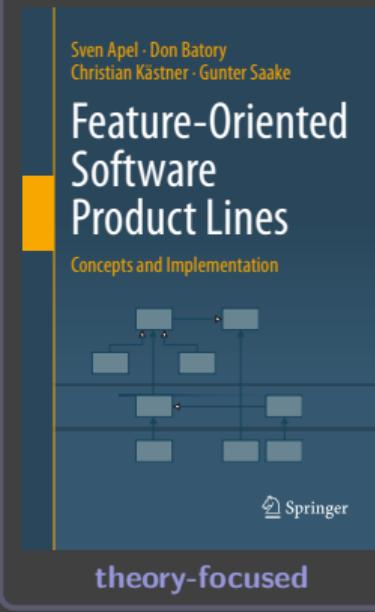
4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

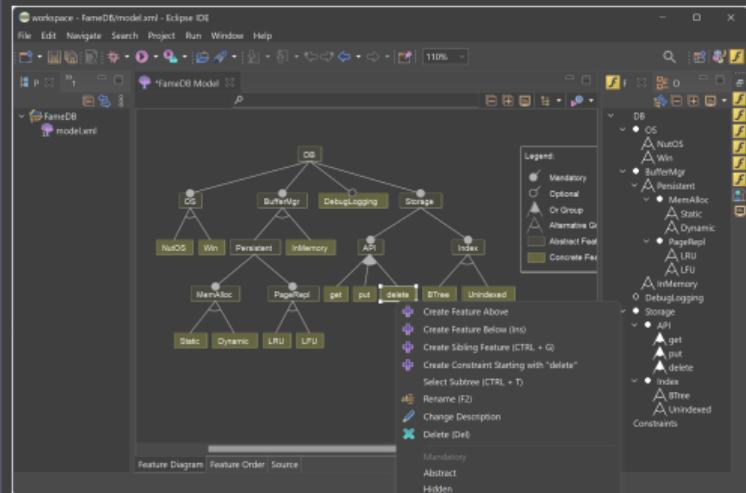
9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

# What You Might Need

## Recommended Literature for Lecture & Exercise



## Recommended Tool Support for the Exercise



# Credit for the Slides



Thomas Thüm



Professor at TU Braunschweig  
software engineering  
FeatureIDE team leader

Timo Kehrer



Professor at University of Bern  
software engineering

Elias Kuiter



PhD student in Magdeburg  
feature-model analysis  
FeatureIDE core developer

# Course Organization – Summary

## Lessons Learned

- focus: how to implement features
- focus: how to model valid combinations
- focus: how to do quality assurance
- course organization

## Further Reading

- Apel et al. 2013 — best book for this lecture
- Meinicke et al. 2017 — more practical guide on tool support

## Practice

- Ask questions on the course organization!
- Form teams for the practical tasks

# FAQ – 1. Introduction

## Lecture 1a

- What is a software product line, feature, product/variant, domain?
- What is customization, handcrafting, mass production, mass customization?
- What are example for each of those?
- What is the one-size-fits-all or swiss-army-knife principle?
- What is the difference between product-line engineering and single-system engineering?
- What are advantages of product-line engineering?

## Lecture 1b

- What are software clones, feature traceability, automated generation, combinatorial explosion, feature interactions, and continuous growth?
- Why are those challenging when developing product lines?
- What are examples for these six fundamental challenges?
- How do those challenges interact with each other?
- How complex is the Linux kernel?
- At which pace is the Linux kernel developed?

## Lecture 1c

- What you should know?
- What you will learn?
- What you might need?
- Who are the authors of this course?
- How is this course organized?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## 2a. Configuration of Runtime Variability

- Variability and Binding Time
- Examples for Configuration Options
  - Command-Line Parameters
  - Preference Dialogs
  - Configuration Files
- Configuration Options and Runtime Variability
- Running Example: Graph Library
- Valid Combinations of Options
- Summary

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## 2b. Realization of Runtime Variability

- Recap and Motivating Example
- Global Variables
- Method Parameters
- Problems of Runtime Variability
  - Reconfiguration at Runtime
  - Code Scattering
  - Code Tangling
  - Code Replication
- Summary

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

## 2c. Design Patterns for Variability

- Recap on Object Orientation
- Recap on Design Patterns
  - Template Method Pattern
  - Abstract Factory Pattern
  - Decorator Pattern
- Trade-Offs and Limitations
  - Template Method and Abstract Factory
  - Diamond Problem and Decorator
- Summary
- FAQ

# 2. Runtime Variability and Design Patterns – Handout

Software Product Lines | Timo Kehrer, Thomas Thüm, Elias Kuiter | April 21, 2023



## **2. Runtime Variability and Design Patterns**

### **2a. Configuration of Runtime Variability**

Variability and Binding Time

Examples for Configuration Options

- Command-Line Parameters

- Preference Dialogs

- Configuration Files

Configuration Options and Runtime Variability

Running Example: Graph Library

Valid Combinations of Options

Summary

### **2b. Realization of Runtime Variability**

### **2c. Design Patterns for Variability**

# Recap: Software Product Lines

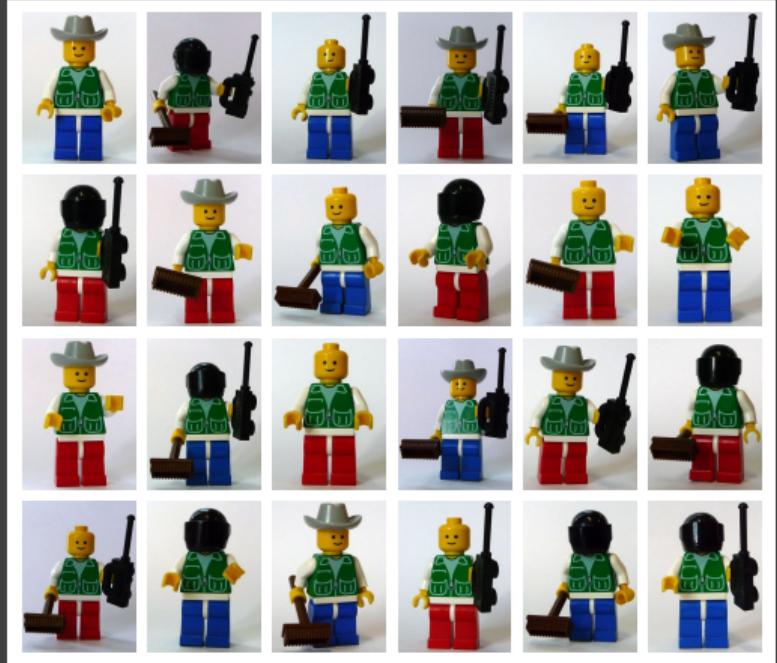
## Software Product Line

[Northrop et al. 2012, p. 5]

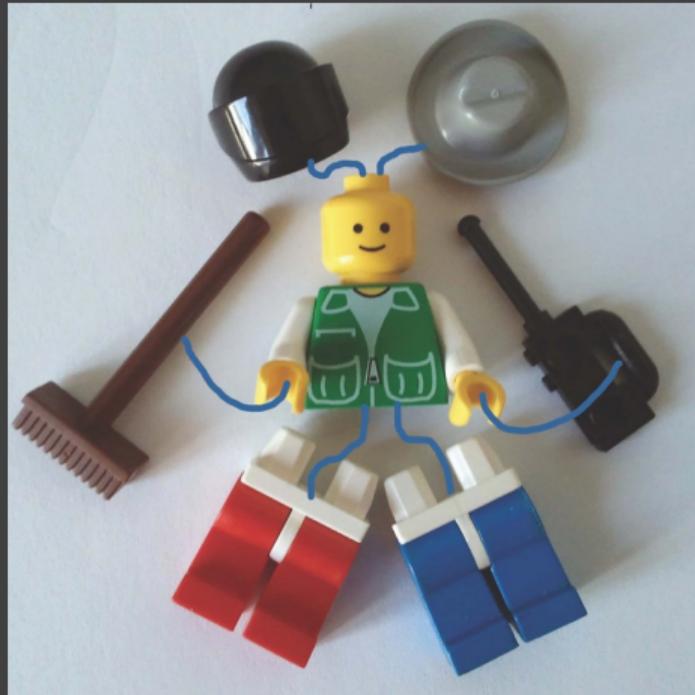
"A **software product line** is

- a set of software-intensive systems (aka. products or variants)
- that share a common, managed set of features
- satisfying the specific needs of a particular market segment or mission (aka. domain)
- and that are developed from a common set of core assets in a prescribed way."

[Software Engineering Institute, Carnegie Mellon University]



# How to Implement Software Product Lines?



## Key Issues

- Systematic reuse of implementation artifacts
- Explicit handling of variability

## Variability

[Apel et al. 2013, p. 48]

**“Variability** is the ability to derive different products from a common set of artifacts.”

## Variability-Intensive System

Any software product line is a variability-intensive system.

# Variability and Binding Times

## Binding Time

[Apel et al. 2013, p. 48]

- Variability offers choices
- Derivation of a product requires to make decisions (aka. binding)
- Decisions may be bound at different binding times

## When? By whom? How?

Lecture 2a: **when** and **by whom**

Lecture 2b: **how**



# Example: Command-Line Parameters

```
C:\>dir /?
Displays a list of files and subdirectories in a directory.

DIR [drive:][path][filename] [/A[[:attributes]]] [/B] [/C] [/D] [/L] [/N]
  [/O[[:sortorder]]] [/P] [/Q] [/R] [/S] [/T[[:timefield]]] [/W] [/X] [/4]

[drive:][path][filename]
      Specifies drive, directory, and/or files to list.

/A      Displays files with specified attributes.
attributes  D Directories          R Read-only files
            H Hidden files        A Files ready for archiving
            S System files        I Not content indexed files
            L Reparse Points     O Offline files
            - Prefix meaning not

/B      Uses bare format (no heading information or summary).
/C      Display the thousand separator in file sizes. This is the
       default. Use /-C to disable display of separator.
/D      Same as wide but files are list sorted by column.
/L      Uses lowercase.
/N      New long list format where filenames are on the far right.
/O      List by files in sorted order.
sortorder   N By name (alphabetic)  S By size (smallest first)
            E By extension (alphabetic) D By date/time (oldest first)
            G Group directories first - Prefix to reverse order

/P      Pauses after each screenful of information.
/Q      Display the owner of the file.
/R      Display alternate data streams of the file.
/S      Displays files in specified directory and all subdirectories.

Press any key to continue . . .
```

Description of configuration options

```
C:\>dir Windows /ah
Volume in drive C is Windows
Volume Serial Number is 1260-4B56

Directory of C:\Windows

12/07/2019  04:54 PM    <DIR>        BitLockerDiscoveryVolumeContents
12/07/2019  11:14 AM    <DIR>        ELAMBKUP
06/21/2021  06:11 AM    <DIR>        Installer
12/07/2019  11:14 AM    <DIR>        LanguageOverlayCache
12/22/2019  08:38 PM           38,224 MFGSTAT.zip
02/25/2020  12:51 AM           128,916 modules.log
12/07/2019  11:09 AM           670 WindowsShell.Manifest
                                         3 File(s)    167,810 bytes
                                         4 Dir(s)   1,075,515,850,752 bytes free

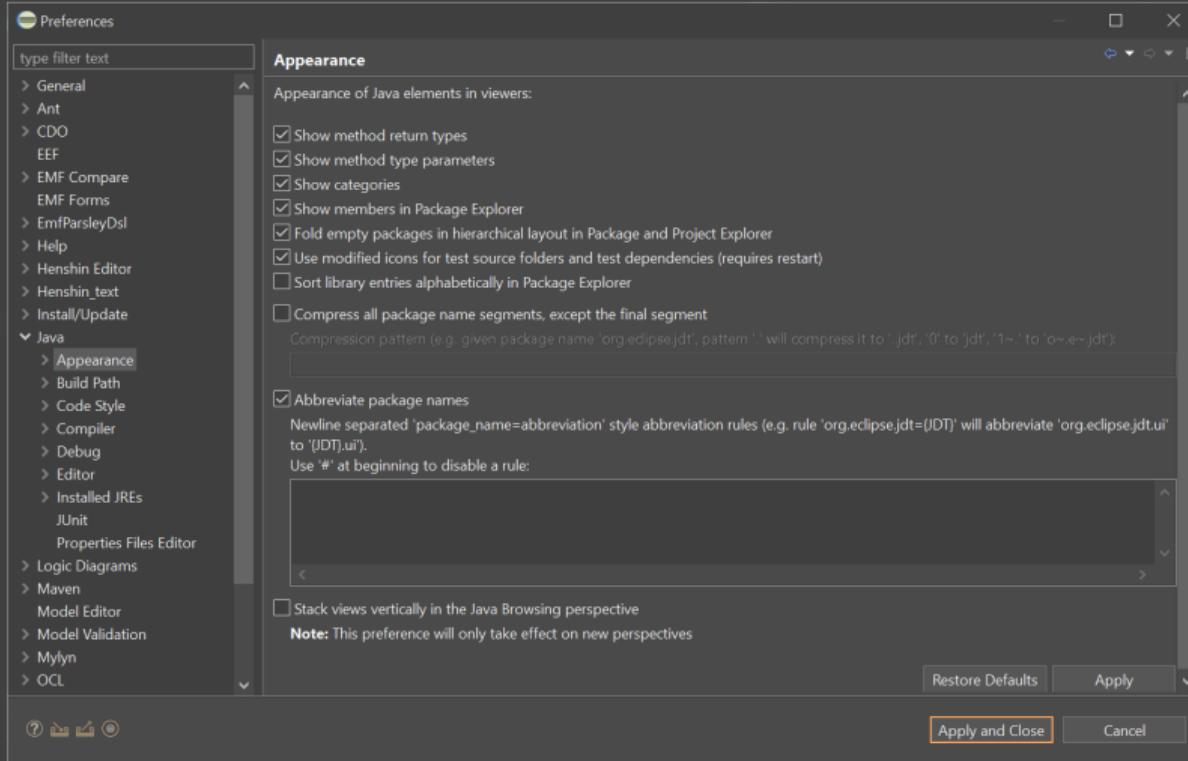
C:\>dir Windows /ah /-c
Volume in drive C is Windows
Volume Serial Number is 1260-4B56

Directory of C:\Windows

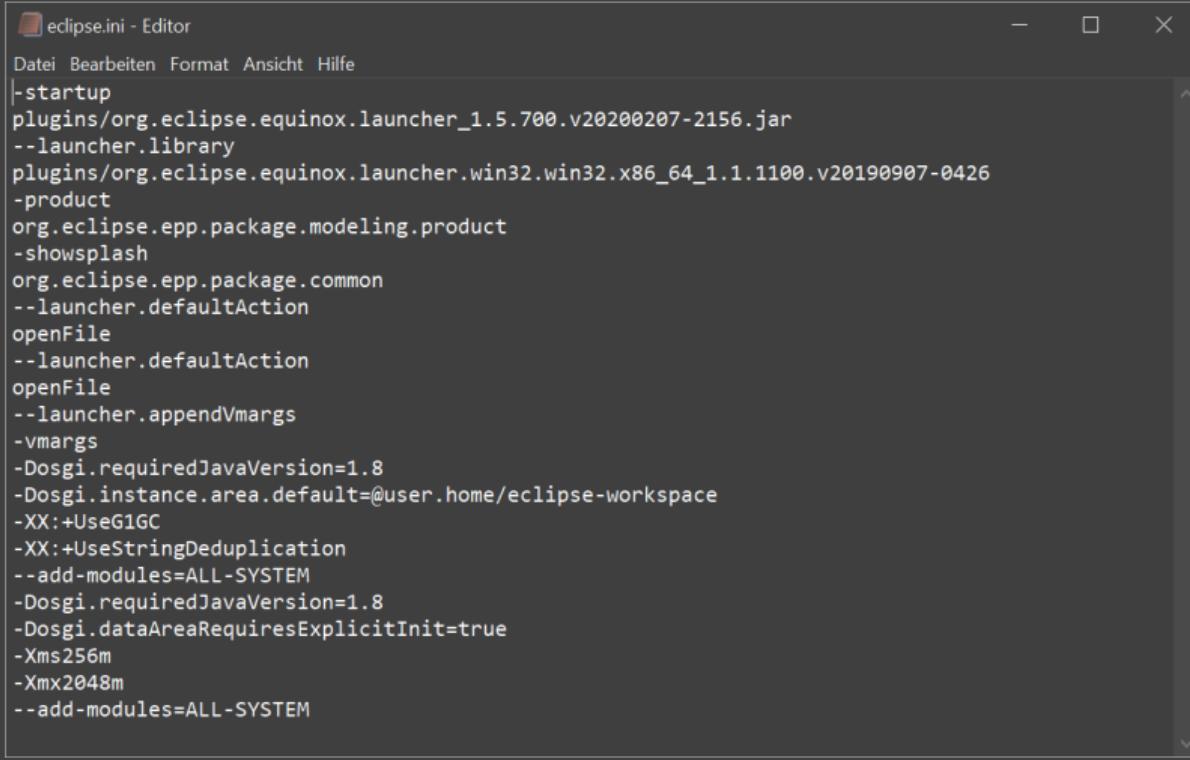
12/07/2019  04:54 PM    <DIR>        BitLockerDiscoveryVolumeContents
12/07/2019  11:14 AM    <DIR>        ELAMBKUP
06/21/2021  06:11 AM    <DIR>        Installer
12/07/2019  11:14 AM    <DIR>        LanguageOverlayCache
12/22/2019  08:38 PM           38224 MFGSTAT.zip
02/25/2020  12:51 AM           128916 modules.log
12/07/2019  11:09 AM           670 WindowsShell.Manifest
                                         3 File(s)    167810 bytes
```

Two different instances? separator , in file sizes

# Example: Preference Dialogs



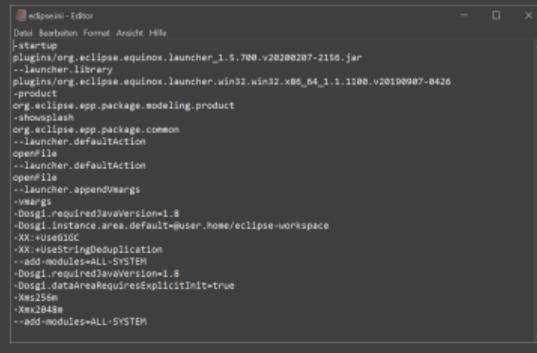
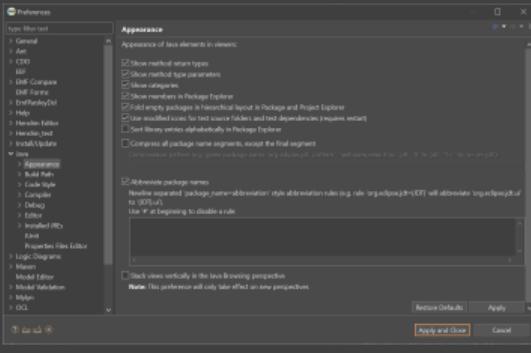
# Example: Configuration Files



```
eclipse.ini - Editor
Datei Bearbeiten Format Ansicht Hilfe
|-startup
plugins/org.eclipse.equinox.launcher_1.5.700.v20200207-2156.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.1100.v20190907-0426
-product
org.eclipse.epp.package.modeling.product
-showsplash
org.eclipse.epp.package.common
--launcher.defaultAction
openFile
--launcher.defaultAction
openFile
--launcher.appendVmargs
-vmargs
-Dosgi.requiredJavaVersion=1.8
-Dosgi.instance.area.default=@user.home/eclipse-workspace
-XX:+UseG1GC
-XX:+UseStringDeduplication
--add-modules=ALL-SYSTEM
-Dosgi.requiredJavaVersion=1.8
-Dosgi.dataAreaRequiresExplicitInit=true
-Xms256m
-Xmx2048m
--add-modules=ALL-SYSTEM
```

# What do these examples have in common?

```
C:\>dir /?  
Displays a list of files and subdirectories in a directory.  
DIR [drive:][:path] [/filename] [/A[lt][ttribute]] [/B[y] [/C[ontact]] [/D[ate]] [/E[xtension]]  
[/I[gnore]] [/O[rdernumber]] [/P[age]] [/R[esume]] [/S[earch]] [/T[imefield]] [/W[ide]] [/X[clude]] [/4[rows]]  
[drive:][:path]\filename  
    Specifies drive, directory, and/or files to list.  
  
/A  
    Displays files with specified attributes.  
attributes  
    D Directories  
    H Hidden Files  
    S System Files  
    L Registry Points  
    - Prefix meaning not  
    R Real-only files  
    A Files ready for archiving  
    C Content indexed files  
    O Offline files  
    N New files  
    P Prefix meaning not  
    M Modified files  
    U Unchanged files  
    W Working files  
    /B  
    Files have format (no heading information or summary).  
    Displays files in brief mode, one file per line. This is the default. Use /C to provide display of separator.  
    /C  
    Same as wide but files are list sorted by column.  
    /L  
    Uses lowercase.  
    /R  
    New long list format where filenames are on the far right.  
    /W  
    Lists files in wide mode, sorted by date.  
    /S  
    Sorts files after each screenful of information.  
    /Q  
    Displays files in quiet mode.  
    /B  
    Displays alternate data streams of the file.  
    /S  
    Displays files in specified directory and all subdirectories.  
Press any key to continue . . .
```



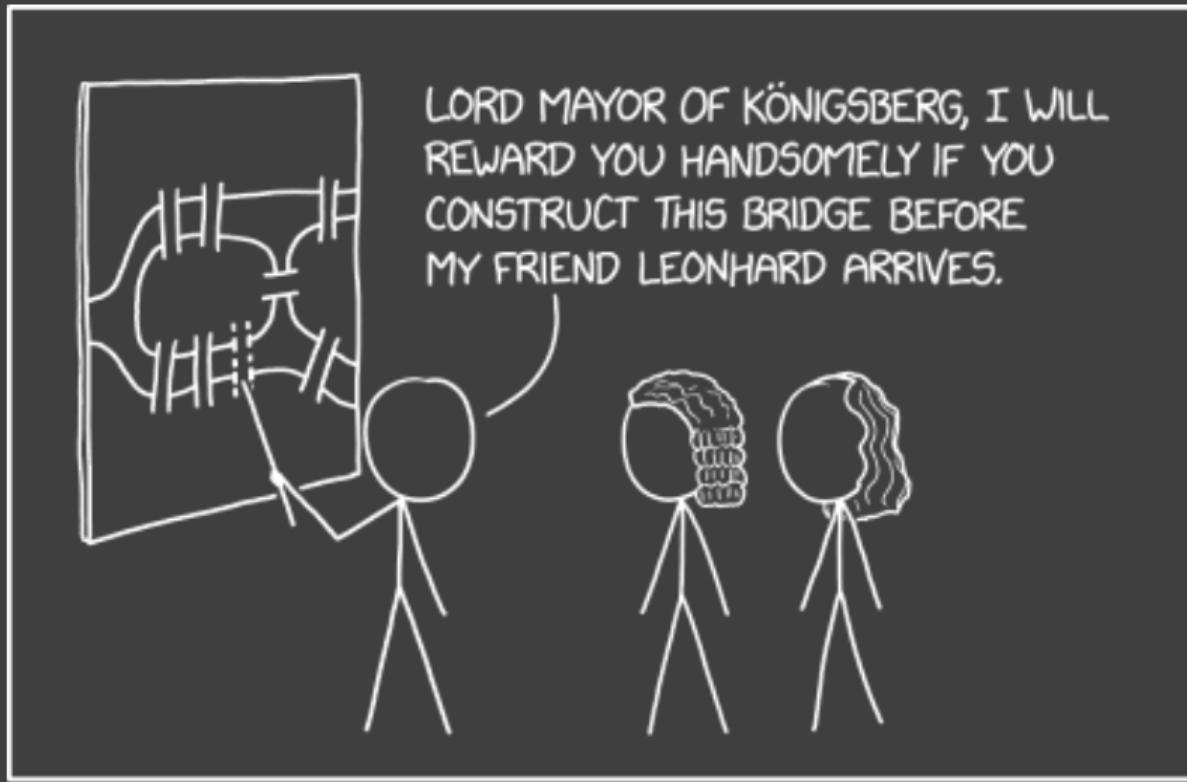
## Configuration Options

- Behavior of a program is determined by configuration options being interpreted at runtime
- Choices offered by variability are decided at runtime
- Configuration may happen interactively (command-line parameters, preference dialogs) or non-interactively (configuration files)

## Runtime Variability

[Apel et al. 2013, p. 49]

Runtime variability is decided after compilation when the program is started (aka. load-time variability) or during program execution.



I TRIED TO USE A TIME MACHINE TO CHEAT ON MY ALGORITHMS  
FINAL BY PREVENTING GRAPH THEORY FROM BEING INVENTED.

# Example: Graph Library

A simple library for graphs providing . . .

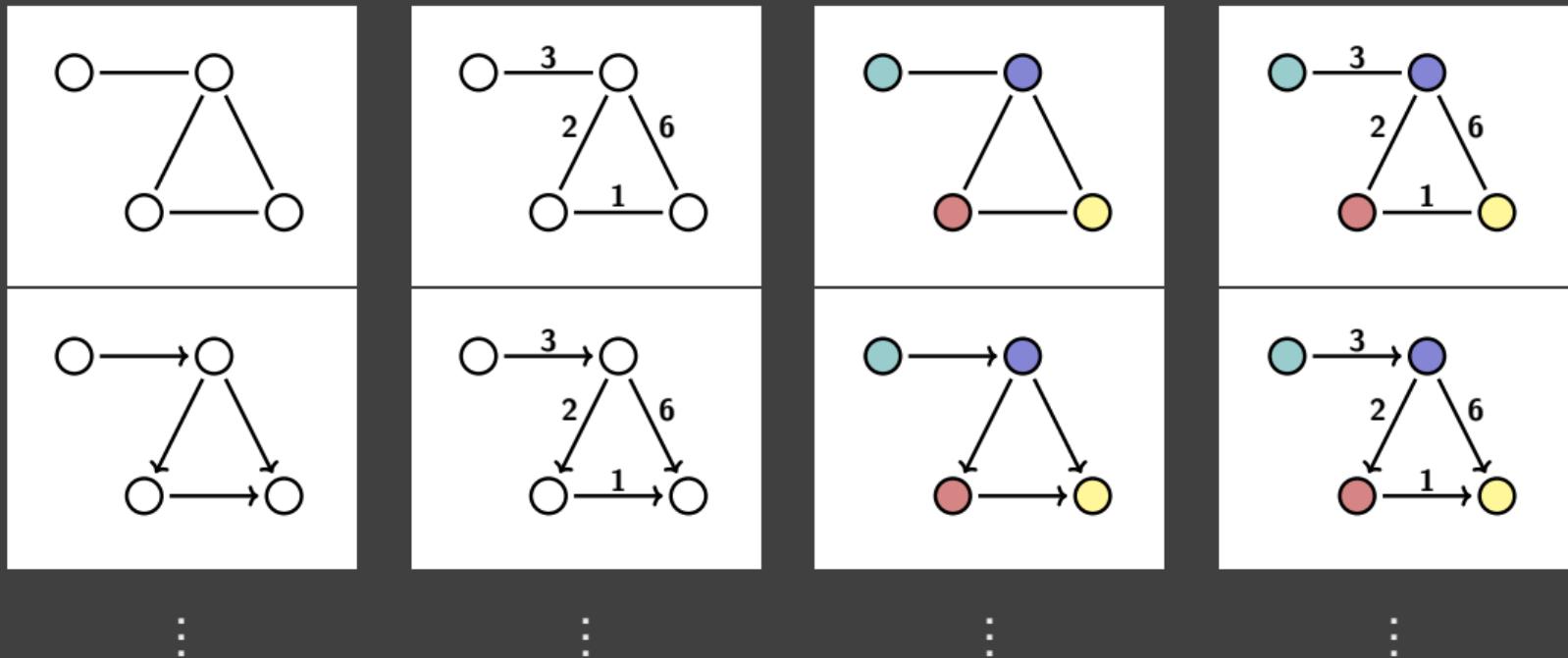
## ... Data Structures

- Directed/undirected edges
- Weighted/unweighted edges
- Colored/uncolored nodes
- ...

## ... and Algorithms

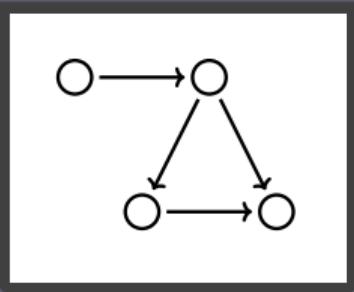
- Vertex numbering
- Vertex coloring
- Shortest path
- Minimum spanning tree
- ...

## Features of a Graph

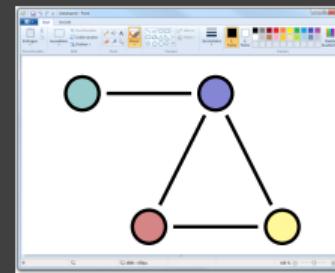
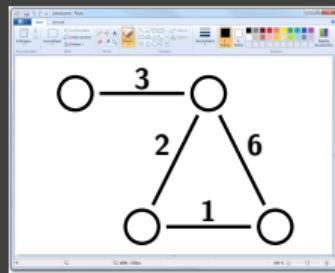
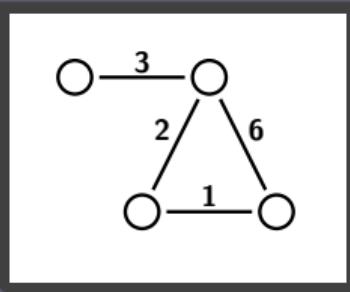


# Features as Configuration Options

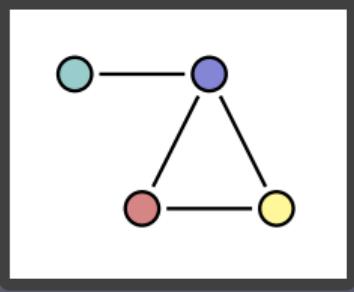
Directed



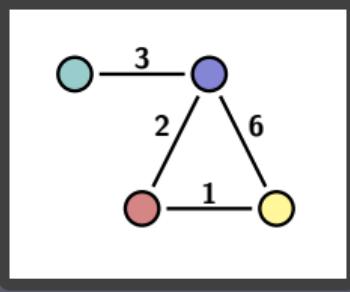
Weighted



Colored



Weighted, Colored



## Configuration of Graph Data Structures

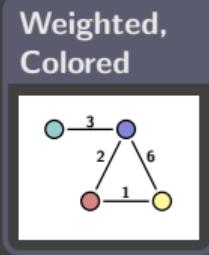
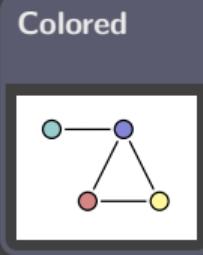
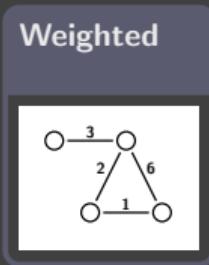
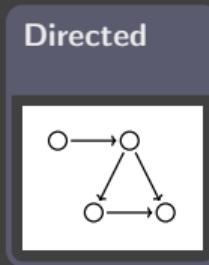
- Typically, configuration options are **flags**
- Their boolean value determines which features are **activated / deactivated**

# Valid Combinations of Options

Algorithm	Graph type	Weights	Coloring
<i>Vertex numbering</i>	*	*	*
<i>Vertex coloring</i>	undirected	*	colored
<i>Shortest path</i>	directed	weighted	*
<i>Minimum spanning tree</i>	undirected	weighted	*
...	...	...	...

## Dependencies Between Features Must Be Checked

- when configuration options are loaded at startup
- whenever options are loaded/changed at runtime



# Configuration of Runtime Variability – Summary

## Lessons Learned

- External setting of configuration options through command-line parameters, preference dialogs, configuration files
- Validity of combinations may be affected by dependencies between features

## Further Reading

- Apel et al. 2013, Chapter 3, pp. 47–49  
— brief introduction of binding times

## Practice

- Do you know any practical examples making use of runtime variability?
- How does the configuration take place?
- Is the configuration checked for validity?

## 2. Runtime Variability and Design Patterns

### 2a. Configuration of Runtime Variability

### 2b. Realization of Runtime Variability

Recap and Motivating Example

Global Variables

Method Parameters

Problems of Runtime Variability

Reconfiguration at Runtime

Code Scattering

Code Tangling

Code Replication

Summary

### 2c. Design Patterns for Variability

# Recap: Variability and Binding Times

## Binding Time

[Apel et al. 2013, p. 48]

- Variability offers choices
- Derivation of a product requires to make decisions (aka. binding)
- Decisions may be bound at different binding times

## When? By whom? How?

Lecture 2a: **when** and **by whom**

Lecture 2b: **how**



# A Non-Variable Graph Implementation

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

```
class Node {  
    int id = 0;  
    Color color = new Color();  
  
    void print() {  
        Color.setDisplayColor(color);  
        System.out.print(id);  
    }  
}
```

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

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

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

# “Symbolic” Feature Traces

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

```
class Node {  
    int id = 0;  
    Color color = new Color();  
  
    void print() {  
        Color.setDisplayColor(color);  
        System.out.print(id);  
    }  
}
```

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

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

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

# Adding Variability

## The Basic Idea

Conditional statements ...  
controlled by configuration options

```
class Graph {  
    ...  
    Edge add(Node n, Node m) {  
        Edge e = new Edge(n, m);  
        if (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 (!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 (COLORED) { color = new Color(); }  
    }  
    void print() {  
        if (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 (WEIGHTED) { weight.print(); }  
    }  
}
```

# Global Variables

```
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(); }  
    }  
}
```

# Special Case: Immutable Global Variables

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

## Idea

Use static configuration when configuration parameters are known at compile time

## Discussion

**Advantage:** Compiler optimizations may remove dead code

**Disadvantage:** No external configuration by the end-user

# Method Parameters

## Idea

- A class exposes its configuration parameters as part of its interface (i.e., method parameters)
- Parameter values are passed along with each method invocation

```
class Graph {  
    boolean weighted, colored;  
    ...  
    Graph(boolean weighted, boolean colored) {  
        this.weighted = weighted; this.colored = colored;  
    }  
    Edge add(Node n, Node m) {  
        Edge e = new Edge(n, m, weighted);  
        if (weighted) { e.weight = new Weight(); }  
        nodes.add(n); nodes.add(m); edges.add(e);  
        return e;  
    }  
}
```

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

## Discussion

**Advantage:** Different instantiations (e.g., colored and uncolored graphs) within the same program  
**Disadvantage:** May lead to methods with many parameters (code smell!)

# Reconfiguration at Runtime?

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

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

## Idea

Alter feature selection without stopping and restarting the program

## Through Experiment

What happens when we change the value of COLORED from false to true (at runtime)?

## Discussion

- Feature-specific code may depend on certain initialization steps or assume certain invariants
- Just updating the values of configuration options does not update the current state of the program (source of bugs!)

[Wang et al. 2023]

# Problem: Code Scattering

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

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

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

## Code Scattering

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

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

# Problem: Code Tangling

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

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

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

Code Tangling

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

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

# Problem: Code Replication (aka. Code Clones)

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

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

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

## Code Replication

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

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

# Realization of Runtime Variability – Summary

## Lessons Learned

- Global (immutable) variables or (lengthy) parameter lists
- Reconfiguration at runtime is possible (in principle)
- Variability is spread over the entire program
- Variable parts are always delivered

## Further Reading

- Apel et al. 2013, Chapter 4
- Meinicke et al. 2017, Section 17.1

## Practice

- Why are code scattering, tangling and replication problematic?
- What are the problems of variable parts being always delivered?

## 2. Runtime Variability and Design Patterns

### 2a. Configuration of Runtime Variability

### 2b. Realization of Runtime Variability

### 2c. Design Patterns for Variability

Recap on Object Orientation

Recap on Design Patterns

Template Method Pattern

Abstract Factory Pattern

Decorator Pattern

#### Trade-Offs and Limitations

Template Method and Abstract Factory

Diamond Problem and Decorator

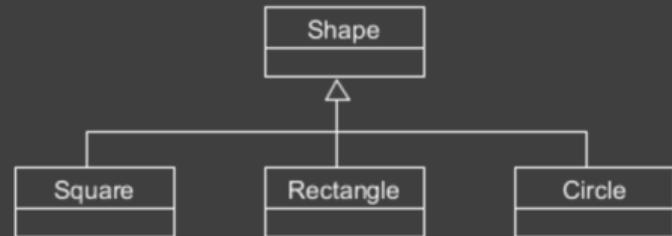
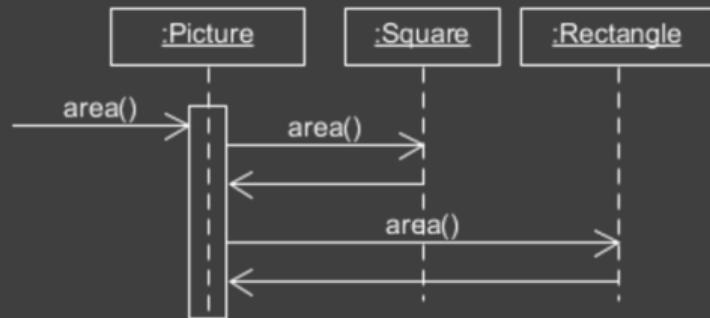
#### Summary

#### FAQ

# Recap: Object Orientation

## Key Concepts

- **Encapsulation:**  
abstraction and information hiding
- **Composition:**  
nested objects
- **Message Passing:**  
delegating responsibility
- **Distribution of Responsibility:**  
separation of concerns
- **Inheritance:**  
conceptual hierarchy, polymorphism, reuse



# Recap: Design Patterns [Gang of Four]

## Design Patterns

- Document common solutions to concrete yet frequently occurring design problems
- Suggest a concrete implementation for a specific object-oriented programming problem

## Design Patterns for Variability

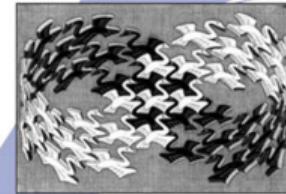
Many Gang of Four (GoF) design patterns for designing software around stable abstractions and interchangeable (i.e., variable) parts, e.g.

- Template Method
- Abstract Factory
- Decorator

# Design Patterns

Elements of Reusable  
Object-Oriented Software

Erich Gamma  
Richard Helm  
Ralph Johnson  
John Vlissides



Cover art © 1994 M.C. Escher / Corbis Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch



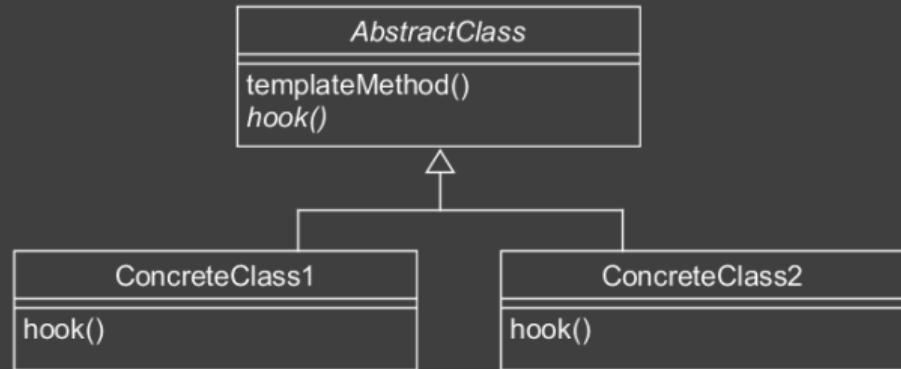
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

# Recap on Design Patterns – Template Method Pattern

## Template Method

[Gang of Four, pp. 325–330]

- **Intent:** “Define the overall structure of an algorithm, while allowing subclasses to refine, or redefine, certain steps.”
- **Motivation:** Avoid code replication by implementing the general workflow of an algorithm once, while allowing for necessary variations.
- **Idea:** A template method defines the skeleton of an algorithm. Concrete methods override the hook methods.

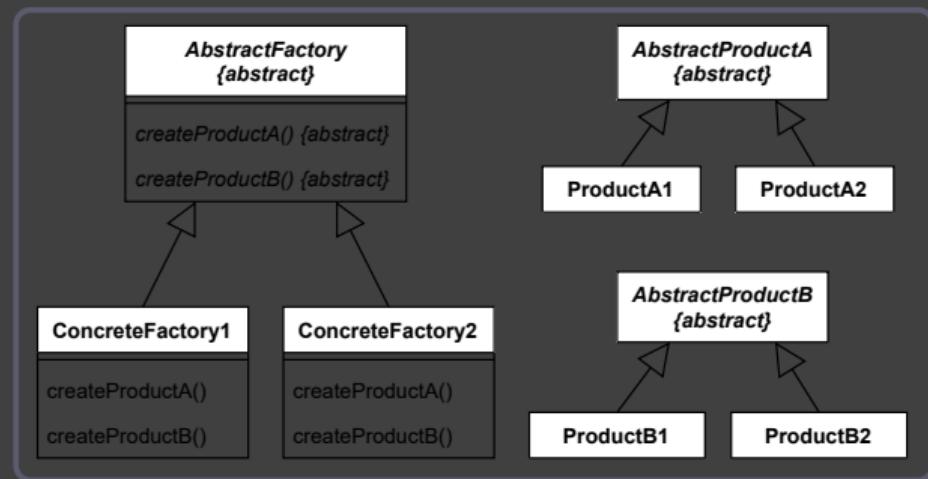


# Recap on Design Patterns – Abstract Factory Pattern

## Abstract Factory

[Gang of Four, pp. 87–95]

- **Intent:** “Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”
- **Motivation:** Avoid case distinctions when creating objects of certain kind, consistently create objects of a particular kind.
- **Idea:** Create classes for the consistent creation of objects.

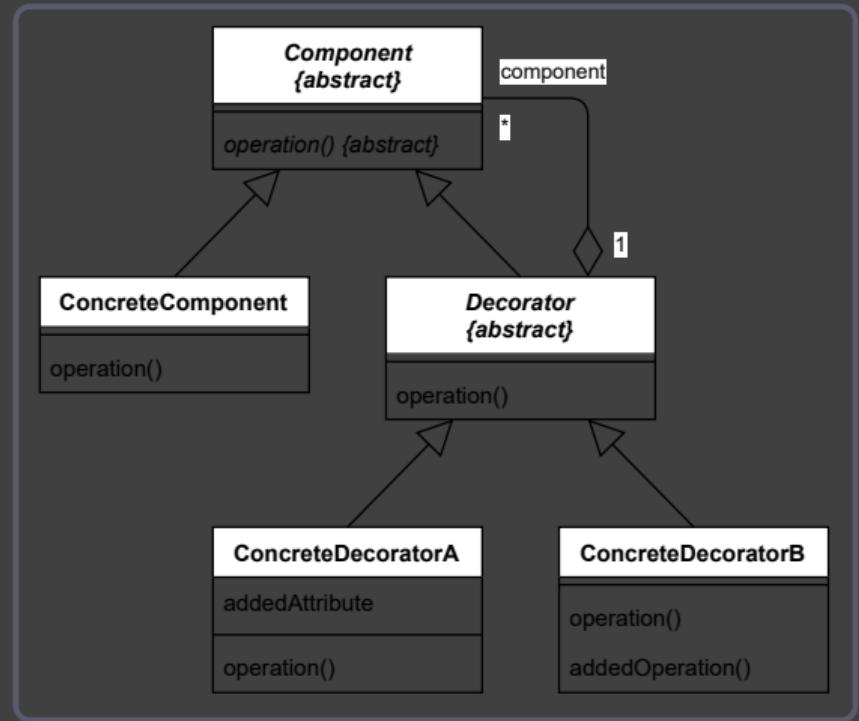


# Recap on Design Patterns – Decorator Pattern

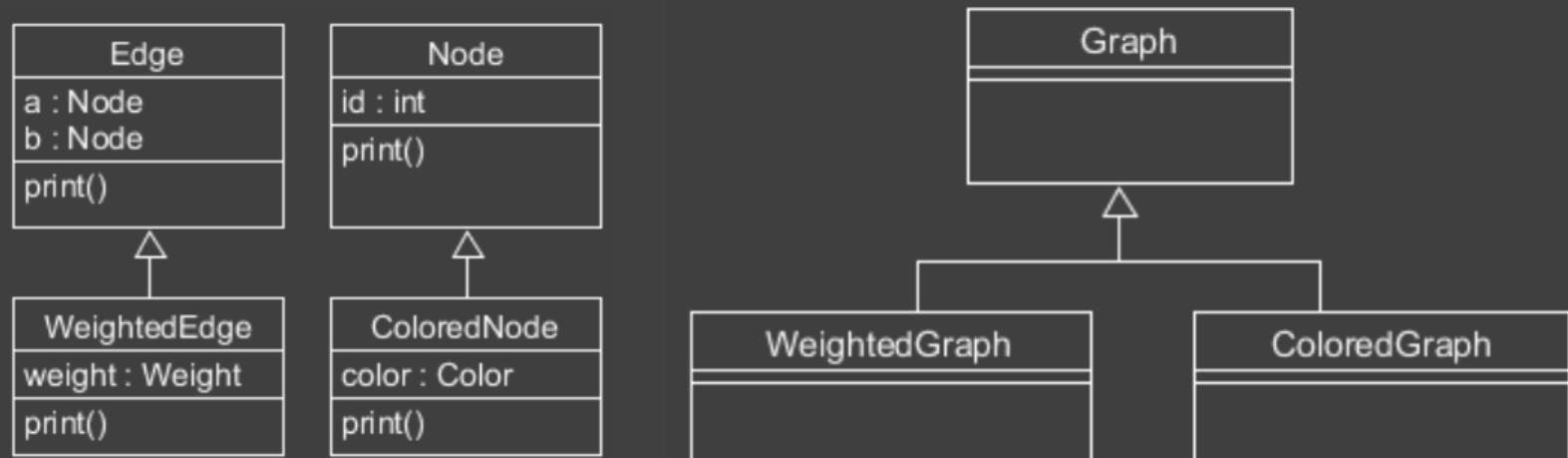
## Decorator

[Gang of Four, pp. 175–184]

- **Intent:** “Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.”
- **Motivation:** Avoid explosion of static classes when combining all additional behaviors with all applicable classes.
- **Idea:** Create decorators and components with the same interface, whereas decorators forward behavior whenever feasible.



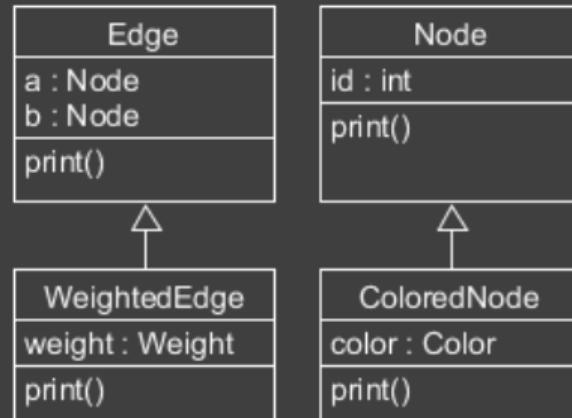
# Object-Oriented Design of our Graph Library



# Instantiation Through Template Method Pattern

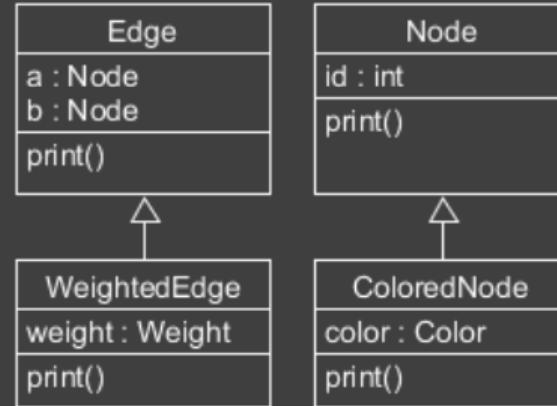
```
class Graph {  
    ...  
    Edge add(Node n, Node m) {  
        Edge e = createEdge();  
        nv.add(n); nv.add(m); ev.add(e);  
        return e;  
    }  
    // hook method (with default implementation)  
    Edge createEdge(Node n, Node m) {  
        return new Edge(n, m);  
    }  
}
```

```
class WeightedGraph extends Graph {  
    ...  
    // override hook method  
    Edge createEdge(Node n, Node m) {  
        Edge e = new WeightedEdge(n, m);  
        e.weight = new Weight();  
        return e;  
    }  
}
```



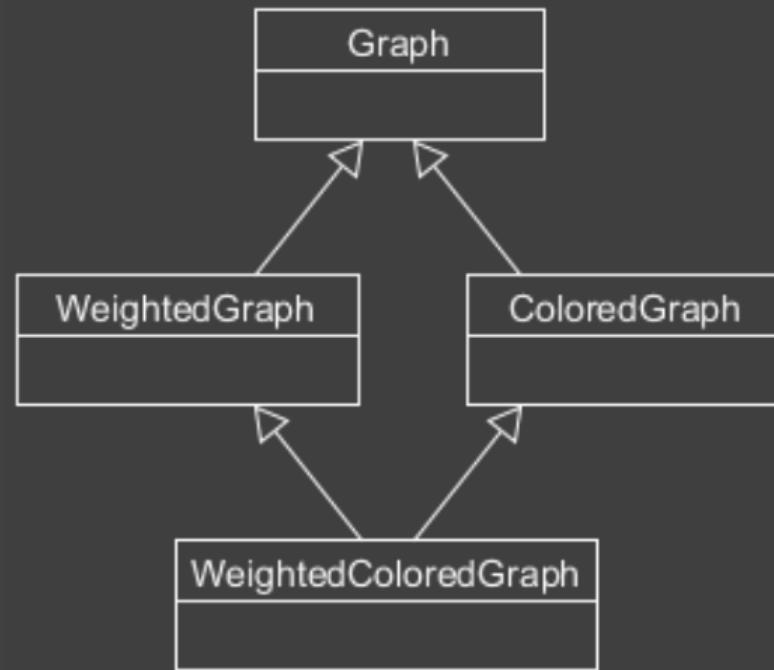
# Instantiation Through Abstract Factory Pattern

```
class Graph {  
    EdgeFactory edgeFactory;  
    ...  
    Graph(EdgeFactory edgeFactory) {  
        this.edgeFactory = edgeFactory;  
    }  
    Edge add(Node n, Node m) {  
        Edge e = edgeFactory.createEdge(n, m);  
        nodes.add(n); nodes.add(m); edges.add(e);  
        return e;  
    }  
}  
  
class EdgeFactory {  
    Edge createEdge(Node a, Node b) {  
        return new Edge(a, b);  
    }  
}
```



```
class WeightedEdgeFactory extends EdgeFactory {  
    Edge createEdge(Node a, Node b) {  
        Edge e = new WeightedEdge(n, m);  
        e.weight = new Weight();  
        return e;  
    }  
}
```

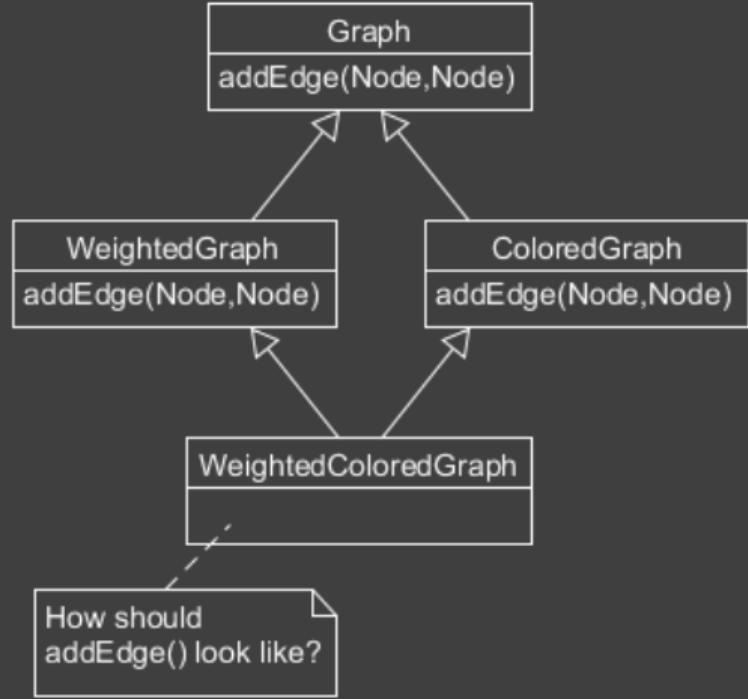
# Feature Combinations?



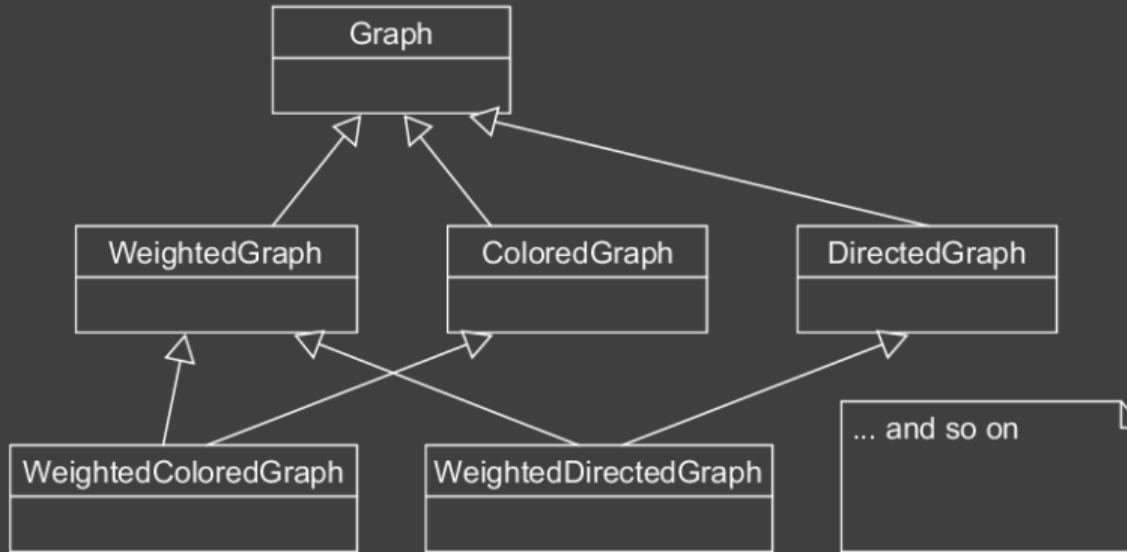
# Diamond Problem

## Multiple Inheritance

- most object-oriented programming languages do not support multiple inheritance (or only provide workarounds)
- critical: how to handle name clashes



# Static Modeling of Feature Combinations

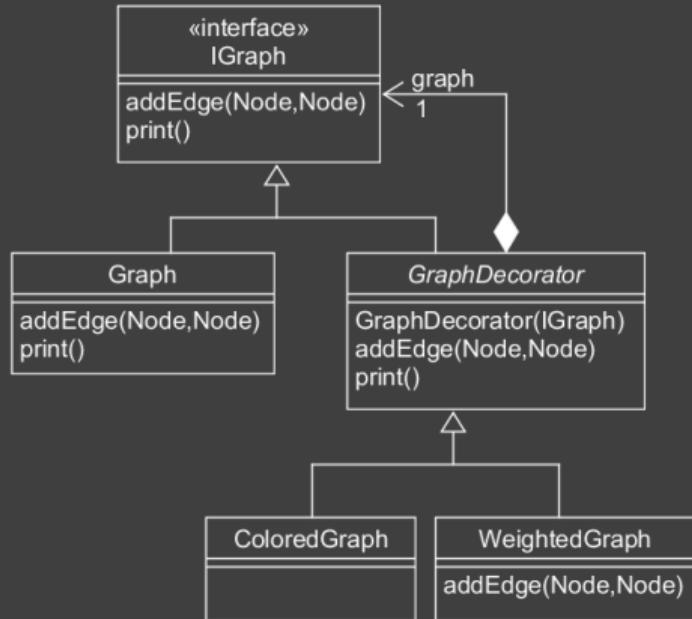


Even if multiple inheritance is supported, statically combining features through inheritance is tedious (or infeasible).

# Decorator Pattern as a Solution?

```
abstract class GraphDecorator implements IGraph {  
    IGraph graph;  
    GraphDecorator(IGraph graph) {  
        this.graph = graph;  
    }  
}
```

```
class WeightedGraph extends GraphDecorator {  
    WeightedGraph(IGraph graph) {  
        super(graph);  
    }  
    Edge add(Node n, Node m) {  
        WeightedEdge e = (WeightedEdge) graph.add(n, m);  
        e.weight = new Weight();  
        return e;  
    }  
    ...  
}
```



## Example Usage

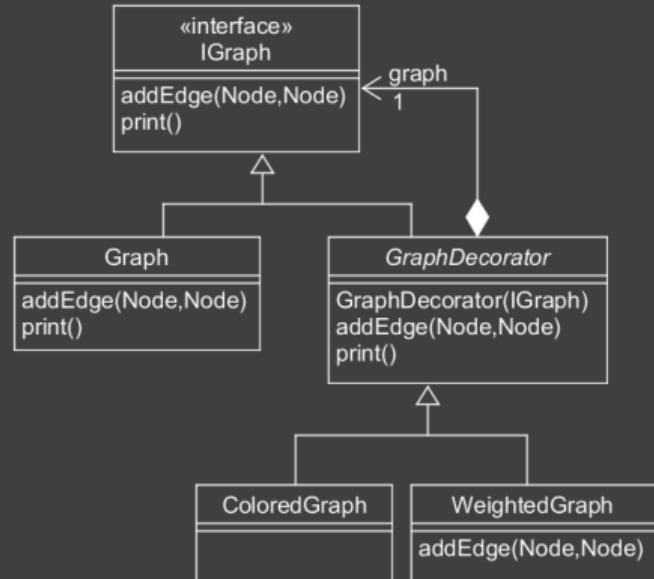
```
IGraph graph = new WeightedGraph(new ColoredGraph(new Graph(new WeightedEdgeFactory())));
```

# Delegation Instead of Inheritance

## Discussion

Extensions (i.e., features) can be combined dynamically, but ...

- must be independent of each other
- cannot add public methods
- runtime overhead due to indirections
- several physical objects are forming a conceptual one (e.g., problems with object identity)



# Design Patterns for Variability – Summary

## Lessons Learned

- Variability through object-orientation and design patterns
- Extension through delegation vs. inheritance
- Limitations and drawbacks w.r.t. feature combinations

## Further Reading

- Meyer 1997, Chapter 3–4
- Gang of Four, Chapter 1–4

## Practice

- What characterizes a modular software design and why can it support variability?
- In which sense are object-oriented solutions more modular than simple conditional statements?
- Do you know of other design patterns supporting variability?

# FAQ – 2. Runtime Variability and Design Patterns

## Lecture 2a

- What is variability, runtime variability, and binding time?
- What is the connection between variability, variability-intensive systems, and software product lines?
- How can configuration options be supplied?
- How can command-line parameters, preference dialogs, and configuration files be used to offer variability?
- What is the principle behind configuration options and when does the configuration happen?
- What are potential features of a graph library?
- What are examples for runtime variability? When are configuration options specified? By whom?
- When is the validity of configuration options checked?

## Lecture 2b

- How to realize runtime variability in source code?
- What is the best strategy?
- How can (immutable) global variables and method parameters be used to realize variability?
- Why is the development of runtime variability challenging?
- What are code scattering, code tangling, and code replication?
- Does runtime variability allow reconfiguration at runtime?
- How to develop new features or variants with runtime variability? Exemplify!
- What are (dis)advantages of runtime variability?
- When (not) to use runtime variability?

## Lecture 2c

- What are design patterns (used for)?
- Why are design patterns relevant for variable software?
- Which design patterns are relevant for variable software?
- What are intent, motivation, and idea for template method, abstract factory, and decorator pattern?
- Given an example class diagram, which pattern is applied or should be applied? Why?
- What is the diamond problem? How does it affect variable software?
- Is multiple inheritance useful to combine features? Exemplify!
- What are limitations of design patterns (for variable software)?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. **Compile-Time Variability with Clone-and-Own**

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 3a. Compile-Time Variability and Clone-and-Own

- Problems with Runtime Variability
- Compile-Time Variability
- Ad-Hoc Clone-and-Own
- Variability with Clone-and-Own
- Problems of Clone-and-Own
- Software Clones
- Discussion
- Summary

### 3b. Clone-and-Own with Version Control

- Software Configuration Management
- Version Control Systems
- Variability with Version Control
- Discussion
- Summary

### 3c. Clone-and-Own with Build Systems

- Build Systems
- Variability with Build Systems
- Discussion
- Summary
- FAQ

# 3. Compile-Time Variability with Clone-and-Own – Handout

Software Product Lines | Timo Kehrer, Thomas Thüm, Elias Kuiter | April 22, 2023

### **3. Compile-Time Variability with Clone-and-Own**

#### **3a. Compile-Time Variability and Clone-and-Own**

Problems with Runtime Variability

Compile-Time Variability

Ad-Hoc Clone-and-Own

Variability with Clone-and-Own

Problems of Clone-and-Own

Software Clones

Discussion

Summary

#### **3b. Clone-and-Own with Version Control**

#### **3c. Clone-and-Own with Build Systems**

# Recap: How to Implement Software Product Lines?



## Key Issues

- Systematic reuse of implementation artifacts
- Explicit handling of variability

## Variability

[Apel et al. 2013, p. 48]

**"Variability** is the ability to derive different products from a common set of artifacts."

## Variability-Intensive System

Any software product line is a variability-intensive system.

# Recap: Variability and Binding Times

## Binding Time

[Apel et al. 2013, p. 48]

- Variability offers choices
- Derivation of a product requires to make decisions (aka. binding)
- Decisions may be bound at different binding times

## When? By whom? How?

Lecture 2a: **when** and **by whom**

Lecture 2b: **how**



# Problems with Runtime Variability

## Basic Principles

### Variability with Configuration Options:

- Conditional statements controlled by configuration options
- Global variables vs. method parameters

### Object-Orientation and Design Patterns:

- Template Method
- Abstract Factory
- Decorator

## Problems of Runtime Variability

### Conditional Statements:

- Code scattering, tangling, and replication

### Design Patterns for Variability:

- Trade-offs and potential negative side effects
- Constraints that may restrict their usage

### In General:

- Variable parts are always delivered
- Not well-suited for compile-time binding

# Compile-Time Variability

## Problems of Runtime Variability

### Conditional Statements:

- Code scattering, tangling, and replication

### Design Patterns for Variability:

- Trade-offs and potential negative side effects
- Constraints that may restrict their usage

### In General:

- Variable parts are always delivered
- Not well-suited for compile-time binding

## Compile-Time Variability

[Apel et al. 2013, p. 49]

“Compile-time variability is decided before or at compile time.”

### Goals:

- Only required source code is compiled
- Smaller and highly optimized variants

### Challenge:

- How to implement options and alternatives (i.e., variability)?

## In this Lecture

Simple concepts and techniques for a few variants

# Ad-Hoc Clone-and-Own

## Clone-and-Own

[Northrop et al. 2012, p. 7]

"Suppose you are developing a new system that seems very similar to one you have built before. You borrow what you can from your previous effort, modify it as necessary, add whatever it takes, and field the product, which then assumes its own maintenance trajectory separate from that of the first product. What you have done is what is called **clone and own**. You certainly have taken economic advantage of previous work; you have reused a part of another system. But now you have two entirely different systems, not two systems built from the same base. This is again **ad hoc reuse**."

## Cloning Whole Products



## Clone-and-Own

- New variants of a software system are created by copying and adapting an existing variant
- Afterwards, cloned variants evolve independently of each other

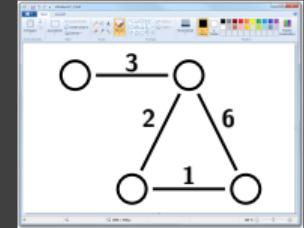
# Example for Ad-Hoc Clone-and-Own

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

initial graph implementation  
providing weighted graphs

```
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();  
    }  
}
```

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

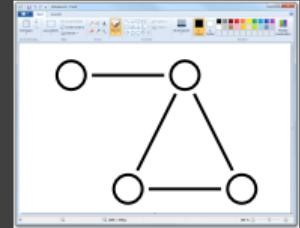


```
public class Node {  
    int id = 0;  
  
    void print() {  
        System.out.print(id);  
    }  
}
```

# Alice's Clone: Unweighted Graphs

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

Alice works with unweighted graphs: she copies and adapts the basic implementation

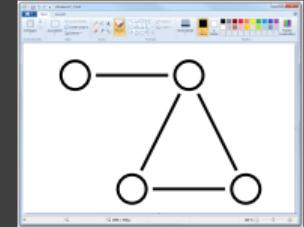


```
class Edge {  
    Node a, b;  
    Weight weight = new Weight();  
  
    Edge(Node a, Node b) {  
        this.a = a; this.b = b;  
    }  
    void print() {  
        a.print(); b.print();  
        weight.print();  
    }  
}
```

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

```
public class Node {  
    int id = 0;  
  
    void print() {  
        System.out.print(id);  
    }  
}
```

# Alice's Clone: Unweighted Graphs



```
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);  
        return e;  
    }  
    void print() {  
        for (int i = 0; i < edges.size(); i++) {  
            ((Edge) edges.get(i)).print();  
        }  
    }  
}
```

Alice works with unweighted graphs: she copies and adapts the basic implementation

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

```
public class Node {  
    int id = 0;  
  
    void print() {  
        System.out.print(id);  
    }  
}
```

# Bob's Clone: Colored Graphs

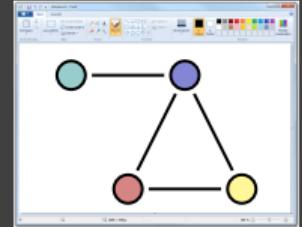
```
public class Graph {  
    List nodes = new ArrayList();  
    List edges = new ArrayList();  
  
    Edge add(Node n, Node m) {  
        Edge e = new Edge(n, m);  
        nodes.add(n); nodes.add(m); edges.add(e);  
        return e;  
    }  
    void print() {  
        for (int i = 0; i < edges.size(); i++) {  
            ((Edge) edges.get(i)).print();  
        }  
    }  
}
```

Bob works with colored graphs: he is a colleague of Alice and knows her variant, so he copies and adapts Alice's variant

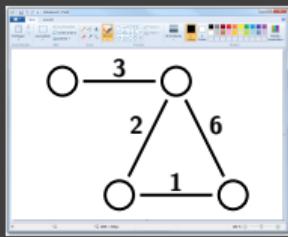
```
class Edge {  
    Node a, b;  
  
    Edge(Node a, Node b) {  
        this.a = a; this.b = b;  
    }  
    void print() {  
        a.print(); b.print();  
    }  
}
```

```
public class Node {  
    int id = 0;  
    Color color = new Color();  
  
    void print() {  
        Color.setDisplayColor(color);  
        System.out.print(id);  
    }  
}
```

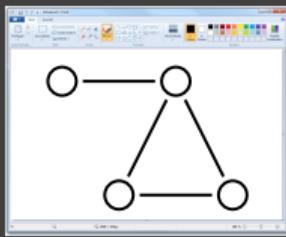
```
public class Color {  
    static void setDisplayColor(  
        Color c) {...}  
}
```



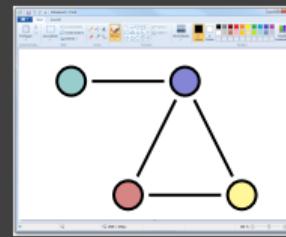
# Why is Clone-and-Own Problematic?



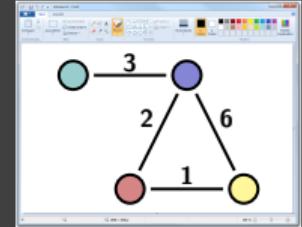
Alice



Bob

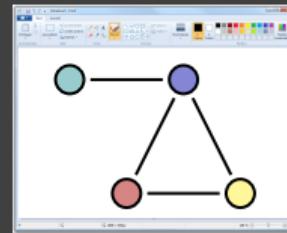
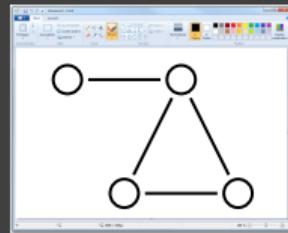
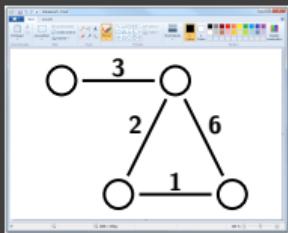


# Clone-and-Own Problems: Feature Combinations

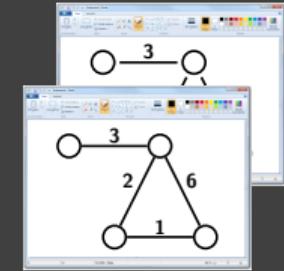


Eve has a new requirement: she wants to work with graphs which are both colored and weighted

- Where to start from?
- Does Eve know about Bob's and Alice's variants?
- If so, how to avoid repeating the work that has been already done by Alice and Bob, respectively?



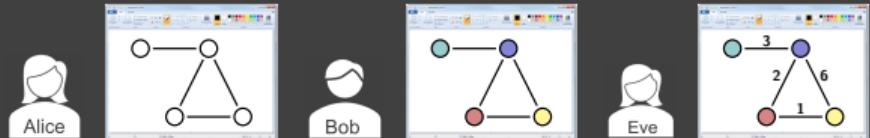
# Clone-and-Own Problems: Evolution & Maintenance



Maintainers of the initial variant refactor the code of the basic implementation

```
public class Graph {  
    ...  
    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);  
        Edge e = add(n, m);  
        e.weight = w;  
        return e;  
    }  
    ...  
}
```

- Who informs Alice, Bob and Eve about the improvement?
- How do they know whether the improvement is relevant for them?
- If so, how to propagate the improvement to their variant?



# Recap: Software Clones

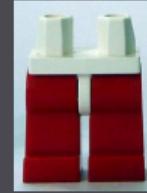
[Lecture 1]

## Software Clone

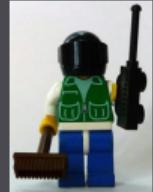
[Rattan et al. 2013, p. 1166]

- = result of copying and pasting existing fragments of the software
- code clones = copied code fragments
- replicates need to be altered consistently
- for example: bugs need to be fixed in all replicated fragments
- in practice: a common source for inconsistencies and bugs

## Cloning Parts of Software



## Cloning Whole Products (Clone-and-Own)



# Types of Software Clones

## Types of Software Clones

[Rattan et al. 2013, p. 1167]

- Type 1: identical except whitespaces and comments
- Type 2: syntactically similar (e.g., changed identifiers, ...)
- Type 3: copied with modifications (e.g., inserted or removed statements)
- Type 4: similar functionality without textual similarities

## Cloning Parts of Software



## Relevant Types for Clone-and-Own?

- Type 1: may happen if clones diverge and comments need to reflect actual changes
- Type 2: may happen if clones diverge and identifier names are not appropriate anymore
- Type 3: actually necessary for clone-and-own
- Type 4: may happen if same functionality is implemented again (simply unknown or merge/cherrypick infeasible)

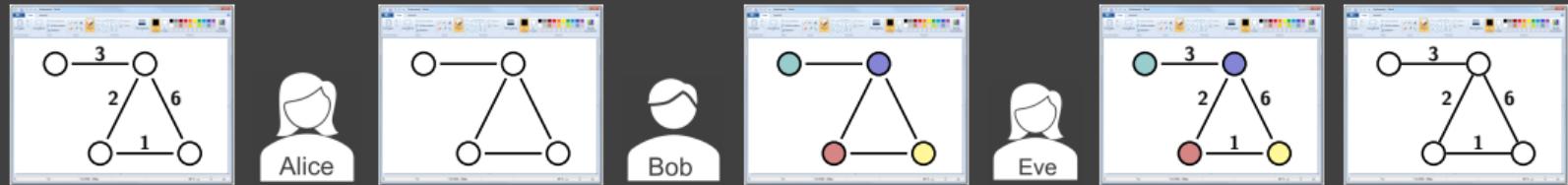
[see Lecture 4]

Every difference is an obstacle for future maintenance (cf. merge and cherrypick)

## Cloning Whole Products (Clone-and-Own)



# Discussion of Clone-and-Own



## Advantages

- Simple and straightforward approach
- Rapid exploration of new ideas
- No upfront investments

## Disadvantages

- No structured and systematic reuse (copy & edit)
- No flexible combination of features
- Maintenance quickly becomes impractical

## Towards Managed Clone-and-Own

- How can we better manage such clone-and-own development?
- The traditional answer: Software Configuration Management
- In the sequel: Software Configuration Management in practice

# Compile-Time Variability and Clone-and-Own – Summary

## Lessons Learned

- Compile-time variability is decided before or at compile time
- In clone-and-own, new variants of a software system are created by copying and adapting an existing variant
- Simple paradigm, but suffering from maintenance problems in the long run

## Further Reading

- Apel et al. 2013, Section 3.1.1, pp. 48–49  
— brief introduction of compile-time variability
- Rattan et al. 2013, Section 2, pp. 1167–1168  
— brief introduction to software clones, their types, reasons, (dis)advantages
- Antkiewicz et al. 2014 — brief introduction to ad-hoc clone-and-own (L0)

## Practice

- What are the reasons why clone-and-own is very popular in practice?
- What is the order of magnitude of the number of variants that can be reasonably maintained in clone-and-own?
- Have you ever applied the principle of clone-and-own? If so, where and how?

### **3. Compile-Time Variability with Clone-and-Own**

#### **3a. Compile-Time Variability and Clone-and-Own**

#### **3b. Clone-and-Own with Version Control**

Software Configuration Management

Version Control Systems

Variability with Version Control

Discussion

Summary

#### **3c. Clone-and-Own with Build Systems**

# Excusus: Software Configuration Management

## Software Configuration Management

Policies, processes, and tools for managing evolving software systems:

- Version control
- System building
- Release management
- Change management
- Collaborative work

## No Software Configuration Management

Lecture 3a: Ad-Hoc Clone-and-Own  
aka. unmanaged clone-and-own

## Version Control

Lecture 3b: Clone-and-Own with Version Control  
instance of managed clone-and-own

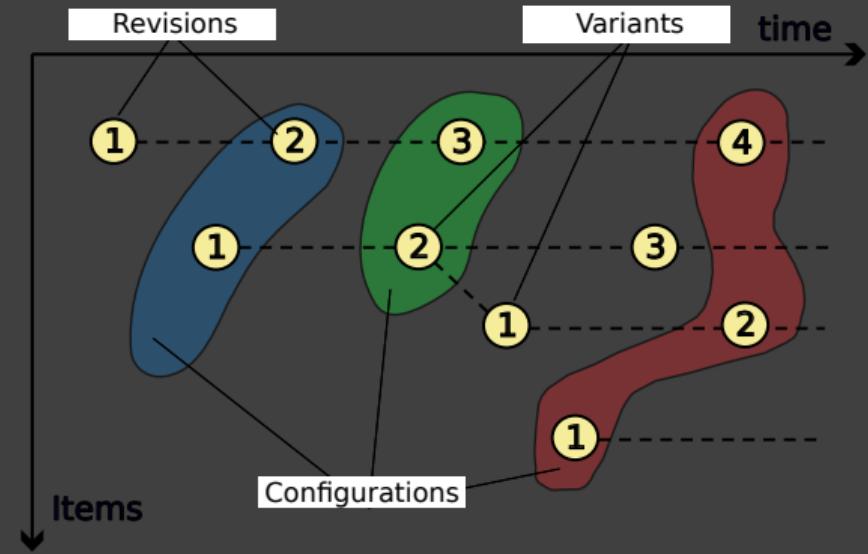
## System Building

Lecture 3c: Clone-and-Own with Build Systems  
instance of managed clone-and-own

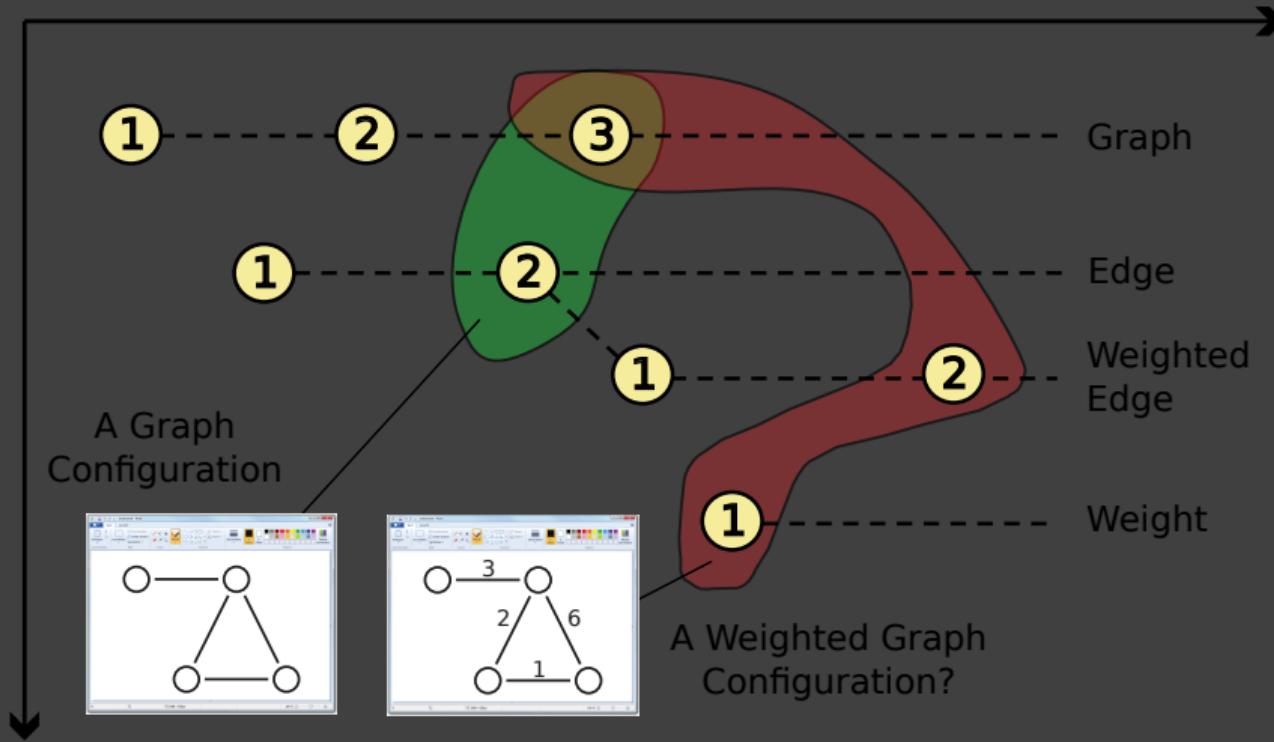
# Excursus: Software Configuration Management

## Basic Terms and Definitions

- **Software Item:** An (atomic) artifact that can be uniquely identified
- **Version:** A modified software item
  - **Revision:** A new version that replaces an old one
  - **Variant:** A version that co-exists with another one
- **Configuration:** A set of software items that together form a functioning (partial) system
- **Baseline:** A stable configuration that represents a point of reference for further development
- **Release:** A baseline delivered to customers

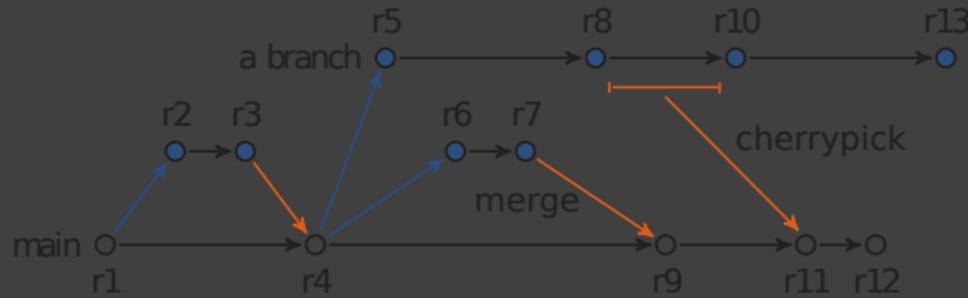


## Example: A Conceptual Organization of our Graph Library



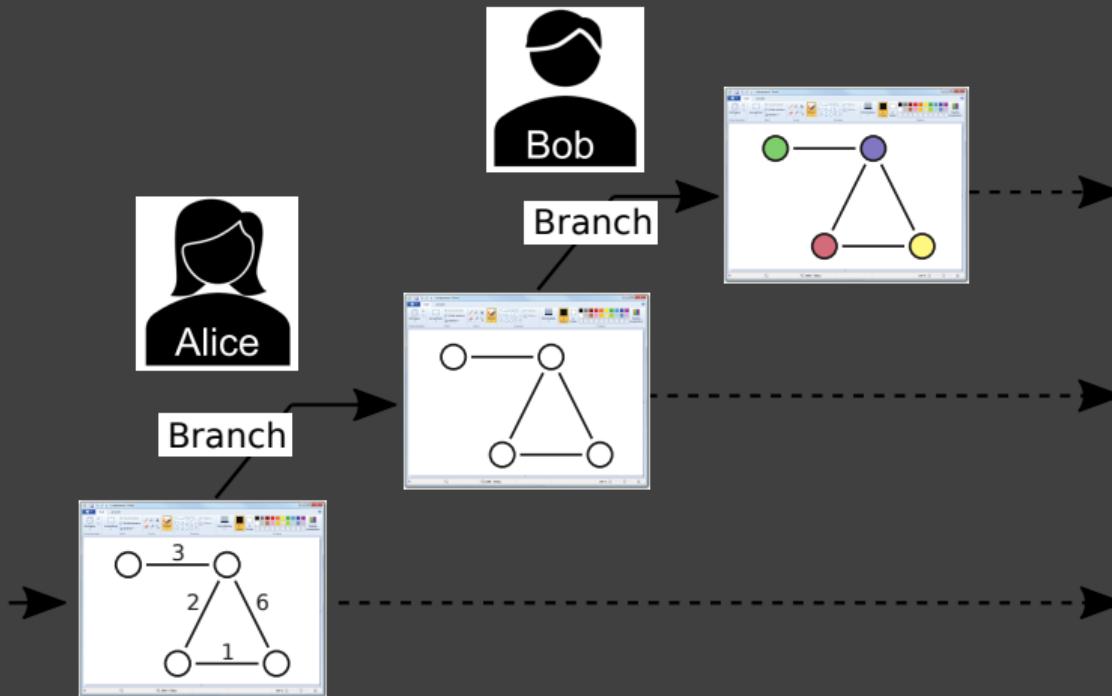
# Tool Support: Version Control Systems

cherrypick := patch( $\Delta(r8, r10), r11$ )

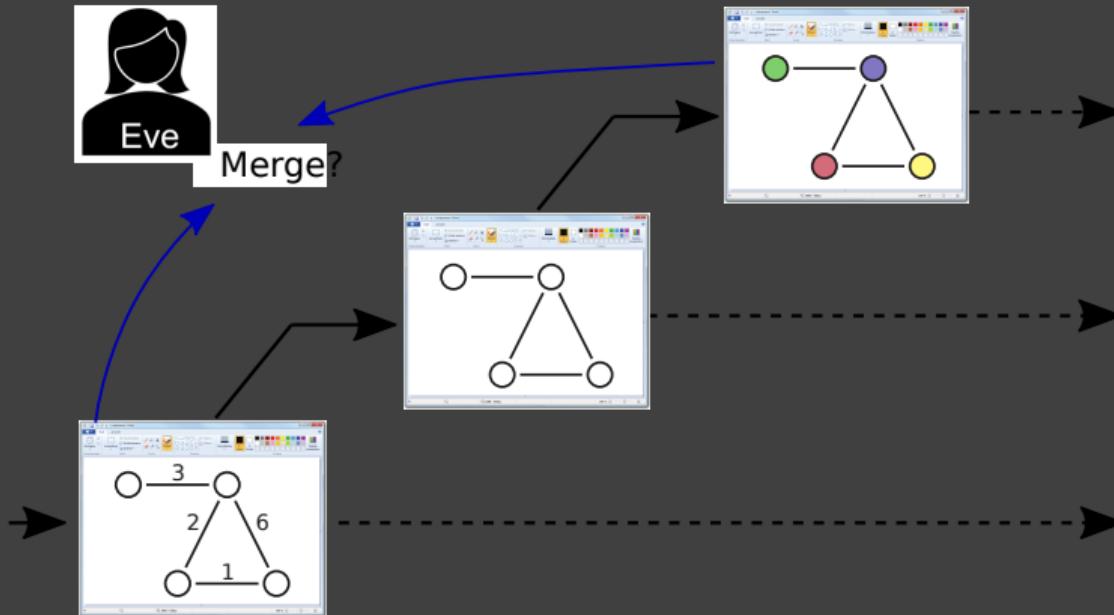


merge := 3-way-merge( $r4, \Delta(r4, r7), \Delta(r4, r9)$ )

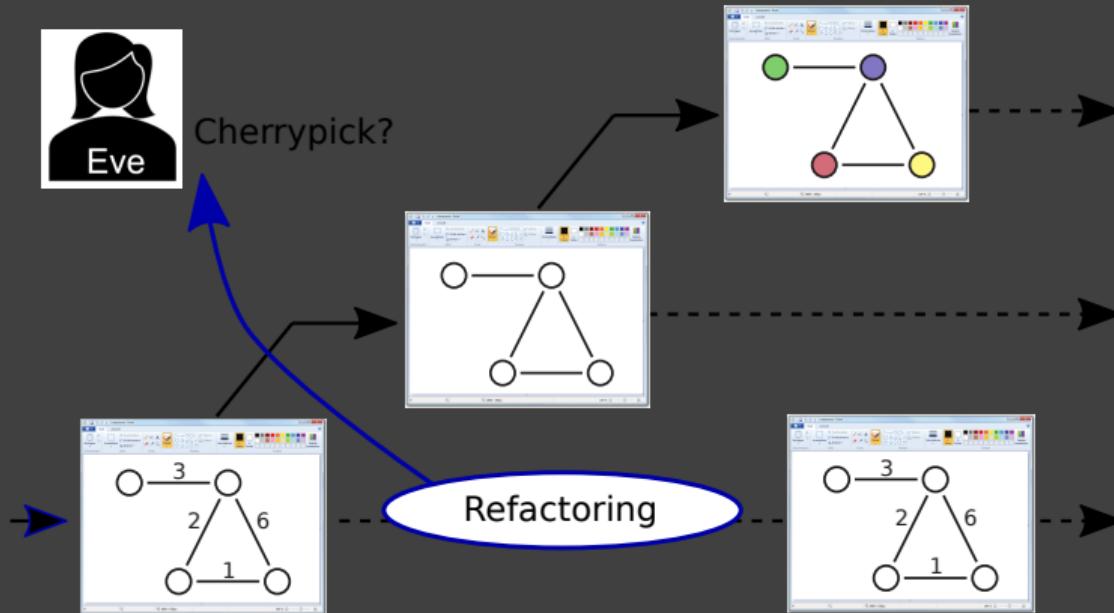
## Example: Graph Library under Version Control



## Example: Graph Library under Version Control



## Example: Graph Library under Version Control



# Clone-and-Own with Version Control

## Observations

- Aka. **managed clone-and-own** (opposed to ad-hoc clone-and-own)
- Supports keeping track of revisions and variants – aka. **provenance**
- Creation of new variants is partially supported by merging of branches
- Propagation of changes between variants is supported by cherrypicking changes

However:

- Versioning is typically limited to entire system variants (i.e., branches)
- No flexible combination of software items

## Advantages

[Apel et al. 2013, p. 104]

- Well-established and stable systems
- Well-known known process
- Good tool integration

## Disadvantages

[Apel et al. 2013, pp. 104–105]

- Development of variants, not features: flexible combination of features not directly possible
- No structured reuse (copy & edit)
- Merging and cherrypicking not fully automated

# No Version Control at All?

## Revision Control $\subset$ Version Control

“Unless only few small variations are required for few customers, the use of version control systems should be restricted to **revision control**.”

[Apel et al. 2013, p. 104]

# Clone-and-Own with Version Control – Summary

## Lessons Learned

- Software configuration management as a traditional discipline of managing the evolution of variability-intensive systems
- Version control systems as a widespread tool supporting clone-and-own in practice

## Further Reading

- Apel et al. 2013, Section 5.1, pp. 99–105 — introduction of variability with version control (not explicitly calling it clone-and-own)
- Staples and Hill 2004 — experience report on managed clone-and-own with version control and build systems
- Antkiewicz et al. 2014 — brief introduction to clone-and-own with version control (L1)

## Practice

- Which software configuration management concepts are supported by version control systems?
- Do you know other version control systems than Git?
- If so, in which way are they different from Git?

### **3. Compile-Time Variability with Clone-and-Own**

#### **3a. Compile-Time Variability and Clone-and-Own**

#### **3b. Clone-and-Own with Version Control**

#### **3c. Clone-and-Own with Build Systems**

Build Systems

Variability with Build Systems

Discussion

Summary

FAQ

# Recap: Software Configuration Management

## Software Configuration Management

Policies, processes, and tools for managing evolving software systems:

- Version control
- System building
- Release management
- Change management
- Collaborative work

## No Software Configuration Management

Lecture 3a: Ad-Hoc Clone-and-Own  
aka. unmanaged clone-and-own

## Version Control

Lecture 3b: Clone-and-Own with Version Control  
instance of managed clone-and-own

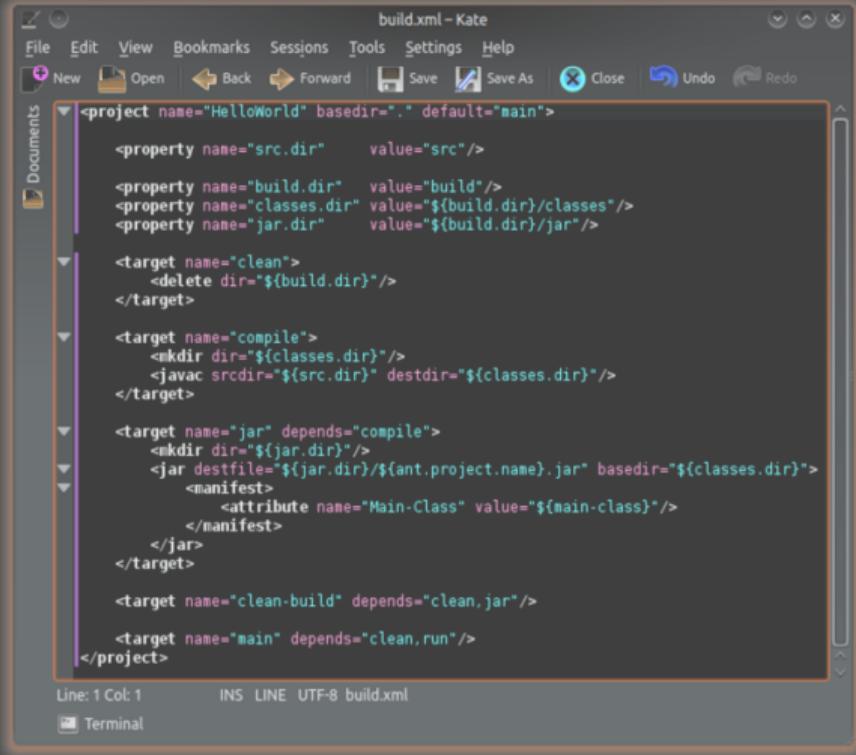
## System Building

Lecture 3c: Clone-and-Own with Build Systems  
instance of managed clone-and-own

# Tool Support: Build Systems

## Build Systems

- Automation of the build process through build scripts
- Multiple steps with dependencies/conditions
  - Copy files,
  - call compiler,
  - start other tools,
  - ...
- Tools:
  - make
  - ant
  - maven ...



The screenshot shows a window titled "build.xml - Kate" containing an Ant build script. The script defines a project named "HelloWorld" with a default target "main". It sets properties for source directories ("src.dir"), build directories ("build.dir" and "classes.dir"), and jar directories ("jar.dir"). It includes three targets: "clean" (deletes the build directory), "compile" (creates the classes directory and compiles source files), and "jar" (creates a jar file named after the project). It also defines a "clean-build" target that depends on "clean" and "jar", and a "main" target that depends on "clean" and "run". The code is color-coded for syntax highlighting.

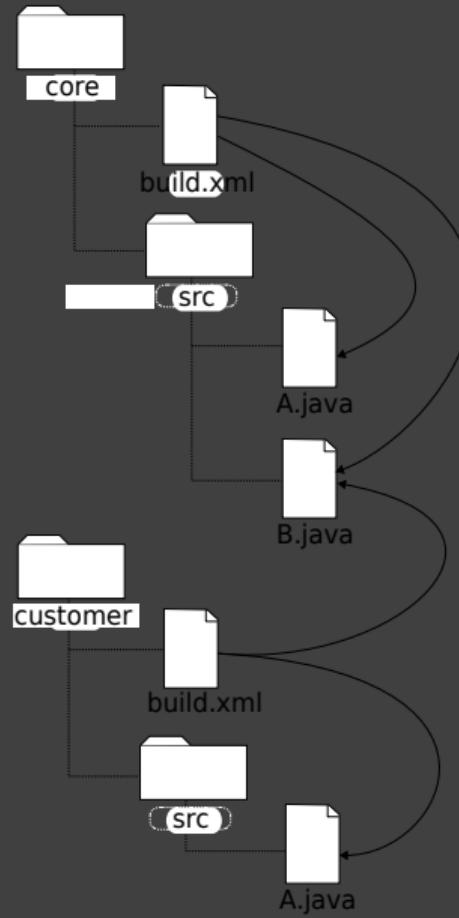
```
<project name="HelloWorld" basedir=". " default="main">  
    <property name="src.dir" value="src"/>  
    <property name="build.dir" value="build"/>  
    <property name="classes.dir" value="${build.dir}/classes"/>  
    <property name="jar.dir" value="${build.dir}/jar"/>  
  
    <target name="clean">  
        <delete dir="${build.dir}"/>  
    </target>  
  
    <target name="compile">  
        <mkdir dir="${classes.dir}"/>  
        <javac srcdir="${src.dir}" destdir="${classes.dir}"/>  
    </target>  
  
    <target name="jar" depends="compile">  
        <mkdir dir="${jar.dir}"/>  
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}">  
            <manifest>  
                <attribute name="Main-Class" value="${main-class}"/>  
            </manifest>  
        </jar>  
    </target>  
  
    <target name="clean-build" depends="clean,jar"/>  
  
    <target name="main" depends="clean,run"/>  
</project>
```

Line: 1 Col: 1      INS LINE UTF-8 build.xml  
Terminal

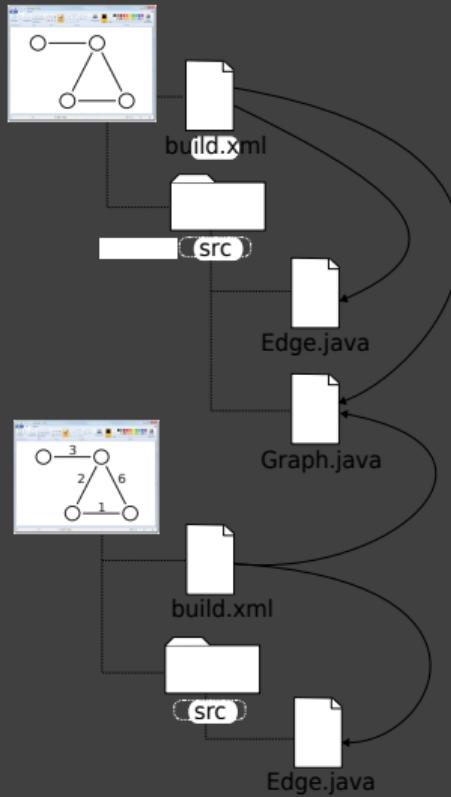
# Variability with Build Systems

## Basic Idea

- One build script per variant
- Include/exclude files when translating
- Overwrite variant-specific files



## Example: Graph Library



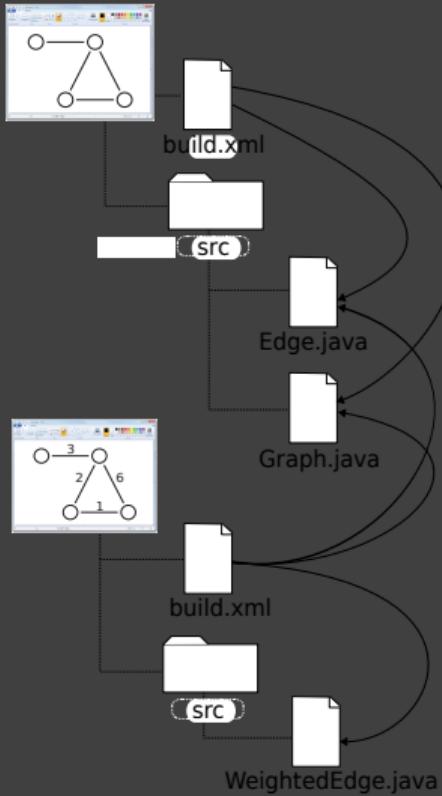
```
class Edge {
    Node a, b;

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

```
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();
    }
}
```

# Example: Graph Library



```
class Graph {  
    EdgeFactory edgeFactory;  
    ...  
    Graph(EdgeFactory edgeFactory) {  
        this.edgeFactory = edgeFactory;  
    }  
    Edge add(Node n, Node m) {  
        Edge e = edgeFactory.createEdge(n, m);  
        nodes.add(n); nodes.add(m); edges.add(e);  
        return e;  
    }  
}
```

```
class Edge {  
    Node a, b;  
    ...  
}
```

```
class WeightedEdge extends Edge {  
    Weight weight = new Weight();  
    ...  
}
```

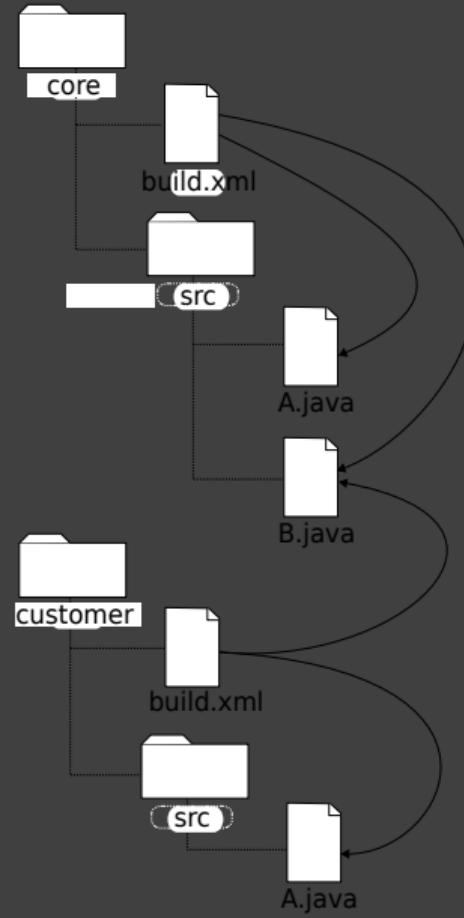
# Clone-and-Own with Build Systems

## Comparison to Version Control Systems

- Supports combination of more fine-grained software items (i.e., files)
- However: Only limited support for provenance

## In General

- Combination of items (i.e., files)  
≠ combination of features
- Changes to the basic variant may have undesired side-effects
  - Some variants are updated but do not need those changes
  - Some variants are updated but incompatible to those changes
  - Variants with copied files are not automatically updated



# Clone-and-Own with Build Systems – Summary

## Lessons Learned

- Variability through build scripts
- Granularity of clones: Individual files
- Combination of files  
≠ combination of features

## Further Reading

- Apel et al. 2013, Section 5.2.2 and 5.2.4, pp. 106–110 — brief introduction of clone-and-own with build systems (not explicitly calling it clone-and-own)
- Staples and Hill 2004 — experience report on managed clone-and-own with version control and build systems

## Practice

- Which software configuration management concepts are supported by build systems?
- What are the commonalities and differences of clone-and-own with version control and clone-and-own with build systems?
- What are the strengths and weaknesses?

# FAQ – 3. Compile-Time Variability with Clone-and-Own

## Lecture 3a

- What are problems of runtime variability?
- What is compile-time variability?
- What is (ad-hoc) clone-and-own?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of (ad-hoc) clone-and-own?
- When (not) to use (ad-hoc) clone-and-own?
- What is better runtime or compile-time variability?

## Lecture 3b

- What is software configuration management and version control (used for)?
- What is the difference between version, revision, variant, configuration, baseline, release, merge, cherrypick, revision control?
- How can version control be used for clone-and-own? Illustrate!
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of version control for variability?
- When (not) to use clone-and-own with version control?
- Shall we use version control at all?

## Lecture 3c

- What are build systems (used for)?
- How can build systems be used for clone-and-own? Illustrate!
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of build systems for variability?
- When (not) to use clone-and-own with build systems?
- Shall we use build systems at all?
- What is the granularity of clones for all three techniques?
- What is the effort to migrate from ad-hoc clone-and-own to managed clone-and-own?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 4a. Feature Models and Configurations

- Recap: Software Product Lines
- Features Have Dependencies
- Specifying Valid Configurations
  - Natural Language
  - Configuration Map
- Feature Models
- Discussion of Feature Models
- Summary

### 4b. Transforming Feature Models

- Representations and Transformations
- UVL, the Universal Variability Language
- Propositional Formulas
- CNF as a Universal Formula Language
- Summary

### 4c. Analyzing Feature Models

- Configurators in the Wild
- Automated Analysis of Feature Models
- SAT, #SAT, and Alisat
- Consistency, Cardinality, and Enumeration
  - Feature Model
  - Features
  - Partial Configurations
- Automated Analyses in FeatureIDE
- Summary
- FAQ

## 4. Feature Modeling – Handout

Software Product Lines | Elias Kuiter, Thomas Thüm, Timo Kehrer | April 26, 2023

# 4. Feature Modeling

## 4a. Feature Models and Configurations

Recap: Software Product Lines

Features Have Dependencies

Specifying Valid Configurations

- Natural Language

- Configuration Map

Feature Models

Discussion of Feature Models

Summary

## 4b. Transforming Feature Models

## 4c. Analyzing Feature Models

# Recap: Software Product Lines

[Lecture 1]

## Software Product Line

[Northrop et al. 2012, p. 5]

“A **software product line** is

- a set of software-intensive systems
- that share a common, managed set of features
- satisfying the specific needs of a particular market segment or mission
- and that are developed from a common set of core assets in a prescribed way.”

[Software Engineering Institute, Carnegie Mellon University]

## Product

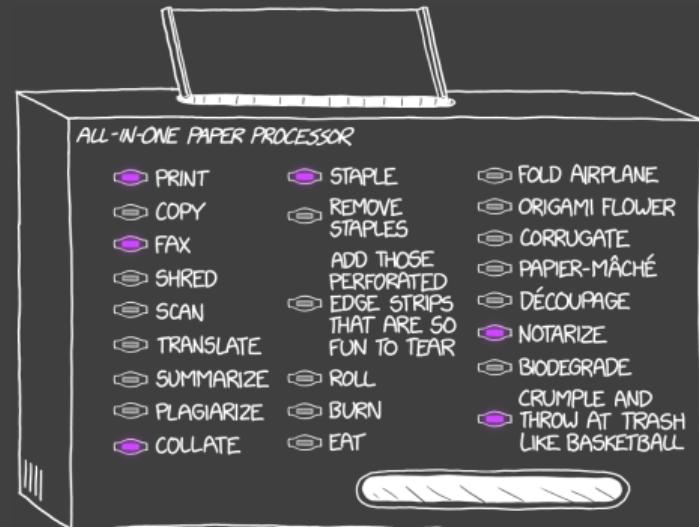
[Apel et al. 2013, p. 19]

“A **product of a product line** is specified by a valid feature selection (a subset of the features of the product line). A feature selection is **valid** if and only if it fulfills all feature dependencies.”

## Feature

[Apel et al. 2013, p. 18]

“A **feature** is a characteristic or end-user-visible behavior of a software system.”



# Features Have Dependencies

Ordering a Waffle ...



... with Sugar



... with Cherries



This is Nice, But ...

- plate and sugar seem to always be included, a fork is only included for some orders  
⇒ limitations seem **arbitrary**
- children get special treatment  
⇒ order process is **unfair**
- what exactly am I paying for?  
⇒ investments are **unclear**

In This Lecture

1. how to **model and configure** features and their dependencies?
2. how to **store and communicate**?
3. how to **analyze and understand**?

# Specifying Valid Configurations

## Configuration

- a **configuration** over a set of features  $F$  selects and deselects features in  $F$
- formally: a pair  $(S, D)$  such that  $S, D \subseteq F$  and  $S, D$  are disjoint ( $S \cap D = \emptyset$ )
- is **complete** if all features are covered ( $S \cup D = F$ ) and **partial** otherwise
- a complete configuration is **valid** if it “makes sense” in the domain and **invalid** otherwise
- we often abbreviate complete configurations with  $S \equiv (S, F \setminus S)$

Feature set  $F = \{\text{ConfigDB}, \text{Get}, \text{Put}, \text{Delete}, \text{Transactions}, \text{Windows}, \text{Linux}\}$

Examples for **complete** configurations:

- **valid** (read-only database on Windows):  
 $(\{C, G, W\}, \{P, D, T, L\})$
- **valid** (fully functional database on Linux):  
 $(\{C, G, P, D, T, L\}, \{W\})$
- **invalid** ( $\notin$  no operating system):  
 $(\{C, G\}, \{P, D, T, W, L\})$
- **invalid** (transactions  $\notin$  read-only database):  
 $(\{C, G, T, L\}, \{P, D, W\})$

Examples for **partial** configurations:

$(\{C, G\}, \{P, D\}), (\emptyset, \emptyset)$

# Specifying Valid Configurations – Natural Language

## Valid Configuration

A complete configuration over  $F$  is valid if it “makes sense” in the domain. ↵ “**makes sense**”?

## Natural Language

- informal description of relationships between features in  $F$
- a complete configuration  $S$  is **valid** if it conforms to the description
  - + succinct
  - sometimes ambiguous
  - not machine-readable

“A **configurable database** has an API that allows for at least one of the request types **Get**, **Put**, or **Delete**. Optionally, the database can support **transactions**, provided that the API allows for Put or Delete requests. Also, the database targets a supported operating system, which is either **Windows** or **Linux**.”

# Specifying Valid Configurations – Configuration Map

## Valid Configuration

A complete configuration over  $F$  is valid if it "makes sense" in the domain. ↵ "makes sense"?

## Configuration Map

- a **configuration map** over  $F$  is a set of complete configurations  $M \subseteq \mathcal{P}(F)$
  - a complete configuration  $S$  is **valid** if it occurs in the configuration map ( $S \in M$ )
  - also known as product map
- + precise  
- not human-readable  
- redundant, explodes in size ( $0 \leq |M| \leq 2^{|F|}$ )

Feature set  $F = \{\text{ConfigDB}, \text{Get}, \text{Put}, \text{Delete}, \text{Transactions}, \text{Windows}, \text{Linux}\}$

Configuration map:

{C, G, W}	{C, G, L}
{C, P, W}	{C, P, L}
{C, G, P, W}	{C, G, P, L}
{C, D, W}	{C, D, L}
{C, G, D, W}	{C, G, D, L}
{C, P, D, W}	{C, P, D, L}
{C, G, P, D, W}	{C, G, P, D, L}
{C, P, T, W}	{C, P, T, L}
{C, G, P, T, W}	{C, G, P, T, L}
{C, D, T, W}	{C, D, T, L}
{C, G, D, T, W}	{C, G, D, T, L}
{C, P, D, T, W}	{C, P, D, T, L}
{C, G, P, D, T, W}	{C, G, P, D, T, L}

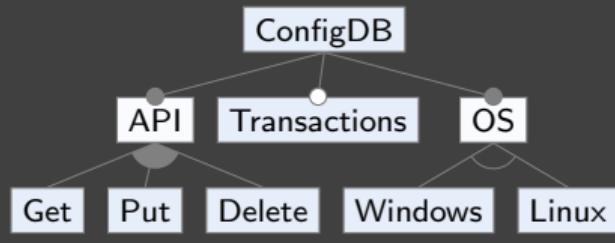
# Specifying Valid Configurations – Configuration Map in Excel

	A	B	C	D	E	F	G
1	ConfigDB	Get	Put	Delete	Transactions	Windows	Linux
2	x	x				x	
3	x		x			x	
4	x	x	x			x	
5	x			x		x	
6	x	x		x		x	
7	x		x	x		x	
8	x	x	x	x		x	
9	x		x		x	x	
10	x	x	x		x	x	
11	x			x	x	x	
12	x	x		x	x	x	
13	x		x	x	x	x	
14	x	x	x	x	x	x	
15	x	x					x
16	x		x				x
17	x	x	x				x
18	x			x			x
19	x	x		x			x
20	x		x	x			x
21	x	x	x	x			x
22	x		x		x		x
23	x	x	x		x		x
24	x			x	x		x
25	x	x		x	x		x
26	x		x	x	x		x
27	x	x	x	x	x		x

Can we do better?

# Feature Models – Syntax

[Apel et al. 2013; Kang et al. 1990, pp. 63–72; Batory 2005]



*Transactions* →  $\text{Put} \vee \text{Delete}$

## Legend:

- Abstract Feature
- Concrete Feature
- Mandatory
- Optional
- ▲ Or Group
- △ Alternative Group

## Feature Model

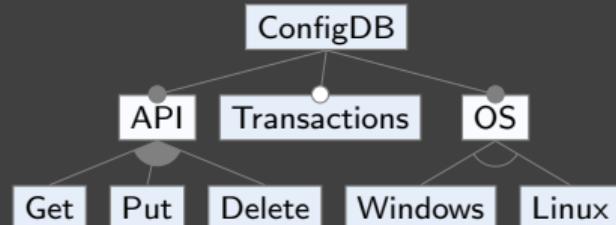
- hierarchy of features
- dependencies between features modeled by tree and cross-tree constraints
- **tree constraints**: defined by the hierarchy
- **cross-tree constraints**: propositional formulas over features
- **abstract features** are used to group other features
- **concrete features** have an implementation
- also known as feature diagram or feature tree
- notation for **optional/mandatory features** and **or/alternative groups**

# Feature Models – Semantics

[Apel et al. 2013; Batory 2005]

## Tree Constraints

- the **root feature** is always required
- each feature requires its parent (aka. **parent-child-relationship**)
- an **optional feature** can be (de-)selected freely when its parent is selected
- a **mandatory feature** is required by its parent
- **or group**: at least one child feature must be selected when the parent is selected
- **alternative group**: exactly one child feature must be selected when the parent is selected

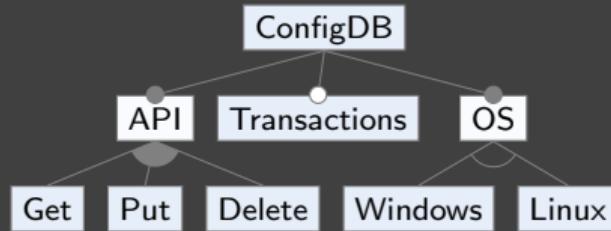


*Transactions* → *Put* ∨ *Delete*

## Cross-Tree Constraints

- a list of **propositional formulas** expressing further dependencies between features
- each cross-tree constraint must be satisfied

# Feature Models – Examples

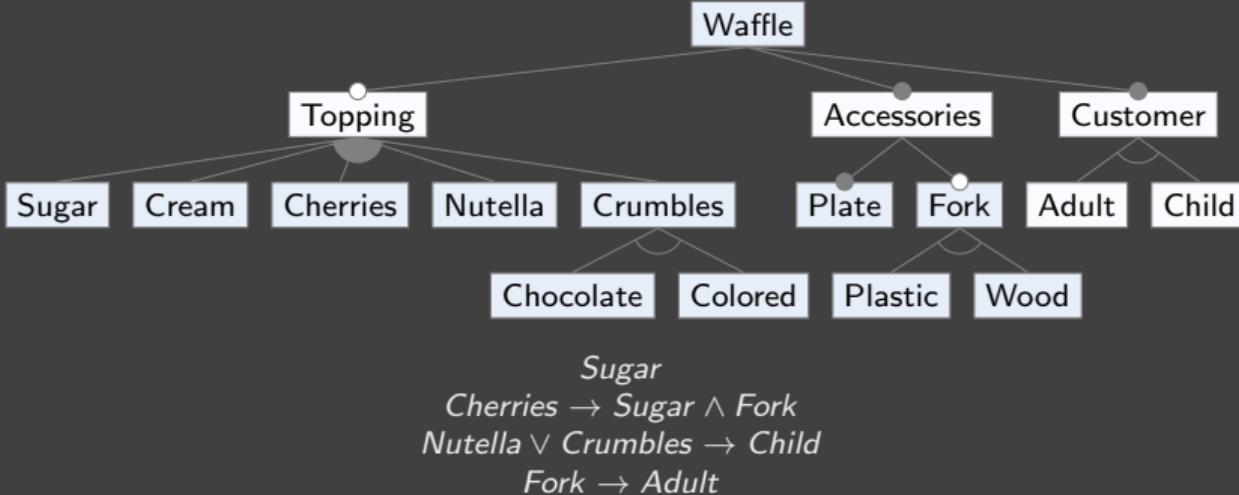


*Transactions → Put ∨ Delete*

## Is This a Valid Configuration?

- **valid** (read-only database on Windows):  
 $(\{C, A, G, O, W\}, \{P, D, T, L\})$
- **valid** (fully functional database on Linux):  
 $(\{C, A, G, P, D, T, O, L\}, \{W\})$
- **invalid** ( $\not\in$  no operating system):  
 $(\{C, A, G\}, \{P, D, T, O, W, L\})$
- **invalid** (transactions  $\not\in$  read-only database):  
 $(\{C, A, G, T, O, L\}, \{P, D, W\})$

# Feature Models – Examples



- every feature (leaf or compound) can be abstract or concrete
- groups can be used anywhere
- directly below groups, no optional or mandatory markers are allowed

# Discussion of Feature Models

## Pro: Making Tacit Knowledge Explicit

"I think the best [about feature modeling] is you can see relationships, to actually know what configurations are allowed and what are not allowed. That was also not so easy to express in the past [...] This is from the developer's point of view. But it's also [...] important, because before we noticed that **the same functionality was implemented twice** within the same project, basically they haven't realized that. They implemented the same features." – Interview with Practitioners [Berger et al. 2014]

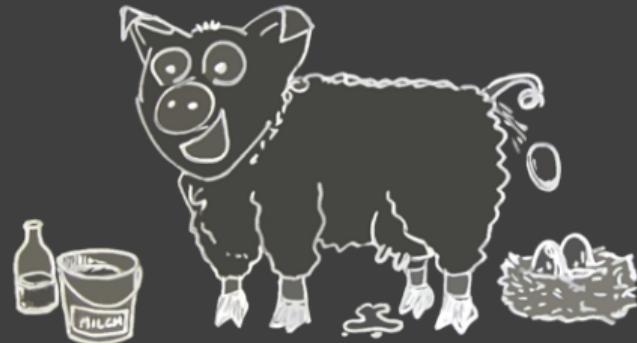
## Pro: Tool Support



, Gears, pure::variants, ...

## Con: Challenges

- **domain scoping:** which features?
- **feature interactions:** which dependencies?
- requires infrastructure, consulting, and training



# Feature Models and Configurations – Summary

## Lessons Learned

- features, dependencies between features, and configurations
- feature models: abstract and concrete features, tree and cross-tree constraints
- tree constraints: optional, mandatory, or group, alternative group

## Further Reading

- Apel et al. 2013, Section 2.3, pp. 26–39  
— introduction to feature modeling
- Thorsten Berger et al. (2013): A Survey of Variability Modeling in Industrial Practice
- Damir Nešić et al. (2019): Principles of Feature Modeling

## Practice

1. sketch a feature model with features  $A, B, C, D, E, F$  that has exactly those 5 valid configurations (pen and paper preferred):  
 $\{A, B\}$        $\{A, C, E\}$        $\{A, C, E, F\}$   
 $\{A, B, D\}$        $\{A, C, F\}$
2. discuss in groups whether your feature models are syntactically correct and specify exactly the above configurations

## 4. Feature Modeling

### 4a. Feature Models and Configurations

### 4b. Transforming Feature Models

Representations and Transformations

UVL, the Universal Variability Language

Propositional Formulas

CNF as a Universal Formula Language

Summary

### 4c. Analyzing Feature Models

# Representations and Transformations

## Natural Language

"A configurable database has an API that allows for at least one of the request types Get, Put, or Delete. Optionally, the database can support transactions, provided that the API allows for Put or Delete requests. Also, the database targets a supported operating system, which is either Windows or Linux."

## Configuration Map

$\{C, G, W\}$

:

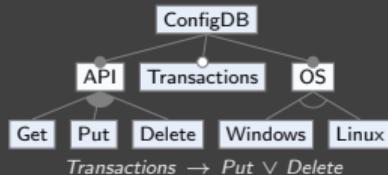
$\{C, G, P, D, T, W\}$

$\{C, G, L\}$

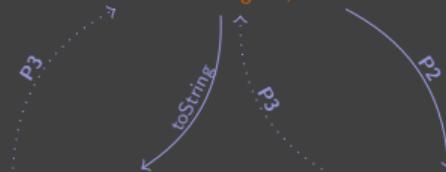
:

$\{C, G, P, D, T, L\}$

## Feature Diagram (Graphical Feature Model)

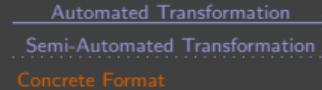


## Feature Model Feature Diagram, P1



## Natural Language Thoughts, Plain Text

## Configuration Map Excel, Set of Sets



## Problems

P1 How to express feature models **textually**?

P2 How to (a) validate configurations and (b) get all valid configurations **automatically**?

P3 (How to reverse engineer feature models?)

# UVL, the Universal Variability Language [UVL]

## features

ConfigDB

**mandatory**

API {abstract}

**or**

Get

Put

Delete

**optional**

Transactions

**mandatory**

OS {abstract}

**alternative**

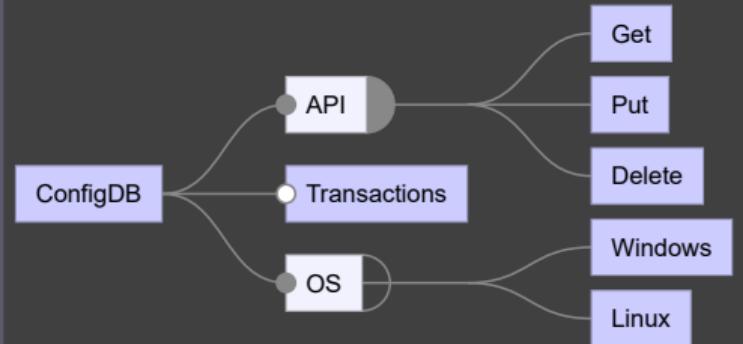
Windows

Linux

## constraints

$\text{Transactions} \Rightarrow \text{Put} \mid \text{Delete}$

## A Feature Model “Sideways”

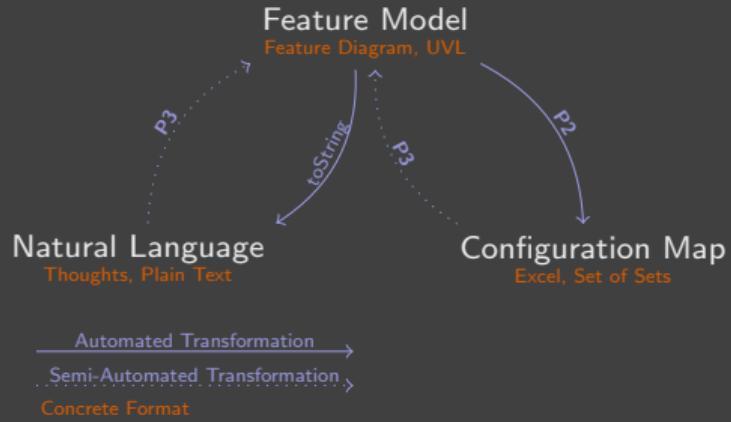


$\text{Transactions} \rightarrow \text{Put} \vee \text{Delete}$

## Universal Variability Language (UVL)

- textual language for feature modeling
- adds advanced modeling constructs (e.g., attributes, cardinalities, submodels, ...)

# Representations and Transformations



## Problems

- P1 How to express feature models **textually**?
- P2 How to (a) validate configurations and (b) get all valid configurations **automatically**?
- P3 (How to reverse engineer feature models?)

## Solutions

- P1 Universal Variability Language  $\Rightarrow$  **Syntax**
- P2 **Semantics?**
- P3 –

# Propositional Formulas – Recap

## Syntax of Propositional Formulas

Inductive definition of **propositional formulas**

- the **Boolean truth values**  $\top, \perp$
- any **Boolean variable**  $X$
- any **negation**  $\neg\phi$  of a formula  $\phi$
- any **conjunction**  $(\phi \wedge \psi)$  of formulas  $\phi$  and  $\psi$
- any **disjunction**  $(\phi \vee \psi)$ , **implication**  $(\phi \rightarrow \psi)$ , or **biimplication**  $(\phi \leftrightarrow \psi)$

## Informal Semantics of Propositional Formulas

$\top$	“true” (or <b>tautology</b> )
$\perp$	“false” (or <b>contradiction</b> )
$\neg\phi$	“not $\phi$ ”
$\phi \wedge \psi$	“ $\phi$ and $\psi$ ”
$\phi \vee \psi$	“ $\phi$ or $\psi$ ” (inclusive or!)
$\phi \rightarrow \psi$	“if $\phi$ , then $\psi$ ” (and else?)
$\phi \leftrightarrow \psi$	“ $\phi$ if and only if $\psi$ ”

## Operator Precedence: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

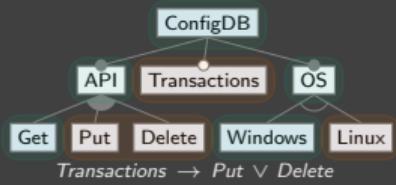
$$\textit{Transactions} \rightarrow (\textit{Put} \vee \textit{Delete})$$

$$\equiv \textit{Transactions} \rightarrow \textit{Put} \vee \textit{Delete}$$

$$\not\equiv (\textit{Transactions} \rightarrow \textit{Put}) \vee \textit{Delete}$$

# Propositional Formulas – Example

A Feature Model  $FM \dots$



$\dots$  as a Propositional Formula  $\Phi(FM)$

$$\begin{aligned}\Phi(FM) = & \text{ConfigDB} \\ \wedge & (\text{API} \leftrightarrow \text{ConfigDB}) \\ \wedge & (\text{Transactions} \rightarrow \text{ConfigDB}) \\ \wedge & (\text{OS} \leftrightarrow \text{ConfigDB}) \\ \wedge & (\text{Get} \vee \text{Put} \vee \text{Delete} \leftrightarrow \text{API}) \\ \wedge & (\text{Windows} \vee \text{Linux} \leftrightarrow \text{OS}) \\ \wedge & \neg(\text{Windows} \wedge \text{Linux}) \\ \wedge & (\text{Transactions} \rightarrow \text{Put} \vee \text{Delete})\end{aligned}$$

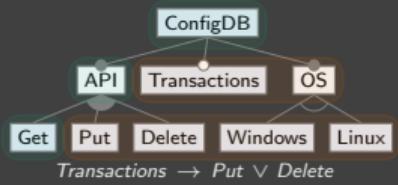
Is This a Valid Configuration?

$$\begin{aligned}\Phi(FM)(\{C, A, G, O, W\}) \\ \equiv \Phi(FM)((\{C, A, G, O, W\}, \{P, D, T, L\})) \\ \equiv C \wedge (A \leftrightarrow C) \wedge (T \rightarrow C) \wedge (O \leftrightarrow C) \\ \wedge (G \vee P \vee D \leftrightarrow A) \wedge (W \vee L \leftrightarrow O) \\ \wedge \neg(W \wedge L) \wedge (T \rightarrow P \vee D) \\ \equiv \top \wedge (\top \leftrightarrow \top) \wedge (\perp \rightarrow \top) \wedge (\top \leftrightarrow \top) \\ \wedge (\top \vee \perp \vee \perp \leftrightarrow \top) \wedge (\top \vee \perp \leftrightarrow \top) \\ \wedge \neg(\top \wedge \perp) \wedge (\perp \rightarrow \perp \vee \perp) \\ \equiv \top \wedge \top \\ \equiv \top\end{aligned}$$

⇒ **configuration is valid**  
(read-only database on Windows)

# Propositional Formulas – Example

A Feature Model  $FM \dots$



$\dots$  as a Propositional Formula  $\Phi(FM)$

$$\begin{aligned}\Phi(FM) = & \text{ConfigDB} \\ \wedge & (\text{API} \leftrightarrow \text{ConfigDB}) \\ \wedge & (\text{Transactions} \rightarrow \text{ConfigDB}) \\ \wedge & (\text{OS} \leftrightarrow \text{ConfigDB}) \\ \wedge & (\text{Get} \vee \text{Put} \vee \text{Delete} \leftrightarrow \text{API}) \\ \wedge & (\text{Windows} \vee \text{Linux} \leftrightarrow \text{OS}) \\ \wedge & \neg(\text{Windows} \wedge \text{Linux}) \\ \wedge & (\text{Transactions} \rightarrow \text{Put} \vee \text{Delete})\end{aligned}$$

Is This a Valid Configuration?

$$\begin{aligned}\Phi(FM)(\{C, A, G\}) \\ \equiv & \Phi(FM)((\{C, A, G\}, \{P, D, T, O, W, L\})) \\ \equiv & C \wedge (A \leftrightarrow C) \wedge (T \rightarrow C) \wedge (O \leftrightarrow C) \\ & \wedge (G \vee P \vee D \leftrightarrow A) \wedge (W \vee L \leftrightarrow O) \\ & \wedge \neg(W \wedge L) \wedge (T \rightarrow P \vee D) \\ \equiv & \top \wedge (\top \leftrightarrow \top) \wedge (\perp \rightarrow \top) \wedge (\perp \leftrightarrow \top) \\ & \wedge (\top \vee \perp \vee \perp \leftrightarrow \top) \wedge (\perp \vee \perp \leftrightarrow \perp) \\ & \wedge \neg(\perp \wedge \perp) \wedge (\perp \rightarrow \perp \vee \perp) \\ \equiv & \top \wedge \top \wedge \top \wedge \perp \wedge \top \wedge \top \wedge \top \wedge \top \wedge \top \\ \equiv & \perp\end{aligned}$$

⇒ **configuration is invalid**  
( $\nexists$  no operating system)

# Propositional Formulas – Algorithm

## Algorithm: Translate $FM$ Into $\Phi(FM)$

1. translate each tree constraint
  - **Root feature:**  $R$  is always required
  - **Optional feature:**  $C$  requires  $P$
  - **Mandatory feature:**  
Optional +  $P$  requires  $C$
  - **Or group:**  
Optional +  $P$  requires at least one  $C_i$
  - **Alternative group:**  
Optional +  $P$  requires exactly one  $C_i$
2. conjoin translated tree constraints  
$$\Phi(TC) \leftarrow \bigwedge_{tc \in TC} \Phi(tc)$$
3. conjoin **cross-tree constraints**  
$$\Phi(CTC) \leftarrow \bigwedge_{ctc \in CTC} ctc$$
4.  $\Phi(FM) \leftarrow \Phi(TC) \wedge \Phi(CTC)$

$$\Phi(\text{Root}) = Root$$

$$\Phi\left(\begin{array}{c} P \\ \bullet \\ C \end{array}\right) = C \rightarrow P$$

$$\Phi\left(\begin{array}{c} P \\ \bullet \\ C \end{array}\right) = C \leftrightarrow P$$

$$\Phi\left(\begin{array}{c} P \\ \bullet \\ C_1 \quad \dots \quad C_n \end{array}\right) = \bigvee_{1 \leq i \leq n} C_i \leftrightarrow P$$

$$\Phi\left(\begin{array}{c} P \\ \bullet \\ C_1 \quad \dots \quad C_n \end{array}\right) = \bigvee_{1 \leq i \leq n} C_i \leftrightarrow P$$

$$\wedge \bigwedge_{1 \leq i < j \leq n} \neg(C_i \wedge C_j)$$

# CNF as a Universal Formula Language

## Recap: Conjunctive Normal Form

- a **literal**  $L$  is a variable  $X$  or its negation  $\neg X$
- a **clause**  $C$  is a disjunction of literals  $\bigvee_j L_j$
- a **conjunctive normal form (CNF)** is a conjunction of clauses  $\bigwedge_i C_i = \bigwedge_i \bigvee_j L_j$
- intuitively: a set of “rules” to be satisfied
- any formula  $\phi$  can be transformed into a CNF  $\phi'$  that is logically equivalent ( $\phi \Leftrightarrow \phi'$ )

## Recap: Laws of Propositional Logic

- implication:  $\phi \rightarrow \psi \Leftrightarrow \neg\phi \vee \psi$
- biimplication:  $\phi \leftrightarrow \psi \Leftrightarrow (\neg\phi \vee \psi) \wedge (\neg\psi \vee \phi)$
- De Morgan's laws:  $\neg(\phi \wedge \psi) \Leftrightarrow \neg\phi \vee \neg\psi$
- distributivity:  $(\phi \wedge \psi) \vee \chi \Leftrightarrow (\phi \vee \chi) \wedge (\psi \vee \chi)$

## Transforming Part of $\Phi(FM)$ into $CNF(\Phi(FM))$

$C$	$C$
$\wedge (T \rightarrow C)$	$\wedge (\neg T \vee C)$
$\wedge (O \leftrightarrow C)$	$\wedge (\neg O \vee C) \wedge (\neg C \vee O)$
$\wedge (W \vee L \leftrightarrow O)$	$\wedge (\neg(W \vee L) \vee O)$
$\wedge \neg(W \wedge L)$	$\wedge (\neg O \vee W \vee L)$
$C$	$C$
$\wedge (\neg T \vee C)$	$\wedge (\neg T \vee C)$
$\wedge (\neg O \vee C) \wedge (\neg C \vee O)$	$\wedge (\neg O \vee C) \wedge (\neg C \vee O)$
$\wedge ((\neg W \wedge \neg L) \vee O)$	$\wedge (\neg W \vee O) \wedge (\neg L \vee O)$
$\wedge (\neg O \vee W \vee L)$	$\wedge (\neg O \vee W \vee L)$
$\wedge (\neg W \vee \neg L)$	$\wedge (\neg W \vee \neg L)$

# CNF as a Universal Formula Language – DIMACS

$C$   
 $\wedge (\neg T \vee C)$   
 $\wedge (\neg O \vee C) \wedge (\neg C \vee O)$   
 $\wedge (\neg W \vee O) \wedge (\neg L \vee O)$   
 $\wedge (\neg O \vee W \vee L)$   
 $\wedge (\neg W \vee \neg L)$

```
c 1 C
c 2 T
c 3 O
c 4 W
c 5 L
p cnf 5 8
1 0
-2 1 0
-3 1 0 -1 3 0
-4 3 0 -5 3 0
-3 4 5 0
-4 5 0
```

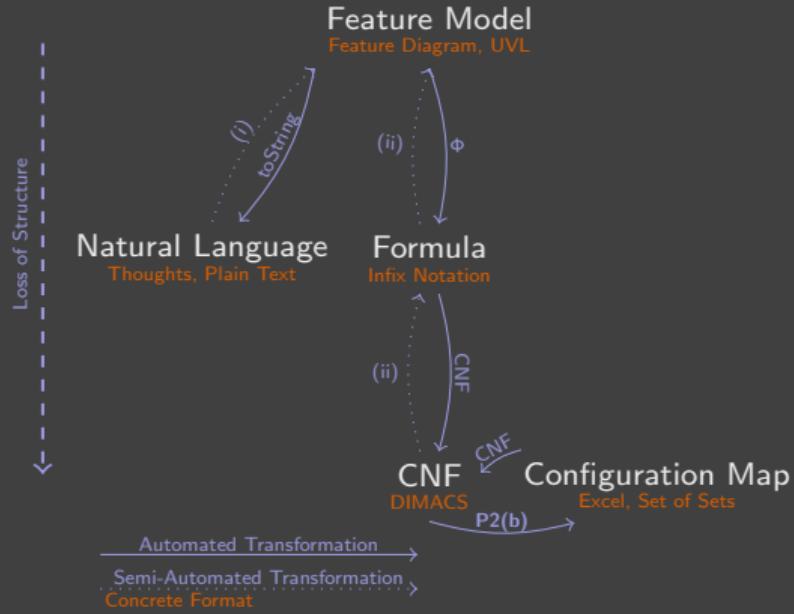
## DIMACS Format

[DIMACS 1993]

- de facto industry standard for storing CNF
- machine-readable, automated analyses, ...
- comments start with c ...
- problem line:  
p cnf #variables #clauses
- clause  $\bigvee_i L_i$  translates to L1 ... Ln 0
- intuitively:

$0 \} \quad \neg \} \quad \text{means} \quad \left\{ \begin{array}{l} \wedge \\ \vee \\ \neg \end{array} \right\}$

# Representations and Transformations



## Problems

- P1 How to express feature models **textually**?
- P2 How to
- validate configurations and
  - get all valid configurations
- automatically**?
- P3 (How to reverse engineer feature models?)

## Solutions

- P1 Universal Variability Language  $\Rightarrow$  **Syntax**
- P2 Propositional Formulas  $\Rightarrow$  **Semantics**
- evaluate feature-model formula
  - Lecture 4c
- P3 (i) e.g., Bakar et al. 2015  
(ii) e.g., Czarnecki and Wasowski 2007

# Transforming Feature Models – Summary

## Lessons Learned

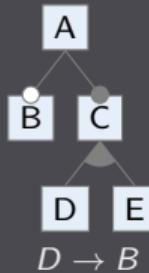
- to understand large configuration spaces, we need formal semantics and machine-readable representations
- propositional formulas satisfy many (though not all) needs for such a representation

## Further Reading

- Don Batory (2005): Feature Models, Grammars, and Propositional Formulas
- UVL — official website for the Universal Variability Language with examples, grammar, literature pointers
- Alexander Knüppel et al. (2017): Is There a Mismatch Between Real-World Feature Models and Product-Line Research?

## Practice

1. translate the following feature diagram into a propositional formula:



2. check formulas of your colleagues

## 4. Feature Modeling

### 4a. Feature Models and Configurations

### 4b. Transforming Feature Models

### 4c. Analyzing Feature Models

Configurators in the Wild

Automated Analysis of Feature Models

SAT, #SAT, and AllSAT

Consistency, Cardinality, and Enumeration

Feature Model

Features

Partial Configurations

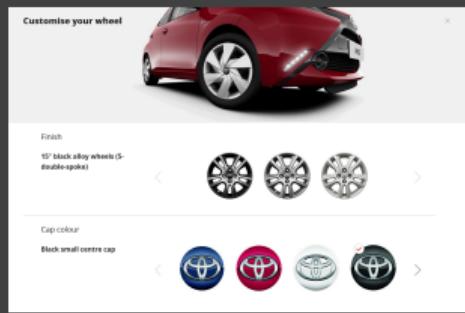
Automated Analyses in FeatureIDE

Summary

FAQ

# Configurators in the Wild – Cars

## Configuring a Car ...



Demo Video

## with a Weird Price

**Your AYGO**  
5 Door Hatchback x-play 1.0 Petrol (69 hp)  
Automatic (Front Wheel Drive - FWD)

Retail price	£11,610.00
Red Pop	
Gleam fabric	
15" black alloy wheels (5-double-spoke)	£500.00
Black small centre cap	
<b>Total</b>	<b>£12,610.00</b>

[Summary and save >](#)

## with 8 Wheels!

**Your customisation**  Change customisation

**Colours & wheels**

Red Pop

15" colour-customisable alloy wheels (5 double-spoke)	£500.00
15" black machined-face alloy wheels (5-double-spoke)	£500.00
Black small centre cap	

**Interior trim**

Gleam fabric

**Total**

£12,610.00

- canceling the dialog was not considered and lead to an invalid state (i.e., configuration)
- humans check these configurations, but some errors are only found during production
- many constraints: appear arbitrary, not explained

# Configurators in the Wild – Cars

## Configuring a German Car

[example from Lecture 1]

Configuration Assistant.

» Show instructions

Your most recent action requires your configuration to be adjusted.

Your choice

+ Enhanced Bluetooth telephone with USB & Voice Control + £ 350.00

Adding

+ BMW Navigation £ 0.00

Removing

- Enhanced Bluetooth with wireless charging - £ 395.00

- Navigation system Professional £ 0.00

- WiFi hotspot preparation £ 0.00

- Media package - Professional - £ 900.00

- Online Entertainment £ 0.00

- Microsoft Office 365 - £ 150.00

Why does the telephone conflict with Microsoft Office?

# Configurators in the Wild – Notebooks

## Configuring a Notebook

### Display

14.0" FHD (1920x1080), LED backlight, 300 nits, 16:9 aspect ratio, 700:1 contrast ratio, 72% gamut, 170° viewing angle, IPS, Touch

SELECTED

14.0" WQHD (2560x1440), LED backlight, 300 nits, 16:9 aspect ratio, 700:1 contrast ratio, 72% gamut, 170° viewing angle, IPS, Touch

+ £91.20

14.0" HDR WQHD (2560x1440) with Dolby Vision™, LED backlight, 500 nits, 16:9 aspect ratio, 1500:1 contrast ratio, 100% gamut, 170° viewing angle, IPS, Touch  
Please note this display is only available with WWAN/mobile broadband.

+ £159.60

### Display

14.0" FHD (1920x1080), LED backlight, 300 nits, 16:9 aspect ratio, 700:1 contrast ratio, 72% gamut, 170° viewing angle, IPS, Touch



14.0" WQHD (2560x1440), LED backlight, 300 nits, 16:9 aspect ratio, 700:1 contrast ratio, 72% gamut, 170° viewing angle, IPS, Touch

+ £91.20

14.0" HDR WQHD (2560x1440) with Dolby Vision™, LED backlight, 500 nits, 16:9 aspect ratio, 1500:1 contrast ratio, 100% gamut, 170° viewing angle, IPS, Touch  
Please note this display is only available with WWAN/mobile broadband

LOADING...

SELECTED

can detect mistakes, but provides no explanations or fixes

# Configurators in the Wild – Notebooks

## Still Configuring a Notebook

### Microsoft Productivity Software

None

SELECTED

Microsoft Office 365 Home

+ £59.99

Microsoft Office 365 Personal

+ £79.99

Microsoft Office Home and Student 2016

+ £119.99

Microsoft Office Home and Business 2016

+ £229.99

Adobe Acrobat Standard 2017 and Microsoft Office Home and Business 2016

+ £399.60

Adobe Acrobat Standard 2017 and Microsoft Office

+ £628.80

### Microsoft Office Not Included

For your best experience, Lenovo recommends selecting a Microsoft Office product with your new purchase.



### NEED HELP DECIDING?

Roll over each product to get specific details on each Office product

allows some feature combinations and not others, prices are opaque

# Automated Analysis of Feature Models

## Open Questions

- How do such configurators work?
- How to avoid inconsistencies?
- How to provide explanations and fixes?
- How to get all valid configurations automatically? (**P2(b)**)

## Automated Analysis of Feature Models

- up until now: **creation** and **transformation** of feature models
- now: **analysis** of feature models to improve our understanding of a configuration space
- for brevity: product = valid configuration

## Asking Questions About Feature Models

- Is a given configuration valid?
- Is there any product at all?  
How many/which products are there?
- Is a given feature (de-)selectable at all?  
How many/which products include it?
- Is a given partial configuration consistent?  
How many/which products include it?
- (Which features always occur together?)
- (Is a given constraint redundant?)
- (How do two feature model versions differ?)
- (Why is ...? How to fix ...?)

# SAT, #SAT, and AllSAT

## Recap: Boolean Satisfiability Problem (SAT)

- **decision problem:** is there any assignment  $A$  that satisfies a given formula?
- formally:  $SAT(\phi) \Leftrightarrow \exists A: \phi(A) = \top$
- known to be **NP-complete**:  
in theory, difficult to solve if  $P \neq NP$ ;  
in practice, solvability depends on domain
- answered by **SAT solvers**:  
highly-optimized, off-the-shelf tools;  
competitively developed over several decades;  
takes a CNF in DIMACS format as input

- $X \rightarrow Y$  is satisfiable
- $X \vee \neg X$  is satisfiable (even a tautology)
- $X \wedge \neg X$  is not satisfiable (why?)

## Sharp Satisfiability Problem (#SAT)

- **counting problem:** how many assignments satisfy a given formula?
- $\#SAT(\phi) = |\{A \mid \phi(A) = \top\}|$
- known to be **#P-complete**:  
at least as hard as SAT (probably harder)
- answered by **#SAT solvers**

## Solution Enumeration Problem (AllSAT)

- **enumeration problem:** which assignments satisfy a given formula?
- $AllSAT(\phi) = \{A \mid \phi(A) = \top\}$
- at least as hard as #SAT (probably harder)
- answered by **AllSAT solvers**

# Automated Analysis of Feature Models

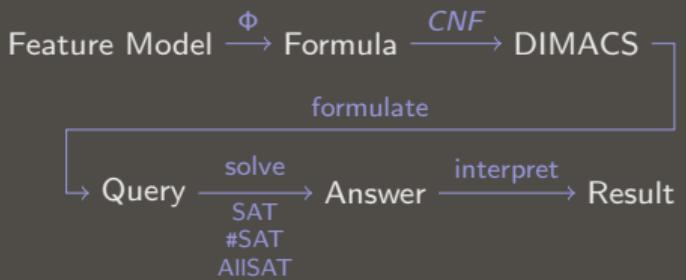
## Asking Questions About Feature Models

- Is a given configuration valid?  $\Rightarrow$  evaluate
- Is there any valid configuration at all?  
How many/which valid configurations are there?
- Is a given feature (de-)selectable at all?  
How many/which valid configurations include it?
- Is a given partial configuration consistent?  
How many/which valid configurations include it?

## Choosing the Right Solver

- “is?”  $\approx$  SAT solver query
- “how many?”  $\approx$  #SAT solver query
- “which?”  $\approx$  AllSAT solver query

## Typical Feature-Model Analysis Process



for brevity, we assume that  $\phi = \text{CNF}(\Phi(FM))$   
for a given feature model  $FM$

# Consistency, Cardinality, and Enumeration – Feature Model

## Consistency of Feature Models (SAT)

### Void/Consistent Feature Model

- are there grave modeling errors?
- is it possible to configure any product at all?

$$\phi \xrightarrow{\text{SAT}} \perp/\top \begin{array}{c} \xrightarrow{\perp} \\[-1ex] \xrightarrow{\top} \end{array} \begin{array}{l} FM \text{ is } \textbf{void} \\ FM \text{ is } \textbf{consistent} \end{array}$$

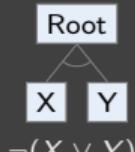
## Cardinality of Feature Models (#SAT)

### How Many Products Are There?

$$\phi \xrightarrow{\#SAT} |C|$$

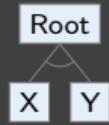
### Variability Factor: Share of Products?

$$\phi \xrightarrow{\#SAT} |C| \longrightarrow \frac{|C|}{2^{|F|}}$$



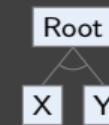
void

$$\neg(X \vee Y)$$



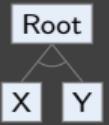
consistent

$$X \vee Y$$



0 products, VF 0

$$\neg(X \vee Y)$$



2 products, VF  $\frac{2}{8}$

$$X \vee Y$$

# Consistency, Cardinality, and Enumeration – Feature Model

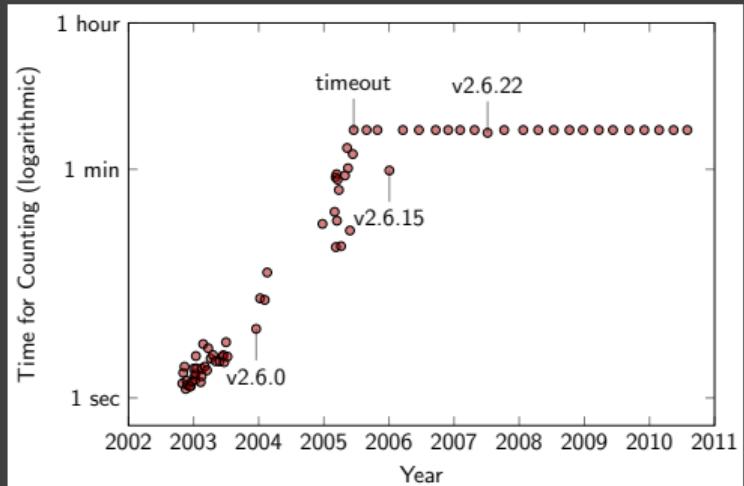
## Feasibility of SAT-Based Analyses

### Is SAT-Based Analysis “Easy”?

- provocative claim: “SAT-based analysis of feature models is easy” [Mendonca et al. 2009]
- easy = performs much better than expected (although NP-complete)
- easy = fast?
  - what about formulating the query? (e.g., CNF transformation)
  - what about many queries? (e.g., what we discuss next)

## Feasibility of #SAT-Based Analyses

### Time to Count Products of Linux



- the solver from 2023 can solve models from 2003
- can we analyze the models from 2023 in 2043?

# Consistency, Cardinality, and Enumeration – Feature Model

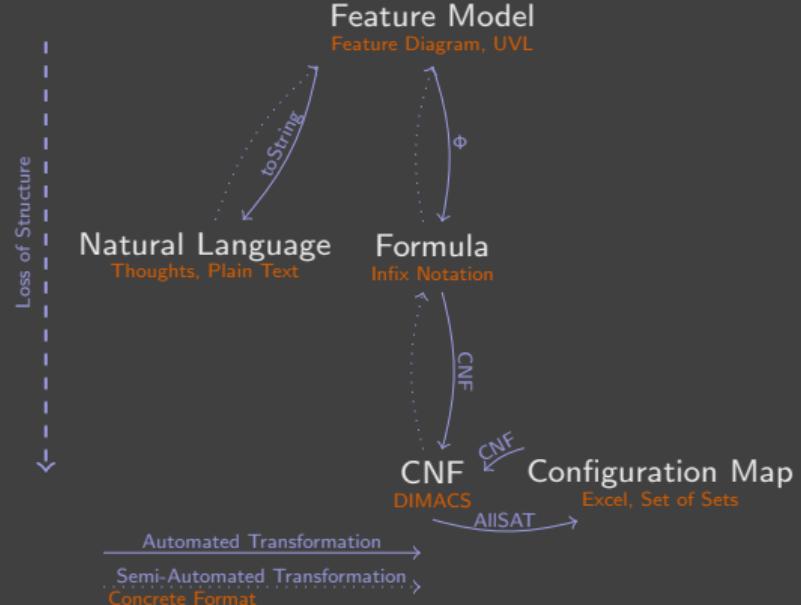
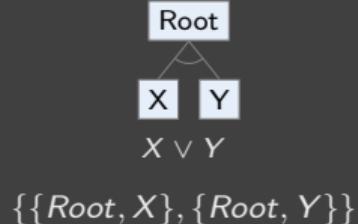
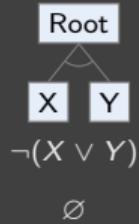
## Enumeration of Feature Models (AllSAT)

### Which Products Are There?

- P2(b): How to get all products?

$$\phi \xrightarrow{\text{AllSAT}} C$$

AllSAT does not scale to realistic feature models!  
50 features, configurations  $\approx 1$  Byte  $\approx 1$  Petabyte



# Consistency, Cardinality, and Enumeration – Features

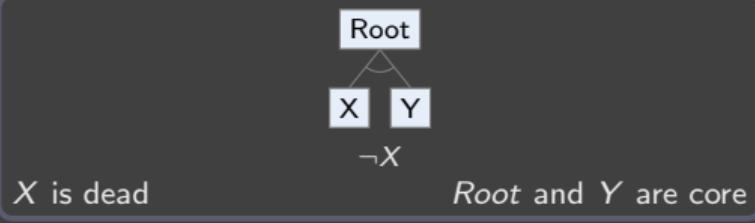
## Consistency of Features (SAT)

### Core/Dead Feature

- can a feature  $F$  be (de-)selected at all?

$$\phi \wedge F \xrightarrow{\text{SAT}} \perp/\top \begin{array}{c} \xrightarrow{\perp} F \text{ is dead} \\ \xrightarrow{\top} F \text{ is not dead} \end{array}$$

$$\phi \wedge \neg F \xrightarrow{\text{SAT}} \perp/\top \begin{array}{c} \xrightarrow{\perp} F \text{ is core} \\ \xrightarrow{\top} F \text{ is not core} \end{array}$$



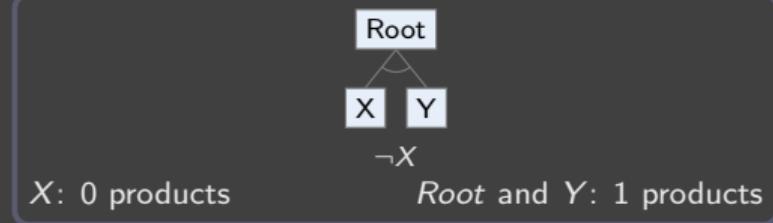
## Cardinality of Features (#SAT)

### How Many Products Include Feature $F$ ?

$$\phi \wedge F \xrightarrow{\#SAT} |\{S \in C \mid F \in S\}|$$

### Commonality: How Constrained is This Feature?

$$\phi \wedge F \xrightarrow{\#SAT} |\{S \in C \mid F \in S\}| \rightarrow \frac{|\{S \in C \mid F \in S\}|}{|C|}$$



# Consistency, Cardinality, and Enumeration – Partial Configurations

## Consistency of Partial Configurations (SAT)

### Valid Partial Configuration

Is a partial configuration  $C = (\mathcal{S}, \mathcal{D})$  consistent with the feature model?

$$\phi \wedge \bigwedge_{s \in \mathcal{S}} s \wedge \bigwedge_{d \in \mathcal{D}} \neg d \xrightarrow{\text{SAT}} \perp / \top \xrightarrow{\perp} C \times \begin{cases} \top \\ \top \end{cases} C \checkmark$$

## Cardinality of Partial Configurations (#SAT)

### How Many Products Are Still Configurable for $C$ ?

$$\phi \wedge \dots \xrightarrow{\#SAT} |\{(S', D') \in C \mid \mathcal{S} \subseteq S', \mathcal{D} \in D'\}|$$

Root



$X \rightarrow Y$

$(\{\text{Root}\}, \{X\}) \checkmark$

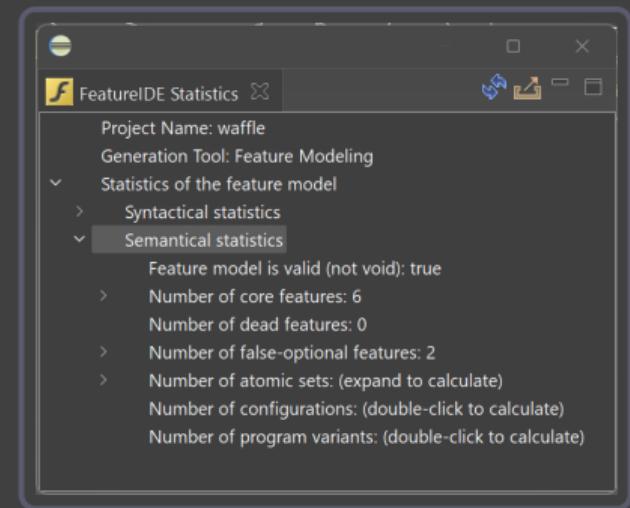
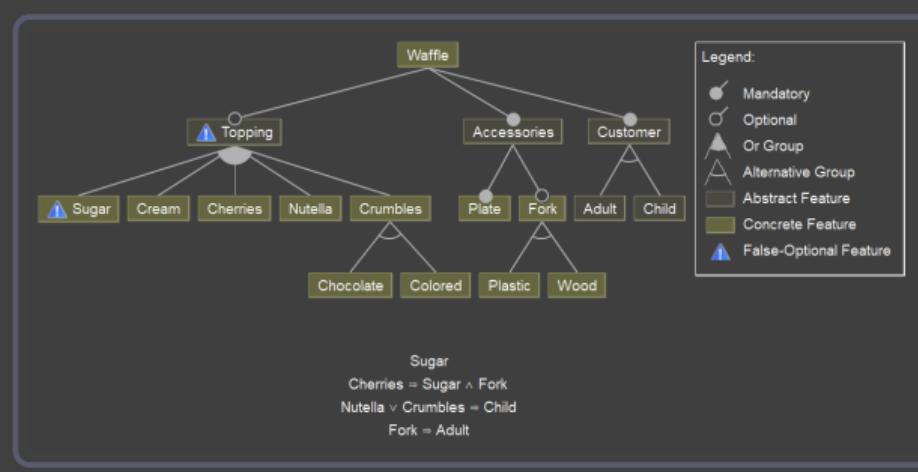
Root



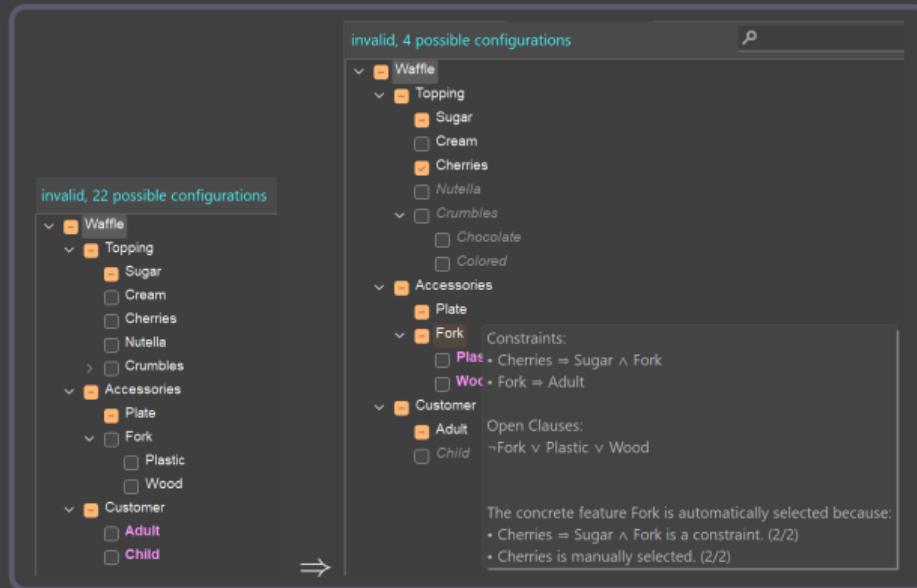
$X \rightarrow Y$

$(\{\text{Root}\}, \{X\})$ : 2 products

# Automated Analyses in FeatureIDE – Feature-Model Editor



# Automated Analyses in FeatureIDE – Configuration Editor

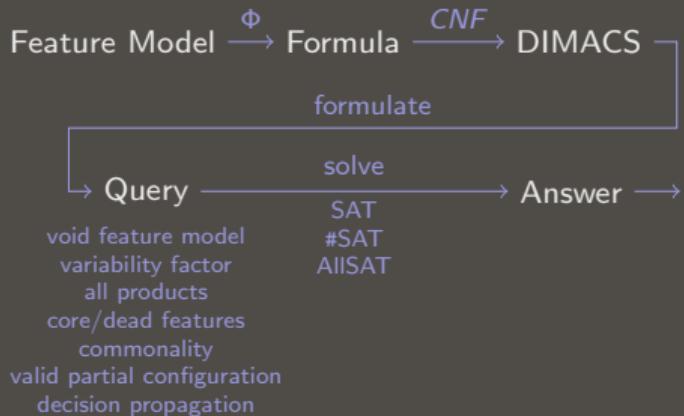


## Decision Propagation

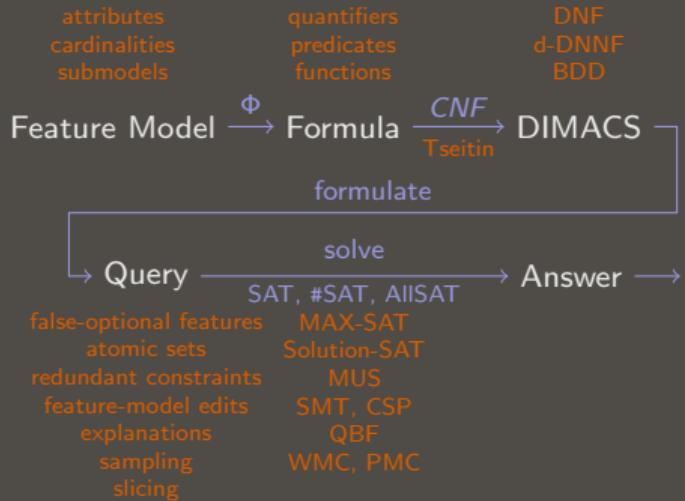
- after each decision (i.e., click) ...
  - ... select features that are now **conditionally core**
  - ... deselect features that are now **conditionally dead**
- this way it is impossible to configure an invalid configuration
- explanations for all propagated decisions

# Automated Analysis of Feature Models

## The Road So Far ...



## ... and Beyond



- develop new analyses
- improve efficiency of existing analyses
- investigate correctness and compositionality

# Analyzing Feature Models – Summary

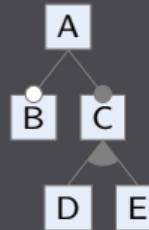
## Lessons Learned

- with solvers, we can build reliable configurators for product lines
- SAT-based analyses: void feature model, core/dead features, decision propagation
- #SAT-based analyses: variability factor, feature commonality

## Further Reading

- Apel et al. 2013, Section 10.1, pp. 244–254
  - introduction to feature-model analysis
- David Benavides et al. (2010): Automated Analysis of Feature Models 20 Years Later: A Literature Review
  - old but extensive literature survey
- Chico Sundermann et al. (2021): Applications of #SAT Solvers on Feature Models
  - experiments on the scalability of #SAT solvers

## Practice



think of a constraint that would make exactly one feature dead

# FAQ – 4. Feature Modeling

## Lecture 4a

- What is feature modeling? When is it needed?
- How can we specify valid combinations of features?
- What is a complete, partial, valid, invalid configuration?
- What are (dis-)advantages of natural language, configuration map, and feature models?
- What is the graphical syntax and semantics of feature models?
- Give an example feature model!

## Lecture 4b

- What representations of feature models are available? Are they equivalent?
- How to represent feature models textually?
- What is UVL (used for)?
- How to identify whether a configuration is valid?
- How to translate feature model into a propositional formula?
- What are DIMACS and KConfig (used for)?
- Would you recommend Excel for feature model? Why (not)?

## Lecture 4c

- Why can configuration become challenging?
- How can we identify problems with feature models and configurations?
- How can feature models be analyzed? What analyses are available?
- What solvers can be used to analyze feature models?
- What is the difference between SAT, #SAT, and ALLSAT?
- Why are solvers useful when creating configurations?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 5a. Features with Build Systems

- How to Implement Features?
- Problems of Ad-Hoc Approaches for Variability
  - Features with Runtime Variability?
  - Features with Clone-and-Own?
- Recap: Clone-and-Own with Build Systems
- Introducing Features to Build Systems
- The Linux Kernel
- Discussion
- Summary

### 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

- Recap: Code Scattering and Tangling
- Feature Traceability Problem
- Feature Traceability with Colors
  - Feature Commander
  - FeatureIDE
- Virtual Separation of Concerns
  - CIDE
- Summary
- FAQ

# 5. Conditional Compilation – Handout

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

## 5. Conditional Compilation

### 5a. Features with Build Systems

How to Implement Features?

Problems of Ad-Hoc Approaches for Variability

- Features with Runtime Variability?

- Features with Clone-and-Own?

Recap: Clone-and-Own with Build Systems

Introducing Features to Build Systems

The Linux Kernel

Discussion

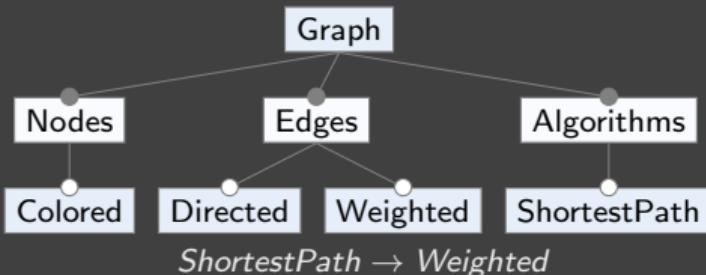
Summary

### 5b. Features with Preprocessors

### 5c. Feature Traceability

# How to Implement Features?

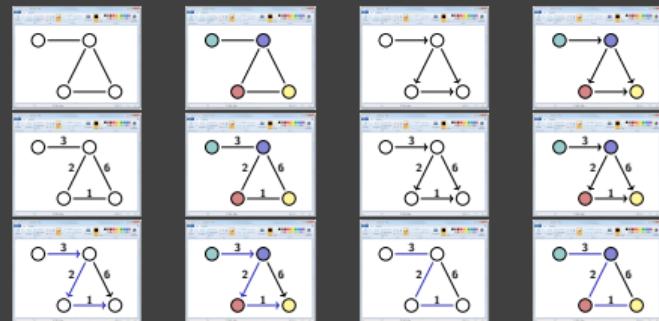
Given a feature model for graphs ...



... 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 Lecture 5 – Lecture 7

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

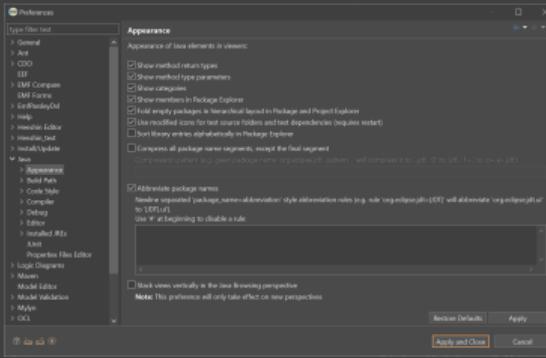
```
public class Config {  
    public static boolean COLORED = true;  
    public static boolean WEIGHTED = false;  
}
```

```
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?



The screenshot shows a Windows Command Prompt window with the following text:

```
C:\>java -f
```

Displays a list of files and subdirectories in a directory.

```
DIR [drive:][[path][filename] [/A[[:] attributes]] [/B] [/C] [/D] [/I] [/M]
     [/O[[:]sortorder]] [/P] [/Q] [/R] [/S] [/T[[:]timefield]] [/W] [/X] [/A]
```

[drive:][[path][filename]] Specifies drive, directory, and/or file to list.

/A Displays files with specified attributes:

attributes	D Directories	R Read-only files
	H Hidden files	A Files ready for archiving
	S System files	I Not content indexed files
	L Registry Points	O Offline files
	M Mount points	N Content indexed files
	P Pruned files	F Filtered files

/B Use bare format (no heading information or summary).

/C Display the thousand separator in file sizes. This is the default. Use '/C' to disable display of separator.

/D Same as wide but files are list sorted by column.

/I New long list where filenames are on the far right. list by files in sorted order.

/M sortorder

By name (alphabetical)	5 By size (smallest first)
0 By extension (alphabetical)	0 By date/time (oldest first)
4 By date/time (newest first)	9 Prefix to reverse order

/P Pauses after each screenful of information.

/Q Display the owner of the file.

/R Display alternate data streams of the file.

/S Displays files in specified directory and all subdirectories.

Press any key to continue . . .

The screenshot shows the 'eclipse.ini' Editor with the following configuration options:

```
eclipse.ini -Editor
```

```
-Dosgi.bundles.Format=Asicht.HFile
-eStartup
plugins/org.eclipse.equinox.launcher_1.5.700.v20200207-2156.jar
--launcher.library
plugins/org.eclipse.equinox.launcher.win32.win32.x86_64_1.1.100.v20190907-0426
--product
org.eclipse.epp.package.modeling.product
--showsplash
org.eclipse.epp.package.common
--launcher.defaultAction
openFile
--launcher.setDefaultAction
openFile
--launcher.appendVmargs
--vmargs
--Dosgi.requiredJavaVersion=1.8
--Dosgi.instance.area.default=user.home/eclipse-workspace
--XX:UseG1GC
--XX:UseStringDeduplication
--add-modules=ALL-SYSTEM
--Dosgi.requiredJavaVersion=1.8
--Dosgi.instanceAreaRequiresExplicitInit=true
-Xms156m
-Xmx2848m
--add-modules=ALL-SYSTEM
```

## 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

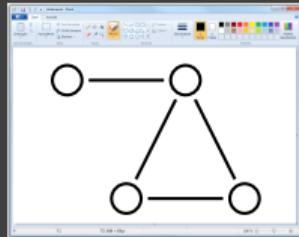
## What is missing?

- 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

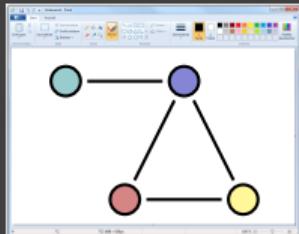
# Problems of Ad-Hoc Approaches for Variability – Features with Clone-and-Own?



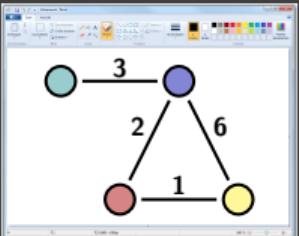
Alice



Bob



Eve



## 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]

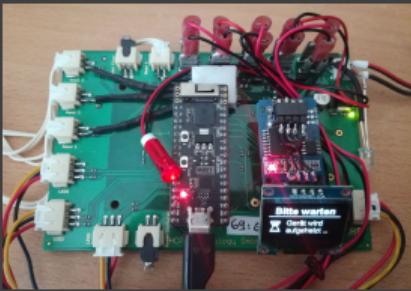
## Case Study: Anesthesia Device

- 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



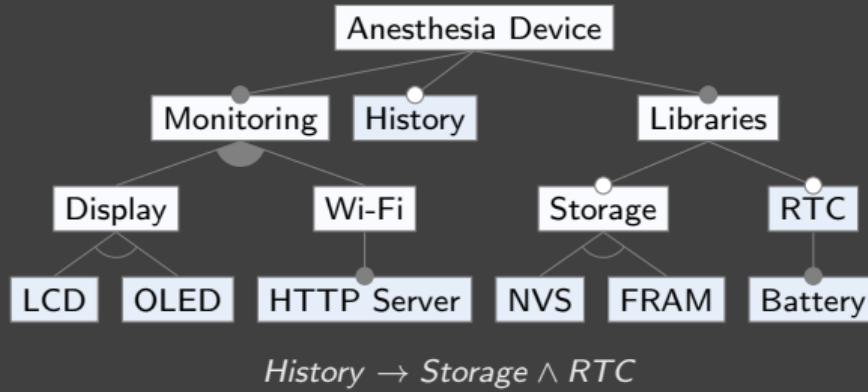
✓	📁	main-variants
✓	📁	config
✓	CFG	cfg_production_oled.mk
✓	CFG	cfg_prototype_lcd_RTC.mk
✓	CFG	cfg_prototype_oled.mk
✓	📁	lib
✓	C	battery.c
✓	C	ds3231.c
✓	C	fram.c
✓	C	i2cdev.c
✓	C	rtc.c
✓	📁	monitor
✓	C	display.c
✓	C	http.c
✓	C	lcd.c
✓	C	oled.c
✓	C	wifi.c
✓	CFG	build.mk
✓	C	history.c
✓	C	main.c

# 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  
⇒ one build script per group of related features



✓	📁 main-features	
✓	📁 lib	
⌚	battery.c	Battery
📦	build.mk	Libraries
⌚	ds3231.c	RTC
⌚	fram.c	FRAM
⌚	i2cddev.c	FRAM
⌚	rtc.c	RTC
✓	📁 monitor	
📦	build.mk	Monitor
⌚	display.c	Display
⌚	http.c	HTTP Server
⌚	lcd.c	LCD
⌚	oled.c	OLED
⌚	wifi.c	Wi-Fi
📦	build.mk	Anesthesia Device
⌚	history.c	History
⌚	main.c	Anesthesia Device

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 "$(_ARCH)" = "x86_64" || "$(_ARCH)" = "aarch64"
```

```
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 v

depends on !64BIT

**select GENERIC\_VDSO\_32**

select ARCH SPLIT ARG64

config X86\_64

def\_bool v

depends on 64BIT

**select ARCH HAS GIGANTIC PAGE**

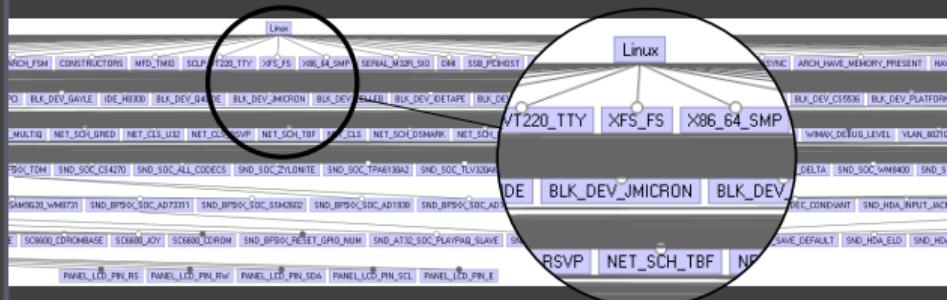
**select ARCH\_SUPPORTS\_INT128 if CC\_HAS\_INT128**

## KConfig Language

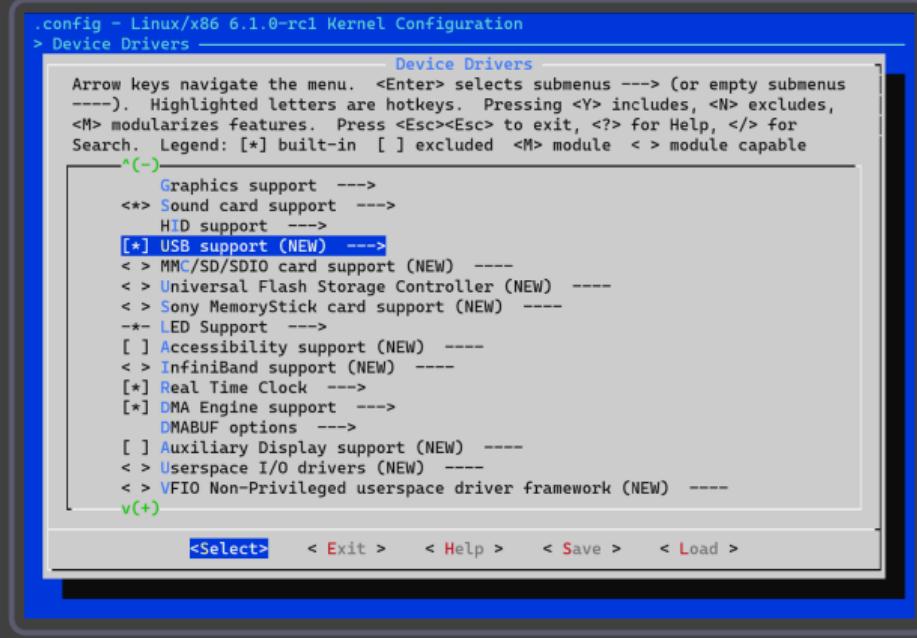
[kernel.org]

- 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 ( $\approx 8\%$ ): static linkage (= include feature)
  - obj-m (< 1%): dynamic linkage (= as module)
  - obj- (0%): no linkage (= exclude feature)
  - obj-\$(...) ( $\approx 91\%$ ): conditional compilation
- full power of Make  $\Rightarrow$  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
```

```
# in the real code, kconfig is (unintuitively?) 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

```
.config - Linux/x86 6.1.0-rc1 Kernel Configuration
> Binary Emulations
    Binary Emulations
        Arrow keys navigate the menu. <Enter> selects submenus
        ---> (or empty submenus ----). Highlighted letters are
        hotkeys. Pressing <Y> includes, <N> excludes, <M>
        modularizes features. Press <Esc><Esc> to exit, <?> for
            [*] IA32 Emulation
            [ ] x32 ABI for 64-bit mode (NEW)
<Select>  < Exit >  < Help >  < Save >  < Load >
```

## 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

## 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 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 inclusion 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

What are differences of the following two implementation techniques for variability?

- (a) clone-and-own with build systems
- (b) features with build systems

## 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 be 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 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]*/                System.out.print("Hello");  
4     System.out.print("Hello");  
5     /*end[Hello]*/  
6     /*if[Beautiful]*/          System.out.print(" beautiful");  
7     System.out.print(" beautiful");  
8     /*end[Beautiful]*/  
9     /*if[Wonderful]*/          System.out.print(" wonderful");  
10    System.out.print(" wonderful");  
11    /*end[Wonderful]*/  
12    /*if[World]*/              System.out.print(" world!");  
13    System.out.print(" world!");  
14    /*end[World]*/  
15  }                          System.out.print(" world!");  
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[]  
3         args) {  
4         //#if Hello  
5         System.out.print("Hello");  
6         //#endif  
7         //#if Beautiful  
8         System.out.print(" beautiful");  
9         //#endif  
10        //#if Wonderful  
11        System.out.print(" wonderful");  
12        //#endif  
13        //#if World  
14        System.out.print(" world!");  
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 Antenna on Command Line

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

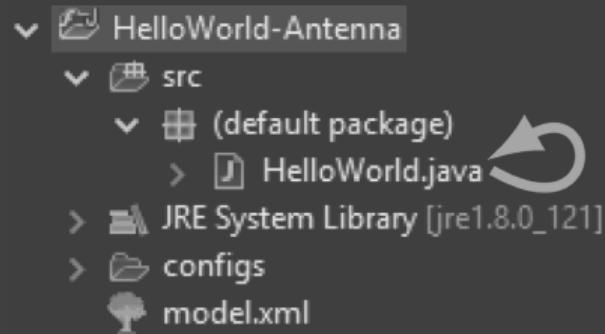
# In-Place and Out-of-Place Preprocessors

[Meinicke et al. 2017]

## In-Place Preprocessor

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

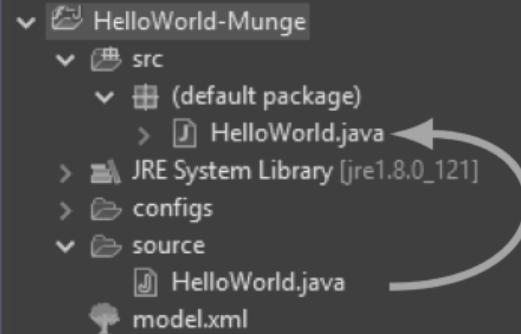
## Antenna, ...



## Out-of-Place Preprocessor

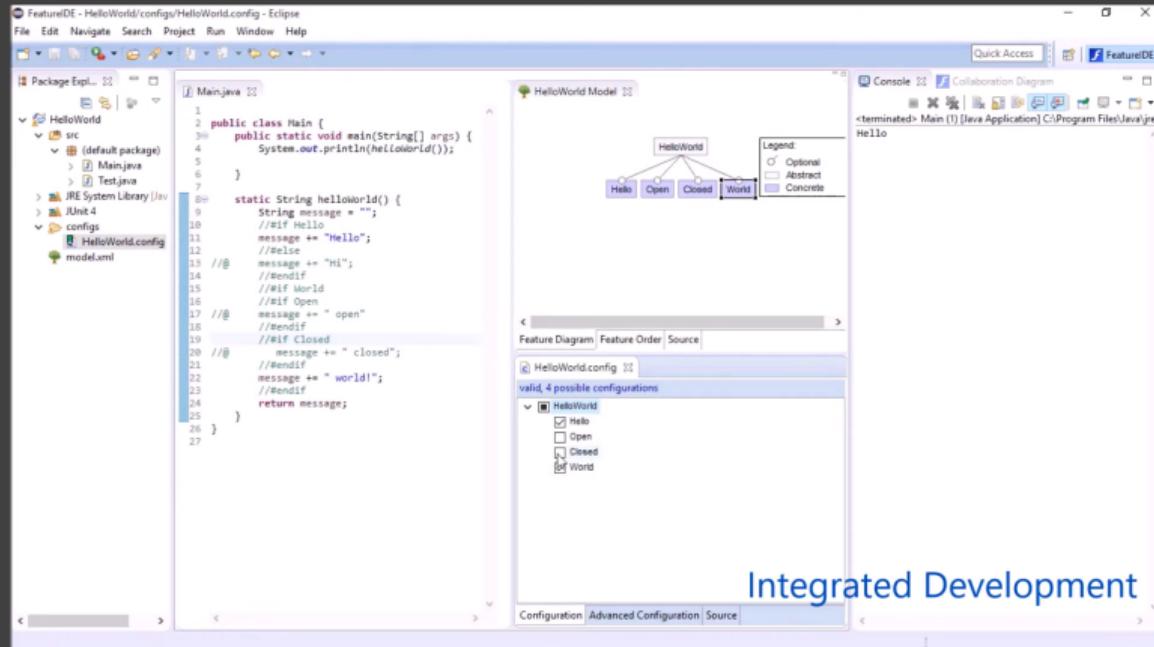
- separate output file generated
- lines deleted where necessary
- often: worse support in IDEs

## 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

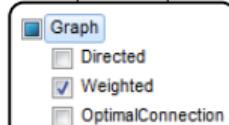
## Powerful Preprocessors

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?

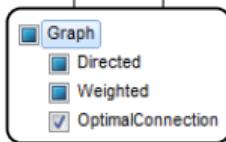
```
class Edge {  
    Node first, second;  
    //ifdef Weighted  
    int weight;  
    //endif  
    Edge(Node first, Node second) {...}  
    boolean equals(Edge e) {  
        return (first == e.first  
            && second == e.first  
            || first == e.second  
            && second == 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));  
    } }
```



```
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

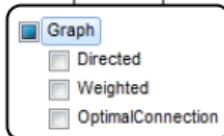
can produce syntactically ill-formed programs:

- 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

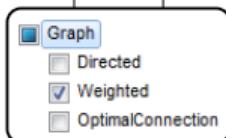
can produce ill-typed programs:

- 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
#endif 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));
#endif USE_ISPTS_FLAG
    IsPts = !result;
}
#endif
#endif
```

# Discussion of Preprocessors

## Advantages

- well-known and mature tools, readily available
- easy to use  
⇒ just annotate and remove
- supports **compile-time variability**
- flexible, arbitrary levels of **granularity**
- can handle code and non-code artifacts (**uniformity**)
- little **preplanning** required  
⇒ variability can be added to an existing project

## Challenges

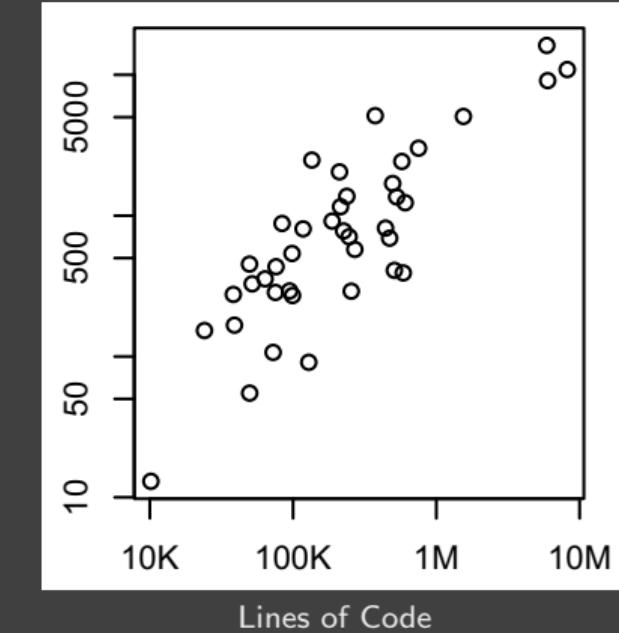
- **scattering** and **tangling**  
⇒ separation of concerns?
- mixes multiple languages in the same development artifact
- may **obfuscate** source code and severely impact its readability
- hard to analyze and process for existing IDEs
- often used in an ad-hoc or **undisciplined** fashion
- prone to subtle syntax, type, or runtime errors which are hard to detect

[Lecture 9–Lecture 11]

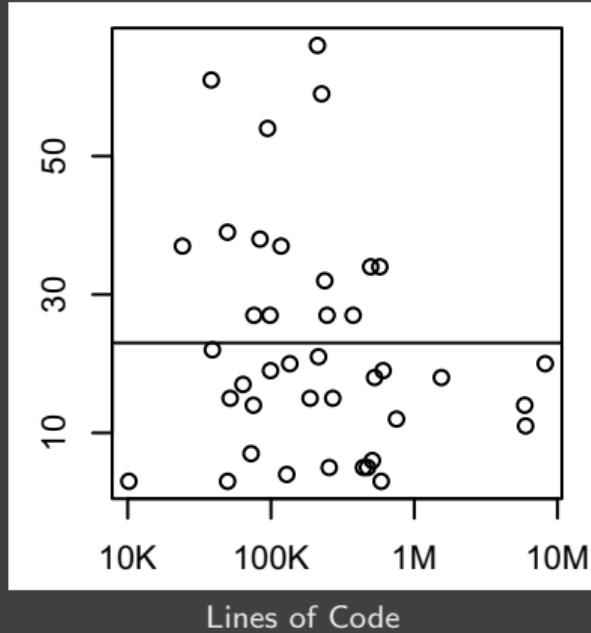
# Preprocessor-Based Product Lines in the Wild

[Liebig et al. 2010]

Number of Features



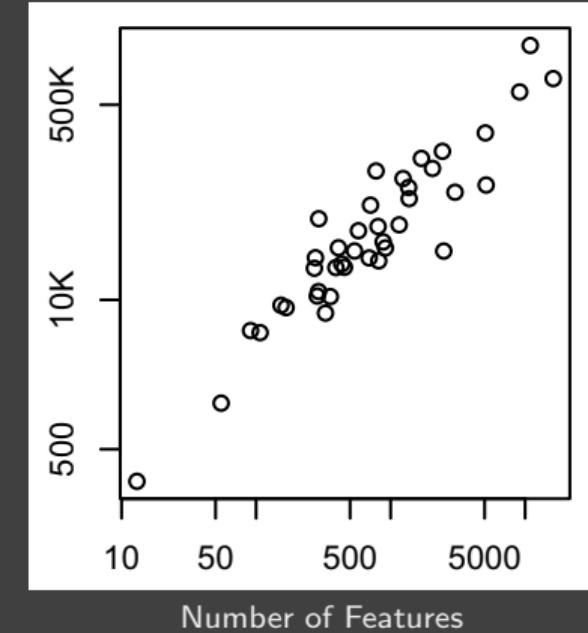
Percentage of Variable Code



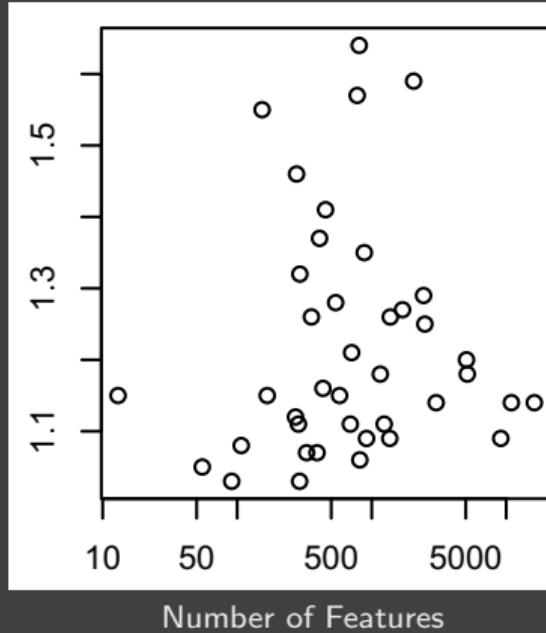
# Preprocessor-Based Product Lines in the Wild

[Liebig et al. 2010]

Lines of Variable Code



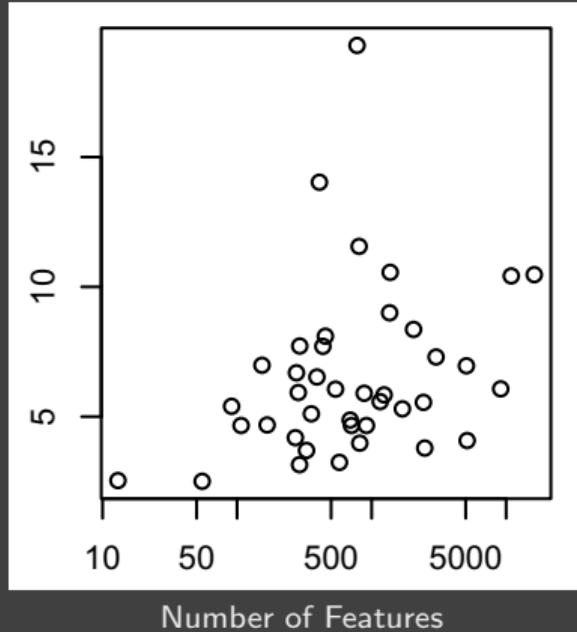
Average Nesting Depth



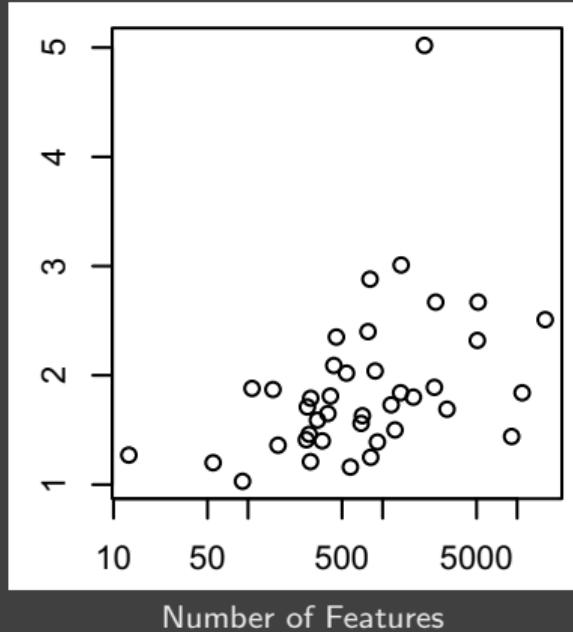
# Preprocessor-Based Product Lines in the Wild

[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

Feature Traceability Problem

Feature Traceability with Colors

  Feature Commander

  FeatureIDE

Virtual Separation of Concerns

  CIDE

Summary

FAQ

# 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 Color {  
    static void  
    setDisplayColor(  
        Color c) {...}  
}
```

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

```
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();  
    }  
}
```

## 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



## 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 gray
- visualizations based on preprocessor directives

## Demo Video

(there is no sound)

The screenshot shows the Feature Commander interface integrated with a code editor. The left side features a tree view of files under 'File Structure' with various nodes colored (blue, yellow, gray) to represent their feature status. The main area displays a snippet of C code with several preprocessor directives (#ifdef, #ifndef) containing feature names like 'CONFIG\_XENO\_OPT\_PIPELINE\_HEAD'. A sidebar on the right contains a 'Colors' palette with a legend for colors (blue, green, yellow, red, purple) and a 'ColorAssignments' section. Callouts provide the following information:

- 'see the file structure and open files by clicking on a node'
- 'get an overview of the feature structure of the current viewport'
- 'see the feature structure for the whole document'
- 'see the feature percentage for each node at one glance'
- 'shows always the inner feature color of the sidebars'
- 'use tooltips to identify features'
- 'click a feature occurrence to navigate to its start'
- 'Machine', 'Interfaces', 'Drivers'
- 'save and load color assignments with two clicks'

```
if (sched->ched_migrate)
    sched->ched_migrate(thread, sched);
/* * WARNING: the scheduling class may have just changed as a
 * result of calling the per-class migration hook.
 */
xsched_set_reached(sched->ched);
thread->ched = sched;

if (!xthread_set_state(thread, XTHREAD_BLOCK_BITS)) {
    xsched_requeue(thread);
    xthread_set_state(thread, XREADY);
}

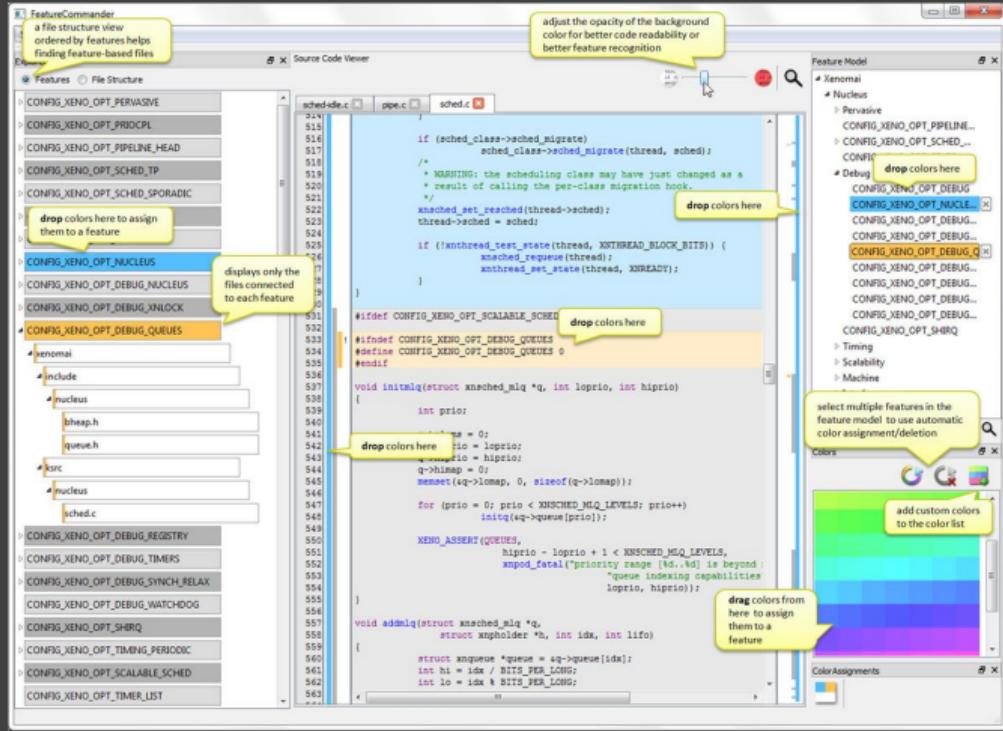
#endif CONFIG_XENO_OPT_SCALABLE_SCHED

#ifndef CONFIG_XENO_OPT_DEBUG_QUEUES
#define CONFIG_XENO_OPT_DEBUG_QUEUES 0
#endif

void initmq(struct xsched_mq *q, int loprio, int hiprio)
{
    int prio;
    q->elems = 0;
    CONFIG_XENO_OPT_NUCLEUS |= loprio;
    q->hiprio |= hiprio;
    q->imap = 0;
    memset(q->lomap, 0, sizeof(q->lomap));
    for (prio = 0; prio < XNSCHED_MQ_LEVELS; prio++)
        initq(&q->queue[prio]);
    XENO_ASSERT(QUEUES,
                hiprio - loprio + 1 < XNSCHED_MQ_LEVELS,
                msg_fatal("priority range [%d..%d] is beyond max queue indexing capabilities",
                          loprio, hiprio));
}

void addmq(struct xsched_mq *q,
           struct amholder *h, int idx, int lifo)
{
    struct xqueue *queue = q->queue[idx];
    int hi = idx / BITS_PER_LONG;
    int lo = idx % BITS_PER_LONG;
```

# 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

The screenshot shows the FeatureIDE interface. On the left is a package explorer with a tree view of the 'Elevator-Antenna-v1.4' project, including source code and resource files. The main area is a code editor for 'Request.java'. The code uses color coding to trace features:

- Red highlights appear around lines 72 and 77, which define a class 'RequestComparator' and its constructor.
- Yellow highlights appear around lines 84 and 88, which implement the 'compare' method for different sorting strategies.
- Green highlights appear around lines 91 and 92, which calculate differences between floor values.

The status bar at the bottom indicates the code is 'Writable' and shows a 1:1 ratio.

## 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?

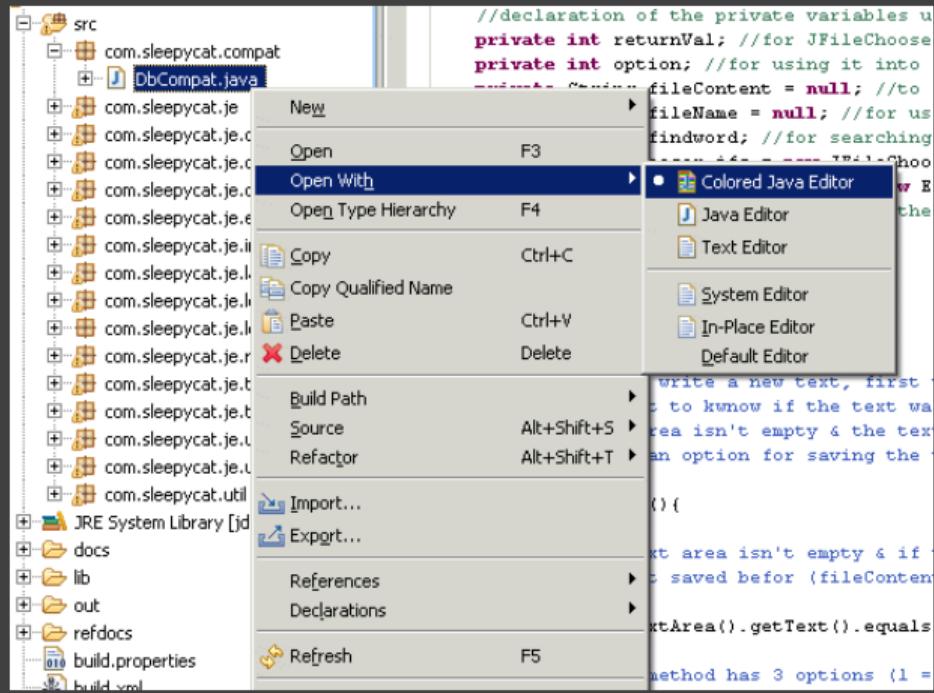
- features physically separated from each other
- dedicated components, services, plug-ins

[Lecture 6]

- dedicated modules, folders, files

[Lecture 7]

# Virtual Separation of Concerns – CIDE [CIDE]



## What is CIDE?

- stands for **Colored Integrated Development Environment**
- 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]

The screenshot shows the CIDE IDE interface. On the left is a code editor with tabs for Notepad.java, Actions.java, and Main.java. The code in Main.java is annotated with various colors: green for the first if-block, yellow for the second, orange for the third, blue for the fourth, and red for the fifth. The code is as follows:

```
        output.setText(t.toString());
        program.setText(t.eval(""));
        equation.setText(base);
        updateQuarkPanel();
    }
}
apply.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (hoa.isSelected()) {
            t = t.apply(new hoa("h" + layerno));
        }
        if (ladvice.isSelected()) {
            t = t.apply(new advice("a" + layerno));
        }
        if (intro.isSelected()) {
            t = t.apply(new intro("i" + layerno));
        }
        if (gadvice.isSelected()) {
            t = t.apply(new gadvice("g" + layerno));
        }
        if (hoa.isSelected() || gadvice.isSelected()
            || ladvice.isSelected() || intro.isSelected())
            hoa.setSelected(false);
            gadvice.setSelected(false);
            ladvice.setSelected(false);
            intro.setSelected(false);
        equation.setText("(" + layerno + ")" + equation.g
            + ")");
    }
});
```

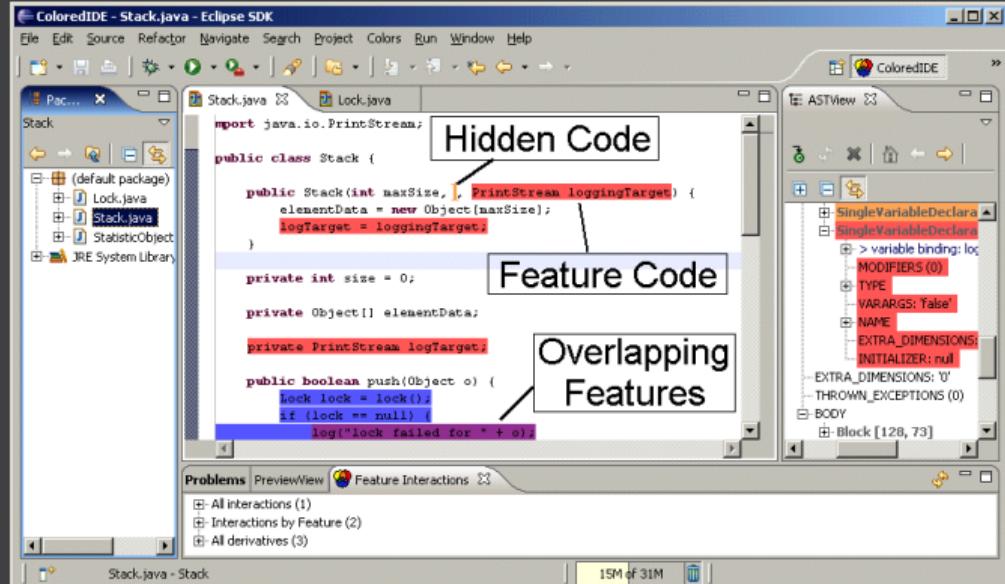
To the right is an "Outline" view titled "ASTView". It displays the Abstract Syntax Tree (AST) structure of the selected code. The tree nodes are color-coded according to their type:

- OC: null (grey)
- IERS (1) (grey)
- RUCTOR: 'false' (grey)
- PARAMETERS (0) (grey)
- N\_TYPE2 (grey)
- ETERS (1) (grey)
- \_DIMENSIONS: '0' (grey)
- VN\_EXCEPTIONS (0) (grey)
- lock [4440, 805] (grey)
- STATEMENTS (5)
  - IfStatement [4450, 73]
    - EXPRESSION
      - MethodInvocation [4454, 16]
    - THEN\_STATEMENT
    - ELSE\_STATEMENT: null
  - IfStatement [4529, 80]
  - IfStatement [4615, 77]
  - IfStatement [4690, 81]
  - IfStatement [4785, 457]

## 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]

```
model.colors Test.java BaseMessaging.java MediaController.java 2
59 public class MediaController extends MediaListController {
60
61     private MediaData media;
62     private NewLabelScreen screen;
63
64     public MediaController (MainUIMidlet midlet, AlbumData albumData,
65                           super(midlet, albumData, albumListScreen);
66 )
67
68     public boolean handleCommand(Command command) {
69         String label = command.getLabel();
70         System.out.println ("<* PhotoController.handleCommand() *> "
71
72         /** Case: Save Add photo */
73         if (label.equals("Add")) {
74             ScreenSingleton.getInstance().setCurrentScreenName(Consi
75             AddMediaToAlbum form = new AddMediaToAlbum("Add new item");
76             form.setCommandListener(this);
77             setCurrentScreen(form);
78             return true;
79         }
80     }
81     // #ifdef includePhotoAlbum
82     // [NC] Added in the scenario 07
83     if (label.equals("View")) {
```

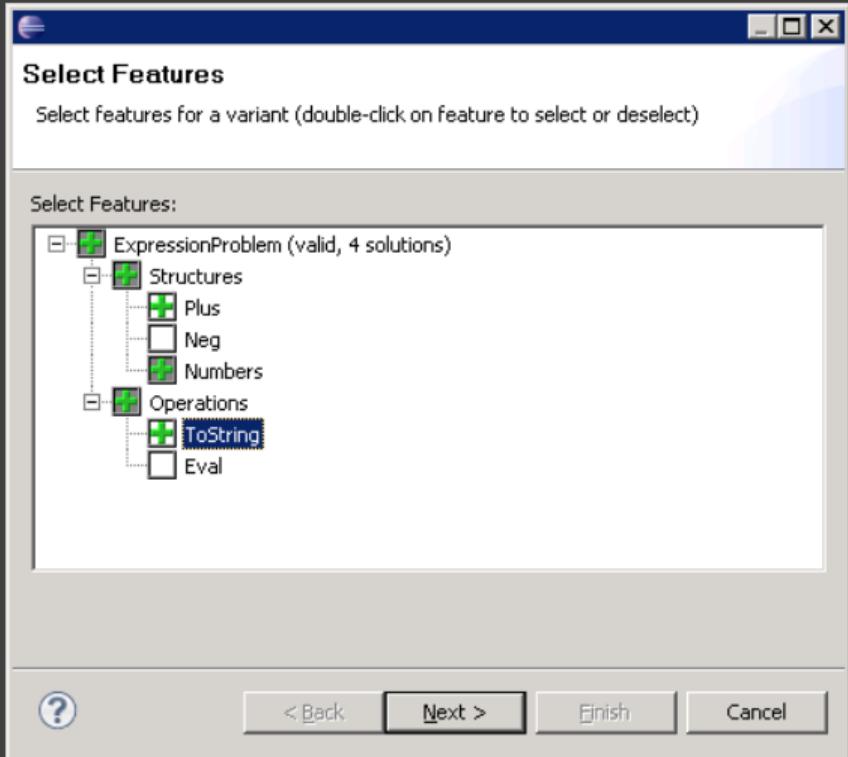
```
model.colors Test.java BaseMessaging.java MediaController.java 2
59 public class MediaController extends MediaListController {
60
61     public boolean handleCommand(Command command) {
62         /** Case: Save Add photo */
63         // #ifdef includePhotoAlbum
64         // [NC] Added in the scenario 07
65         //##ifdef
66         // #ifdef includeMusic
67         // [NC] Added in the scenario 07
68         if (label.equals("Play")) {
69             String selectedMediaName = getSelectedMediaName();
70             return playMultiMedia(selectedMediaName);
71
72         /** Case: Add photo */
73         }
74         //##endif
75         if (label.equals("Save Item")) {
76             try {
77                 // #ifdef includeMusic
78                 // [NC] Added in the scenario 07
79                 if (getAlbumData() instanceof MusicAlbumData) {
80                     getAlbumData().loadMediaDataFromRMS(getCurrentS
81                     MediaData mymedia = getAlbumData().getMediaInfo(
82                     MultiMediaData mmedi = new MultiMediaData(mymed
83                     getAlbumData().updateMediaInfo(mymedia, mmedi);
84                 }
85             } //##endif
86
87         }
88     }
89
90     // #endif
91     // [NC] Added in the scenario 07
92 }
```

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

# Virtual Separation of Concerns – CIDE [CIDE]

## Why configuration?

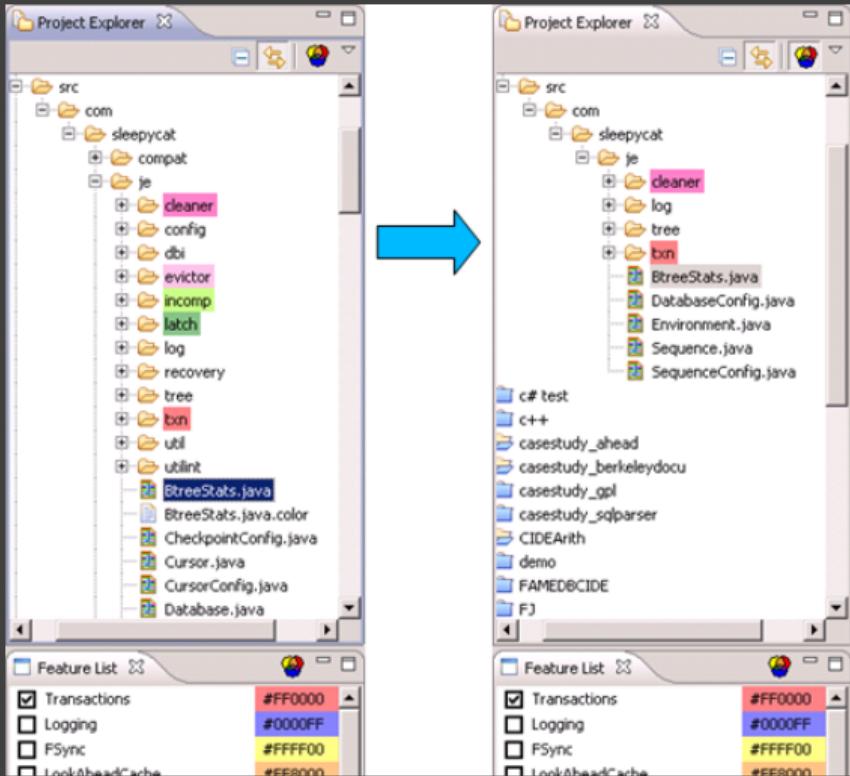
- features specified in FeatureIDE feature model
- configuration created in FeatureIDE configuration editor
- configuration used to generate and visualize variant
- ...



# 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?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. **Modular Features**
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 6a. Components

- Recap: How to Implement Features?
- Modularity
- Software Components
- Example: A Color Component
- Component-Based Product Lines
- Discussion of Components
- Summary

### 6b. Services and Microservices

- (Micro-)Services
- Services vs. Components
- Microservice Architectures
- Implementation of Product Lines
- Service Composition
- Summary

### 6c. Frameworks with Plug-Ins

- Hot Spots and Plug-Ins
- Preplanned Extensions
- Basic Design Principles
- Plug-In Loading and Management
- Frameworks in the Wild
- Implementation of Product Lines
- Discussion
- Summary
- FAQ

# 6. Modular Features – Handout

Software Product Lines | Timo Kehrer, Thomas Thüm, Elias Kuiter | May 26, 2023

# **6. Modular Features**

## **6a. Components**

Recap: How to Implement Features?

Modularity

Software Components

Example: A Color Component

Component-Based Product Lines

Discussion of Components

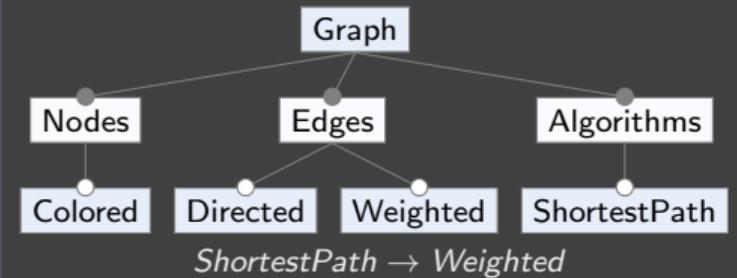
Summary

## **6b. Services and Microservices**

## **6c. Frameworks with Plug-Ins**

# Recap: How to Implement Features?

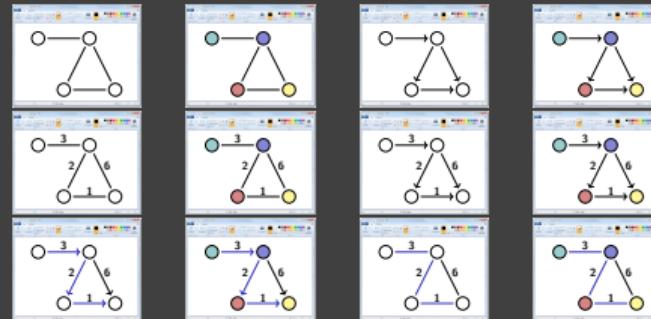
Given a feature model for graphs ...



... 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 Lecture 5 – Lecture 7

# Recap: Features with Build Systems

✓	📁 main-features	
✓	📁 lib	
C	battery.c	Battery
🔗	build.mk	Libraries
C	ds3231.c	RTC
C	fram.c	FRAM
C	i2cdev.c	FRAM
C	rtc.c	RTC
✓	📁 monitor	
🔗	build.mk	Monitor
C	display.c	Display
C	http.c	HTTP Server
C	lcd.c	LCD
C	oled.c	OLED
C	wifi.c	Wi-Fi
🔗	build.mk	Anesthesia Device
C	history.c	History
C	main.c	Anesthesia Device

## Conditional Compilation with Build Systems

- exploit the expressiveness of a build system's configuration language
- include and exclude individual files or entire directories based on feature selection

## Major Challenges

[Lecture 5]

- 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 inclusion and exclusion of individual lines or chunks of code  
⇒ **high-level use only!**

# Recap: Features with 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  
            || first == e.second  
            && second == e.first  
            ) && weight == e.weight;  
//##ifndef Directed  
    }  
    void testEquality() {  
        Node a = new Node();  
        Node b = new Node();  
        Edge e = new Edge(a, b);  
        Assert.assertTrue(e.equals(e));  
    } }
```

## Conditional Compilation with Preprocessors

- use conditional compilation facilities provided by preprocessors
- annotate and potentially remove code fragments based on feature selection

## Major Challenges

[Lecture 5]

- **scattering** and **tangling**  
⇒ 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]

# Modularity

## Modularization

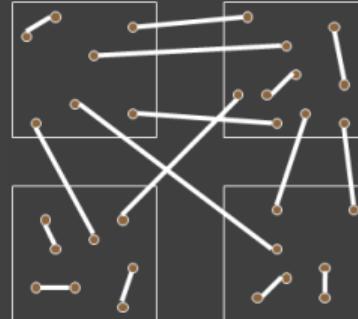
consistent application of **information hiding** and **data encapsulation** to achieve:

- strong logical connection between the inner parts of a module (high cohesion)
- precisely defined, minimal interfaces (low coupling)

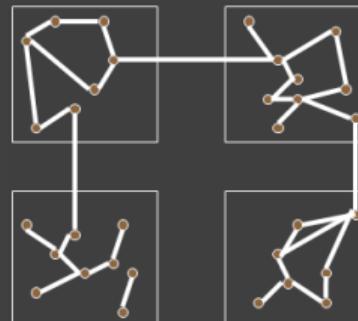
## Coupling and Cohesion

- **cohesion**: measure of how well the parts of a module work together  
⇒ intra-module communication
- **coupling**: measure of the complexity of communication across modules  
⇒ inter-module communication

## High Coupling, Low Cohesion



## Low Coupling, High Cohesion



# Why Modularity?

## Traditional Reasons

- modules can be developed independently of each other (collaborative work)
- easier to maintain because changes can be made locally
- data encapsulation promotes stability and reliability
- software is easier to understand
- hiding complexity behind interfaces
- decomposition = divide and conquer

## Modularization and Software Product Lines

- **reuse**: parts of the software can be *reused*
- **alternatives**: modules can be *exchanged by alternative implementations*
- **variability**: modules can be *reassembled in a new context* (e.g., in other projects)

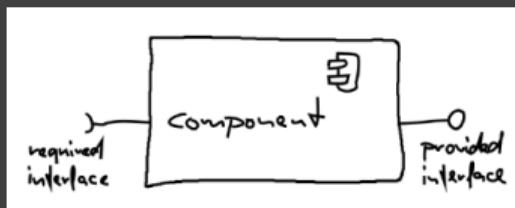


# Components

## Component

[Szyperski 2002]

A software component is a unit of composition with contractually specified **interfaces** and explicit **context dependencies** only. A software component can be deployed independently and is **subject to composition** by third parties.



## Context/Deployment Dependencies

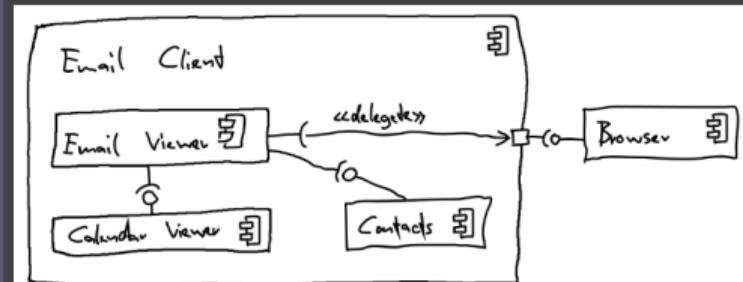
typically container or middleware (e.g., JavaEE, CORBA, OSGi, etc.)

## Composition and Reuse

Components . . .

- are composed with other components to form software systems
- are supposed to be re-usable in other software systems
- may stem from third-party vendors: markets for components, make-or-buy-decisions

## UML Component Diagrams



# Components vs. Objects/Classes

## Commonalities

Numerous similar principles:

- encapsulation and information hiding
- accessibility through public interfaces
- (de-)composition and nested objects/components
- etc.

## Differences

- **objects are smaller** than components by focusing on detailed implementation problems (components aim at abstracting from implementation details)
- **object are less cohesive and stronger coupled** than components due to (deliberately) delegating lots of responsibilities to other objects whereas components aim at maximizing cohesion and minimizing coupling
- **classes are reused through inheritance and polymorphism** whereas components are reused by being integrated into a component architecture

# Example: A Color Component

[Apel et al. 2013]

## A Reusable Component in Java

- assume that storing and printing colors is non-trivial (e.g., in our graph library)
- implement color management as a reusable component, using Java's visibility mechanism to enforce encapsulation

```
package components.color;

// public API
public class ColorComponent {
    public Color createColor(int r, int g, int b) { /* ... */ }
    public void printColor(Color color) { /* ... */ }
    public void mapColor(Object o, Color c) { /* ... */ }
    public Color getColor(Object o) { /* ... */ }

    // just one component instance
    public static ColorComponent getInstance() { return instance; }
    private static ColorComponent instance = new ColorComponent();
    private ColorComponent() { super(); }
}

public interface Color { /* ... */ }

// hidden implementation
class ColorImpl implements Color { /* ... */ }
class ColorPrinter { /* ... */ }
class ColorMapping { /* ... */ }
```

# Example: A Color Component

```
public class Graph {  
    private List<Node> nodes = new ArrayList<Node>();  
    public Node add() {  
        Node n = null;  
        if (Config.COLORED) {  
            Color c = ColorComponent.getInstance().createColor(0, 0, 0);  
            n = new Node(nodes.size(), c);  
        } else {  
            n = new Node(nodes.size());  
        }  
        nodes.add(n);  
        return n;  
    }  
    // ...  
}
```

```
public class Node {  
    private int id;  
    private ColorComponent colorComp =  
        ColorComponent.getInstance();  
    public Node(int id) { this.id = id; }  
    public Node(int id, Color c) {  
        this(id);  
        colorComp.mapColor(this, c);  
    }  
    public void print() {  
        if (Config.COLORED) {  
            Color c = colorComp.getColor(this);  
            colorComp.printColor(c);  
        }  
        System.out.print(id);  
    }  
}
```

- we can **reuse** the color component when implementing the color feature for the graph library and also for other applications
- however: we need to write custom code to connect our implementation with the component  
⇒ **glue code**

# Component-Based Product Lines

## General Idea

- every feature is implemented by a dedicated component
- feature selection determines which components shall be integrated to form an application

## Vision



## Reality



## Glue Code and Customization

- developers must connect components through glue code  
exception: components are only exchanged against alternative components with identical interface
- components may contain run-time variability  
e.g., color manager in our example may be parameterized by color model RGB or CMYK

# The Library Scaling Problem

## Decomposition and Reuse



(Tangram)

### What is the optimal size of a component?

- large components (vertical scaling): typically not widely reusable
- small components (horizontal scaling): limited payoff for component integrators

### Practical Compromise

- mostly vertically-scaled components within a few important, narrow domains (e.g., user interface construction systems)
- in other words: there is no general market for arbitrary components, but a few specialized market segments

# Discussion of Components

## Advantages

- supports **compile-time variability**
- modular implementation with reduced scattering and tangling (compared to runtime variability and preprocessors)

## Challenges

- requires **glue code** for every product:  
clone-and-own for glue code? glue code with runtime variability?
- no automated generation based on feature selection
- requires preplanning to identify good level of granularity (cf. library scaling problem)
- no support for fine-granular variability  
⇒ often combined with runtime variability within components

# Components – Summary

## Lessons Learned

- Modularity = information hiding and data encapsulation
- Components foster a modular software architecture and design
- Reuse within and beyond product lines
- No automated product derivation, glue code is necessary

## Further Reading

- Szyperski 2002
- Apel et al. 2013, Chapter 4.4

## Practice

- Why is feature modeling relevant for component-based product lines?
- How can product-line engineering help to find the right trade-offs regarding the library scaling problem?

## 6. Modular Features

### 6a. Components

### 6b. Services and Microservices

(Micro-)Services

Services vs. Components

Microservice Architectures

Implementation of Product Lines

Service Composition

Summary

### 6c. Frameworks with Plug-Ins

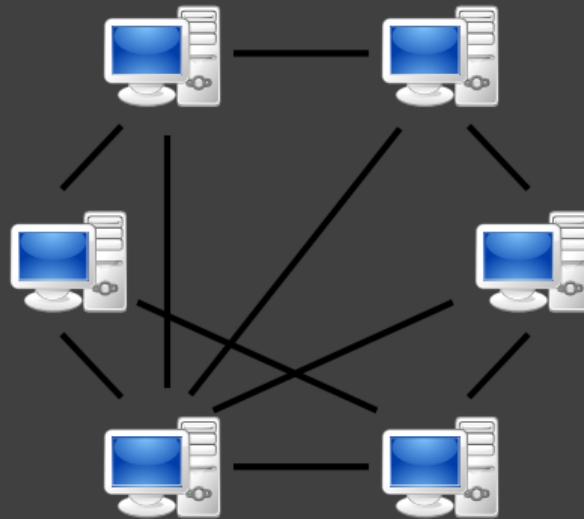
# (Micro-)Services

## (Micro-)Service

“[A (micro-)service is a module] implemented and operated as a small yet independent system, offering access to its internal logic and data through a well-defined network interface.” Jamshidi et al., 2018

## (Micro-)Service Architecture

“A [micro]service is a cohesive, independent process interacting via messages. A microservice architecture [service-oriented architecture] is a distributed application where all its modules are microservices.” Dragoni et al., 2017



# Services vs. Components

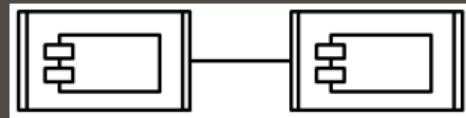
## Component

Intra-process communication (i.e., method calls)



## Service

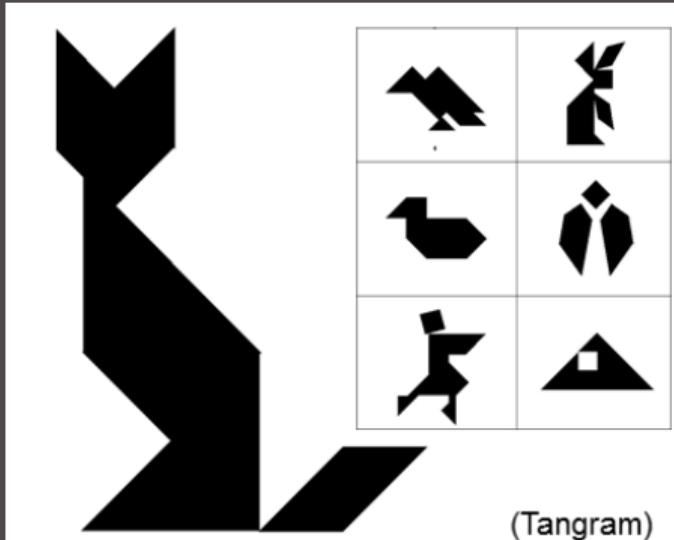
Inter-process communication (e.g., REST API)



As a consequence, each (micro-)service can be implemented using different technology stacks, whereas components are bound to the same technology (given by container or middleware).

# Microservice Architecture: Motivation

## Recap: The Library Scaling Problem



### How small is a Microservice?

- Components are very unspecific of how to deal with the general requirement “not too small but not too big”.
- On the contrary, there is a clear philosophy behind microservice architectures, largely driven by organizational constraints wrt. agile teams and continuous delivery.

“If the codebase is too big to be **managed by a small team**, looking to break it down is very sensible. [...] The smaller the service, the more you **maximize the benefits and downsides** of microservice architecture.” Sam Newman, 2015

# Microservice Architecture: Philosophy and Principles

## Conway's Law

"Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations." Melvin Edward Conway, 1968

## Single Responsibility Principle

"Gather together the things that change for the same reasons. Separate those things that change for different reasons." Robert C. Martin, 2014

"You build it, you run it." Amazon CTO Werner Vogel, 2006

## Consequences

- Microservices are supposed to be split along business capabilities (e.g., purchase, sale, ...) instead of technical concerns (e.g., UI, persistence, ...)
- Each microservice is built (full stack) *and* operated by a small agile team that takes over full responsibility (cf. DevOps)

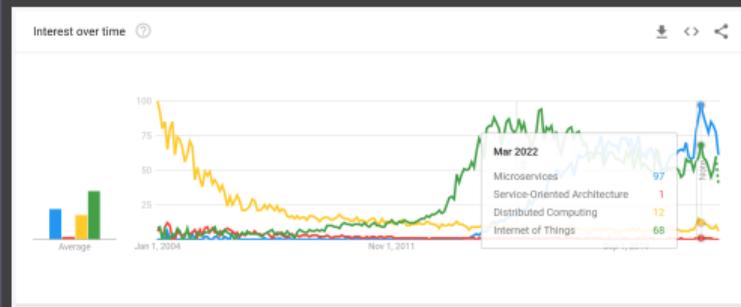
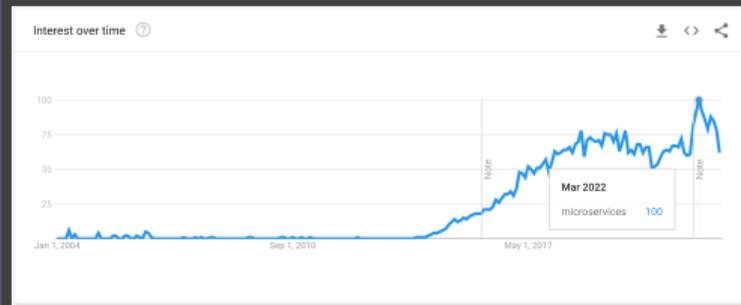
"[A microservice] could be re-written in two weeks." Jon Eaves, 2014

"Every team should be small enough that it can be fed with two pizzas." Jeff Bezos, 2018

# Traditional Promises of Microservices

- **Scalability:** Microservices are small enough to be developed by a small, agile team.
- **Continuous integration/deployment:** Microservices can be deployed independently of each other.
- **Heterogeneity:** Each microservice can be implemented using its own technology stack.
- **Fault tolerance:** The crash of a single microservice should not lead to a crash of the entire system.
- **Efficiency:** Configuration of execution environment can be optimized per microservice.
- **Modernization:** A microservice can be easily replaced by an alternative one (even re-implemented from scratch).

## The Microservice “Hype”



# Implementation of Product Lines

**Recap: Component-Based Implementation**



**Plenty of Glue Code**



**Same Idea**

- Features are implemented as services.
- Feature selection determines the services to be composed.

**However**

"Standardized" service composition instead of highly individual glue code.

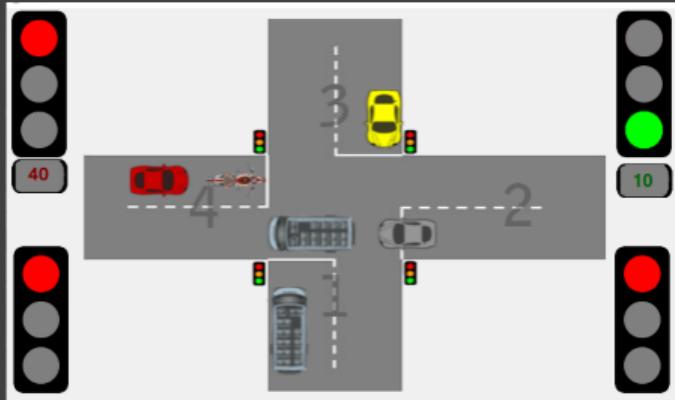


# Service Composition

## Orchestration

Description of an executable (business-)process as a combination of services (centralized perspective).

Web Services Business Process Execution Language (WS-BPEL)



## Choreography

Each service describes its own task within a service composition (decentralized perspective).

Web Services Choreography Description Language (WS-CDL)



# Services and Microservices – Summary

## Lessons Learned

- Services are another kind of module implemented and operated independently of each other
- Microservices have a clear philosophy regarding their size, driven by organizational constraints
- Reuse within and beyond product lines
- No automated product derivation, orchestration or choreography is necessary

## Further Reading

- Apel et al. 2013, Chapter 4.4

## Practice

- We have talked a lot about promises of microservices. Do you also see any drawbacks?
- In a component-based product-line implementation, practitioners often rely on clone-and-own for glue code. How could we handle variability in service orchestrations?

## 6. Modular Features

### 6a. Components

### 6b. Services and Microservices

### 6c. Frameworks with Plug-Ins

Hot Spots and Plug-Ins

Preplanned Extensions

Basic Design Principles

Plug-In Loading and Management

Frameworks in the Wild

Implementation of Product Lines

Discussion

Summary

FAQ

# Framework with Plug-Ins

## Framework and Hot Spot

[Apel et al. 2013, pp. 80–81]

A **framework** is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuse at a larger granularity than classes. A framework is open for **extension** at explicit **hot spots** (aka. extension point).

## Plug-In

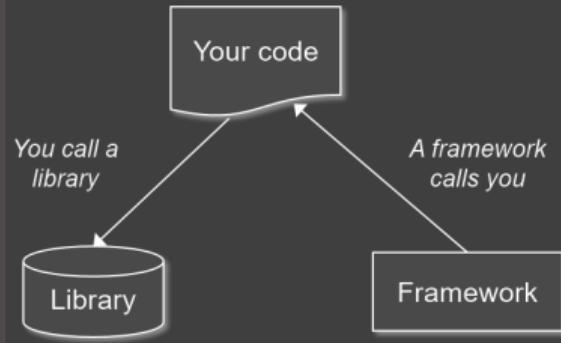
[Apel et al. 2013, pp. 80–81]

A **plug-in** extends hot spots of a [...] framework with custom behavior. A plug-in can be separately compiled and deployed.

Frameworks with plug-ins are also called **black-box frameworks**: Developers need to understand interfaces, but not the internal framework implementation.

## Inversion of Control

Hollywood principle: “Don’t call us, we call you”



- Can be understood in terms of the observer and/or strategy pattern: The framework exposes explicit hot spots, at which plug-ins can register observers and strategies.
- Requires **preplanning** for possible future extensions

## Real-World Example: Preplanned Bike Extensions



bike lock



front wheel brake



rear wheel brake



kickstand

## Real-World Example: Preplanned Bike Extensions



+



=

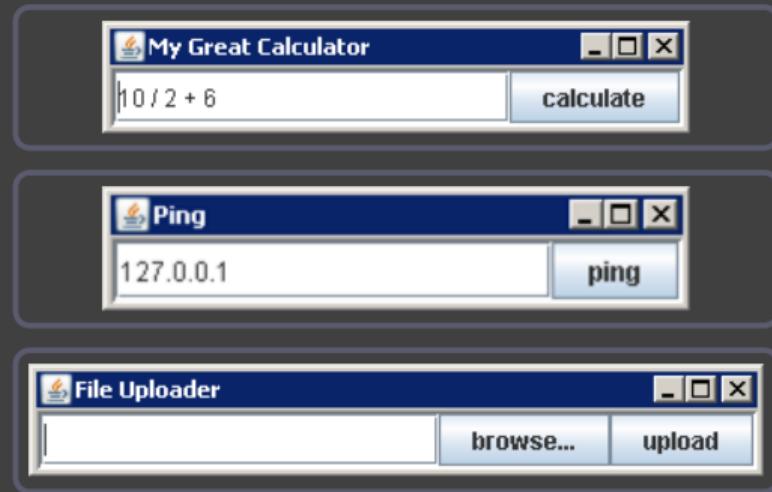


framework with  
extension points

plug-ins

framework with plug-ins

# Simple Java Example: A Family of Dialogs [Apel et al. 2013]



- All dialogs have a similar structure (basically textfield + button)
- Large parts of the source code are identical:
  - Main method,
  - Initialization of windows, textfield and button
  - layouting,
  - window closing and disposal,
  - etc.

# Simple Java Example: A Family of Dialogs

```
public class Calc extends JFrame {  
    private JTextField textfield;  
    public static void main(String[] args) {  
        new Calc().setVisible(true);  
    }  
    public Calc() { init(); }  
    protected void init() {  
        JPanel p = new JPanel(new BorderLayout());  
        p.setBorder(new BevelBorder(/* ... */));  
        JButton button = new JButton();  
        button.setText("calculate");  
        p.add(button, BorderLayout.EAST);  
        textfield = new JTextField("");  
        textfield.setText("10 / 2 + 6");  
        textfield.setPreferredSize(new Dimension(350, 40));  
        p.add(textfield, BorderLayout.WEST);  
        button.addActionListener(new ActionListener()  
            { /* calculate */});  
        this.setContentPane(p);  
        this.setTitle("My Great Calculator");  
        this.pack();  
        // ...  
    }  
}
```

```
public class Ping extends JFrame {  
    private JTextField textfield;  
    public static void main(String[] args) {  
        new Calc().setVisible(true);  
    }  
    public Calc() { init(); }  
    protected void init() {  
        JPanel p = new JPanel(new BorderLayout());  
        p.setBorder(new BevelBorder(/* ... */));  
        JButton button = new JButton();  
        button.setText("ping");  
        p.add(button, BorderLayout.EAST);  
        textfield = new JTextField("");  
        textfield.setText("127.0.0.1");  
        textfield.setPreferredSize(new Dimension(350, 40));  
        p.add(textfield, BorderLayout.WEST);  
        button.addActionListener(new ActionListener()  
            { /* calculate */});  
        this.setContentPane(p);  
        this.setTitle("Ping");  
        this.pack();  
        // ...  
    }  
}
```

# Simple Java Example: A Family of Dialogs

plug-in implementation hidden from application

```
public class Application extends JFrame {  
    private Plugin plugin;  
    // ...  
    public Application(Plugin plugin) {  
        this.plugin = plugin;  
        plugin.setApplication(this);  
        init();  
    }  
    protected void init() {  
        JPanel p = new JPanel(new BorderLayout());  
        p.setBorder(new BevelBorder(/*...*/));  
        JButton button = new JButton();  
        button.setText(plugin.getButtonText());  
        p.add(button, BorderLayout.EAST);  
        JTextField textfield = new JTextField("");  
        textfield.setText(plugin.getInitialText());  
        textfield.setPreferredSize(new Dimension(200, 20));  
        p.add(textfield, BorderLayout.WEST);  
        button.addActionListener(/*... plugin.buttonClicked(); ...*/);  
        this.setContentPane(p);  
        this.setTitle(plugin.getApplicationTitle());  
        // ...  
    }  
    public String getInput() {  
        return textfield.getText();  
    }  
}
```

```
public interface Plugin {  
    String getApplicationTitle();  
    String getButtonText();  
    String getInitialText();  
    void buttonClicked();  
    void setApplication(Application app);  
}
```

```
public class CalcPlugin implements Plugin {  
    private Application application;  
  
    public String getApplicationTitle() {  
        return "My Great Calculator";  
    }  
    public String getButtonText() {  
        return "calculate";  
    }  
    public String getInitialText() {  
        return "10 / 2 + 6";  
    }  
    public void buttonClicked() {  
        calculate(application.getInput());  
    }  
    public void setApplication(Application app) {  
        application = app;  
    }  
    private void calculate(String expression) {  
        /* calculate */  
    }  
}
```

# Simple Example: A Family of Dialogs

application implementation hidden from plug-in

```
public class Application extends JFrame implements InputProvider {
    private Plugin plugin;
    // ...
    public Application(Plugin plugin) {
        this.plugin = plugin;
        plugin.setInputProvider(this);
        init();
    }
    protected void init() {
        JPanel p = new JPanel(new BorderLayout());
        p.setBorder(new BevelBorder(/*...*/));
        JButton button = new JButton();
        button.setText(plugin.getButtonText());
        p.add(button, BorderLayout.EAST);
        JTextField textfield = new JTextField("");
        textfield.setText(plugin.getInitialText());
        textfield.setPreferredSize(new Dimension(200, 20));
        p.add(textfield, BorderLayout.WEST);
        button.addActionListener(/* ... plugin.buttonClicked(); ... */);
        // ...
    }
    public String getInput() {
        return textfield.getText();
    }
}
```

```
public interface InputProvider {
    public String getInput();
}
```

```
public interface Plugin {
    String getApplicationTitle();
    String getButtonText();
    String getInitialText();
    void buttonClicked();
    void setInputProvider(InputProvider p);
}
```

```
public class CalcPlugin implements Plugin {
    private InputProvider inputProvider;

    public String getApplicationTitle() { /*...*/ }
    public String getButtonText() { /*...*/ }
    public String getInitialText() { /*...*/ }
    public void buttonClicked() {
        calculate(inputProvider.getInput());
    }
    public void setInputProvider(InputProvider p) {
        inputProvider = p;
    }
    private void calculate(String expression) {
        /* calculate */
    }
}
```

# Plug-In Loading and Management

## Simple Example vs. Reality

simplification in our previous example:

- a single extension point supporting the registration of a single plug-in
- plug-in implementation known at compile-time

typical requirements in practice:

- an extension point typically supports the registration of arbitrarily many plug-ins
- a single plug-in may extend several extension points
- a plug-in may add new extension points to the framework (framework of frameworks)
- plug-in implementation provided by third parties

## Plug-In Loader

- Searches in a dedicated directory for DLL/JAR/XML files
- Tests whether file implements a plug-in
- Checks dependencies
- Initializes plug-ins

## Plug-In Manager

GUI and/or console interface for plug-in administration and configuration

# Example: Plug-In Loading and Management

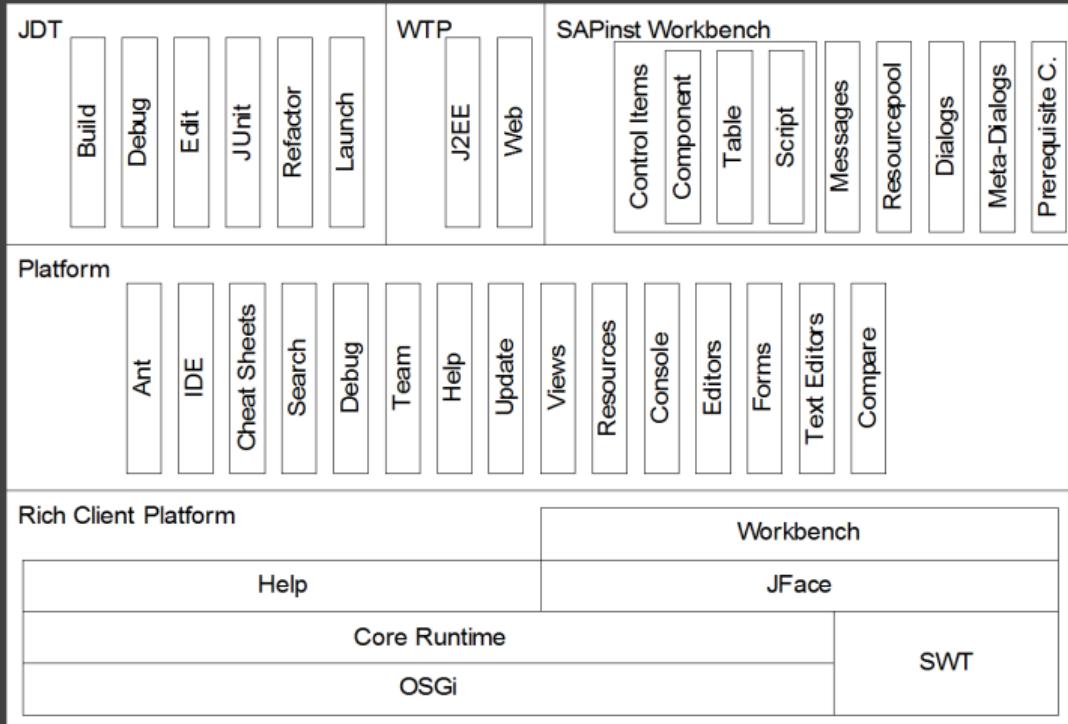
## Plug-In Loader using Java Reflection

```
public class Starter {  
    public static void main(String[] args) {  
        if (args.length != 1)  
            System.out.println("Plugin name not specified");  
        else {  
            String pluginName = args[0];  
            try {  
                Class pluginClass = Class.forName(pluginName);  
                new Application((Plugin)  
                    pluginClass.newInstance()).setVisible(true);  
            } catch (Exception e) {  
                System.out.println("Cannot load plugin " +  
                    pluginName + ", reason: " + e);  
            }  
        }  
    }  
}
```

## Handling multiple Plug-Ins

```
public class Application {  
    private List<Plugin> plugins;  
  
    public Application(List<Plugin> plugins) {  
        this.plugins = plugins;  
        for (Plugin plugin : plugins) {  
            plugin.init();  
        }  
    }  
  
    public Message processMsg(Message msg) {  
        for (Plugin plugin : plugins) {  
            msg = plugin.process(msg);  
            // ...  
        }  
        return msg;  
    }  
}
```

# Frameworks in the Wild: Eclipse



## Versatile IDE

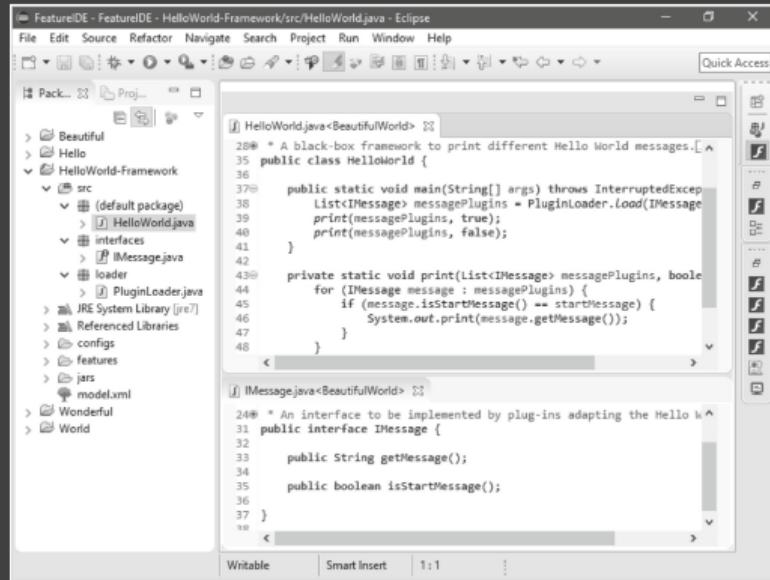
- Lots of common functionality required by any IDE (e.g., editors, incremental project build, etc.)
- Only language-specific extensions need to be registered (e.g., syntax highlighting, compiler, etc.)

## Specifically in Eclipse

- Actually a set of (recursively nested) frameworks
- Largely declarative description of extension points

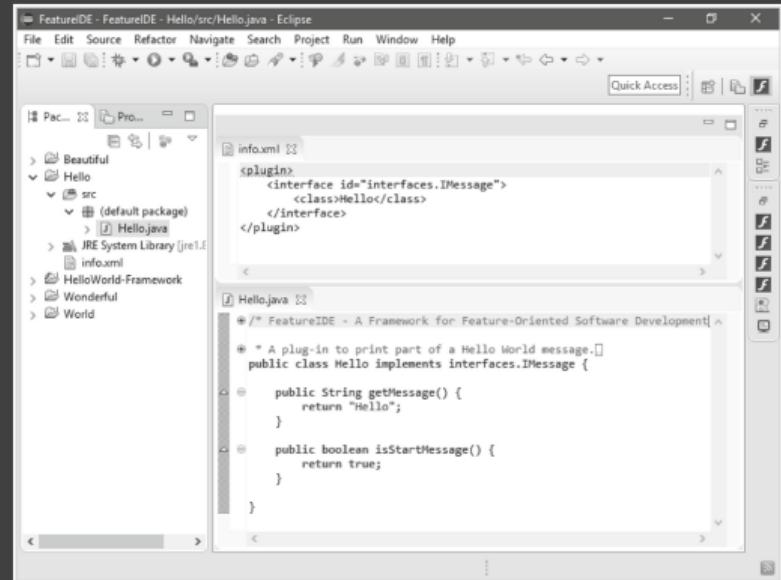
# Frameworks in the Wild: Eclipse

[Meinicke et al. 2017]



The screenshot shows the Eclipse IDE interface with two code editors open. The left editor displays `HelloWorld.java` which contains a main method that prints different messages based on a list of plugins. The right editor displays `IMessage.java`, an interface with methods for getting a message and checking if it's a start message.

```
20 * A black-box framework to print different Hello World messages.
21
22
23 public class HelloWorld {
24
25     public static void main(String[] args) throws InterruptedException {
26         List<IMessage> messagePlugins = PluginLoader.load(IMessage.class);
27         print(messagePlugins, true);
28         print(messagePlugins, false);
29     }
30
31     private static void print(List<IMessage> messagePlugins, boolean
32         for (IMessage message : messagePlugins) {
33             if (message.isStartMessage() == startMessage) {
34                 System.out.print(message.getMessage());
35             }
36         }
37     }
38
39     /**
40      * An interface to be implemented by plug-ins adapting the Hello World
41      */
42     public interface IMessage {
43
44         public String getMessage();
45
46         public boolean isStartMessage();
47     }
48 }
```



The screenshot shows the Eclipse IDE interface with two code editors open. The left editor displays `info.xml`, which defines a plugin with a specific ID and class. The right editor displays `Hello.java`, a class that implements the `IMessage` interface, returning "Hello" as the message and `true` as the start message.

```
<plugin>
    <interface id="interfaces.IMessage">
        <class>Hello</class>
    </interface>
</plugin>
```

```
/*
 * FeatureIDE - A Framework for Feature-Oriented Software Development
 */
public class Hello implements interfaces.IMessage {

    public String getMessage() {
        return "Hello";
    }

    public boolean isStartMessage() {
        return true;
    }
}
```

## Frameworks in the Wild: Further Examples

- Other IDEs (e.g., IntelliJ)
- Unit test frameworks (e.g., JUnit)
- Frontend frameworks (e.g., React, Angular)
- Backend frameworks (e.g., Spring Boot, Ruby on Rails, Django)
- Multimedia frameworks (e.g., DirectX)
- Raster/vector graphics editors (e.g., Adobe Photoshop, MS Visio)
- Instant messenger frameworks (e.g., Miranda, Trillian)
- Compiler frameworks (e.g., LLVM, Polyglot, abc, JustAddJ)
- Web browsers (e.g., Firefox)
- E-Mail clients (e.g., Thunderbird)
- etc.

# Implementation of Product Lines

## Recap: Service-Based Implementation



Still needs some specification of “composition” (cf. orchestration vs. choreography)



## Same Idea

- Features are implemented by different plug-ins
- Feature selection determines the plug-ins to be loaded and registered

neither glue code nor explicit service composition required



full automation comes at a price (cf. preplanning problem)

# Example: Extending Basic Graphs by Plug-Ins?

```
public class Graph {  
    private List<GraphPlugin> plugins = new ArrayList<GraphPlugin>();  
    // ...  
    public void registerPlugin(GraphPlugin p) {  
        plugins.add(p);  
    }  
    public void addNode(int id, Color c){  
        Node n = new Node(id);  
        notifyAdd(n, c);  
        nodes.add(n);  
    }  
    public void print() {  
        for (Node n : nodes) {  
            notifyPrint(n);  
            // ...  
        }  
        // ...  
    }  
    private void notifyAdd(Node n, Color c) {  
        for (GraphPlugin p : plugins) {  
            p.aboutToAdd(n, c);  
        }  
    }  
    private void notifyPrint(Node n) {  
        for (GraphPlugin p : plugins) {  
            p.aboutToPrint(n);  
        }  
    }  
    // ...  
}
```

```
public interface GraphPlugin {  
    public void aboutToAdd(Node n, Color c);  
    public void aboutToAdd(Edge e, Weight w);  
    public void aboutToPrint(Node n);  
    public void aboutToPrint(Edge e);  
}
```

```
public class ColorPlugin implements GraphPlugin {  
    private Map<Node, Color> map = new HashMap<Node, Color>();  
  
    public void aboutToAdd(Node n, Color c) {  
        map.put(n, c);  
    }  
  
    public void aboutToAdd(Edge e, Weight w) {  
        // do nothing  
    }  
  
    public void aboutToPrint(Node n) {  
        Color c = map.get(n);  
        Color.setDisplayColor(c);  
    }  
  
    public void aboutToPrint(Edge e) {  
        // do nothing  
    }  
}
```

# Discussion

In our example, we can observe that:

- There are lots of empty methods in the ColorPlugin
- The Framework consults all registered plug-ins before printing a node or edge

## General Challenge: Cross-cutting Concerns

Implementing cross-cutting concerns as plug-ins

- typically leads to huge interfaces, large parts of which are irrelevant for a dedicated plug-in
- causes lots of communication overhead between plug-ins and framework

If we were not familiar with our graph library, would we anticipate that:

- Colors and weights should be part of the Plugin interface?
- Every plug-in needs to be notified that the framework is about to print a node or edge?

## Generally known as Preplanning Problem

- Hard to identify and foresee the relevant hot spots and nature of extensions
- Developing a framework needs lots of expertise and excellent domain knowledge

# Frameworks with Plug-Ins – Summary

## Lessons Learned

- A framework is open to extension by integrating plug-ins at explicit hot spots.
- The framework takes full control over all plug-ins (load-time and run-time).
- Enables full automation but requires preplanning effort.

## Further Reading

- Apel et al. 2013, Chapter 4.3

## Practice

- What are possible extensions for a bike?
- How are bike extensions affected by the preplanning problem?



# FAQ – 6. Modular Features

## Lecture 6a

- What are problems of previous implementation techniques?
- How to implement product lines with components?
- What is modularity, cohesion, coupling, glue code? Why does it matter?
- Why is it hard to find the right size for components?
- What is the library scaling problem? How can product-line engineering help?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of components?
- When (not) to implement product lines with components?

## Lecture 6b

- What is a (micro-)service, orchestration, choreography?
- What are commonalities and differences between components and services?
- How to implement product lines with (micro-)services?
- How are microservices related to Conway's law, single responsibility principle, agile development, DevOps?
- How to compose services?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of services?
- When (not) to implement product lines with services?

## Lecture 6c

- What are (black-box) frameworks, plug-ins, extension points/hot spots, extensions?
- What is inversion of control? Why is it useful?
- How to hide implementation details from both, framework and plug-ins? Why is it useful?
- How to implement product lines with plug-ins?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of plug-ins?
- When (not) to implement product lines with plug-ins?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 7a. Limitations of Object Orientation

- Recap on Modularity
- Preplanning Problem
- Crosscutting Concerns
- Tyranny of the Dominant Decomposition
- Example: Arithmetic Expressions
- Summary

### 7b. Feature-Oriented Programming

- Motivation
- Feature Modules
- Feature Composition
- Feature Modules in Java
- Principle of Uniformity
- Discussion
- Summary

### 7c. Aspect-Oriented Programming

- Recap and Motivation
- Aspects and Aspect Weaving
- Static vs. Dynamic Extensions
- Quantification
- Executing Additional Code
- Aspects for Product Lines
- Discussion
- Summary
- FAQ

# 7. Languages for Features – Handout

Software Product Lines | Timo Kehrer, Thomas Thüm, Elias Kuiter | June 2, 2023

# 7. Languages for Features

## 7a. Limitations of Object Orientation

Recap on Modularity

Preplanning Problem

Crosscutting Concerns

Tyranny of the Dominant Decomposition

Example: Arithmetic Expressions

Summary

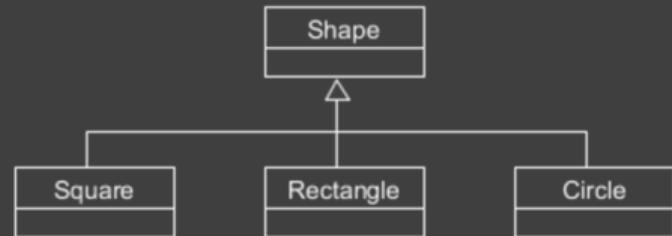
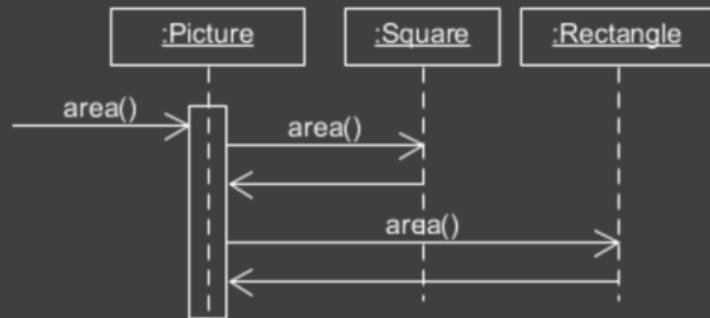
## 7b. Feature-Oriented Programming

## 7c. Aspect-Oriented Programming

# Recap: Object Orientation

## Key Concepts

- **Encapsulation:**  
abstraction and information hiding
- **Composition:**  
nested objects
- **Message Passing:**  
delegating responsibility
- **Distribution of Responsibility:**  
separation of concerns
- **Inheritance:**  
conceptual hierarchy, polymorphism, reuse



# Recap: Design Patterns [Gang of Four]

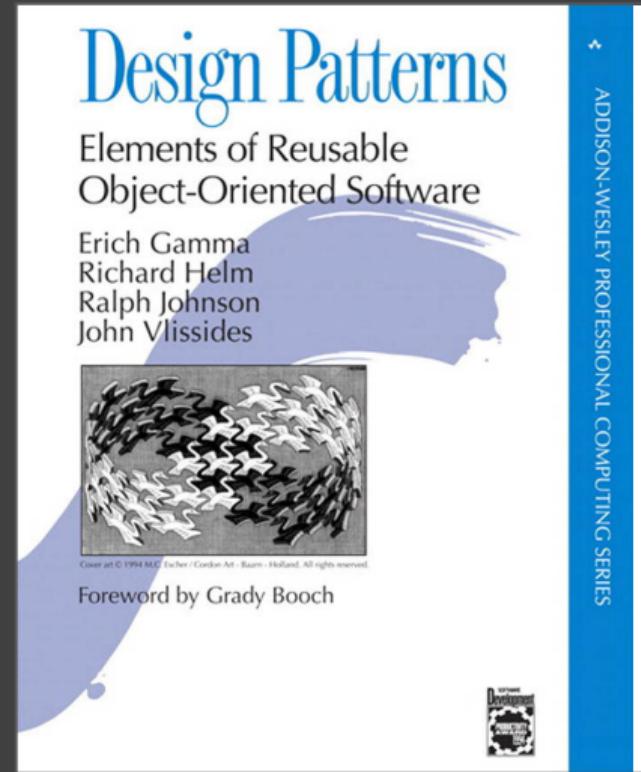
## Design Patterns

- Document common solutions to concrete yet frequently occurring design problems
- Suggest a concrete implementation for a specific object-oriented programming problem

## Design Patterns for Variability

Many Gang of Four (GoF) design patterns for designing software around stable abstractions and interchangeable (i.e., variable) parts, e.g.

- Template Method
- Abstract Factory
- Decorator



# Recap: Modularity - Components, Services, and Frameworks

## Component-/Service-Based SPLs



Specification of “composition” (glue code, orchestration)



## Framework-Based SPLs



Neither glue code nor service composition required.

# Example: Extending Basic Graphs by Plug-Ins?

```
public class Graph {  
    private List<GraphPlugin> plugins = new ArrayList<GraphPlugin>();  
    // ...  
    public void registerPlugin(GraphPlugin p) {  
        plugins.add(p);  
    }  
    public void addNode(int id, Color c){  
        Node n = new Node(id);  
        notifyAdd(n, c);  
        nodes.add(n);  
    }  
    public void print() {  
        for (Node n : nodes) {  
            notifyPrint(n);  
            // ...  
        }  
        // ...  
    }  
    private void notifyAdd(Node n, Color c) {  
        for (GraphPlugin p : plugins) {  
            p.aboutToAdd(n, c);  
        }  
    }  
    private void notifyPrint(Node n) {  
        for (GraphPlugin p : plugins) {  
            p.aboutToPrint(n);  
        }  
    }  
    // ...  
}
```

```
public interface GraphPlugin {  
    public void aboutToAdd(Node n, Color c);  
    public void aboutToAdd(Edge e, Weight w);  
    public void aboutToPrint(Node n);  
    public void aboutToPrint(Edge e);  
}
```

```
public class ColorPlugin implements GraphPlugin {  
    private Map<Node, Color> map = new HashMap<Node, Color>();  
  
    public void aboutToAdd(Node n, Color c) {  
        map.put(n, c);  
    }  
  
    public void aboutToAdd(Edge e, Weight w) {  
        // do nothing  
    }  
  
    public void aboutToPrint(Node n) {  
        Color c = map.get(n);  
        Color.setDisplayColor(c);  
    }  
  
    public void aboutToPrint(Edge e) {  
        // do nothing  
    }  
}
```

# Recap on Modularity

In our example, we can observe that:

- There are lots of empty methods in the ColorPlugin
- The Framework consults all registered plug-ins before printing a node or edge

## General Challenge: Cross-cutting Concerns

Implementing cross-cutting concerns as plug-ins

- typically leads to huge interfaces, large parts of which are irrelevant for a dedicated plug-in
- causes lots of communication overhead between plug-ins and framework

If we were not familiar with our graph library, would we anticipate that:

- Colors and weights should be part of the Plugin interface?
- Every plug-in needs to be notified that the framework is about to print a node or edge?

## Generally known as Preplanning Problem

- Hard to identify and foresee the relevant hot spots and nature of extensions
- Developing a framework needs lots of expertise and excellent domain knowledge

# Preplanning Problem

## Components, Services, and Frameworks

Extensions are not possible ad-hoc, but must be foreseen and planned in advance:

- Frameworks: Hot spots / extension points
  - Components/services: Provided and required interfaces
- ⇒ **No modular extension without a suitable extension point / interface!**

## And classical OO language concepts?

Subtyping and polymorphism allow for ad-hoc extensions to some extent, but:

- Often, client code and/or basic implementation need to be adapted, too (non-modular)
- Design patterns require preplanning of potential future extensions
- Limited flexibility of inheritance hierarchies (no “mix and match” supporting arbitrary feature combinations)

⇒ **No variable extension without loosing modularity!**

# Crosscutting Concerns

## Concern

[Apel et al. 2013, pp. 55]

A concern is an area of interest or focus in a system, and features are the concerns of primary interest in product-line engineering.

## Crosscutting (Concern)

[Apel et al. 2013, pp. 55]

Crosscutting is a structural relationship between the representations of two concerns. It is an alternative to hierarchical and block structure.

# Tyranny of the Dominant Decomposition

## Tyranny of the Dominant Decomposition

- Many concerns can be modularized, but not always at the same time.
- Developers choose a decomposition, but some other concerns cut across.
- Simultaneous modularization along different dimensions is not possible.

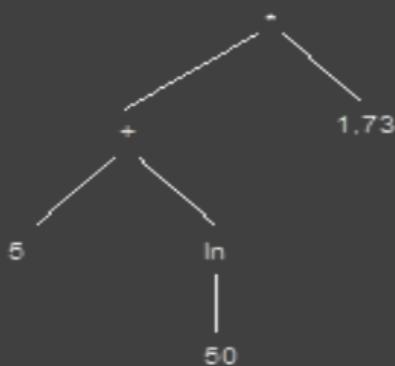
Crosscutting concerns are inherently difficult to separate using traditional mechanisms.

- Logging: Each time a method is called.
- Caching/Pooling: Each time an object is created.
- Synchronization/Locking: Extension of many methods with lock/unlock calls.

Features in a software product line are often cross-cutting (e.g., color and weight in our graph example).

## Example: Arithmetic Expressions

$(5 + \ln(50)) * 1.73$



- Arithmetic expressions are stored in a tree structure.
- Terms (i.e., sub-trees) can be evaluated and printed.

### Question

How to separate data structures and operations such that both can be extended independently of each other?

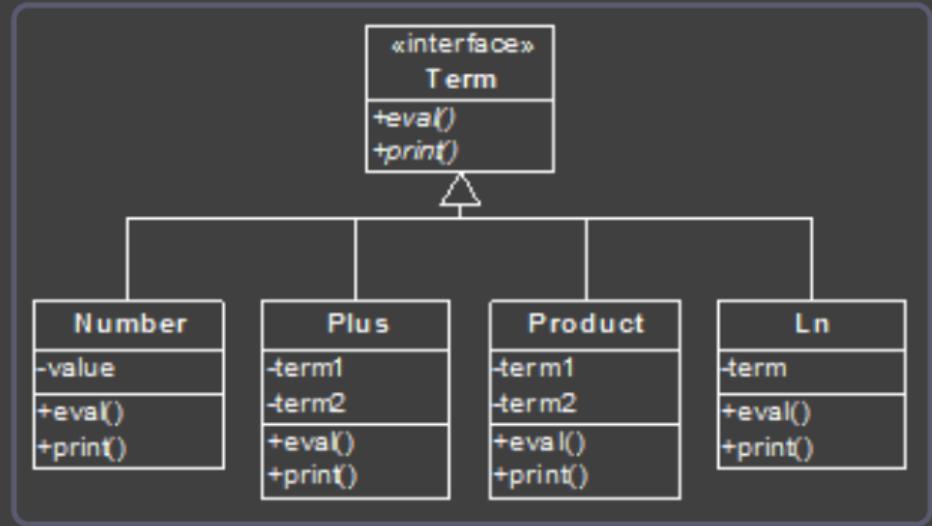
# Arithmetic Expressions – Data-Centric Decomposition

## Implementation Variant 1: Data-Centric

- Recursive class structure (composite pattern)
- For each operation (eval, print, ...) there is a dedicated method in each class (Number, Plus, ...)

## Terms are modular, but...

- New operations, e.g. drawTree or simplify, cannot simply be added
- All existing classes must be adjusted!
- Operations cut across the expressions.



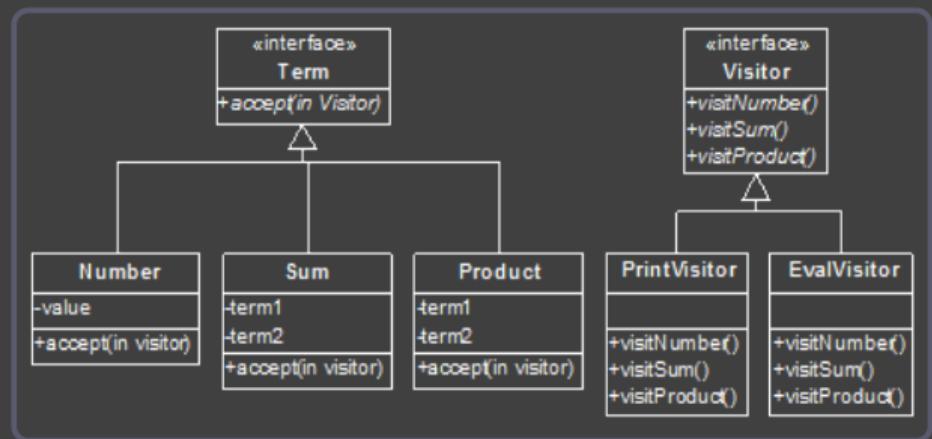
# Arithmetic Expressions – Operation-Centric Decomposition

## Implementation Variant 2: Operation-Centric

- Just a single accept method per class (visitor pattern).
- Each operation is implemented by a dedicated visitor.

## Operations are modular, but...

- New expressions, e.g. Min or Power, cannot simply be added
- For each new class, all visitor classes must be adjusted
- Expressions cut across operations



# Lessons Learned from the Simple Example

Hardly possible to modularize expressions and operations at the same time!

## Data-Centric Decomposition

- New expressions can be added directly: modular.
- New operations must be added to all classes: not modular.

## Operation-Centric Decomposition

- New operations can be added as another visitor: modular.
- For new expressions, all existing visitors must be extended: not modular.

# Limitations of Object Orientation – Summary

## Lessons Learned

Important problems of previous approaches:

- Inflexible inheritance hierarchies (especially with runtime variability, frameworks, components, services)
- Feature traceability (especially with runtime variability, branches, build systems, preprocessors)
- Preplanning problem (esp. with frameworks, components, services)
- Cross-cutting issues (esp. with frameworks, components, services)

## Practice

Looking at our graph implementation serving as running example throughout the course:

- Which concern is the dominant one regarding modular decomposition?
- What are crosscutting concerns?
- Can we restructure the implementation to come up with a different decomposition?

## Further Reading

s. previous lectures

# 7. Languages for Features

## 7a. Limitations of Object Orientation

## 7b. Feature-Oriented Programming

Motivation

Feature Modules

Feature Composition

Feature Modules in Java

Principle of Uniformity

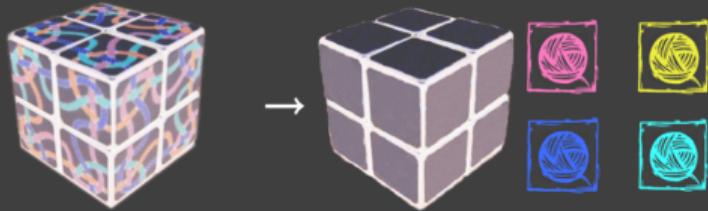
Discussion

Summary

## 7c. Aspect-Oriented Programming

# Motivation

## Modularization of Cross-Cutting Concerns



## Flexible Extension / Minimal Preplanning



## Feature Traceability



Achieving all this requires novel implementation techniques that overcome the limitations of classical object-oriented paradigms.

# Background: Collaboration-Based Design

## Inspiration: Collaborations in the Real World

- People collaborate to achieve a common goal.
- A collaboration typically comprises several persons playing different roles.
- Persons may play multiple roles by participating in different collaborations.

## Mentor-Student Collaboration

- A person in the role of a mentor has responsibilities to instruct students on certain topics.
- A person in the role of a student has responsibilities to study the offered material.

## Collaborations in Java

[Apel et al. 2013, pp. 131]

- A **collaboration** is a set of interacting classes, each class playing a distinct role, to achieve a certain function or capability.
- A **role** defines the responsibilities a class takes in a collaboration.

- Different classes play different roles within a collaboration.
- A class plays different roles in different collaborations.
- A role encapsulates the behavior/functionality of a class relevant to a collaboration.

## Example: Collaborations, Classes and Roles

		Classes												
		Graph	Edge	Node	Weight	Color								
Collaborations	Base	<table border="1"><tr><td>Graph</td></tr><tr><td>Edge add(Node,Node)</td></tr><tr><td>void print()</td></tr></table>	Graph	Edge add(Node,Node)	void print()	<table border="1"><tr><td>Edge</td></tr><tr><td>Node a,b</td></tr><tr><td>void print()</td></tr></table>	Edge	Node a,b	void print()	<table border="1"><tr><td>Node</td></tr><tr><td>void print()</td></tr></table>	Node	void print()		
Graph														
Edge add(Node,Node)														
void print()														
Edge														
Node a,b														
void print()														
Node														
void print()														
Directed	<table border="1"><tr><td>Graph</td></tr><tr><td>Edge add(Node,Node)</td></tr></table>	Graph	Edge add(Node,Node)	<table border="1"><tr><td>Edge</td></tr><tr><td>Node start</td></tr><tr><td>void print()</td></tr></table>	Edge	Node start	void print()							
Graph														
Edge add(Node,Node)														
Edge														
Node start														
void print()														
Weighted	<table border="1"><tr><td>Graph</td></tr><tr><td>Edge add(Node,Node)</td></tr><tr><td>Edge add(Node,Node,Weight)</td></tr></table>	Graph	Edge add(Node,Node)	Edge add(Node,Node,Weight)	<table border="1"><tr><td>Edge</td></tr><tr><td>Weight weight</td></tr><tr><td>void print()</td></tr></table>	Edge	Weight weight	void print()		<table border="1"><tr><td>Weight</td></tr><tr><td>...</td></tr></table>	Weight	...		
Graph														
Edge add(Node,Node)														
Edge add(Node,Node,Weight)														
Edge														
Weight weight														
void print()														
Weight														
...														
Colored			<table border="1"><tr><td>Node</td></tr><tr><td>Color color</td></tr><tr><td>void print()</td></tr></table>	Node	Color color	void print()		<table border="1"><tr><td>Color</td></tr><tr><td>...</td></tr></table>	Color	...				
Node														
Color color														
void print()														
Color														
...														

# Feature Composition

## Feature Modules

- Each collaboration mapped to a feature and is called a feature module (or layer).
- Feature modules may refine a base implementation by adding new elements or by modifying and extending existing ones.

## Feature Module Composition

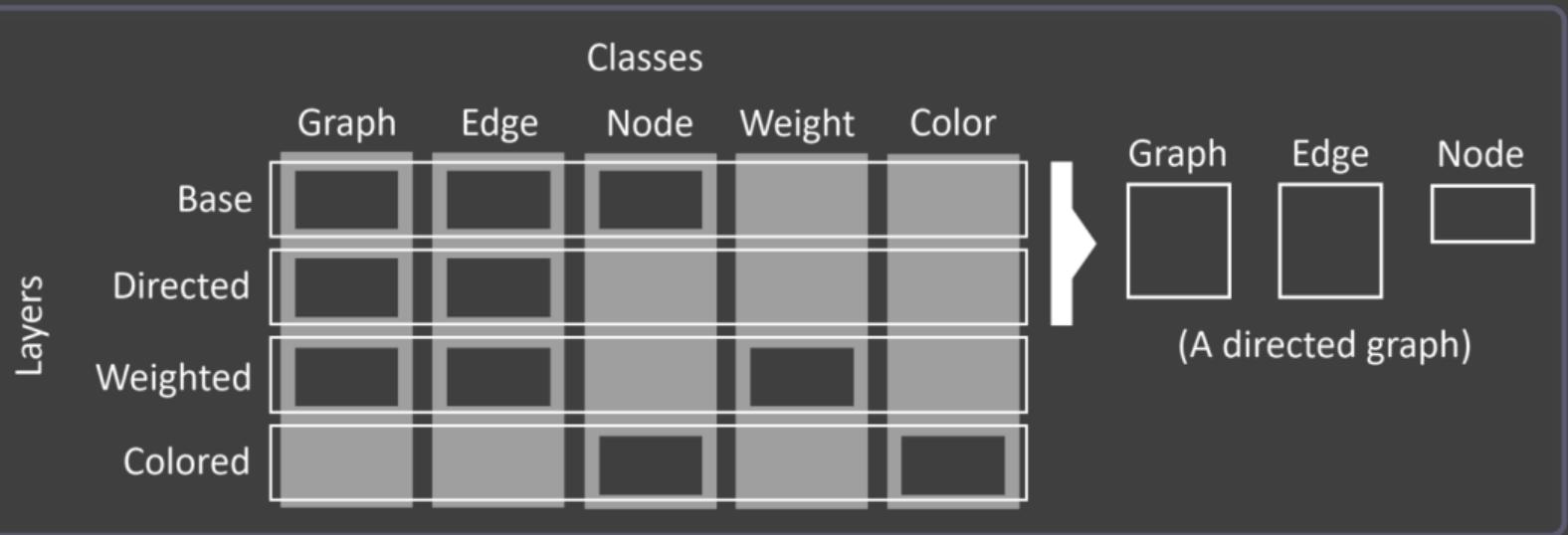
Selected feature modules may be superimposed by lining-up classes according to the roles they play.



# Feature Modules and Feature Module Composition

## Open Questions

- How to bundle classes to feature modules and specify their refinements?
- How to handle refinements during composition of feature modules?



# Jak: A Java Extension for Feature-Oriented Programming

[Batory et al. 2004]

## Layers

- keyword **layer** denotes the feature a class belongs to
- layer = feature module = collaboration

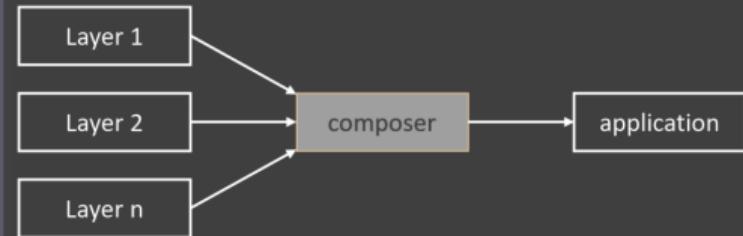
## Class Refinement

A class refinement (keyword **refines**) can add new members to a class and extend existing methods.

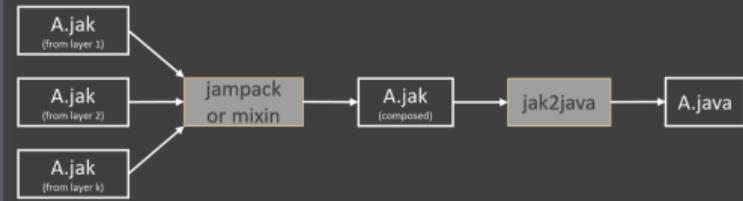
## Composer

- AHEAD (Algebraic Hierarchical Equations for Application Design) + jampack/mixin
- Application constructed by composing layers
- Internally, the composer invokes a variety of tools to perform its task

## Composer (High-Level View)



## Composer (Jak File Composition)



# Jak: Layers



- Layer (i.e., feature module) Base consists of the classes Graph, Node, and Edge
- The three classes collaborate to provide the functionality to construct and display graph structures.

## Graph.jak

```
class Graph {  
    private List nodes = new ArrayList();  
    private 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);  
        return e;  
    }  
    void print() { ... }  
}
```

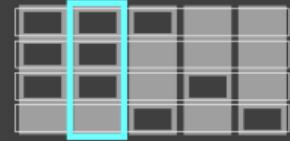
## Node.jak

```
class Node {  
    private int id = 0;  
  
    void print() {  
        System.out.print(id);  
    }  
}
```

## Edge.jak

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

# Jak: Class Refinement – New Members



## Mixin-Based Inheritance

Subclasses are abstract in the sense that they can be applied to **different** concrete superclasses.

## Refinement Chain

A refinement chain is a linear inheritance chain where the bottom-most class of the chain is the only class that is meant to be instantiated.

## New Members

A stepwise refinement can add new members (i.e., fields and methods) to a class.

### Edge.jak

```
class Edge {  
    ...  
}
```

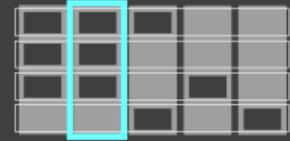
### Edge.jak

```
refines class Edge {  
    private Node start;  
    ...  
}
```

### Edge.jak

```
refines class Edge {  
    private Weight weight;  
    ...  
}
```

# Jak: Class Refinement – Method Extensions



## Mixin-Based Inheritance

Subclasses are abstract in the sense that they can be applied to **different** concrete superclasses.

## Refinement Chain

A refinement chain is a linear inheritance chain where the bottom-most class of the chain is the only class that is meant to be instantiated.

## Method Extension

A method extension is implemented by method overriding and calling the overridden method via the keyword Super.

### Edge.jak in Layer Base

```
class Edge {  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

### Edge.jak in Layer Directed

```
refines class Edge {  
    ...  
    void print() {  
        Super().print();  
        System.out.print(" directed from " + start);  
    }  
}
```

### Edge.jak in Layer Weighted

```
refines class Edge {  
    ...  
    void print() {  
        Super().print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

# AHEAD: Composition Using Jampack

## Jampack

- Jampack superimposes the refinement chain into a single class.
- Super calls are integrated by method inlining (cf. optimization in compiler construction).

## Composition Result

```
class Edge {  
    private Node start;  
    private Weight weight;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
        System.out.print(" directed from " + start);  
        System.out.print(" weighted with " + weight);  
    }  
}
```

# AHEAD: Composition Using Mixin

## Mixin

- Mixin retains layer relationships as an inheritance chain.
- Produces a single file that contains a (linear) inheritance hierarchy where only the bottom-most class is “public”.
- Super calls are integrated by method inlining (cf. optimization in compiler construction).

## Do Not Confuse with Mixins

mixins are a language concept to decompose classes into parts without inheritance and rather similar to Jampack

## Composition Result

```
class Edge$$Base {  
    void print() { ... }  
}  
class Edge$$Directed extends Edge$$Base {  
    private Node start;  
    void print() {  
        super.print();  
        System.out.print(" directed from " + start);  
    }  
}  
class Edge extends Edge$$Directed {  
    private Weight weight;  
    void print() {  
        super.print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

# Jampack vs. Mixin

## Jampack

- Assignment of generated code to roles disappears after generation
- Local variables can be accessed from within refined methods

## Mixin

- Code overhead and method call indirections negatively impact runtime performance
- Feature modularity preserved even after composition

# Jampack and Mixin in Practice

## Recommended Usage

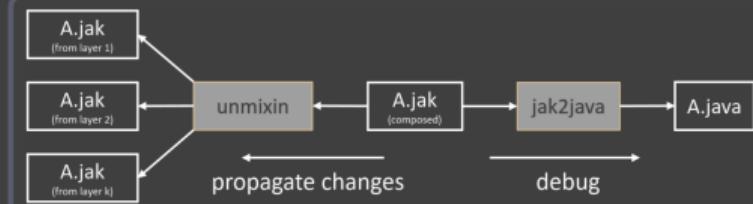
- Use Mixin during development and iterative refinement (debugging)
- Use Jampack when a production version of a class is to be produced (performance)

## Un mixin

Automatically propagates changes from the composed .jak file back to its original layer files

## Un mixin and Debugging

- Make changes to a mixin-composed .jak file during debugging
- Then automatically back-propagate changes to the layer files



# Composition: Order Matters!

## (a) Edge.jak

```
class Edge {  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

## (b) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" directed from " + start);  
    }  
}
```

## (c) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

Class refinements themselves are (largely) independent of the order in which they are eventually composed.

## Composition Order (a), (b), (c)

```
class Edge {  
    private Node start;  
    private Weight weight;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
        System.out.print(" directed from " + start);  
        System.out.print(" weighted with " + weight);  
    }  
}
```

However, the order in which features are applied is important (e.g., earlier features in the sequence may add elements that are refined by later features).

# Composition: Order Matters!

## (a) Edge.jak

```
class Edge {  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

## (b) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" directed from " + start);  
    }  
}
```

## (c) Edge.jak

```
refines class Edge { ...  
    void print() {  
        Super().print();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

Class refinements themselves are (largely) independent of the order in which they are eventually composed.

## Composition Order (a), (c), (b)

```
class Edge {  
    private Node start;  
    private Weight weight;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
        System.out.print(" weighted with " + weight);  
        System.out.print(" directed from " + start);  
    }  
}
```

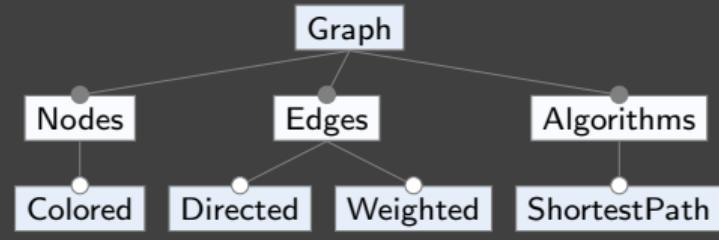
However, the order in which features are applied is important (e.g., earlier features in the sequence may add elements that are refined by later features).

# Composition: Order Matters!

The order in which compositions are to be applied is an input parameter of the composition tool.

## Composition Order in FeatureIDE

- In FeatureIDE, a total order can be defined based on the feature model
- Default: depth-first traversal of the feature model (only concrete features)

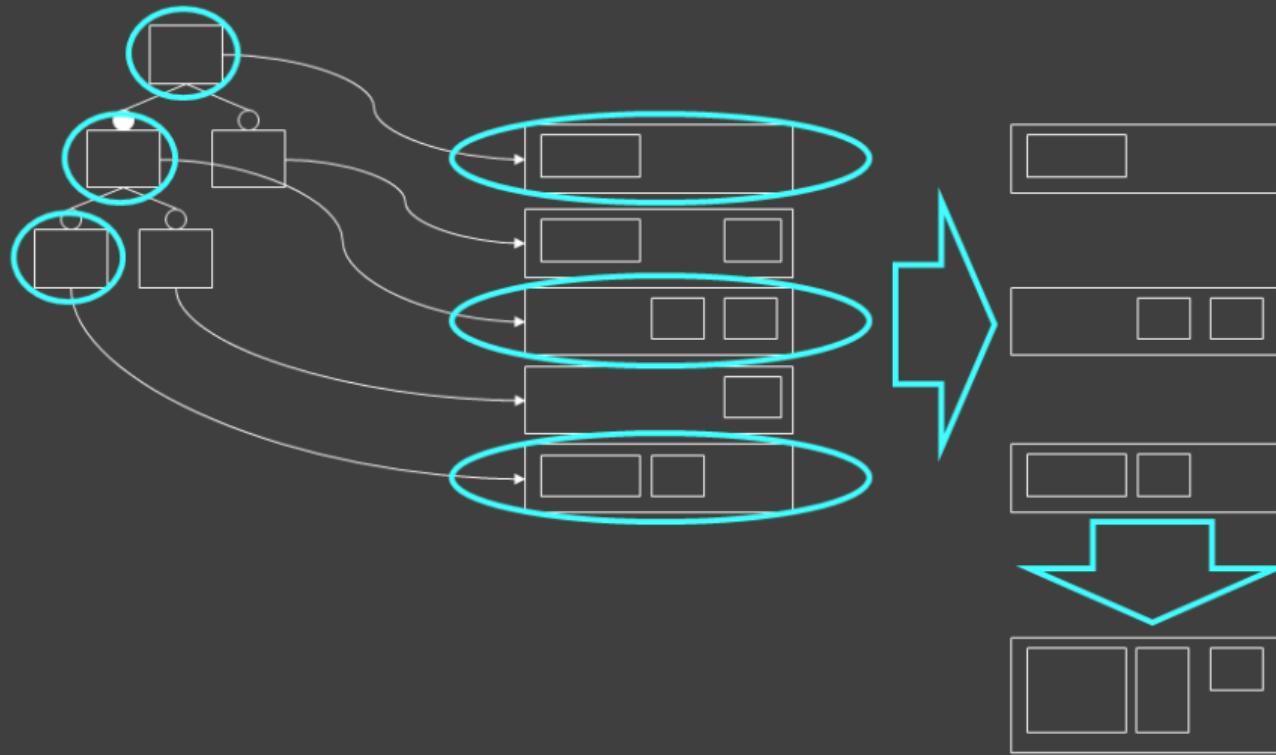


*ShortestPath → Weighted*

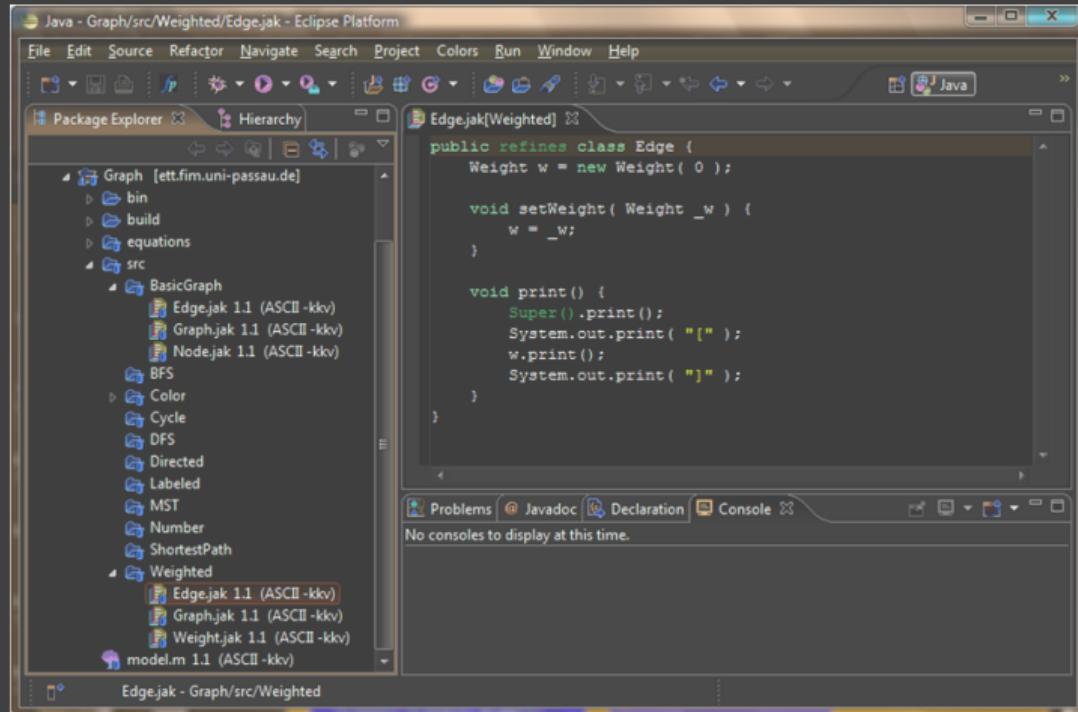
## Default Order for Example

Graph, Colored, Directed, Weighted, ShortestPath

# Big Picture: Product-Line Implementation and Product Generation



# Practical Organization of Feature Modules



- In most FOP tools, feature modules are represented by (nested) file-system directories
- Classes and their refinements are stored in files inside the corresponding containment hierarchies

# Beyond Jak: Uniformity

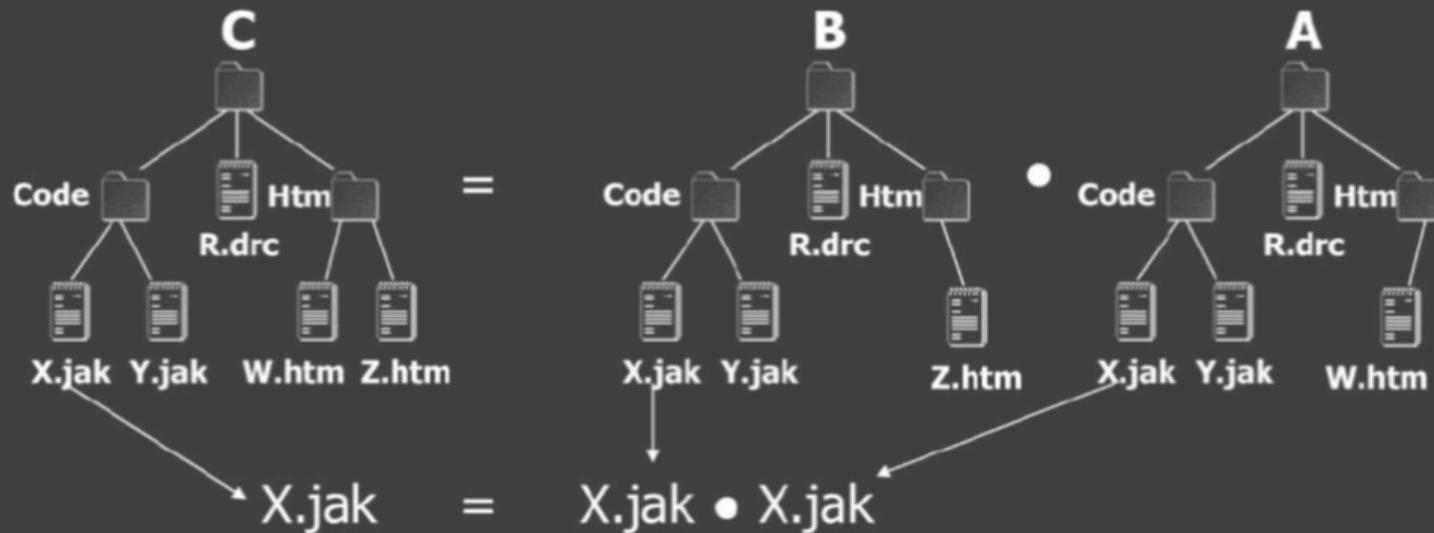
## Motivation

- Software does not only consist of Java source code, but also other programming languages, build scripts, documentation, models, etc.
- All software artifacts must be refined
- Integration of different artifacts in collaborations

## Idea

- Each feature is represented by a containment hierarchy:
  - Directory structure organizes the feature's artifacts.
  - At the file level, there may be heterogeneous artifacts.
- Composing features means composing containment hierarchies and, to this end, composing corresponding artifacts recursively by name and type
- For each artifact type, a different implementation of the composition operator “ $\bullet$ ” has to be provided in AHEAD (just like the Jak-composition tool)

## Beyond Jak: Uniformity



# FeatureHouse: A Model and Framework for FOP

## Goal

**Language-independent model and tool chain** to enhance given languages rapidly with support for feature-oriented programming

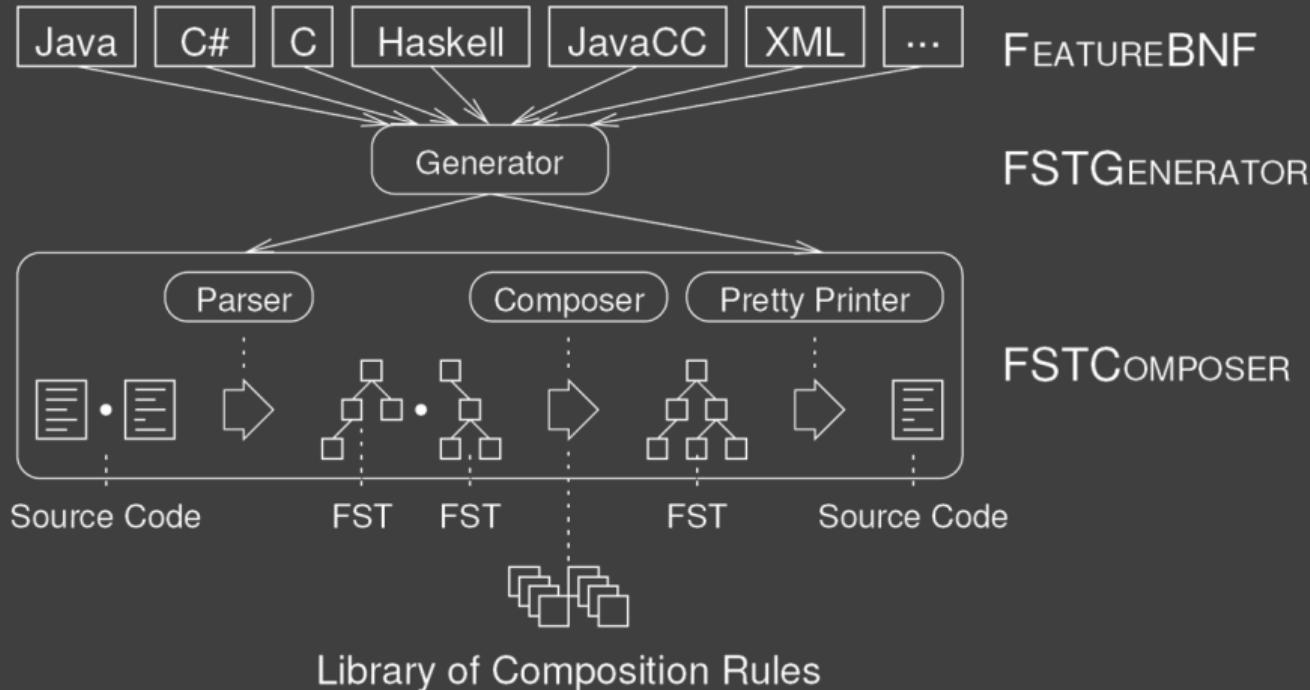
## Assumption

- A feature may be represented as tree, known as **Feature Structure Tree (FST)**
- Example; Java: Packages, Classes, Methods and Fields

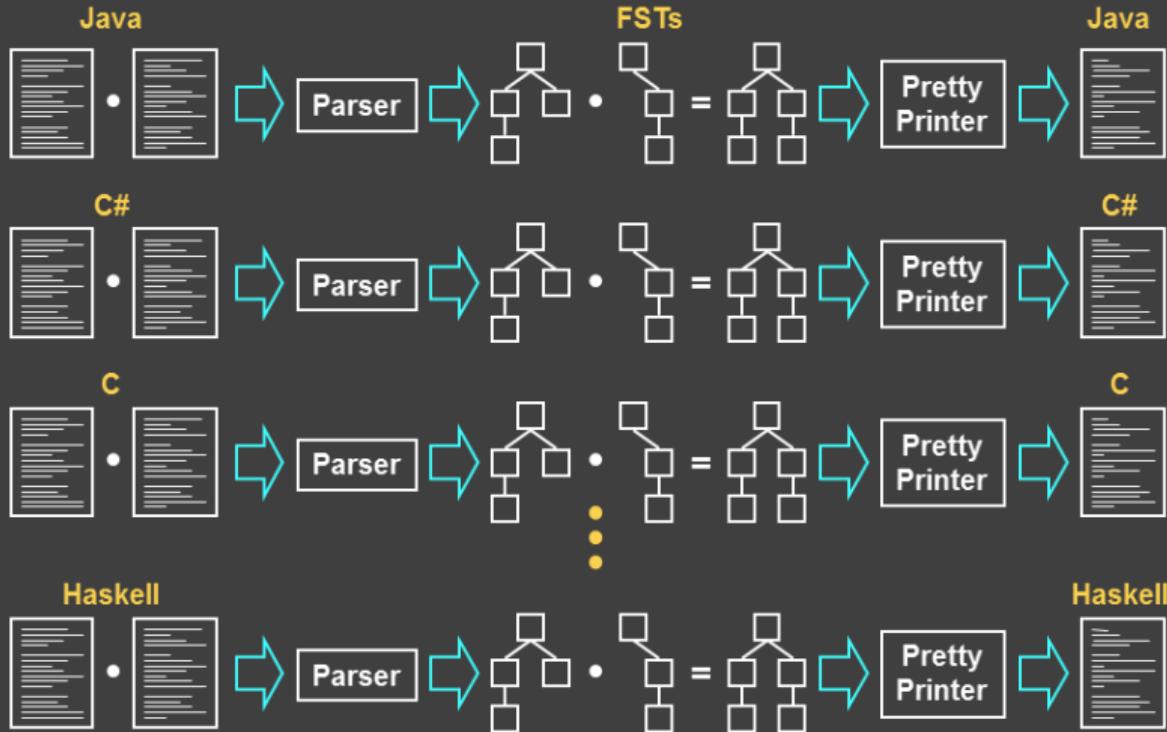
## Idea

- Composition = Superimposition of FSTs (i.e., recursively superimpose nodes of FST, starting with the root node)
- Inner nodes: Can be safely superimposed if they are identical (superimposed parents and same name), or added if non-identical
- Leaf nodes: Type-specific resolution of conflicts

# FeatureHouse: Overview



# FeatureHouse: Composition



# Example: Java Support in FeatureHouse

```
class Edge {  
    private Node a, b;  
    void print() {  
        System.out.print("Edge between " + a + " and " + b);  
    }  
}
```

```
class Edge {  
    private Node start;  
    void print() {  
        original();  
        System.out.print(" directed from " + start);  
    }  
}
```

```
class Edge {  
    private Weight weight;  
    void print() {  
        original();  
        System.out.print(" weighted with " + weight);  
    }  
}
```

## Differences Compared to Jak

- No explicit keyword refines
- Calling the method from previous refinement using keyword original

# Discussion

## Advantages

- Easy to use language-based mechanism, requires only minimal language extensions.
- Conceptually uniformly applicable to code and noncode artifacts.
- Separation of (possibly crosscutting) feature code into distinct feature modules.
- Little preplanning required due to mixin-based extension mechanism.
- Direct feature traceability from a feature to its implementation in a feature module.

## Disadvantages

- Requires adoption of a language extension and composition tools.
- Tools need to be constructed for every language (although with the help of a framework).
- Only academic tools so far, little experience in practice.
- Granularity restricted to method-level (or other named structural entities).

# Feature-Oriented Programming – Summary

## Lessons Learned

- Idea: Mixin-based inheritance getting rid of the traditional limitations of inflexible inheritance hierarchies.
- Supports encapsulation of (cross-cutting) concerns and feature traceability by design.
- Academic approach not widely adopted in industry.

## Further Reading

- Batory et al.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering, 30(6), 2004.
- Apel et al.: Language-Independent and Automated Software Composition: The FeatureHouse Experience. IEEE TSE, 39(1), 2013.
- Apel et al. 2013, Chapter 6.1
- Meinicke et al. 2017, Part 4

## Practice

- How is class refinement in FOP different from inheritance in OOP?
- To some extent, FOP can be considered as static counterpart to the Decorator design pattern in OOP. Why?
- In which sense does FOP violate the classical principles of information hiding and encapsulation of OOP? What are the consequences?

# 7. Languages for Features

## 7a. Limitations of Object Orientation

## 7b. Feature-Oriented Programming

## 7c. Aspect-Oriented Programming

Recap and Motivation

Aspects and Aspect Weaving

Static vs. Dynamic Extensions

Quantification

Executing Additional Code

Aspects for Product Lines

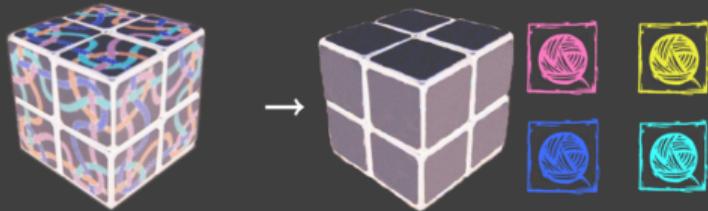
Discussion

Summary

FAQ

# Motivation

## Modularization of Cross-Cutting Concerns



## Flexible Extension / Minimal Preplanning



## Feature Traceability



Achieving all this requires novel implementation techniques that overcome the limitations of classical object-oriented paradigms.

# Aspects and Aspect Weaving

## Aspect

[Apel et al. 2013, pp. 143–145]

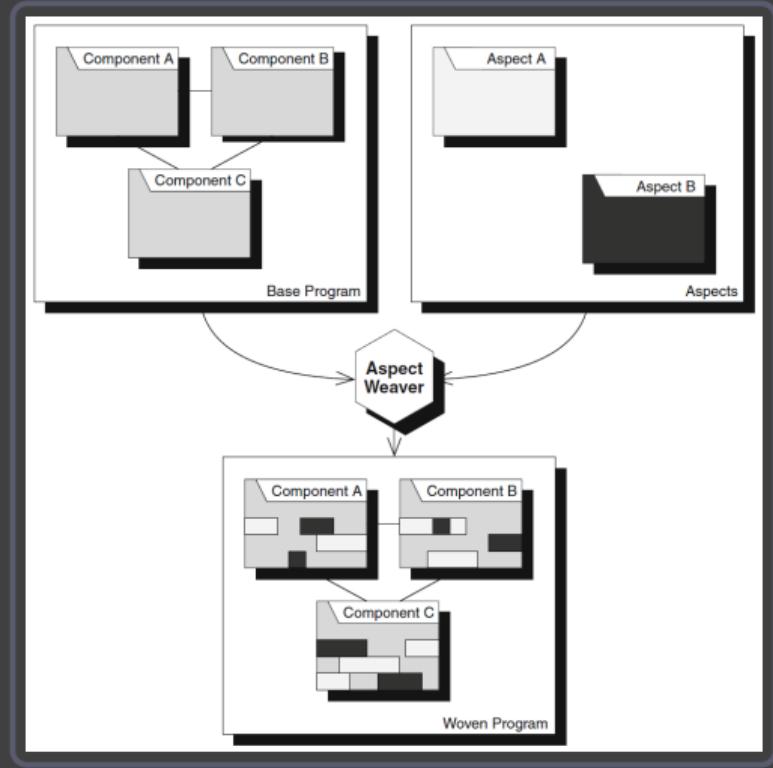
An aspect encapsulates the implementation of a crosscutting concern.

## Aspect Weaving

[Apel et al. 2013, pp. 143–145]

An aspect weaver merges the separate aspects of a program and the base program at user-selected program locations.

- Localizing a crosscutting concern within one code unit eliminates code scattering and tangling.
- An aspect can affect multiple other concerns with one piece of code, thereby avoiding code replication.

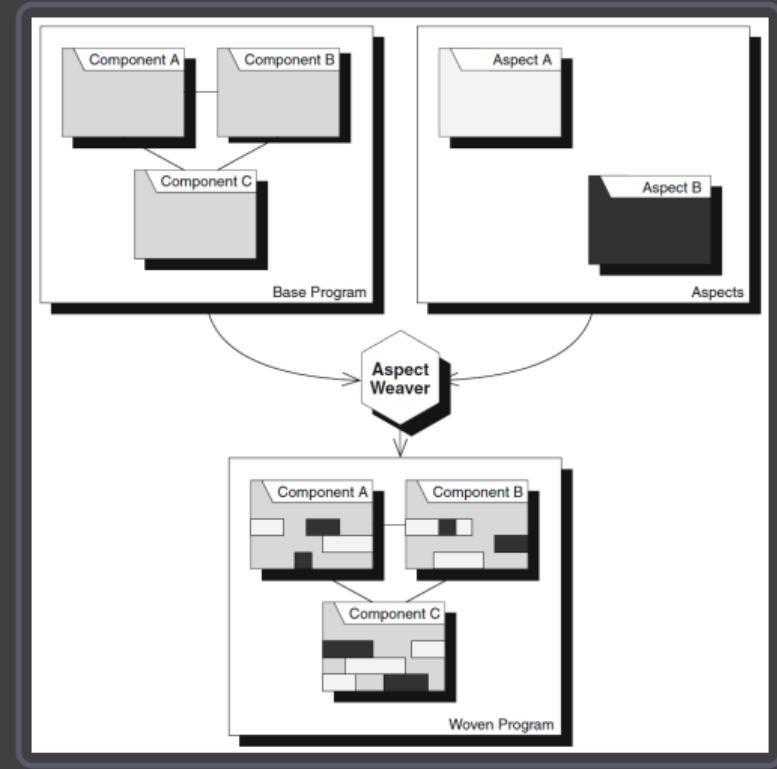


# Aspects and Aspect Weaving in Java: AspectJ

AspectJ is an aspect-oriented language extension of Java.

- Base program is written in Java (i.e., components = classes)
- Aspects are written in Java but typically include a multitude of new language constructs introduced by AspectJ
- Aspect weaver (aka. AspectJ Compiler) follows a compile-time binding approach (though certain decisions are made at runtime, s. later).

AspectJ is the most popular and widely used aspect-oriented language, all examples in this lecture will be given in AspectJ.



# Static Extensions through Inter-Type Declarations

## Inter-Type Declaration

[Apel et al. 2013, pp. 143–145]

An inter-type declaration injects a method, field, or interface from inside an aspect into an existing class or interface.

## Typical Usage

Add field X / method Y to class Z

```
aspect Weighted {  
    private int Edge.weight = 0;  
    public void Edge.setWeight(int w) {  
        weight = w;  
    }  
}
```

# Dynamic Extensions through Join Points

## Joint Point

[Apel et al. 2013, pp. 143–145]

A join point is an event in the execution of a program at which aspects can be woven into the program.

## Advice

Code which is being executed when a join point matches.

## Join-Points in AspectJ:

- Calling/executing a method/constructor
- Access to a field (read or write)
- Catching an Exception
- Execution of a Advice
- ...

```
class Test {  
    MathUtil u;  
    public void main() {  
        u = new MathUtil(); // constructor call  
        int i = 2;  
        i = u.twice(i); // method-call  
        System.out.println(i); // method-call  
    }  
}  
class MathUtil {  
    public int twice(int i) {  
        return i * 2; // method-execution  
    }  
}
```

## Open Question

How to specify the join points an aspect (i.e., an advice) affects?

# Quantification through Pointcuts

## Pointcut

[Apel et al. 2013, pp. 143–145]

A pointcut is a declarative specification of the join points that an aspect affects. It is a predicate that determines whether a given join point matches.

## Quantification

[Apel et al. 2013, pp. 143–145]

Quantification is the process of selecting multiple join points based on a declarative specification (that is, based on a pointcut).

- Execute Advice X whenever the method `setWeight` of class `Edge` is called
- Execute Advice Y whenever any field in class `Edge` is accessed
- Execute Advice Z whenever a public method is called anywhere in the system and the method `initialize` has been called beforehand

## Anonymous Pointcut

```
aspect A1 {  
    after() : execution(int MathUtil.twice(int)) {  
        System.out.println("MathUtil.twice executed");  
    }  
}
```

## Explicit Pointcut

```
aspect A2 {  
    pointcut executeTwice() :  
        execution(int MathUtil.twice(int));  
    after() : executeTwice() {  
        System.out.println("MathUtil.twice executed");  
    }  
}
```

# Quantification over other Join Points

## Call of a method

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println(" MathUtil.twice called");  
    }  
}
```

## Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

# Quantification over other Join Points

## Call of a method

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println("MathUtil.twice called");  
    }  
}
```

## Constructor call

```
aspect A1 {  
    after() : call(MathUtil.new()) {  
        System.out.println("MathUtil created");  
    }  
}
```

## Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
  
    class MathUtil {  
        public int twice(int i) {  
            return i * 2;  
        }  
    }  
}
```

# Quantification over other Join Points

## Call of a method

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println(" MathUtil.twice called");  
    }  
}
```

## Constructor call

```
aspect A1 {  
    after() : call(MathUtil.new()) {  
        System.out.println(" MathUtil created");  
    }  
}
```

## And many more

- get/set: field access (read/write)
- etc.

## Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

# Further Quantification Options

## Pointcuts with “Wildcards”

```
aspect A1 {  
    pointcut P1(): execution(int MathUtil.twice(int));  
    pointcut P2(): execution(* MathUtil.twice(int));  
    pointcut P3(): execution(int MathUtil.twice(*));  
    pointcut P4(): execution(int *.twice(int));  
    pointcut P5(): execution(int MathUtil.twice(..));  
    pointcut P6(): execution(int *Util.tw*(int));  
    pointcut P7(): execution(int *.twice(int));  
    pointcut P8(): execution(int MathUtil+.twice(int));  
}
```

## Logical Connections of Pointcuts

```
aspect A1 {  
    pointcut P2(): call(* MathUtil.*(..)) && !call(* MathUtil.twice(*));  
    pointcut P3(): execution(* MathUtil.twice(..)) && args(int);  
}
```

## Wildcard Symbols

- \* Exactly one value
- .. Arbitrary many values
- + Class or any subclass

## Logical Connectors

Pointcuts can be connected by usual logical operators  
`&&`, `||`, and `!`

# Advice (Pieces of Advice)

- Additional code before, after or instead of the join point: **before**, **after**, **around**
- Around-Advice allows to continue the original join point using the keyword **proceed**
- Keyword **proceed** corresponds to **original/Super** in FOP and **super** in OOP

```
public class Test2 {  
    void foo() {  
        System.out.println(" foo() executed");  
    }  
}  
aspect AdviceTest {  
    before(): execution(void Test2.foo()) {  
        System.out.println(" before foo()");  
    }  
    after(): execution(void Test2.foo()) {  
        System.out.println(" after foo()");  
    }  
    void around(): execution(void Test2.foo()) {  
        System.out.println(" around begin");  
        proceed();  
        System.out.println(" around end");  
    }  
    after() returning (): execution(void Test2.foo()) {  
        System.out.println(" after returning from foo()");  
    }  
    after() throwing (RuntimeException e): execution(void Test2.foo()) {  
        System.out.println(" after foo() throwing "+e);  
    }  
}
```

# thisJoinPoint

In an advice, `thisJoinPoint` can be used to get more information about the current join point.

```
aspect A1 {  
    after() : call(int MathUtil.twice(int)) {  
        System.out.println(thisJoinPoint);  
        System.out.println(thisJoinPoint.getSignature());  
        System.out.println(thisJoinPoint.getKind());  
        System.out.println(thisJoinPoint.getSourceLocation());  
    }  
}
```

## Output

```
call(int MathUtil.twice(int))  
int MathUtil.twice(int)  
method—call  
Test.java:5
```

# Parameterized Pointcuts

```
aspect A1 {  
    pointcut execTwice(int value) :  
        execution(int MathUtil.twice(int)) && args(value);  
    after(int value) : execTwice(value) {  
        System.out.println("MathUtil.twice executed with parameter "  
            + value);  
    }  
}
```

## Base Program

```
class Test {  
    public static void main(String[] args) {  
        MathUtil u = new MathUtil();  
        int i = 2;  
        i = u.twice(i);  
        i = u.twice(i);  
        System.out.println(i);  
    }  
}  
  
class MathUtil {  
    public int twice(int i) {  
        return i * 2;  
    }  
}
```

## Pointcuts this and target

- **execution:** this and target capture the object on which the method is called
- **call, set, and get:** this captures the object that calls the method or accesses the field; target captures the object on which the method is called or the field is accessed.

```
aspect A1 {  
    pointcut P1(Main s, MathUtil t):  
        call(* MathUtil.twice(*))  
        && this(s)  
        && target(t);  
}
```

# Order Matters: Aspect Precedence

Order in which aspect weaver process the aspects may be relevant when multiple aspects extend the same join point.

## Example

- 1st aspect implements synchronization with around advice
- 2nd aspect implements logging with after-advice on the same join point
- Depending on the order of weaving, the logging code will be synchronized or not

## Explicit definition using declare precedence

```
aspect DoubleWeight {  
    declare precedence : *, Weight, DoubleWeight;  
    [...]  
}
```

# Aspect Weaving: Behind the Scenes

## Weaving in AspectJ (Conceptually)

- Inter-type declarations are added to respective classes
- Each advice is converted into a method
- Pointcuts: Add method call from the join points to the advice
- Dynamic extensions: Insert source code at all potential join points that checks dynamic conditions and, if conditions hold, calls the advice method.

## Weaving in AspectJ (Technically)

AspectJ Compiler; conceptual effect is only visible in Bytecode

## Other Options

- Source transformation: Base Program + Aspects → Java (s. FOP/Jak).
- Evaluation at runtime: Meta-Object Protocol, interpreted languages, ...

# Typical (Traditional) Aspects

- Logging, Tracing, Profiling
- Adding identical code to a large number of methods

## Record time to execute my public methods

```
aspect Profiler {  
    Object around() : execution(public * com.company..*.*(..)) {  
        long start = System.currentTimeMillis();  
        try {  
            return proceed();  
        } finally {  
            long end = System.currentTimeMillis();  
            printDuration(start, end,  
                         thisJoinPoint.getSignature());  
        }  
    }  
    // implement recordTime...  
}
```

# Aspects for Product Lines

## Basic Idea

- Implement one aspect per feature.
  - Feature selection determines the aspects which are included in the weaving process.
- 
- Aspects encapsulate changes to be made to existing classes.
  - However, aspects do not encapsulate new classes introduced by a feature (only nested classes within an aspect)

## A Color Feature for Graphs

```
aspect ColorFeature {  
    Color Node.color = new Color();  
  
    before(Node n): execution(void print()) && this(n) {  
        Color.setDisplayColor(n.color);  
    }  
  
    static class Color {  
        ...  
    }  
}
```

# Controversial: Obliviousness and Fragile-Pointcut Problem

## Principle of Obliviousness

Base program is (deliberately) supposed to be oblivious wrt. the aspects that “hook into” the system:

- Base program developers implement their concerns as if there were no aspects.
- Aspect programmers extend the base program.

## Obliviousness worsens the fragile-pointcut problem

Because the base programmer does not know about aspects, it is more likely that changes may break aspect bindings and can remain unnoticed for a long time.

## Fragile-Pointcut Problem

Base program may be modified such that the set of join points changes in an undesired way:

- Join points may be removed accidentally
- Join points may be captured by aspects accidentally

```
class Chess {  
    void drawKing() {...}  
    void drawQueen() {...}  
    void drawKnight() {...}  
    void draw() {...} // new method matches pointcut!  
}  
aspect UpdateDisplay {  
    pointcut drawn : execution(* draw*(..));  
}
```

# Discussion

## Advantages

- Separation of (possibly crosscutting) feature code into distinct aspects.
- Direct feature traceability from feature to its implementation in an aspect.
- Little or no preplanning effort required.
- Fine-grained variability driven by the join-point model of the aspect-oriented language.

## Disadvantages

- Requires adoption of a rather complex extension mechanism (new language and paradigm).
- No unifying theory like no language-independent framework.
- Program evolution and maintenance affected by fragile-pointcut problem.

# Aspect-Oriented Programming – Summary

## Lessons Learned

- Idea: “In programs P, whenever condition C arises, perform action A”
- AspectJ: Sophisticated joint-point model and powerful language to quantify over join points (through pointcuts).
- Supports encapsulation of (cross-cutting) concerns and feature traceability by design.
- Practical acceptance limited due to fragile-pointcut problem.

## Further Reading

- Kiczales et al: Aspect-oriented programming. Proc. Europ. Conf. Object-Oriented Programming. 1997
- Filman et al.: Aspect-oriented software development. Addison-Wesley. 2005
- Apel et al. 2013, Chapter 6.2

## Practice

- Which features particularly benefit from the concept of quantification?
- What similarities and differences do you see between FOP and AOP?

# FAQ – 7. Languages for Features

## Lecture 7a

- What are problems of previous implementation techniques?
- What is the preplanning problem?
- What are (crosscutting) concerns?
- What is the tyranny of the dominant decomposition?
- What are crosscutting concerns when implementing arithmetic expressions?
- Why cannot all concerns be modularized with object orientation and design patterns?

## Lecture 7b

- What is feature-oriented programming and collaboration-based design? What are collaborations, roles, feature modules, class refinements?
- How to compose feature modules?
- What is the difference between Mixin and Jampack? What is better?
- Why does the order matter when composing feature modules? Where does it come from?
- What is the principle of uniformity?
- How to implement product lines with feature modules?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of feature modules?

## Lecture 7c

- What is aspect-oriented programming? What are aspects, aspect weaving, join points, pointcuts, pieces of advice, AspectJ?
- What are static/dynamic extensions, quantification, before/after/around advice, inter-type declarations?
- What is aspect precedence (good for)? How is it different from feature modules?
- What are obliviousness and fragile-pointcut problem?
- What are commonalities and differences between feature modules and aspects?
- How to implement product lines with aspects?
- What are (dis)advantages of aspects?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 8a. Process Model for Product Lines

- Recap: Process Models
- Domain and Application Engineering
- Analysis and Design
- Implementation and Testing
- Overview on Domain and Application Engineering
- Problem and Solution Space
- Summary

### 8b. Implementation of Product Lines

- Recap: Runtime Variability
- Recap: Clone-and-Own
- Recap: Conditional Compilation
- Recap: Modular Features
- Recap: Languages for Features
- Comparison of Implementation Techniques
- Summary

### 8c. Adoption of Product Lines

- Product-Line Adoption
- Proactive Adoption Strategy
- Extractive Adoption Strategy
- Reactive Adoption Strategy
- A Modern Process Model for Adopting and Evolving Product Lines
- Summary
- FAQ

# 8. Development Process – Handout

Software Product Lines | Thomas Thüm, Elias Kuiter, Timo Kehrer | June 14, 2023

# **8. Development Process**

## **8a. Process Model for Product Lines**

Recap: Process Models

Domain and Application Engineering

Analysis and Design

Implementation and Testing

Overview on Domain and Application Engineering

Problem and Solution Space

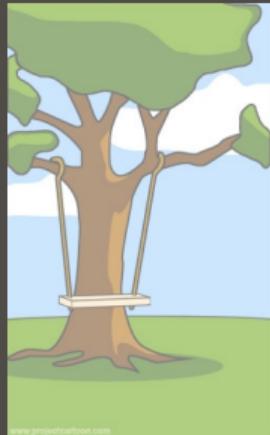
Summary

## **8b. Implementation of Product Lines**

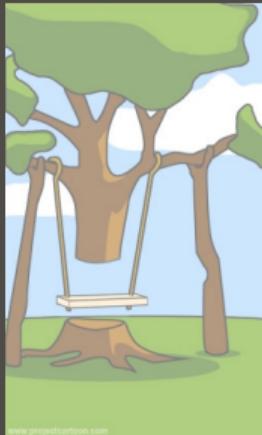
## **8c. Adoption of Product Lines**

# Recap: Process Models

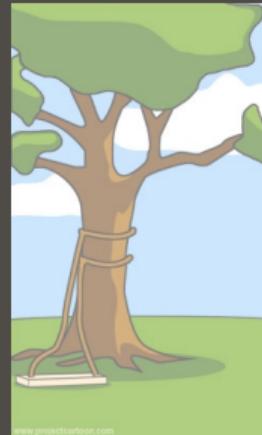
## Recap: The Software Life Cycle



Analysis



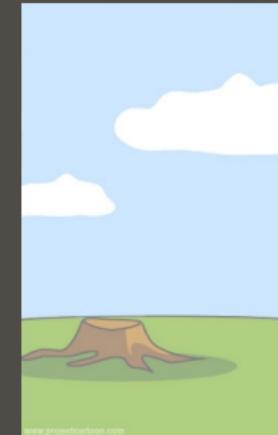
Design



Implementation



Testing



Maintenance

### Process Models for Single-System Engineering

waterfall model, V model, scrum, ...

### Process Models for Product-Line Engineering

???

# Recap: Domain

## Recap: Domain

[Lecture 1]

- “A **domain** is an area of knowledge that:
- is scoped to maximize the satisfaction of the requirements of its stakeholders,
  - includes a set of concepts and terminology understood by practitioners in that area,
  - and includes the knowledge of how to build software systems (or parts of software systems) in that area.”

# Domain and Application Engineering

## A Process Model for Product-Line Engineering

idea: split development into two phases, one for product line and one for products

### Domain Engineering

[Apel et al. 2013, pp. 21–22]

**“Domain engineering** is the process of analyzing the domain of a product line and developing reusable artifacts.”

### Domain Engineering

[Apel et al. 2013, p. 21]

- development for reuse
- prepares artifacts to be used in products (or during application engineering)
- goal: reduce effort per product (i.e., effort during application engineering)

### Application Engineering

[Apel et al. 2013, p. 21]

**“Application engineering** has the goal of developing a specific product for the needs of a particular customer (or other stakeholder).”

### Application Engineering

[Apel et al. 2013, p. 21]

- development with reuse
- build products using artifacts from domain engineering
- repeated for every product
- “application” of the product line (i.e., suitable for application and system software)



Product-Line Requirements

## Domain Engineering



Domain Analysis



Domain Design



Domain Implementation



Domain Testing



Product Requirements

## Application Engineering



Application Analysis



Application Design



Application Implementation



Application Testing



Product

# Domain and Application Analysis



## Domain Analysis [Apel et al. 2013, p. 21, Pohl et al. 2005, p. 25]

- requirements analysis for a product line
- define scope of the product line
- which features are in scope?
- which combinations of features are in scope?
- typically results in a feature model and features mapped to requirements

## Domain Scoping [Apel et al. 2013, p. 22]

which requirements of a domain are in scope for the product line?

- domain experts collect requirements (e.g., from existing systems, interviews, potential customers)
- often economical decision by managers

## Application Analysis [Apel et al. 2013, pp. 21–25]

- requirements analysis for a product
- based on the output of domain analysis
- ideally: customer requirements mapped to a feature selection
- alternative strategies for new and unsupported requirements:
  1. requirement is out of scope (i.e., no product made available)
  2. document for custom development (i.e., development in application engineering)
  3. integrate into domain analysis (i.e., development in domain engineering)
- best strategy depends on the situation

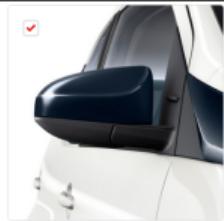
# Domain Scoping in Practice



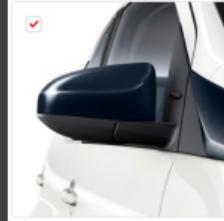
Mirror cover, Cyan Splash, left side  
£22.00  
More info ⓘ



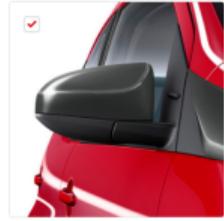
Mirror cover, Cyan Splash, right side  
£22.00  
More info ⓘ



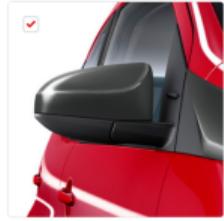
Mirror cover, Dark Blue Twinkle, left side  
£22.00  
More info ⓘ



Mirror cover, Dark Blue Twinkle, right side  
£22.00  
More Info ⓘ



Mirror cover, Electro Grey, left side  
£22.00  
More info ⓘ



Mirror cover, Electro Grey, right side  
£22.00  
More info ⓘ

## Your AYGO

5 Door Hatchback x-play 1.0 Petrol (69 hp)  
5-Speed Manual (Front Wheel Drive - FWD)

Retail price £10,910.00

Red Pop

Gleam fabric

15" steel wheels with wheel caps (6-spoke)

Mirror cover, Bold Black, right side £22.00 ×

Mirror cover, Bold Black, left side £22.00 ×

Mirror cover, Cyan Splash, left side £22.00 ×

Mirror cover, Cyan Splash, right side £22.00 ×

Mirror cover, Dark Blue Twinkle, left side £22.00 ×

Mirror cover, Dark Blue Twinkle, right side £22.00 ×

Mirror cover, Electro Grey, left side £22.00 ×

Mirror cover, Electro Grey, right side £22.00 ×

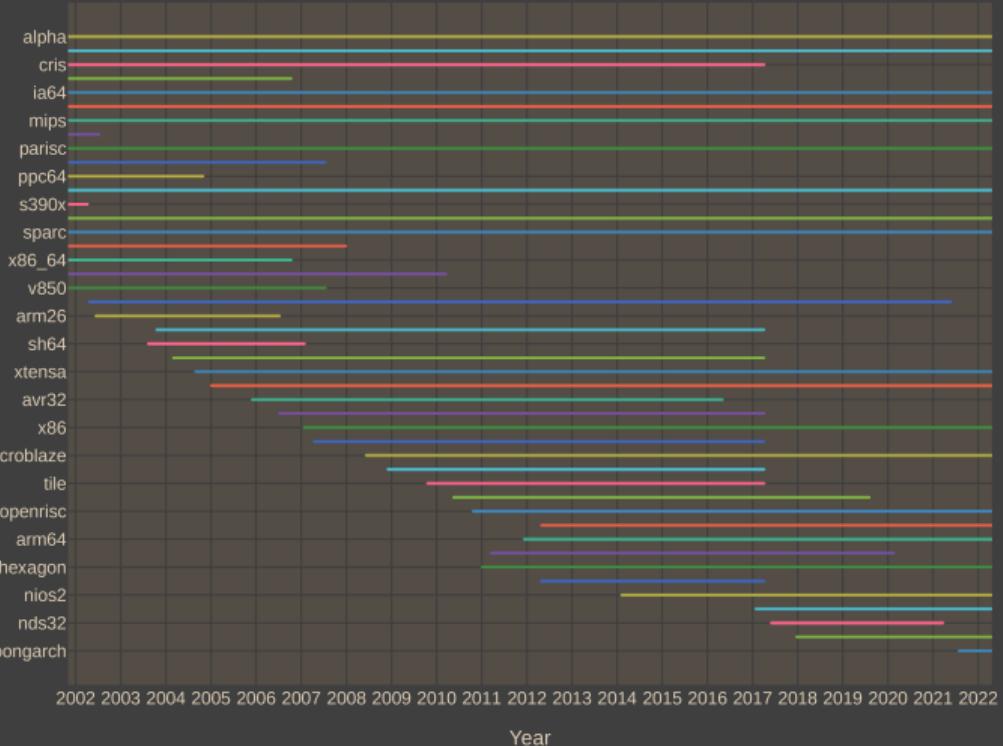
## Aygo Mirror Covers

Can customers choose . . .

- mirror covers? Yes!
- different covers for left and right side? Yes!
- multiple covers for the same side? Yes!
- between 2, 3, 4, . . . colors and which ones?

# Domain Scoping in Practice

Architectures of the Linux Kernel



## Linux Kernel Architectures

- in Linux,  $\approx$  20 CPU architectures are carefully maintained in 2023
- to keep it manageable, obsolete architectures are regularly removed
- requires an answer to “is this architecture still used by a relevant number of people?”

# Domain and Application Design



## Domain Design

[Pohl et al. 2005, p. 26]

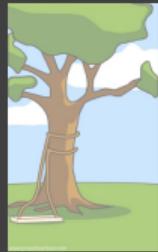
- development of a reference architecture (e.g., client-server or pipe-and-filter)
- common, high-level structure for all products
- decision on implementation technique
  - runtime variability: (immutable) global variables, method parameters
  - clone-and-own: ad-hoc, with version control, with build systems
  - conditional compilation: build systems, preprocessors
  - modular features: components, services, frameworks with plug-ins
  - modularization of crosscutting concerns: feature-oriented, aspect-oriented programming
  - combinations thereof

## Application Design

[Pohl et al. 2005, pp. 32–33]

- create application architecture
- derived from reference architecture
- based on feature selection
- design decisions for application-specific requirements

# Domain and Application Implementation



## Domain Implementation

[Apel et al. 2013, p. 21]

- development of reusable artifacts
- implementation of features identified during domain analysis
- implementation largely depends on the implementation technique chosen in domain design

## Application Implementation

[Apel et al. 2013, p. 21]

- development of products based on reusable artifacts
- ideally: fully automated generation (aka. **product derivation**)
- full automation not feasible . . .
  1. when custom development is needed (i.e., for application-specific requirements)
  2. for clone-and-own, components, services

# Domain and Application Testing



## Domain Testing

[Pohl et al. 2005, p. 27]

- validation and verification of reusable artifacts
- development of reusable tests
- testing of features in isolation, if possible
  - [more in Lecture 10]
- testing of sample products
  - [more in Lecture 11]

## Application Testing

[Pohl et al. 2005, pp. 33–34]

- testing of the application
- reuse of test artifacts from domain testing
- new test artifacts for custom development



Product-Line Requirements

### Domain Engineering



Domain Analysis



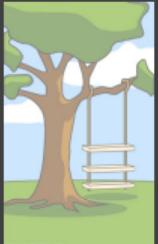
Domain Design



Domain Implementation



Domain Testing



Product Requirements

### Application Engineering



Application Analysis



Application Design



Application Implementation



Application Testing



Product

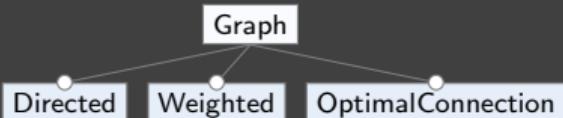
# Problem and Solution Space

## Problem Space

[Apel et al. 2013, p. 21]

"The **problem space** takes the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. Features are, in fact, domain abstractions that characterize the problem space."

[Lecture 4]



## Solution Space

[Apel et al. 2013, p. 21]

"The **solution space** represents the developer's and vendor's perspectives. It is characterized by the terminology of the developer, which includes names of functions, classes, and program parameters. The solution space covers the design, implementation, and validation and verification of features and their combinations in suitable ways to facilitate systematic reuse."

[Lecture 2, Lecture 3, Lecture 5, Lecture 6, Lecture 7]

```
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;  
    }  
    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));  
    } }
```

# Process Model for Product Lines – Summary

## Lessons Learned

- domain engineering: domain analysis, domain design, domain implementation, domain testing
- application engineering: application analysis, application design, application implementation, application testing
- domain scoping, product-line requirements, product requirements
- problem space and solution space

## Further Reading

- Apel et al. 2013
- Pohl et al. 2005

## Practice

1. Where to spend more resources in domain engineering or in application engineering?
2. What should not be done in domain engineering?

## 8. Development Process

### 8a. Process Model for Product Lines

### 8b. Implementation of Product Lines

Recap: Runtime Variability

Recap: Clone-and-Own

Recap: Conditional Compilation

Recap: Modular Features

Recap: Languages for Features

Comparison of Implementation Techniques

Summary

### 8c. Adoption of Product Lines

# Recap: Runtime Variability

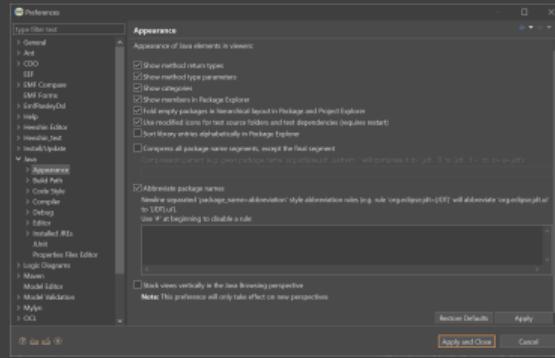
```
public class Config {  
    public static boolean COLORED = true;  
    public static boolean WEIGHTED = false;  
}
```

```
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(); }  
    }  
}
```

# Recap: Runtime Variability



The screenshot shows a Windows Command Prompt window with the following text:  
C:\>dir /?  
Displays a list of files and subdirectories in a directory.  
DIR [drive:][[path][filename] [/A[[:] attributes]] [/B] [/C] [/O] [/I] [/N] [/H] [/X] [/P] [/R] [/T[[:] timefield]] [/W] [/K]  
[drive:][[path][filename]]  
Specifies drive, directory, and/or files to list.  
/A Displays files with specified attributes.  
attributes D Directories H Hidden files R Read-only files  
S System files I No content indexed files  
L Registry Points O Offline files  
/B Use brief format (no heading information or summary).  
/C Use the thousand separator in file sizes. This is the default. Use /-C to disable display of separator.  
/D Same as wide but files are list sorted by column.  
/I /N New long list format where filenames are on the far right.  
/O List by files in sorted order.  
sortorder N By name (alphabetical) S By size (smallest first)  
Y By extension (alphanumeric) D By date/time (oldest first)  
G Sort files by group first Prefix to reverse order  
/P Pauses after each screenful of information.  
/Q Display the owner of the file.  
/R Display alternate data streams of the file.  
/S Displays files in specified directory and all subdirectories.  
Press any key to continue . . .

The screenshot shows the 'eclipse.ini - Editor' window with the following configuration options:  
Date: 2018-07-17 Time: 10:45:20  
-startos windows  
-launcher.library plugins/org.eclipse.equinox.launcher\_1.5.700.v20200207-2156.jar  
--launcher.library plugins/org.eclipse.equinox.launcher.win32.win32.x86\_64\_1.1.100.v20190907-0426  
-product org.eclipse.epp.package.modeling.product  
-showsplash  
org.eclipse.epp.package.common  
--launcher.defaultAction openFile  
--launcher.defaultAction  
--launcher.appendVmargs  
-vmargs  
-Dosgi.requiredJavaVersion=1.8  
-Dosgi.instance.area.default=user.home/eclipse-workspace  
-XX:+UseG1GC  
-XX:+UseStringDeduplication  
--add-modules=ALL-SYSTEM  
-Dosgi.requiredJavaVersion=1.8  
-Dosgi.instanceAreaRequiresExplicitInit=true  
-Xms156m  
-Xmx2048m  
--add-modules=ALL-SYSTEM

## 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

# Recap: 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

# Recap: Runtime Variability

## Basic Principles

### Variability with Configuration Options:

- Conditional statements controlled by configuration options
- Global variables vs. method parameters

### Object-Orientation and Design Patterns:

- Template Method
- Abstract Factory
- Decorator

## Problems of Runtime Variability

### Conditional Statements:

- Code scattering, tangling, and replication

### Design Patterns for Variability:

- Trade-offs and potential negative side effects
- Constraints that may restrict their usage

### In General:

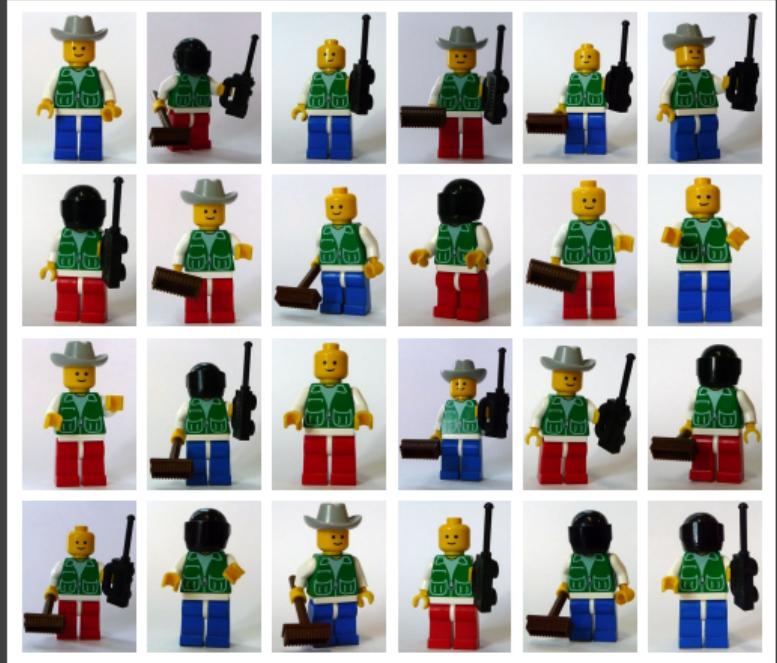
- Variable parts are always delivered
- Not well-suited for compile-time binding

# Recap: Clone-and-Own

## Clone-and-Own

- New variants of a software system are created by copying and adapting an existing variant.
- Afterwards, cloned variants evolve independently of each other.

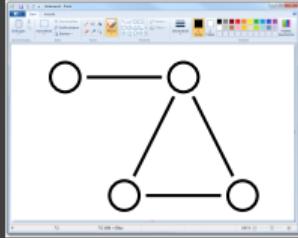
## Cloning Whole Products (Clone-and-Own)



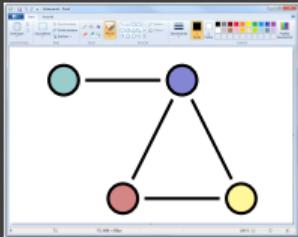
# Recap: Clone-and-Own



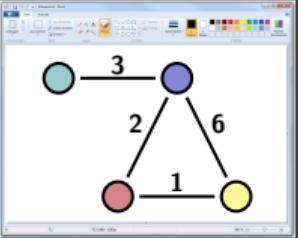
Alice



Bob



Eve



## 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?

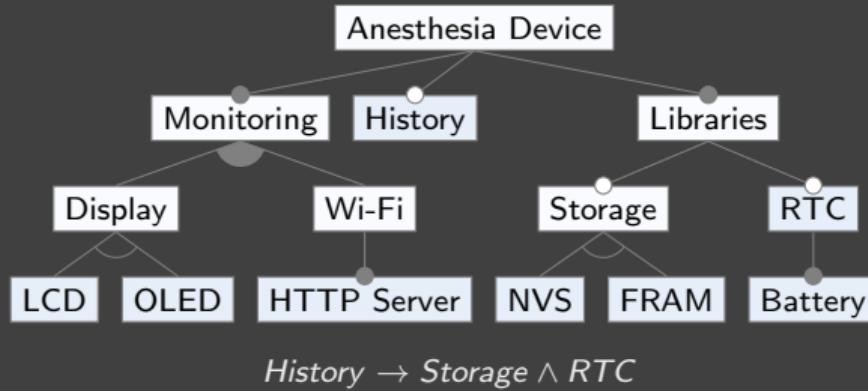
- compile-time variability only for implemented products
- no automated generation:
  - for clone-and-own (with version control systems)
- automated generation based on build script and extra files:
  - for clone-and-own with build systems
- no free feature selection (i.e., configuration)

# Recap: Conditional Compilation – 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  
⇒ one build script per group of related features



✓	📁 main-features	
✓	📁 lib	
⌚	battery.c	Battery
⌚	build.mk	Libraries
⌚	ds3231.c	RTC
⌚	fram.c	FRAM
⌚	i2cdev.c	FRAM
⌚	rtc.c	RTC
✓	📁 monitor	
⌚	build.mk	Monitor
⌚	display.c	Display
⌚	http.c	HTTP Server
⌚	lcd.c	LCD
⌚	oled.c	OLED
⌚	wifi.c	Wi-Fi
⌚	build.mk	Anesthesia Device
⌚	history.c	History
⌚	main.c	Anesthesia Device

# Recap: Conditional Compilation – 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 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 inclusion and exclusion of individual lines or chunks of code  
⇒ high-level use **only!**

# Recap: Conditional Compilation – Preprocessors

## 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 included header files
- preprocessors typically do not remove line breaks to not influence line numbers reported by compilers

# Recap: Conditional Compilation – Preprocessors

## Advantages

- well-known and mature tools, readily available
- easy to use  
⇒ just annotate and remove
- supports **compile-time variability**
- flexible, arbitrary levels of **granularity**
- can handle code and non-code artifacts (**uniformity**)
- little **preplanning** required  
⇒ variability can be added to an existing project

## Challenges

- **scattering** and **tangling**  
⇒ 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]

# Recap: Modular Features – Components

## General Idea

- every feature is implemented by a dedicated component
- feature selection determines which components shall be integrated to form an application

## Reality



## Vision



## Glue Code and Customization

- developers must connect components through glue code  
exception: components are only exchanged against alternative components with identical interface
- components may contain run-time variability  
e.g., color manager in our example may be parameterized by color model RGB or CMYK

# Recap: Modular Features – Services

## Recap: Component-Based Implementation



## Plenty of Glue Code



## Same Idea

- Features are implemented as services.
- Feature selection determines the services to be composed.

## However

"Standardized" service composition instead of highly individual glue code.



# Recap: Modular Features – Frameworks with Plug-Ins

## Recap: Service-Based Implementation



Still needs some specification of “composition” (cf. orchestration vs. choreography)



## Same Idea

- Features are implemented by different plug-ins
- Feature selection determines the plug-ins to be loaded and registered

neither glue code nor explicit service composition required



full automation comes at a price (cf. preplanning problem)

# Recap: Modular Features – Frameworks with Plug-Ins

In our example, we can observe that:

- There are lots of empty methods in the ColorPlugin
- The Framework consults all registered plug-ins before printing a node or edge

## General Challenge: Cross-cutting Concerns

Implementing cross-cutting concerns as plug-ins

- typically leads to huge interfaces, large parts of which are irrelevant for a dedicated plug-in
- causes lots of communication overhead between plug-ins and framework

If we were not familiar with our graph library, would we anticipate that:

- Colors and weights should be part of the Plugin interface?
- Every plug-in needs to be notified that the framework is about to print a node or edge?

## Generally known as Preplanning Problem

- Hard to identify and foresee the relevant hot spots and nature of extensions
- Developing a framework needs lots of expertise and excellent domain knowledge

# Recap: Languages for Features – Feature-Oriented Programming

## Feature Modules

- Each collaboration mapped to a feature and is called a feature module (or layer).
- Feature modules may refine a base implementation by adding new elements or by modifying and extending existing ones.

## Feature Module Composition

Selected feature modules may be superimposed by lining-up classes according to the roles they play.



# Recap: Languages for Features – Feature-Oriented Programming

## Advantages

- Easy to use language-based mechanism, requires only minimal language extensions.
- Conceptually uniformly applicable to code and noncode artifacts.
- Separation of (possibly crosscutting) feature code into distinct feature modules.
- Little preplanning required due to mixin-based extension mechanism.
- Direct feature traceability from a feature to its implementation in a feature module.

## Disadvantages

- Requires adoption of a language extension and composition tools.
- Tools need to be constructed for every language (although with the help of a framework).
- Only academic tools so far, little experience in practice.
- Granularity restricted to method-level (or other named structural entities).

# Recap: Languages for Features – Aspect-Oriented Programming

## Basic Idea

- Implement one aspect per feature.
  - Feature selection determines the aspects which are included in the weaving process.
- 
- Aspects encapsulate changes to be made to existing classes.
  - However, aspects do not encapsulate new classes introduced by a feature (only nested classes within an aspect)

## A Color Feature for Graphs

```
aspect ColorFeature {  
    Color Node.color = new Color();  
  
    before(Node n): execution(void print()) && this(n) {  
        Color.setDisplayColor(n.color);  
    }  
  
    static class Color {  
        ...  
    }  
}
```

# Recap: Languages for Features – Aspect-Oriented Programming

## Advantages

- Separation of (possibly crosscutting) feature code into distinct aspects.
- Direct feature traceability from feature to its implementation in an aspect.
- Little or no preplanning effort required.
- Fine-grained variability driven by the join-point model of the aspect-oriented language.

## Disadvantages

- Requires adoption of a rather complex extension mechanism (new language and paradigm).
- No unifying theory like no language-independent framework.
- Program evolution and maintenance affected by fragile-pointcut problem.

# Comparison of Implementation Techniques

	Compile-Time Variability	Features	Product Generation	Feature Traceability
Runtime Variability	no (very limited for immutable global variables)	only fine grained	yes (except for preference dialogs)	no
Clone-and-Own	yes (only for implemented products)	no	no (limited generation for clone-and-own with build systems)	no
Build Systems (for Conditional Compilation)	yes	only coarse grained	yes	with tool support
Preprocessors (for Conditional Compilation)	yes	only fine grained	yes	with tool support
Components/Services	yes	only coarse grained	no (except pure exchange)	only coarse grained
Frameworks with Plug-Ins	yes	only coarse grained	yes	only coarse grained
Feature Modules/Aspects	yes	yes	yes	yes

## Further Criteria

interfaces between features? code duplication necessary? modularization of cross-cutting concerns? ...

# Implementation of Product Lines – Summary

## Lessons Learned

- nine implementation techniques
- + three implementation strategies for runtime variability / clone-and-own
- + three configuration strategies for runtime variability
- choice based on four criteria, but there are more

## Further Reading

Apel et al. 2013

## Practice

1. Which techniques enable interfaces between features?
2. Which techniques require most code clones?
3. Which techniques can modularize cross-cutting concerns?

## **8. Development Process**

### **8a. Process Model for Product Lines**

### **8b. Implementation of Product Lines**

### **8c. Adoption of Product Lines**

Product-Line Adoption

Proactive Adoption Strategy

Extractive Adoption Strategy

Reactive Adoption Strategy

A Modern Process Model for Adopting and Evolving Product Lines

Summary

FAQ

# Product-Line Adoption

How to introduce a product line in practice?

# Proactive Adoption Strategy

[Apel et al. 2013, p. 40]

## Proactive Adoption

- development of a product line from scratch
- process model as presented in Part 1
- often seen as idealistic, academic
- comparable to the waterfall model

### Advantages

- desired variability planned first
- potentially higher code quality

### Disadvantages

- high up-front investment and risks
- late time-to-market
- production stop: developers develop product line rather products
- no reuse of existing products

# Extractive Adoption Strategy

[Apel et al. 2013, pp. 40–41]

## Extractive Adoption

- migrate one existing product into a product line (with or without runtime variability)
- or: migrate several cloned products into a product line (cf. clone-and-own)
- often motivated by maintenance problems after inconsistent evolution
- requires identification of commonalities and variabilities
- extraction of reusable artifacts
- very common in practice

## Advantages

- lower risks and up-front investment
- all products remain in production

## Disadvantages

- code quality depends on tools for extraction
- limited choice of implementation techniques

# Reactive Adoption Strategy

[Apel et al. 2013, pp. 41–42]

## Reactive Adoption

- start with one or a few products
- incrementally develop more features resulting in more products
- gradually reach the ideal product line
- requires to identify order among features (products)
- comparable to agile methods

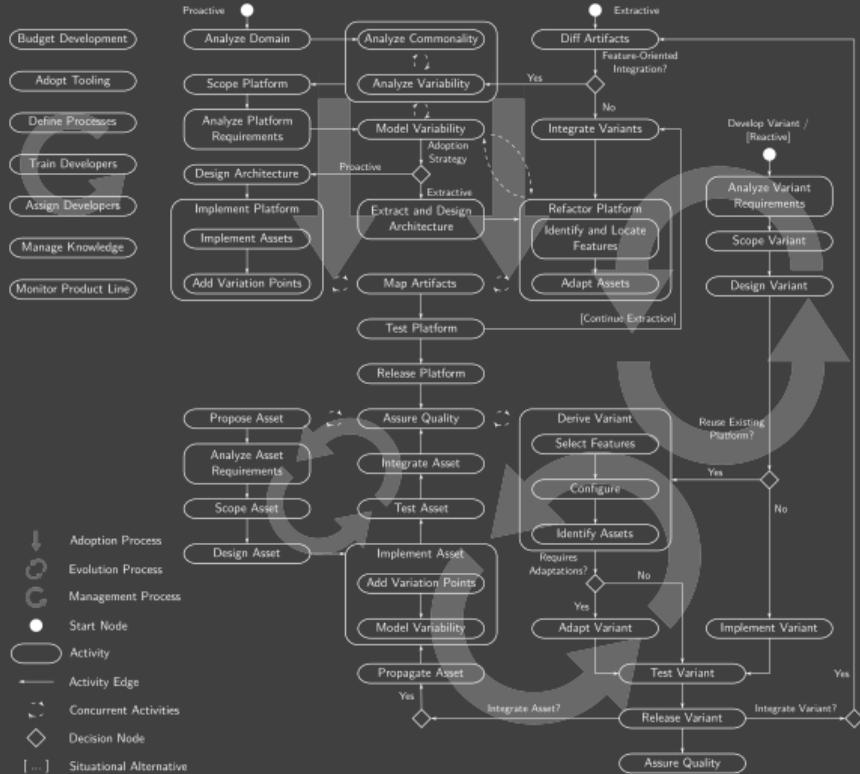
## Advantages

- less up-front investment than the proactive strategy
- also applicable for evolution of a product line (not only adoption)

## Disadvantages

- more changes to architecture and design necessary as not all features planned up-front
- no reuse of existing products

# A Modern Process Model for Adopting and Evolving Product Lines



## promote-pl

[Krüger et al. 2020]

- domain and application engineering only reflect the proactive adoption strategy accurately
- promote-pl is a modern process model that also integrates the reactive and extractive adoption strategy
- more complex, but aligns better with modern development practices

# Adoption of Product Lines – Summary

## Lessons Learned

- adoption strategies to introduce a product line
- proactive, extractive, reactive
- all applied in practice

## Further Reading

Apel et al. 2013, pp. 39–42

## Practice

Are combinations of the adoption strategies feasible?

# FAQ – 8. Development Process

## Lecture 8a

- How do process models for single-system engineering differ to those for product-line engineering?
- What are the two main phases when developing product lines?
- What are (the phases in) domain and application engineering?
- What is happening in which phase?
- How do phases interplay with each other?
- What is domain scoping? Why is it relevant?
- What is the difference between problem and solution space?

## Lecture 8b

- How to implement product lines?
- Which implementation techniques exist? What are further variations of those techniques?
- How to develop new features or variants? Exemplify!
- What are (dis)advantages of each technique?
- When to prefer one implementation technique over another?
- Which techniques support compile-time variability, features, product generation, feature traceability, interfaces between features, reduction of code duplication, modularization of crosscutting concerns?

## Lecture 8c

- How to introduce a product line?
- What are strategies for product-line adoption?
- What are (dis)advantages of product-line adoption strategies?
- When to prefer one adoption strategy over another?
- Are combinations of adoption strategies feasible?
- What is the connection between process models and adoption strategies for product lines?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 9a. What is a Feature Interaction?

- Examples for Feature Interactions
- Feature Interactions
- Example Interactions with Preprocessors
- Higher-Order Interactions
- Summary

### 9b. How to Handle Feature Interactions?

- Motivation and Goals
- Strategy S1: Adapt Feature Model
- Strategy S2: Orthogonal Implementation
- Strategy S3: Duplicate Implementations
- Strategy S4: Move Source Code
- Strategy S5: Conditional Compilation
- Strategy S6: Derivative Modules
- Overview and Discussion
- Summary

### 9c. How to Avoid Feature Interactions?

- The Choice of Features
- Documentation of Interactions
- Summary
- FAQ

# 9. Feature Interactions – Handout

Software Product Lines | Thomas Thüm, Timo Kehrer, Elias Kuiter | June 21, 2023

# 9. Feature Interactions

## 9a. What is a Feature Interaction?

Examples for Feature Interactions

Feature Interactions

Example Interactions with Preprocessors

Higher-Order Interactions

Summary

## 9b. How to Handle Feature Interactions?

## 9c. How to Avoid Feature Interactions?

# An Interaction when Customizing Clothes

## Customization of Clothes

- platforms to create and buy clothes
- creator preselects clothes with certain colors
- customization with pictures in different colors
- where is the problem?

The image displays two side-by-side screenshots of a website for customizing clothing, specifically shirts, featuring the University of Ulm logo.

**Screenshot 1 (Left):** Shows the homepage of "uulm Shop". The header includes the "uulm" logo, a search bar, and navigation links for "All designs", "Men", "Women", "Organic products", "Accessories", and "Topics". Below the header, a section titled "All designs" shows four examples of shirts with the "uulm" logo in different configurations: "Logo white print only on back & polo shirt", "Logo black print only on back & polo shirt", "Signature mark white print only on chest", and "Signature mark black print only on chest". At the bottom of the page are links for "Privacy", "Legal information", "Tracking", "Copyright information", "Help", and payment method icons for "PayPal", "VISA", and "MasterCard". A footer link "Open Your free Spreadshirt Now" is also present.

**Screenshot 2 (Right):** Shows a product page for a "Signature mark black (print only on chest)" shirt. The page includes a detailed description of the item, color selection options, and a "Buy now" button. It also features a grid of other shirt designs, including "Logo white print only on back & polo shirt" and "Signature mark white print only on chest". The footer of this page is identical to the one in Screenshot 1.

[myspreadshop.de]

# An Interaction when Customizing Clothes

## The Problem: Unwanted Interaction of Colors



**Wir brauchen eine Entscheidung von Dir**

Hallo,

vielen Dank für Deine Bestellung! Bevor wir Deine Ware bedrucken können, brauchen wir eine Entscheidung von Dir.

**Bei der Produktion sind wir auf ein kleines Problem gestoßen:**

Die Farbe Deines Designs ist der Farbe Deines Produkts zu ähnlich. Wenn der Kontrast zwischen Druck- und Stofffarbe zu gering ausfällt, wird das Design erfahrungsgemäß schlecht zu erkennen sein.

## The Solution: Choose Other Colors

Unser Lösungsvorschlag:

Wir können die Farbe des Produkts für Dich ändern. In einzelnen Fällen ist es auch möglich, die Farbe des Designs anzupassen. Oder wir belassen die Bestellung genauso wie sie ist, weil Du Dich bewusst für einen geringen Kontrast entschieden hast.

**Bestellung einsehen**

Bitte kontaktiere schnellstmöglich unseren Kundenservice, indem Du auf diese Mail antwortest oder anrufst unter 0341 59 400 5900 (Mo-Fr 9-18 Uhr)

Wenn wir bis zum 02.08.2021 keine Antwort erhalten, werden wir Deine Bestellung stornieren und gegebenenfalls den Rechnungsbetrag erstatten.

Wir freuen uns auf Deine Rückmeldung.

Viele Grüße  
Dein Team von Spreadshirt

# An Interaction when Customizing Clothes

The Problem: Unwanted Interaction of Colors

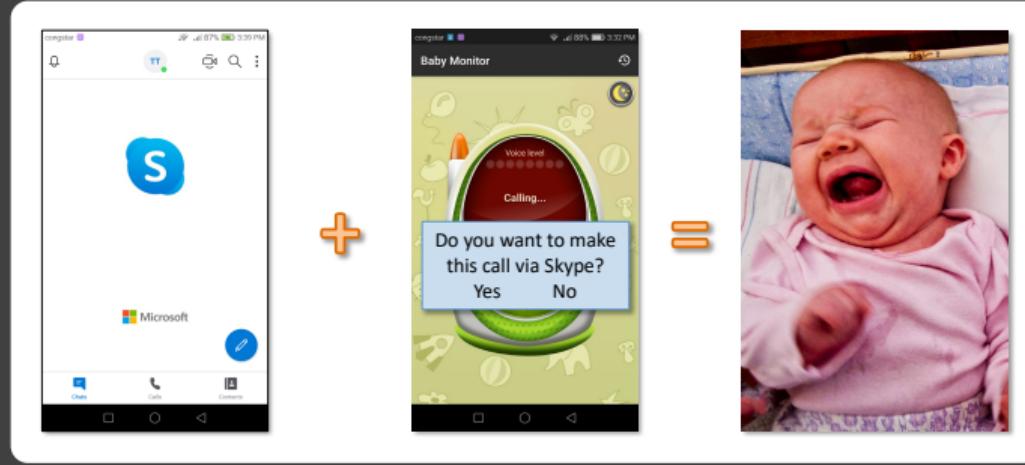


The Solution: Choose Other Colors



seems that contrast is checked for each order (cf. application engineering)  
and not for each published design (cf. domain engineering)

# An Interaction of Android Apps



## Skype vs BabyMonitor

- Skype app installed and used for years
- BabyMonitor installed, carefully tried and used for months
- BabyMonitor can call any other number (i.e., works without internet)
- automatic update of the Skype app
- update causes a question to be asked for every call
- what is the problem?

# Feature Interactions

## Feature Interaction

[Apel et al. 2013, p. 214]

"A **feature interaction** between two or more features is an emergent behavior that cannot be easily deduced from the behaviors associated with the individual features involved."  
for short: interaction

## Inadvertent Interaction

[Apel et al. 2013, p. 214]

"An **inadvertent feature interaction** occurs when a feature influences the behavior of another feature in an unexpected way (for example, regarding the expected control flow, program or data state, or visible behavior)."

here simplified as: wanted vs unwanted

## Feature-Interaction Problem

[Apel et al. 2013, p. 214]

"The **feature-interaction problem** is to detect, manage, and resolve (inadvertent) feature interactions among features."

## Feature Interactions

- detection discussed in next two lectures

[Lecture 10 and Lecture 11]

- resolving interactions (see Part II)
- managing interactions (see Part III)

## What's Next?

- interactions due to the absence of features
- interactions in source code
- interactions with the base code

## A Common Interaction of Toasters



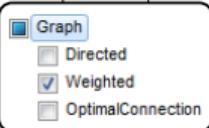
no interaction for two toasts (i.e.,  $T_1 \wedge T_2$  shown)  
and for no toasts (i.e.,  $\neg T_1 \wedge \neg T_2$  not shown)



unwanted interaction for one toast  
(i.e.,  $T_1 \wedge \neg T_2$  shown and  $\neg T_1 \wedge T_2$  not shown)

# Example Interactions with 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));  
    } }
```



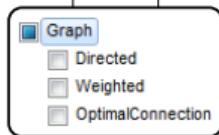
```
class Edge {  
    Node first, second;  
    int weight;  
    Edge(Node first, Node second) {...}  
    boolean equals(Edge e) {  
        return (first == e.first  
            && second == e.first  
            || first == e.second  
            && second == e.first  
            ) && weight ==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

## No Interaction?

- configuration for undirected, weighted edges
- product can be compiled
- what is the problem?

# Example Interactions with 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));  
    } }
```



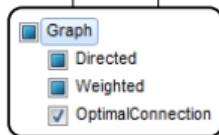
```
class Edge {  
    Node first, second;  
  
    Edge(Node first, Node second) {...}  
    boolean equals(Edge e) {  
        return (first == e.first  
            && second == e.first  
            || first == e.second  
            && second == e.first  
            ) && weight == e.weight ;  
    }  
    void testEquality() {  
        Node a = new Node();  
        Node b = new Node();  
        Edge e = new Edge(a, b);  
        Assert.assertTrue(e.equals(e));  
    } }
```

## Static Interaction

- configuration for undirected, unweighted edges
- product cannot be compiled due to static feature interaction
- field **weight** used for undirected edges but defined in feature **Weighted**
- occurs whenever features **Directed** and **Weighted** are both not selected

# Example Interactions with 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));  
    } }
```



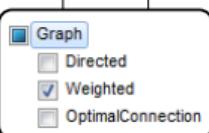
```
class Edge {  
    Node first, second;  
    int weight;  
    Edge(Node first, Node second) {...}  
    boolean equals(Edge e) {  
        return (first == e.first  
            && second == e.first  
            ) && weight == e.weight;  
    }  
    void testEquality() {  
        Node a = new Node();  
        Node b = new Node();  
        Edge e = new Edge(a, b);  
        Assert.assertTrue(e.equals(e));  
    } }
```

## Other Static Interaction

- configuration for directed, weighted edges
- product cannot be compiled due to static feature interaction
- semicolon and bracket missing for every configuration with feature **Directed**
- feature **Directed** has inadvertent interaction with base code

# Example Interactions with 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));  
    } }
```

## Dynamic Interaction?

- again: configuration for directed, weighted edges
- product can be compiled but test fails
- not a static interaction
- defect in the base code
- no interaction at all

## Detection of Interactions

- static interactions [Lecture 10]
- dynamic interactions [Lecture 11]
- next: interactions of more than two features

# Higher-Order Interactions

## Kinds of Interactions

- wanted and unwanted interactions
- static and dynamic interactions
- one-wise interactions (interaction of features with the base code and faulty features)
- pair-wise interactions (between two features)
- higher-order interactions (between more than two features)

## Variability Bug Database

[VBDb]

- database of known feature interactions
- operating system Linux (43 interactions)
- web server Apache (23)
- system tool Busybox (18)
- 3D printer firmware Marlin (14)

## Patterns of Feature Interactions

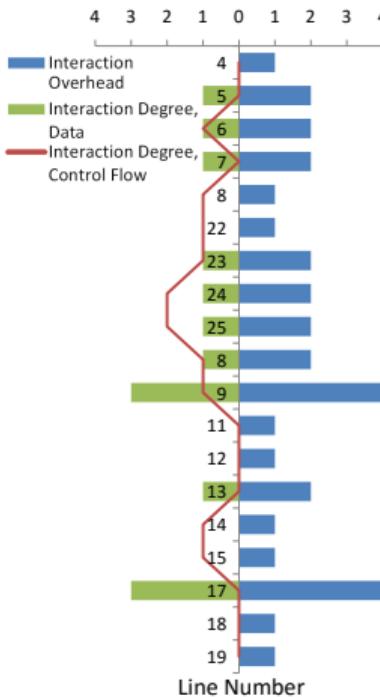
[VBDb]

L	precondition	M	B	A	$\Sigma$
21	<b>some enabled:</b>	9	7	14	49
5	$a$	6	3	7	21
10	$a \wedge b$	3	3	5	21
5	$a \wedge b \wedge c$		1		6
1	$a \wedge b \wedge c \wedge d \wedge e$				1
20	<b>some-enabled-one-disabled:</b>	4	11	10	45
3	$\neg a$	1	6	10	20
13	$a \wedge \neg b$	3	4		20
3	$a \wedge b \wedge \neg c$		1		4
1	$a \wedge b \wedge c \wedge d \wedge \neg e$				1
2	<b>other configurations:</b>	1		1	4
1	$\neg a \wedge \neg b$				1
	$a \wedge \neg b \wedge \neg c$	1			1
1	$a \wedge \neg b \wedge \neg c \wedge \neg d \wedge \neg e$			1	2
43	<b>TOTAL</b>	14	18	23	98

# Interaction on Data and Control Flow

[Meinicke et al. 2016]

```
1 boolean STATISTICS, SMILEY, WEATHER,
2     FAHRENHEIT, SECURE_LOGIN;
3 void createHtml() {
4     String c = wpGetContent();
5     if (SMILEY)
6         c = c.replace(":]", getSmiley(":]"));
7     if (WEATHER) {
8         String weather = getWeather();
9         c = c.replace("[:weather:]", weather);
10    }
11    String head = initHeader();
12    print("<html><head>" + head + "</head><body>");
13    if (STATISTICS) {
14        long time = getCurrentTime();
15        printStatistics(time);
16    }
17    print("<div>" + c + "</div>");
18    String foot = wpGenFooter();
19    print("<hr/>" + foot + "</body></html>");
20 }
21 String getWeather() {
22     float temperature = getCelsius();
23     if (FAHRENHEIT)
24         return (temperature * 1.8 + 32) + "°F";
25     return temperature + "°C";
26 }
```

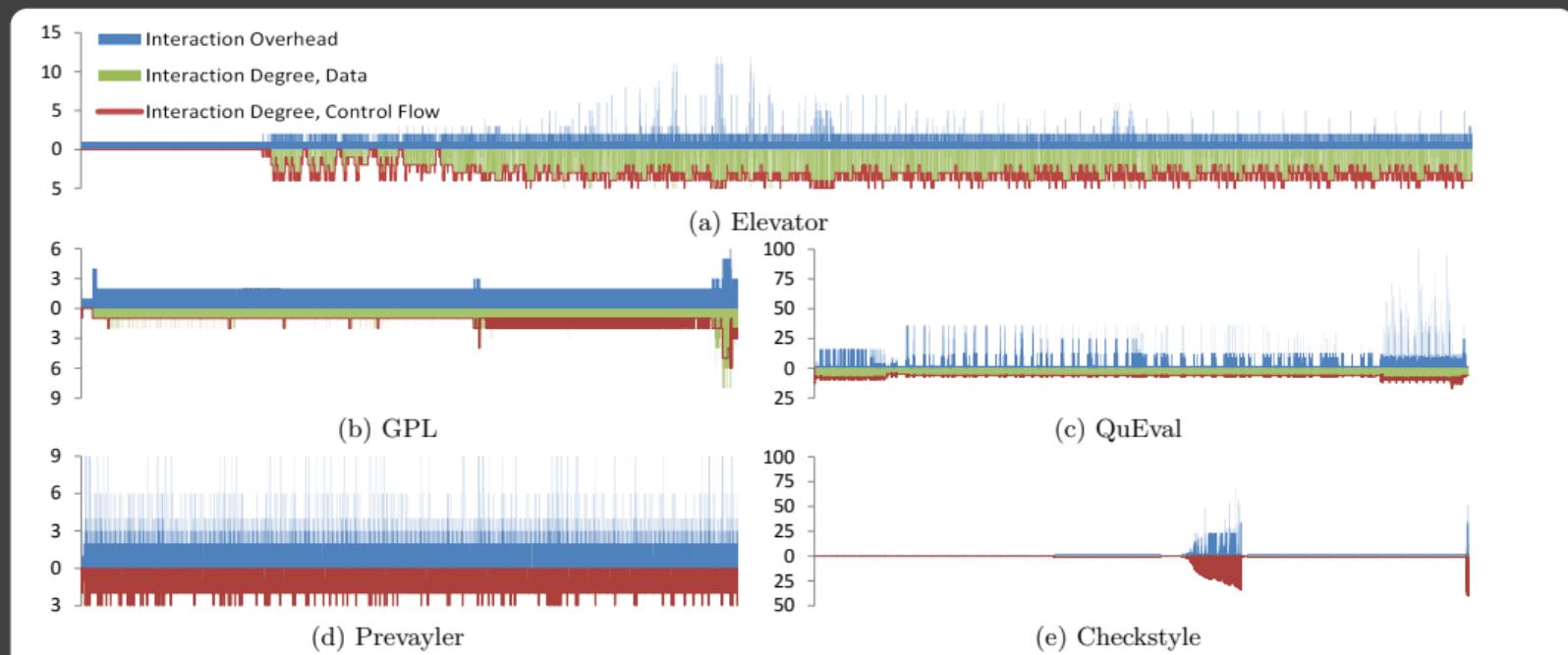


## How do Features Interact?

- given a program with runtime variability
- given one test case (i.e., concrete inputs)
- how much does the execution depend on the configuration?
- how many values for each variable? (green)
- how many different control flows? (red)
- blue color not relevant here (minimal overhead during simultaneous execution)

# Execution Traces in Configurable Systems

[Meinicke et al. 2016]



insights: not all features interact. some statements may lead to higher interactions than others.

# What is a Feature Interaction? – Summary

## Lessons Learned

- feature interaction and feature-interaction problem
- wanted/unwanted, static/dynamic, one-wise/pair-wise/higher-order
- examples: customization of clothes, Android apps, toaster, preprocessor code, runtime variability, Variability Bug Database

## Further Reading

- Apel et al. 2013, Chapter 9, pp. 213–217

## Practice

Do you know further examples of feature interactions?

## 9. Feature Interactions

### 9a. What is a Feature Interaction?

### 9b. How to Handle Feature Interactions?

Motivation and Goals

Strategy S1: Adapt Feature Model

Strategy S2: Orthogonal Implementation

Strategy S3: Duplicate Implementations

Strategy S4: Move Source Code

Strategy S5: Conditional Compilation

Strategy S6: Derivative Modules

Overview and Discussion

Summary

### 9c. How to Avoid Feature Interactions?

# Handling/Implementing Feature Interactions

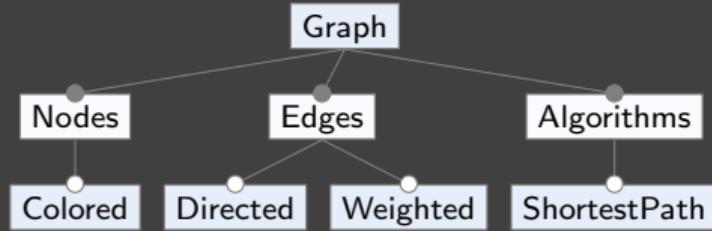
## Assumptions

- Interacting features have been already identified
- Interaction is a pairwise interaction (i.e., two features)

## Problem Description

- Find a strategy to handle the interaction
- Strategy does not introduce the optional-feature problem

# Running Example: A Feature Interaction in our Graph Library



## Optional-Feature Problem

- Domain view: *Weighted* and *ShortestPath* can be deliberately selected independent of each other.
- Implementation view: *ShortestPath* requires *Weighted* due to an implementation dependency.

### Feature Module BasicGraph

```
class Edge {  
    Node a, b; ...  
}
```

### Feature Module Weighted

```
refines class Edge {  
    double weight; ...  
}
```

### Feature Module ShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        Edge e1, e2;  
        ...  
        if (e1.weight > e2.weight)  
        ...  
    }  
}
```

# Handling Feature Interactions: General Goals

## Question

What makes a good strategy to implement coordination code for feature interactions (while solving/avoiding the optional-feature problem)?

### 1. Variability

For every valid configuration (according to feature model), we can generate a product that implements this configuration.

### 2. Implementation Effort

Should not require overwhelming implementation effort (would not be attractive in practice).

### 3. Binary Size and Performance

Should not increase binary size or decrease performance of products compared to an individual implementation of each product.

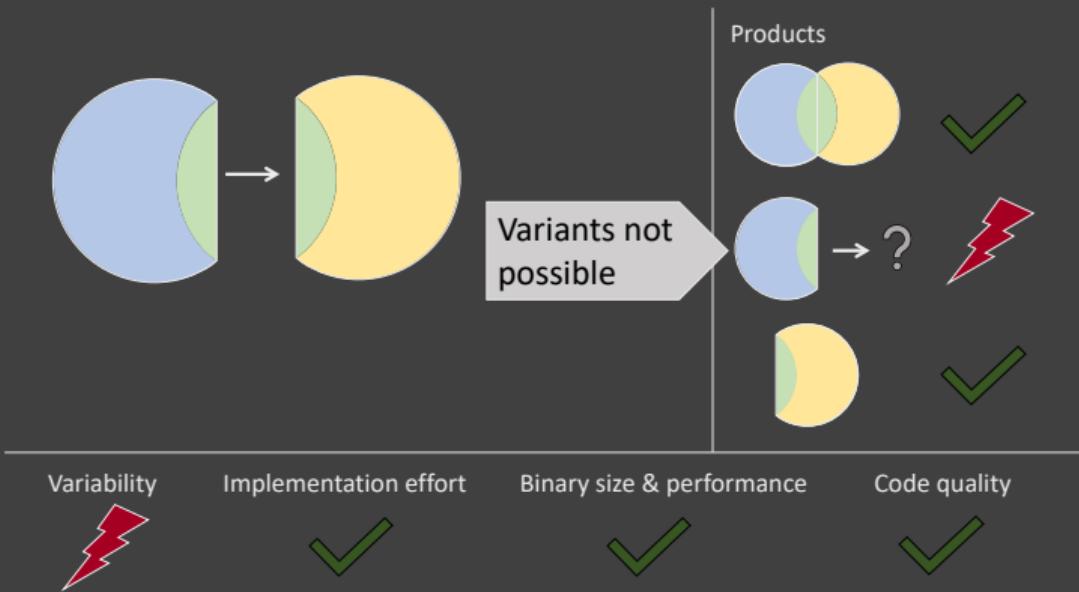
### 4. Code Quality

Should not reduce code quality, which would make the product line harder to maintain.

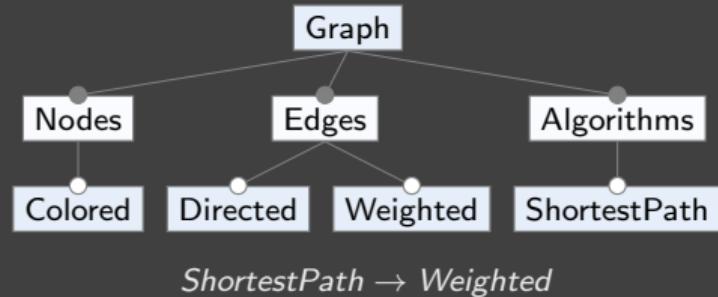
## Strategy S1: Adapt Feature Model – (a) Add Domain Dependency

### Strategy S1a

Declare implementation dependency as domain dependency in the feature model.



## Strategy S1: Adapt Feature Model – (a) Add Domain Dependency



Same implementation as before, but we make implementation dependency explicit: *ShortestPath* requires *Weighted*.

### Feature Module BasicGraph

```
class Edge {  
    Node a, b; ...  
}
```

### Feature Module Weighted

```
refines class Edge {  
    double weight; ...  
}
```

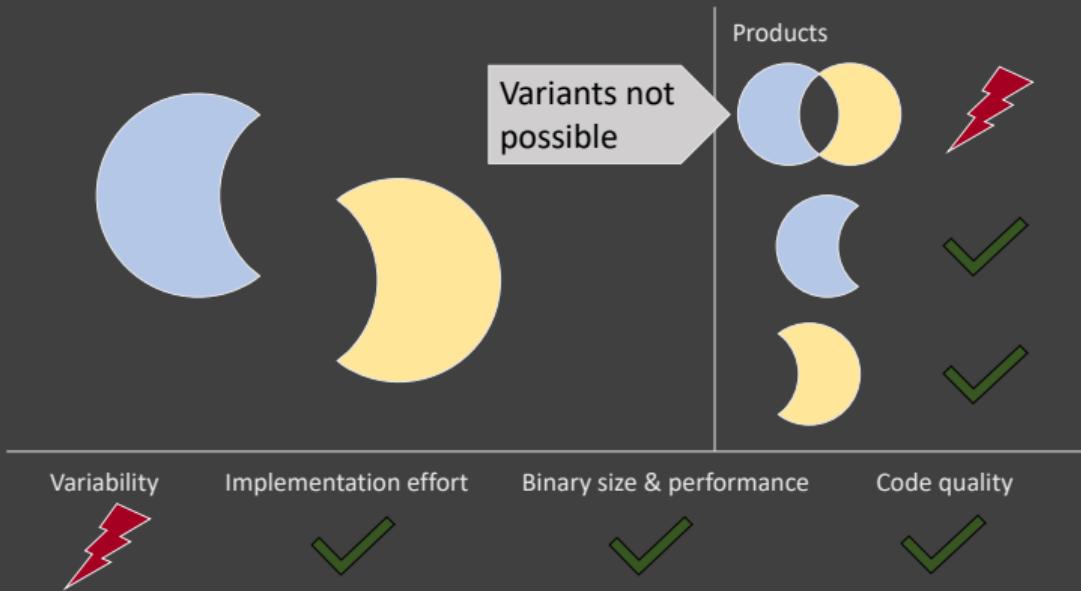
### Feature Module ShortestPath

```
refines class Graph {  
    List<Node> shortestPath(Node a, Node b){  
        Edge e1, e2;  
        ...  
        if (e1.weight > e2.weight)  
            ...  
    }  
}
```

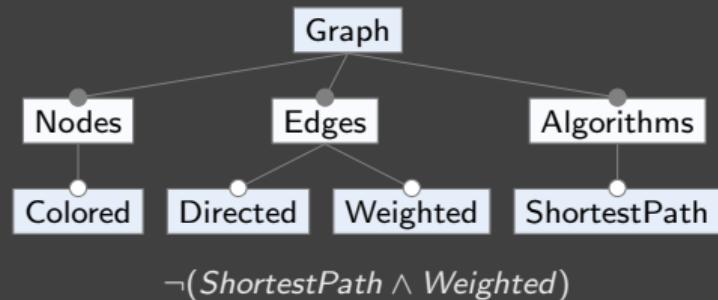
## Strategy S1: Adapt Feature Model – (b) Exclude Feature Combination

### Strategy S1b

Declare problematic feature combinations as mutually exclusive in the feature model.



# Strategy S1: Adapt Feature Model – (b) Exclude Feature Combination



We may safely assume any uniform weight because *ShortestPath* and *Weighted* are mutually exclusive.

## Feature Module BasicGraph

```
class Edge {  
    Node a, b; ...  
}
```

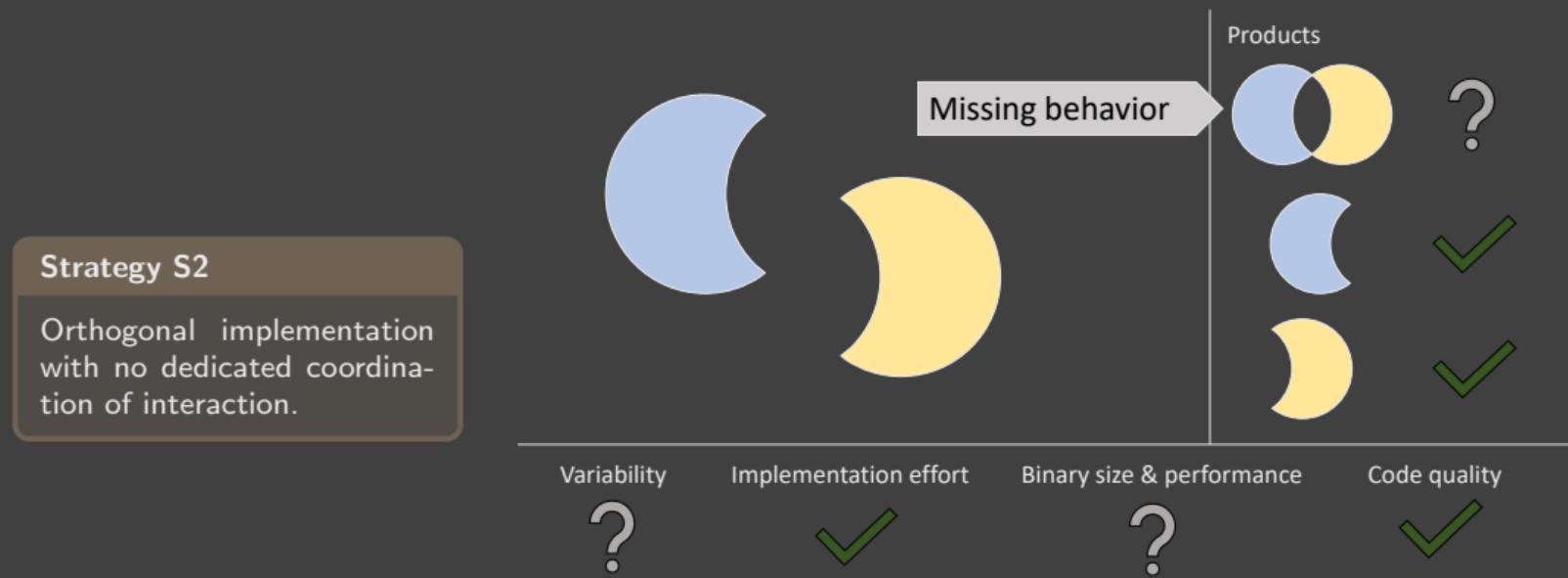
## Feature Module Weighted

```
refines class Edge {  
    double weight; ...  
}
```

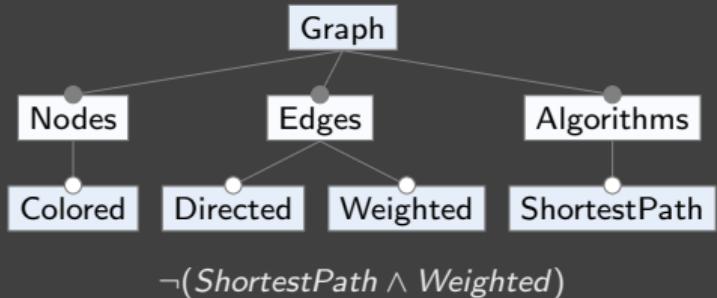
## Feature Module ShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        double w1 = 1.0;  
        double w2 = 1.0;  
        ...  
        if (w1 > w2)  
            ...  
    }  
}
```

## Strategy S2: Orthogonal Implementation



## Strategy S2: Orthogonal Implementation



Calculation of shortest path ignores weights but merely counts the number of edges on a path.

### Feature Module BasicGraph

```
class Edge {  
    Node a, b; ...  
}
```

### Feature Module Weighted

```
refines class Edge {  
    double weight; ...  
}
```

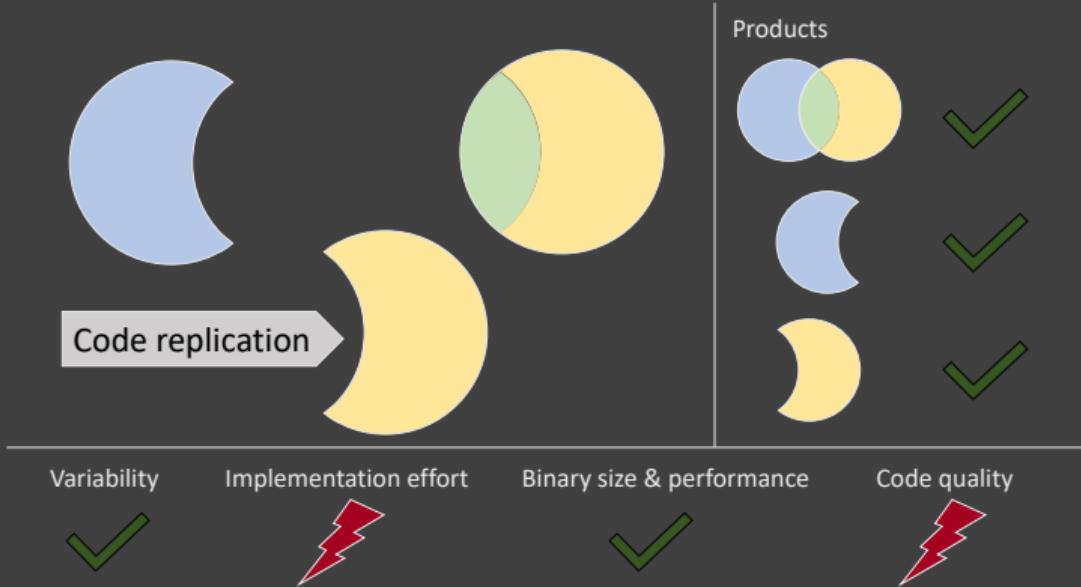
### Feature Module ShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        // ignore weights  
        ...  
    }  
}
```

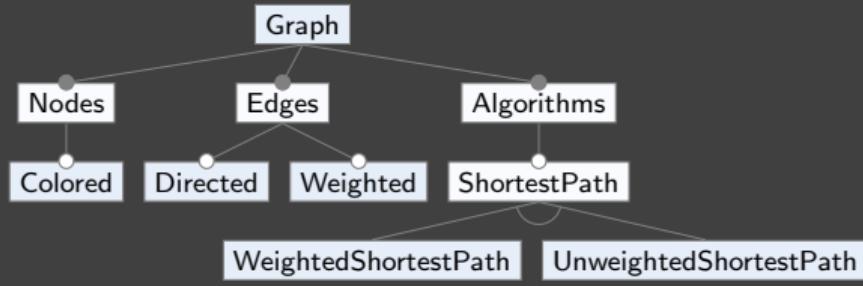
## Strategy S3: Duplicate Implementations

### Strategy S3

Multiple implementations of a feature, with and without coordination code.



# Strategy S3: Duplicate Implementations



*WeightedShortestPath*  $\leftrightarrow$  *ShortestPath*  $\wedge$  *Weighted*

*UnweightedShortestPath*  $\leftrightarrow$  *ShortestPath*  $\wedge$   $\neg$  *Weighted*

One of two different implementations chosen based on selection of interacting features.

## Feature Module UnweightedShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        ...  
        ...  
        // ignore weights  
        ...  
    }  
}
```

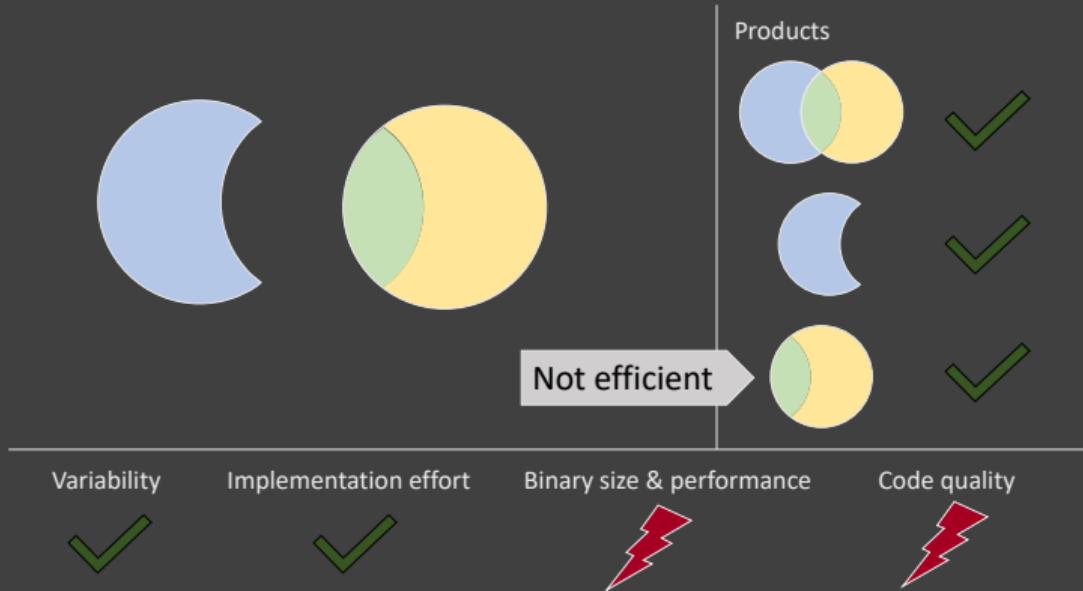
## Feature Module WeightedShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        Edge e1, e2;  
        ...  
        if (e1.weight > e2.weight)  
        ...  
    }  
}
```

## Strategy S4: Move Source Code

### Strategy S4

Move all the coordination code to one of the features (or to a third one all interacting features depend on).



# Strategy S4: Move Source Code

## Feature Module BasicGraph

```
class Edge {  
    Node a, b;  
    double weight = 1.0;  
    ...  
}
```

## Feature Module Weighted

```
refines class Edge {  
    ...  
}
```

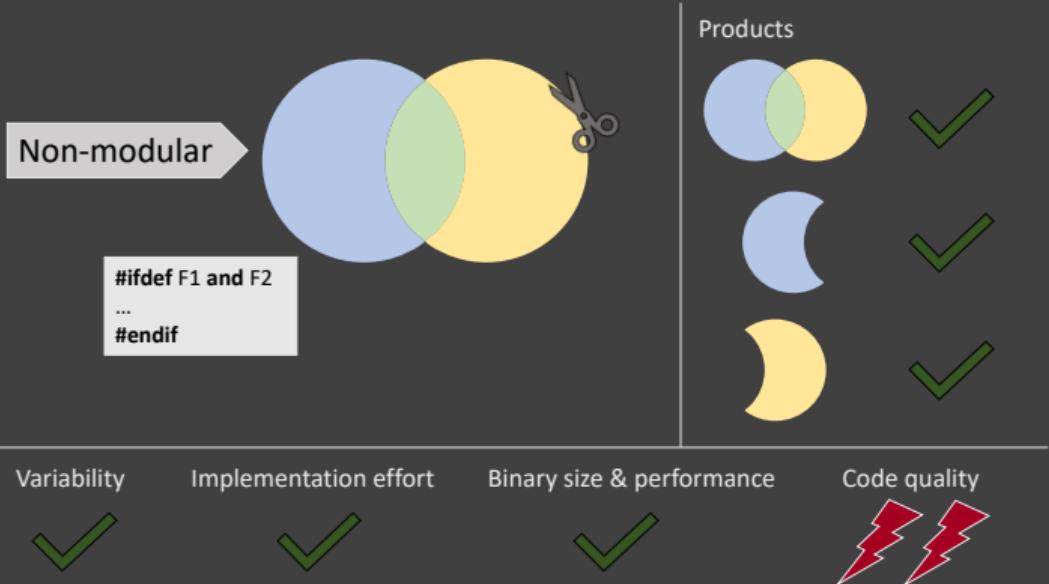
## Feature Module ShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        Edge e1, e2;  
        ...  
        if (e1.weight > e2.weight)  
            ...  
    }  
}
```

## Strategy S5: Conditional Compilation

### Strategy S5

Coordination code is only executed if both features are selected.



# Strategy S5: Conditional Compilation

## Feature Module BasicGraph

```
class Edge {  
    Node a, b; ...  
}
```

## Feature Module Weighted

```
refines class Edge {  
    double weight; ...  
}
```

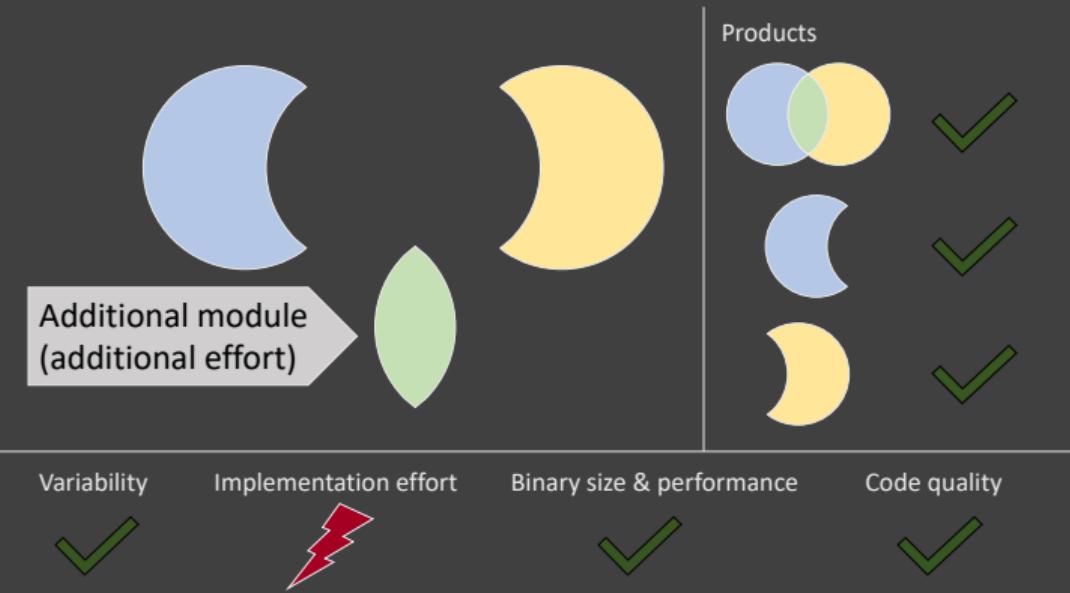
## Feature Module ShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        Edge e1, e2;  
        ...  
        #ifdef WEIGHTED  
            if (e1.weight > e2.weight) ...  
        #endif  
        ...  
    }  
}
```

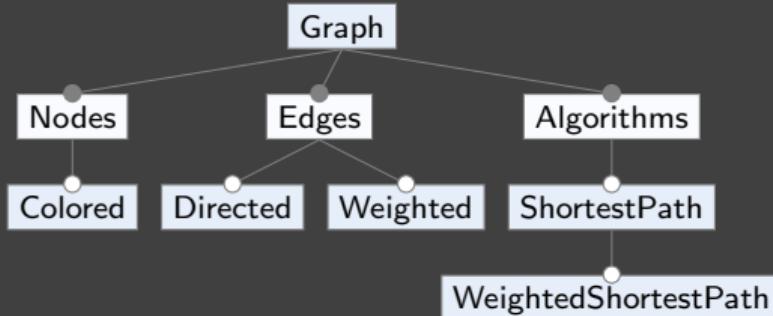
## Strategy S6: Derivative Modules

### Strategy S6

Create a dedicated module for code that coordinates features.



# Strategy S6: Derivative Modules



*WeightedShortestPath*  $\leftrightarrow$  *ShortestPath*  $\wedge$  *Weighted*

Derivative module is selected only if both interacting features are selected.

## Feature Module ShortestPath

```
refines class Graph {  
    List shortestPath(Node a, Node b){  
        Edge e1, e2;  
        ...  
        if (isLonger(e1,e2))  
            ...  
    }  
    boolean isLonger(Edge e1, Edge e2){  
        return false;  
    }  
}
```

## Feature Module WeightedShortestPath

```
refines class Graph {  
    boolean isLonger(Edge e1, Edge e2){  
        return e1.weight > e2.weight;  
    }  
}
```

# Overview and Discussion

Solution	Variability	Effort	Size & performance	Quality
S1: Adapt Feature Model	⚡	✓	✓	✓
S2: Orthogonal implementation	?	✓	?	✓
S3: Duplicate implementations	✓	⚡	✓	⚡
S4: Move source code	✓	✓	⚡	⚡
S5: Conditional compilation	✓	✓	✓	⚡⚡
S6: Derivative modules	✓	⚡	✓	✓

# How to Handle Feature Interactions? – Summary

## Lessons Learned

- Adaptation of feature model to avoid (undesired) feature interactions.
- Strategies to implement coordination code for known feature interactions.
- Discussion of the strengths and weaknesses of each of the strategies.

## Further Reading

- Kästner et al.: On the impact of the optional feature problem: Analysis and case studies. SPLC 2009.
- Apel et al. 2013, Chapter 9

## Practice

Looking back at our graph library and the feature interaction between *ShortestPath* and *Weighted*. Which strategy would you choose to handle this interaction? Why?

Can you think of other feature interactions for the graph library (you may also add additional features)? Again, how would you handle them?

# **9. Feature Interactions**

## **9a. What is a Feature Interaction?**

## **9b. How to Handle Feature Interactions?**

## **9c. How to Avoid Feature Interactions?**

The Choice of Features

Documentation of Interactions

Summary

FAQ

# The Choice of Features

John Ferguson Smart (2017)



[uci.edu]

John Carmack (born 1970)

"The important point is that the cost of adding a feature isn't just the time it takes to code it. The cost also includes the addition of an obstacle to future expansion. [...] The trick is to pick the features that don't fight each other."

# The Choice of Features



**Henry Ford, 1909**

"Any customer can have a car painted any color that he wants so long as it is black."

**Why only black?**

[Apel et al. 2013]

- black color dried faster
- faster production
- more products and cheaper production

# Documentation of Interactions – Incompatibilities of Lenovo Hardware

## Documentation of Remaining Interactions

- not all interactions can be prevented/fixed
- how to apply strategy S1 (see Part II) if there is no feature model?
- what to document?

## Lenovo's Option Compatibility Matrices

- 7 Excel files for current products  
+ archive for old products
- Excel file for computers contains 32 tables (series)
- table for ThinkPad X has 28 columns (models) and  
> 500 rows (accessories)
- 14k cells contain > 400 different footnotes
- a footnote explains one incompatibility

The screenshot shows a web browser window with the Lenovo logo at the top. Below it, there are navigation icons for shopping cart, user profile, and menu. A search bar is also present. The main content area features a heading "Knowledge Base & Guides" with a magnifying glass icon. Below this, a sub-section titled "Accessories and Options Compatibility Matrix (OCM)" is shown. A brief description follows: "The Lenovo OCM are Microsoft Excel files that have compatibility information for Lenovo accessories. To quickly share this page, use this url: [www.lenovo.com/accessoriesguide](http://www.lenovo.com/accessoriesguide)". Below this is a table titled "Description" with several rows of compatibility matrices for different product categories, each with a date. At the bottom, there is a section titled "Archive OCM (For older product)" with a single row in a table.

Description	Version
ThinkPad, ThinkCentre, ThinkStation, Ideapad and Ideacentre Option Compatibility Matrix	Aug 2021
Commercial Monitors - Option Compatibility Matrix	Aug 2021
Consumer Monitors - Option Compatibility Matrix	Aug 2021
Storage - Option Compatibility Matrix	November 2019
ThinkServer - Option Compatibility Matrix	October 2019
ThinkSystem and System X - Option Compatibility Matrix	October 2019
Network function support - Option Compatibility Matrix	Aug 2021

Description
Archive - ThinkPad, ThinkCentre, ThinkStation, Ideapad and Ideacentre Option Compatibility Matrix

# Documentation of Interactions – Incompatibilities of Lenovo Hardware

- columns contain notebooks
- rows contain accessories
- X indicates compatibility
- numbers indicate a known incompatibility

# Documentation of Interactions – Incompatibilities of Lenovo Hardware

A	B
236 313	Compatible with 45W or 65W slim-tip charged notebooks only Only can support touch screen system. 4X80N95874/GX80N07827.KR,Colombia; 4X80Q75521/GX80Q75525.Jordan,PH, Saudi Arabia, Singapore, Thailand 4X80Q75522/GX80Q75526:Argentina,Bolivia,Brazil,Colombia,Ecuador,Malaysia,Mexico,Paraguay,Peru,Uruguay,Venezuela; 4X80Q75523/GX80Q75527:Israel, Russia, UAE 4X80Q75524/GX80Q75528:Algeria, Azerbaijan, Bahrain, Belarus, Botswana, Chile, Cote d'ivoire, Egypt, India, Indonesia, KZ, Kenya, Kuwait, Lebanon, Moldova, Morocco, Nigeria, Oman, Pakistan, Qatar, SA, Sri Lanka, Tunisia, Uganda, Ukraine, UAE 4X80N95873/GX80N07825:Albania,Angola,AU,Austria,Bangladesh,Belgium, BA, Bulgaria, Canada, Croatia, Cyprus, CZ, Denmark, Estonia, Finland, France, Georgia, Germany, Greece, HK, Hungary, Iceland, Ireland, Italy, Latvia, Lithuania, LU, Macedonia, NL, New Zealand, Norway, Poland, Portugal, Romania, Serbia and Montenegro, Slovakia, Slovenia, Spain, Sweden, CH, Taiwan, Turkey, TM, UK, US, Uzbekistan, Vietnam
237 314	4X80N95873.CN,JP, GX80N07824.CN, GX80N20629.JP
238 315	If assemble two more 3.5" Hard Drive , the 3.5" HDD bracket kit 4XF0Q63396 is needed.
239 316	Please ensure to use v44 BIOS or above.
240 317	Please ensure to use v30 BIOS or above
241 318	It can only be used on 900W power supply model, but not on 690w power supply model.
242 319	The 2.5" bracket (4XF0G94539 or 4XF0P01009) may be needed when install a 2.5" HDD or SSD based on user's configuration
243 320	The 2.5" bracket (4XF0P01010) may be needed when install a 2.5" HDD or SSD based on user's configuration
244 321	Only compatible with those that bundled a 45W USB-C adapter
	With AC power adapter connected: PXE boot supported. Wake On LAN from Hibernation, or Power-Off mode. MAC address pass through
245 322	Without AC power adapter connected: PXE boot supported. MAC address pass through For the computer handle with a hole, install a locking clip. For the computer without a hole, install the two cage nuts on the mounting flange. Align the holes in the stopper with the

- 314: pen requires touch screen (cf. S1)
- 315/319/320: extra module needed (cf. S6)
- 316/317: fixed in newer BIOS versions
- 318/321: two modules with a different power supply (cf. S3)

# How to Avoid Feature Interactions? – Summary

## Lessons Learned

- reduction of variability
- which features are actually needed?
- documentation of interactions that cannot be avoided

## Further Reading

Apel et al. 2013

## Practice

Who checks the compatibility of Lenovo products?

# FAQ – 9. Feature Interactions

## Lecture 9a

- What are (inadvertent) feature interactions? Give examples!
- Are feature interactions limited to product lines?
- What is the feature-interaction problem? Why is it critical for product lines?
- What is the difference of static/dynamic, one-wise/pair-wise/higher-order interactions?
- What are typical patterns of interactions?
- How can features interact on control flow and data?

## Lecture 9b

- How to resolve feature interactions?
- What is the optional feature problem?
- What are typical goals when resolving feature interactions?
- Name/explain strategies to resolve feature interactions!
- What are (dis)advantages of those strategies?

## Lecture 9c

- How to cope with feature interactions?
- How to reduce variability?
- What are unused, unnecessary, and shopping-list-features?
- How to document feature interactions?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 10a. Analysis Strategies

- Recap: Quality Assurance
- Automated Analysis of Product Lines
- Product-Based Strategies
- Feature-Based Strategies
- Family-Based Strategies
- Classification of Strategies
- Summary

### 10b. Analyzing Feature Mappings

- Automated Analysis of Feature Mappings
- Presence Conditions
- Detecting Dead Code
- Detecting Superfluous Annotations
- Joining the Problem and Solution Space
- Analyzing Feature Modules
- Feature-Mapping Analyses in FeatureIDE
- Summary

### 10c. Analyzing Variable Code

- Automated Analysis of Variable Code
- Variability-Aware Type Checking
- Analyzing Feature Modules
- Analyzing Conditional Compilation
- Discussion
- Product-Line Analyses in the Wild
- Summary
- FAQ

# 10. Product-Line Analyses – Handout

Software Product Lines | Elias Kuiter, Thomas Thüm, Timo Kehrer | June 9, 2023

# 10. Product-Line Analyses

## 10a. Analysis Strategies

Recap: Quality Assurance

Automated Analysis of Product Lines

Product-Based Strategies

Feature-Based Strategies

Family-Based Strategies

Classification of Strategies

Summary

## 10b. Analyzing Feature Mappings

## 10c. Analyzing Variable Code

# Recap: Quality Assurance

[Ludewig and Licher 2013]

- last lecture:  
how to **avoid** variability bugs (esp. feature interactions)
- this + next lecture:  
how to **find** variability bugs



CAN YOU TAKE A  
LOOK AT THE BUG  
I JUST OPENED?

UH OH.



IS THIS A NORMAL BUG, OR  
ONE OF THOSE HORRIFYING  
ONES THAT PROVE YOUR  
WHOLE PROJECT IS BROKEN  
BEYOND REPAIR AND SHOULD  
BE BURNED TO THE GROUND?



IT'S A NORMAL  
ONE THIS TIME,  
I PROMISE.

OK, WHAT'S  
THE BUG?



THE SERVER CRASHES  
IF A USER'S PASSWORD  
IS A RESOLVABLE URL.

I'LL GET THE  
LIGHTER FLUID.



# Automated Analysis of Product Lines

## Typical Program Analyses

- code metrics
- type checking
- theorem proving
- data-flow analysis
- performance analysis
- ...



## What is a Program Analysis?

- analyzes properties of a **program** (e.g., correctness, performance, and safety)
- can be used to automatically find bugs, bottlenecks, and other **vulnerabilities**

## Asking Questions About Product Lines

- Which product has the most lines of code? [ref]
- Which products have type errors? [ref]
- Which products violate specifications? [ref]
- Which products have unsafe data flows? [ref]
- Which is the fastest product?  
Which product has the smallest binary? [ref]
- ...

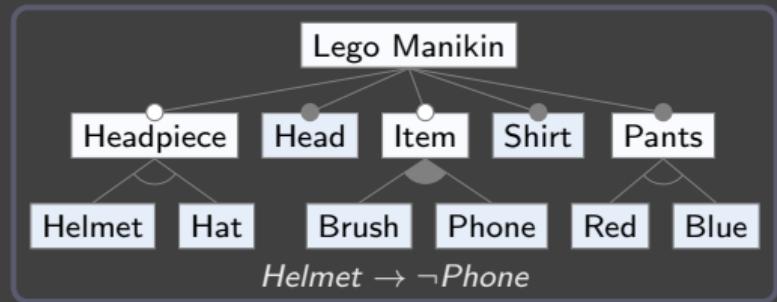
## What is a Product-Line Analysis?

- analyzes properties of an entire **product line**
- can be roughly classified by its **strategy**:
  - product-based
  - feature-based
  - family-based

# Product-Based Strategies

## Intuition

- to analyze the product line, just analyze **each product**
  - individually
  - in isolation
  - possibly in parallel
- e.g., compile and verify each product

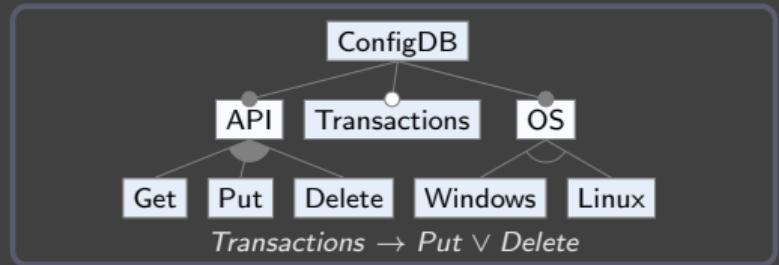


# Product-Based Strategies

## Algorithm

**Require:** a product line  $pl$ ; algorithms  $\gamma$ ,  $\alpha$ ,  $\sigma$   
 $C \leftarrow \text{AllSAT}(\phi(FM_{pl}))$   $\triangleright$  enumerate valid config's  
 $results \leftarrow []$   
**for all**  $S \in C$  **do**  $\triangleright$  for each valid config  
     $p \leftarrow \gamma(S)$   $\triangleright$  generate product  
     $results += \alpha(p)$   $\triangleright$  add analysis result  
**end for**  
**return**  $\sigma(results)$

- $\gamma$  **generates** (e.g., compiles) products (e.g., make, gradle, FeatureHouse, npm, ...)
- $\alpha$  **analyzes** the product (e.g., run verifier)
- $\sigma$  **summarizes** the results (e.g., each individual call to  $\alpha$  must succeed)



$\sigma([\alpha(\gamma(\{C, G, W\}))])$	$\alpha(\gamma(\{C, G, L\}))$
$\alpha(\gamma(\{C, P, W\}))$	$\alpha(\gamma(\{C, P, L\}))$
$\alpha(\gamma(\{C, G, P, W\}))$	$\alpha(\gamma(\{C, G, P, L\}))$
$\alpha(\gamma(\{C, D, W\}))$	$\alpha(\gamma(\{C, D, L\}))$
$\alpha(\gamma(\{C, G, D, W\}))$	$\alpha(\gamma(\{C, G, D, L\}))$
$\alpha(\gamma(\{C, P, D, W\}))$	$\alpha(\gamma(\{C, P, D, L\}))$
$\alpha(\gamma(\{C, G, P, D, W\}))$	$\alpha(\gamma(\{C, G, P, D, L\}))$
$\alpha(\gamma(\{C, P, T, W\}))$	$\alpha(\gamma(\{C, P, T, L\}))$
$\alpha(\gamma(\{C, G, P, T, W\}))$	$\alpha(\gamma(\{C, G, P, T, L\}))$
$\alpha(\gamma(\{C, D, T, W\}))$	$\alpha(\gamma(\{C, D, T, L\}))$
$\alpha(\gamma(\{C, G, D, T, W\}))$	$\alpha(\gamma(\{C, G, D, T, L\}))$
$\alpha(\gamma(\{C, P, D, T, W\}))$	$\alpha(\gamma(\{C, P, D, T, L\}))$
$\alpha(\gamma(\{C, G, P, D, T, W\}))$	$\alpha(\gamma(\{C, G, P, D, T, L\}))$

# Classification of Strategies



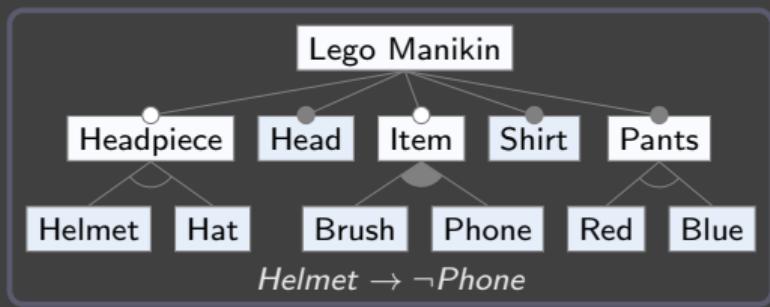
## Product-Based Strategy

- analyze individual **products**
- + sound, complete
- + uses off-the-shelf generator  $\gamma$  and analysis  $\alpha$
- redundant effort
- does not scale well

# Feature-Based Strategies

## Intuition

- to analyze the product line, just analyze **each feature** individually
- ignore all relations to other features
- e.g., compile and verify each component  
⇒ requires **interfaces between features**  
(components, services, plug-ins)

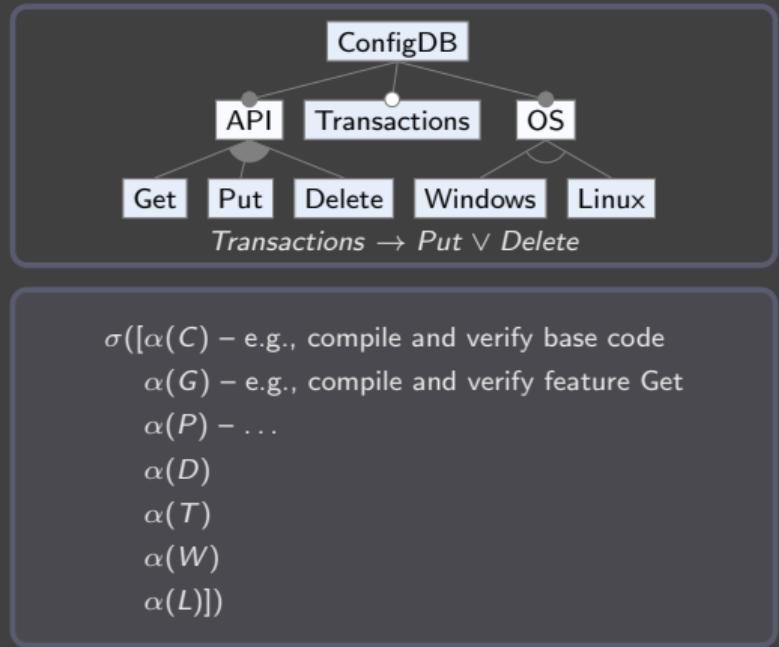


# Feature-Based Strategies

## Algorithm

```
Require: a product line  $pl$ ; algorithms  $\alpha, \sigma$ 
results  $\leftarrow []$ 
for all  $f \in F_{pl}$  do            $\triangleright$  for each feature
    results  $+ = \alpha(f)$        $\triangleright$  add analysis result
end for
return  $\sigma(results)$ 
```

- $\alpha$  **analyzes** the feature (e.g., compiles and verifies the component)
- $\sigma$  **summarizes** the results (see product-based)



# Classification of Strategies



## Product-Based Strategy

- analyze individual **products**
- + sound, complete
- + uses off-the-shelf generator  $\gamma$  and analysis  $\alpha$
- redundant effort
- does not scale well

## Feature-Based Strategy

- analyze individual **features**
- + sound, efficient
- analysis  $\alpha$  requires features with interfaces
- incomplete: misses all feature interactions

# Family-Based Strategies

## Intuition

- analyze the product line (or **family**) as a whole
- requirement: the analysis should give the same result as a product-based analysis
- makes use of the feature model and artifacts
- analysis is **hand-crafted**, no generic algorithm  
⇒ typically: reduction to SAT problems

## Today's Examples

- analyzing **feature mappings**
  - analyzing **variable code**
- ⇒ here: for **conditional compilation** and **feature-oriented programming**



# Classification of Strategies



## Product-Based Strategy

- analyze individual **products**
- + sound, complete
- + uses off-the-shelf generator  $\gamma$  and analysis  $\alpha$
- redundant effort
- does not scale well

## Feature-Based Strategy

- analyze individual **features**
- + sound, efficient
- analysis  $\alpha$  requires features with interfaces
- incomplete: misses all feature interactions

## Family-Based Strategy

- analyze the **product line**
- + sound, complete, efficient
- requires careful, hand-crafted analysis  $\alpha$

# Analysis Strategies – Summary

## Lessons Learned

- product-line analyses are needed for quality assurance
- **product-based**: simple, but does not scale
- **feature-based**: fairly simple, but misses interactions
- **family-based**: efficient, but most complex

## Further Reading

- Apel et al. 2013, Chapter 10
- Thüm et al. 2014

## Practice

Can you imagine other analysis strategies than product-based, feature-based, and family-based? How could such strategies look like?

# 10. Product-Line Analyses

## 10a. Analysis Strategies

## 10b. Analyzing Feature Mappings

Automated Analysis of Feature Mappings

Presence Conditions

Detecting Dead Code

Detecting Superfluous Annotations

Joining the Problem and Solution Space

Analyzing Feature Modules

Feature-Mapping Analyses in FeatureIDE

Summary

## 10c. Analyzing Variable Code

# Automated Analysis of Feature Mappings

## Recap: A Typical Product Line

- embedded or systems programming (e.g., Linux)
- implemented with conditional compilation
  - build systems (e.g., KBuild)
  - preprocessors (e.g., CPP)
- feature traceability only implicit  
⇒ there is code scattering and tangling

## Recap: Feature Mapping

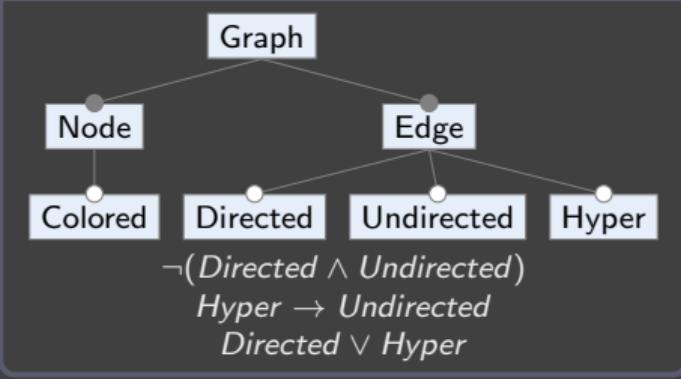
- specifies which features correspond to which artifacts (individual files/lines, components/feature modules/aspects)
- connects the problem space to the solution space

## Asking Questions About the Feature Mapping

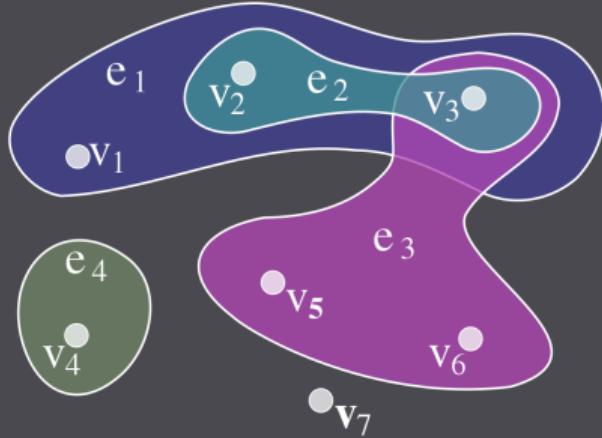
- Is the code even included in any product?
- Are there contradictory or unnecessary preprocessor annotations in the code?
- How scattered and tangled is the code?
- ...

# Automated Analysis of Feature Mappings

## Running Example: Graph Product Line



## An Undirected Hypergraph



# Presence Conditions

## Presence Condition

A **presence condition (PC)** for a code location (i.e., a line or file) is a formula that describes the circumstances under which the code location is included in a product.

- useful for implementation techniques with code scattering and tangling
- e.g., build systems (file PCs) or preprocessors (line PCs)
- here: line PCs for the C preprocessor

## Presence Conditions

T  
T  
Colored  
Colored  
Colored  
T  
T  
T  
*Directed*  
*Directed*  
 $\neg Dir \wedge Hyper$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\quad \quad \quad \neg Dir \wedge \neg Hy$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\quad \quad \quad \neg Dir \wedge \neg Hy$   
T

## graph.cpp

```
class Node {  
    string label;  
#ifdef COLORED  
    string color;  
#endif  
};  
  
class Edge {  
#ifdef DIRECTED  
    Node fromNode, toNode;  
#elifdef HYPER  
#ifdef UNDIRECTED  
    set<Node> nodeSet;  
#elifdef DIRECTED  
    map<Node, set<Node>> nodeMap;  
#endif  
#else  
#ifndef DIRECTED  
    pair<Node, Node> nodePair;  
#endif  
#endif  
};
```

# Detecting Dead Code

## Dead Code

A line or file of code is **dead** when

- no product includes it.
- or, equivalently:  
its presence condition  $PC$  is  
contradictory (i.e.,  $PC \Rightarrow \perp$ ).

calculated by querying a **satisfiability solver** whether  $PC$  is not satisfiable  
(i.e.,  $\neg SAT(PC)$ )

## What causes dead code?

- confusion due to nested `#ifdef`
- domain modeling mistakes
- can be intended!

[Hentze et al. 2021]

## Presence Conditions

$\top$   
 $\top$   
 $Colored$   
 $Colored$   
 $Colored$   
 $\top$   
 $\top$   
 $\top$   
 $Directed$   
 $Directed$   
 $\neg Dir \wedge Hyper$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge \neg Hy$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy$   
 $\top$

## graph.cpp

```
class Node {  
    string label;  
#ifdef COLORED  
    string color;  
#endif  
};  
  
class Edge {  
#ifdef DIRECTED  
    Node fromNode, toNode;  
#elifdef HYPER  
#ifdef UNDIRECTED  
    set<Node> nodeSet;  
#elifdef DIRECTED  
    map<Node, set<Node>> nodeMap;  
#endif  
#else  
#ifndef DIRECTED  
    pair<Node, Node> nodePair;  
#endif  
#endif  
};
```

# Detecting Superfluous Annotations

## Superfluous Annotation

- An annotation is **superfluous**
- when it can be omitted without consequences.
  - or, equivalently:  
its presence condition  $PC$  is implied by the enclosing presence condition  $PC'$  (i.e.,  $PC' \Rightarrow PC$ ).

calculated by querying a **satisfiability solver** whether  $PC' \wedge \neg PC$  is not satisfiable (i.e.,  $\neg SAT(PC' \wedge \neg PC)$ )

- $PC' = \neg Dir \wedge \neg Hy$
- $PC = \neg Dir \wedge \neg Hy \wedge \neg Dir$

## Presence Conditions

- $\top$   
 $\top$   
 $Colored$   
 $Colored$   
 $Colored$   
 $\top$   
 $\top$   
 $\top$   
 $Directed$   
 $Directed$   
 $\neg Dir \wedge Hyper$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\quad \neg Dir \wedge \neg Hy$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\quad \neg Dir \wedge \neg Hy$   
 $\top$

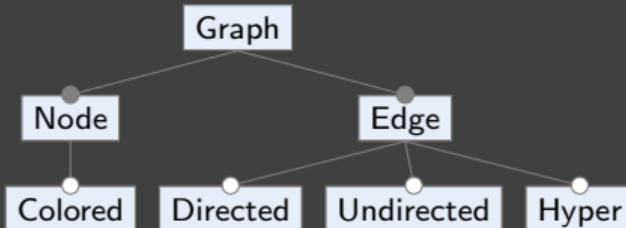
## graph.cpp

```
class Node {  
    string label;  
#ifdef COLORED  
    string color;  
#endif  
};  
  
class Edge {  
#ifdef DIRECTED  
    Node fromNode, toNode;  
#elifdef HYPER  
#ifdef UNDIRECTED  
    set<Node> nodeSet;  
#elifdef DIRECTED  
    map<Node, set<Node>> nodeMap;  
#endif  
#else  
#ifndef DIRECTED  
    pair<Node, Node> nodePair;  
#endif  
#endif  
};
```

# Joining the Problem and Solution Space

- right now, we only consider **line PCs** (from the preprocessor)
- but: a line is only included if its file is included, too  
⇒ we also have to consider **file PCs** (from the build system)
- also: we want to ignore invalid configurations  
⇒ we also have to consider the **feature model FM**
- idea: **join** feature model, file, and line presence condition:  
 $PC_{location} := \Phi(FM) \wedge PC_{file} \wedge PC_{line}$

Suppose we have the feature model ...



... and two files: node.cpp ...

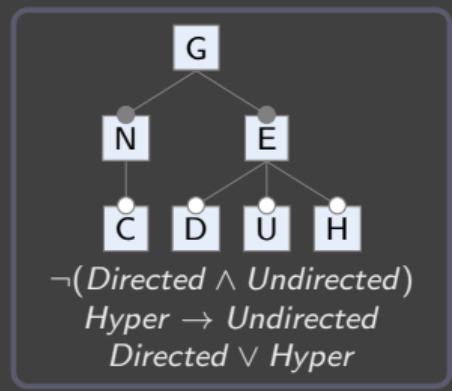
```
class Node {  
    string label;  
#ifdef COLORED  
    string color;  
#endif  
};
```

... and edge.cpp

```
class Edge {  
#ifdef DIRECTED  
    Node fromNode, toNode;  
#elifdef HYPER  
#ifndef UNDIRECTED  
    set<Node> nodeSet;  
#endif  
#else  
#ifndef DIRECTED  
    pair<Node, Node> nodePair;  
#endif  
#endif  
};
```

# Joining the Problem and Solution Space

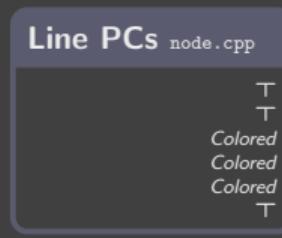
## Problem Space



$\downarrow \Phi$

$\text{Graph} \wedge \text{Node} \wedge \text{Edge}$   
 $\wedge \neg(\text{Directed} \wedge \text{Undirected})$   
 $\wedge (\text{Hyper} \rightarrow \text{Undirected})$   
 $\wedge (\text{Directed} \vee \text{Hyper})$

## Solution Space →



$\text{Line PCs edge.cpp}$

T  
Directed  
Directed  
 $\neg\text{Dir} \wedge \text{Hyper}$   
 $\neg\text{Dir} \wedge \text{Hy} \wedge \text{Un}$   
 $\neg\text{Dir} \wedge \text{Hy} \wedge \text{Un}$   
 $\neg\text{Dir} \wedge \text{Hy} \wedge \neg\text{Un} \wedge \text{Dir}$   
 $\neg\text{Dir} \wedge \text{Hy} \wedge \neg\text{Un} \wedge \text{Dir}$   
 $\neg\text{Dir} \wedge \text{Hy} \wedge \neg\text{Un} \wedge \text{Dir}$   
 $\neg\text{Dir} \wedge \neg\text{Hy}$   
 $\neg\text{Dir} \wedge \neg\text{Hy} \wedge \neg\text{Dir}$   
 $\neg\text{Dir} \wedge \neg\text{Hy} \wedge \neg\text{Dir}$   
 $\neg\text{Dir} \wedge \neg\text{Hy} \wedge \neg\text{Dir}$   
T

**node.cpp**

```
class Node {  
    string label;  
#ifdef COLORED  
    string color;  
#endif  
};
```

**edge.cpp**

```
class Edge {  
#ifdef DIRECTED  
    Node fromNode, toNode;  
#elifdef HYPER  
#ifdef UNDIRECTED  
    set<Node> nodeSet;  
#elifdef DIRECTED  
    map<Node, set<Node>> nodeMap;  
#endif  
#else  
#ifndef DIRECTED  
    pair<Node, Node> nodePair;  
#endif  
#endif  
};
```

# Joining the Problem and Solution Space

## Feature-Model Formula

*Graph*  $\wedge$  *Node*  $\wedge$  *Edge*  
 $\wedge \neg(Directed \wedge Undirected)$   
 $\wedge(Hyper \rightarrow Undirected)$   
 $\wedge(Directed \vee Hyper)$

## File PC *edge.cpp*

Edge

## Line PCs *edge.cpp*

$\top$   
*Directed*  
*Directed*  
 $\neg Dir \wedge Hyper$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge Un$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge Hy \wedge \neg Un \wedge Dir$   
 $\neg Dir \wedge \neg Hy$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy \wedge \neg Dir$   
 $\neg Dir \wedge \neg Hy$   
 $\top$

## *edge.cpp*

```
class Edge {
    #ifdef DIRECTED
        Node fromNode, toNode;
    #elifdef HYPER
    #ifdef UNDIRECTED
        set<Node> nodeSet;
    #elifdef DIRECTED
        map<Node, set<Node>> nodeMap;
    #endif
    #else
    #ifndef DIRECTED
        pair<Node, Node> nodePair;
    #endif
    #endif
};
```

$$PC_{location} := \Phi(FM)$$

$$= G \wedge N \wedge E \wedge \neg(D \wedge U) \wedge (H \rightarrow U) \wedge (D \vee H) \wedge E$$

$$\Leftrightarrow G \wedge N \wedge E \wedge \neg(D \wedge U) \wedge (H \rightarrow U) \wedge (D \vee H) \wedge E$$

$$\Rightarrow (D \vee H) \wedge \neg D \wedge \neg H$$

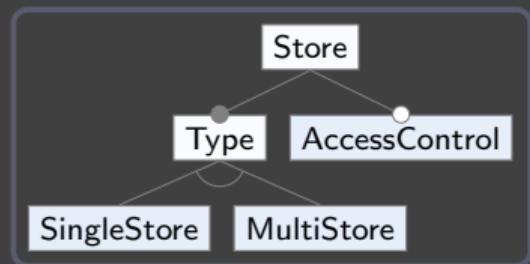
$$\Rightarrow \perp - \text{so this code is dead after all!}$$

$$\wedge PC_{edge.cpp} \wedge PC_{pair<Node, Node> nodePair};$$

$$\wedge \neg D \wedge \neg H \wedge \neg D$$

$$\wedge \neg D \wedge \neg H \wedge \neg D$$

# Analyzing Feature Modules



## Feature-Model Formula

$$\Phi(FM) = \text{Store} \wedge \text{Type} \wedge (\text{SS} \vee \text{MS}) \wedge (\neg \text{SS} \vee \neg \text{MS})$$

## Valid Configurations

{SS}  
{SS, AC}

{MS}  
{MS, AC}

Feature module *SingleStore*

```
class Store {  
    private Object value;  
    Object read() { return value; }  
    void set(Object nvalue) { value = nvalue; }  
}
```

Feature module *MultiStore*

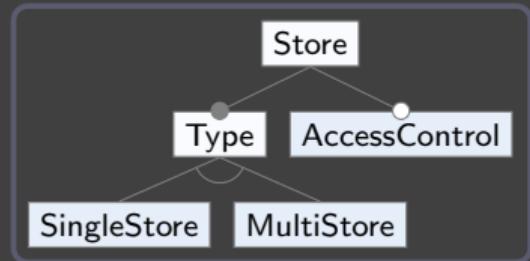
```
class Store {  
    private LinkedList values = new LinkedList();  
    Object read() { return values.getFirst(); }  
    Object[] readAll() { return values.toArray(); }  
    void set(Object nvalue) { values.addFirst(nvalue); }  
}
```

Feature module *AccessControl*

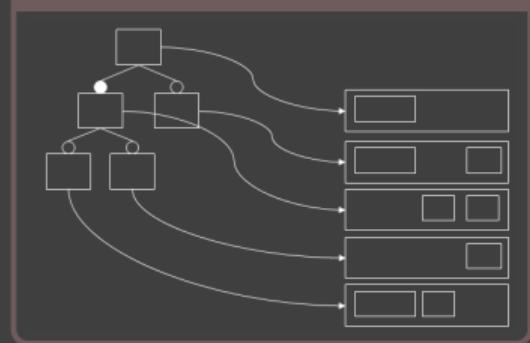
```
refines class Store {  
    private boolean sealed = false;  
    Object read() {  
        if (!sealed) { return Super.read(); }  
        else { throw new RuntimeException("Access.denied!"); }  
    }  
    void set(Object nvalue) {  
        if (!sealed) { Super.set(nvalue); }  
        else { throw new RuntimeException("Access.denied!"); }  
    }  
}
```

Is there dead code? Are there superfluous annotations?

# Analyzing Feature Modules



Recap: 1:1 Feature Mapping!



```
class Store {
    private Object value;
    Object read() { return value; }
    void set(Object nvalue) { value = nvalue; }
}
```

Feature module *SingleStore*

```
class Store {
    private LinkedList values = new LinkedList();
    Object read() { return values.getFirst(); }
    Object[] readAll() { return values.toArray(); }
    void set(Object nvalue) { values.addFirst(nvalue); }
}
```

Feature module *MultiStore*

```
refines class Store {
    private boolean sealed = false;
    Object read() {
        if (!sealed) { return Super.read(); }
        else { throw new RuntimeException("Access.denied!"); }
    }
    void set(Object nvalue) {
        if (!sealed) { Super.set(nvalue); }
        else { throw new RuntimeException("Access.denied!"); }
    }
}
```

Feature module *AccessControl*

Is Are there dead code dead features? Are there superfluous annotations redundant constraints?

# Feature-Mapping Analyses in FeatureIDE

The screenshot shows the FeatureIDE interface. On the left is a code editor for 'Main.java' containing Java code with annotations. An annotation at line 13 is highlighted with a yellow background and a tooltip: 'Antenna: This expression is a contradiction and causes a dead code block.' On the right is a 'HelloWorld Model' window displaying a feature diagram. The diagram shows a root node 'HelloWorld' branching into 'Hello', 'Open', 'Closed', and 'World'. A legend indicates: Mandatory (solid circle), Optional (hollow circle), Abstract (dashed circle), and Concrete (green box). Below the diagram is a 'Feature Diagram' tab, and further down is a configuration table for 'HelloWorld.config' showing two valid configurations: 'HelloWorld' (checkbox checked) and 'Hello' (checkbox checked).

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println(helloWorld());  
4     }  
5     static String helloWorld() {  
6         String message = "";  
7         //#if Hello  
8         message += "Hello";  
9         //#endif  
10        //#if World  
11        //#if Open  
12        message += " open"  
13        //#endif  
14        //#if Closed  
15        message += " closed";  
16        //#endif  
17        message += " world!";  
18        //#endif  
19        return message;  
20    }  
21 }  
22 }
```

demo video available (minute 3 and 4): dead code block, superfluous annotations, generation of all products, error propagation, unit testing

## Discussion

- we can now **identify anomalies**:
  - dead (unused) code
  - mistakes in preprocessor annotations
  - disagreements between problem and solution space
- but: we only analyze the feature mapping and **ignore the actual code**
  - pro: simple, language-independent
  - con: can only find simple anomalies
- difficulty depends on the feature traceability (harder for conditional compilation than for FOP)

# Analyzing Feature Mappings – Summary

## Lessons Learned

- feature-mapping analyses alleviate the impact of code scattering and tangling
- they are usually not necessary when there is good feature traceability
- they cannot detect bugs in the actual code

## Further Reading

- Apel et al. 2013, Chapter 10

## Practice

Above, we assumed that we know all presence conditions already. How can we automatically extract presence conditions from code that uses the C pre-processor? What problems might occur?

# 10. Product-Line Analyses

## 10a. Analysis Strategies

## 10b. Analyzing Feature Mappings

## 10c. Analyzing Variable Code

Automated Analysis of Variable Code

Variability-Aware Type Checking

Analyzing Feature Modules

Analyzing Conditional Compilation

Discussion

Product-Line Analyses in the Wild

Summary

FAQ

# Automated Analysis of Variable Code

## Asking Questions About the Feature Mapping ...

- Are there contradictory or unnecessary preprocessor annotations in the code?
- Is the code even included in any product?
- If so, in how many products is the code included?
- ...

only finds code-agnostic anomalies

## ... and the Variable Code

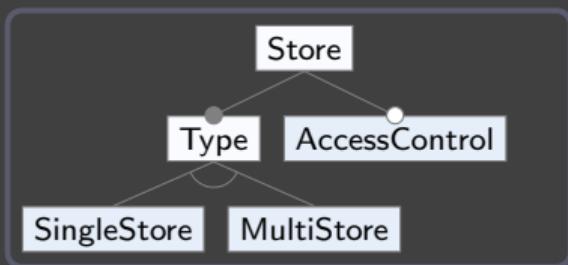
- Can every product be generated (e.g., compiled)?  
⇒ to find all **syntax and type errors**
- Do all tests succeed for every product?  
⇒ to find some **runtime and logic errors**
- Does every product adhere to its specification?  
⇒ to **rule out** runtime and logic errors
- ...

now: analyze (non-)functional properties of all products

## Today's Example

type checking for FOP and conditional compilation

# Variability-Aware Type Checking – Analyzing Feature Modules



## Feature-Model Formula

$$\Phi(FM) = \text{Store} \wedge \text{Type} \wedge (\text{SS} \vee \text{MS}) \wedge (\neg \text{SS} \vee \neg \text{MS})$$

## Valid Configurations

{SS}  
{SS, AC}

{MS}  
{MS, AC}

Is there a type error in **any** product?  
What about {SS, AC}?

```
class Store {  
    private Object value;  
    Object read() { return value; }  
    void set(Object nvalue) { value = nvalue; }  
}
```

Feature module *SingleStore*

```
class Store {  
    private LinkedList values = new LinkedList();  
    Object read() { return values.getFirst(); }  
    Object[] readAll() { return values.toArray(); }  
    void set(Object nvalue) { values.addFirst(nvalue); }  
}
```

Feature module *MultiStore*

```
refines class Store {  
    private boolean sealed = false;  
    Object read() {  
        if (!sealed) { return Super.read(); }  
        else { throw new RuntimeException("Access..denied!"); }  
    }  
    Object[] readAll() {  
        if (!sealed) { return Super.readAll(); }  
        else { throw new RuntimeException("Access..denied!"); }  
    }  
    void set(Object nvalue) {  
        if (!sealed) { Super.set(nvalue); }  
        else { throw new RuntimeException("Access..denied!"); }  
    }  
}
```

Feature module *AccessControl*

# Variability-Aware Type Checking – Analyzing Feature Modules

## Reachability Condition of *id*

guarantees that a given **reference** to *id* is also **defined** somewhere:

$$\Phi(FM) \Rightarrow (PC_{ref}^{id} \rightarrow \bigvee_{def} PC_{def}^{id})$$

or, with a SAT solver:

$$\neg SAT(\Phi(FM) \wedge PC_{ref} \wedge \bigwedge_{def} \neg PC_{def})$$

$$\Phi(FM) \Rightarrow (AC \rightarrow SS \vee MS) \text{ holds,}$$

$$\Phi(FM) \Rightarrow (AC \rightarrow MS) \text{ does not}$$

$\Rightarrow \{SS, AC\}$  has no `readAll!`!

## Type-Safe Product-Line

in a **type-safe** SPL, all references must always be defined (i.e., **all** reachability conditions must hold)  
... and many more conditions ...

```
class Store {  
    private Object value;  
    Object read() { return value; }  
    void set(Object nvalue) { value = nvalue; }  
}
```

Feature module *SingleStore*

*VM*  $\Rightarrow (AccessControl \Rightarrow SingleStore \vee MultiStore)$

```
class Store {  
    private LinkedList values = new LinkedList();  
    Object read() { return values.getFirst(); }  
    Object[] readAll() { return values.toArray(); }  
    void set(Object nvalue) { values.addFirst(nvalue); }  
}
```

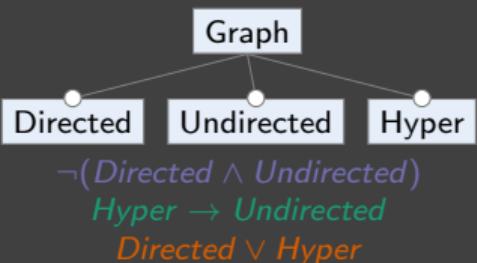
Feature module *MultiStore*

```
refines class Store {  
    private boolean sealed = false;  
    Object read() {  
        if (!sealed) { return Super.read(); }  
        else { throw new RuntimeException("Access..denied!"); }  
    }  
    Object[] readAll() {  
        if (!sealed) { return Super.readAll(); }  
        else { throw new RuntimeException("Access..denied!"); }  
    }  
    void set(Object nvalue) {  
        if (!sealed) { Super.set(nvalue); }  
        else { throw new RuntimeException("Access..denied!"); }  
    }  
}
```

Feature module *AccessControl*

*VM*  $\Rightarrow (AccessControl \Rightarrow MultiStore)$

# Variability-Aware Type Checking – Analyzing Conditional Compilation



Reachability Condition of  $id$

$$\Phi(FM) \Rightarrow (PC_{ref}^{id} \rightarrow \bigvee_{def} PC_{def}^{id})$$

Conflict Condition of  $id$ , def's  $d_i$

guarantees that no definition of  $id$  **conflicts** with another:

$$\Phi(FM) \Rightarrow \bigwedge_{d_1 \neq d_2} \neg(PC_{d_1}^{id} \wedge PC_{d_2}^{id})$$

Is  $e.nodes$  reachable?

$\Phi(FM) \Rightarrow (\top \rightarrow Dir \vee (Hy \wedge Un) \vee (Hy \wedge Dir))$   
holds, because each graph is **directed** or an (**undirected**) **hypergraph**

Does  $e.nodes$  conflict?

$\Phi(FM) \Rightarrow (\neg(Dir \wedge (Hy \wedge Un)) \wedge \neg(Dir \wedge (Hy \wedge Dir)) \wedge \neg((Hy \wedge Un) \wedge (Hy \wedge Dir)))$   
holds, because a graph is **never directed** and an (**undirected**) **hypergraph** at the same time

all reachable, no conflicts

graph.cpp

```

class Node { ... };

class Edge {
#ifndef DIRECTED
    pair<Node, Node> nodes;
#endif
#ifndef HYPER
    set<Node> nodes;
#endif
#ifndef UNDIRECTED
    map<Node, set<Node>> nodes;
#endif
};

std::ostream& operator<<( std::ostream &s, const Edge &e) {
    return s << e.nodes;
}
  
```

# Variability-Aware Type Checking – Discussion

## Just the Tip of the Iceberg

- here, we only discussed reachability and conflict conditions
- but: actual type checking requires a table of all identifiers, their types, and their PCs (and a lot more SAT queries)
- the practical difficulty depends:
  - FOP (due to superimposition)  
⇒ no conflict conditions required
  - good feature traceability (e.g., FOP)  
⇒ trivial PCs, simpler implementation
  - ignoring the feature model  
⇒ better performance (false positives!)

## The TypeChef Project

[Kästner et al. 2011]

- a variability-aware lexer, parser framework, and type system for C code with `#ifdef`'s
- skips preprocessing, instead builds an abstract syntax tree (AST) annotated with presence conditions
- poster with examples
- does it scale?

Busybox (811 features): "We need 57 minutes to type check all modules." [\[ref\]](#)

Linux (6065 features): "We successfully parsed [it in] roughly 85 hours on a single machine." [\[ref\]](#)

# Product-Line Analyses in the Wild – Product-Line Complexity

## Six Classes of Product-Line Complexity [Thüm 2021]

In a timeframe of 24h ...

- NC** Products cannot be generated automatically
- C1** All products can be generated and **tested**
- C2** Not C1, but all **products** can be **generated**
- C3** Not C2, but all **configurations** can be **generated** (AllSAT)
- C4** Not C3, but the **number of valid configurations** can be computed (#SAT)
- C5** Not C4, but **whether there is a valid configuration** can be computed (SAT)
- C6** It cannot be computed whether there is a valid configuration

## Examples

- NC** all product lines with mandatory custom development in application engineering (e.g., components and services with glue code, white-box frameworks)
- C1** < 2000 products for 1 min per product
- C2** < 90000 products for 1 s per product
- C3** <  $10^{13}$  configurations for 1 ns per configuration
- C4** older versions of Linux/Automotive05
- C5** newer versions of Linux/Automotive05 (see Sundermann et al. 2020)
- C6** No example known

# Product-Line Analyses in the Wild – Automated Analysis . . .

## Lecture 4c

### ... of Feature Models

analyze only the feature model

- void, core/dead features
- decision propagation
- atomic sets, redundant constraints
- ...

## Lecture 10b

### ... of Feature Mappings

analyze the feature mapping  
(considering the feature model)

- dead code
- superfluous annotations
- degree of code scattering and tangling
- ...

## Lecture 10c

### ... of Variable Code

analyze the variable code (considering the feature model and feature mapping)

- parsing, type checking
- static analysis
- model checking, theorem proving
- ...

here: **family-based** analysis strategies for **conditional compilation** and **feature-oriented programming**

# Analyzing Variable Code – Summary

## Lessons Learned

- with family-based analyses of variable code, we can analyze (non-)functional properties of all products at once
- type checking all products at once is possible for product lines up to medium size
- for huge product lines (e.g., Linux), it is infeasible

## Further Reading

- Apel et al. 2013, Chapter 10
- Kästner et al. 2011

## Practice

Suppose you have a preprocessor-based product line (with `#ifdef`'s). If you could turn it into a single, large runtime-variable product (with `if`'s), you could use an off-the-shelf compiler to find any type error in any product.

Is this possible? What problems might occur?

[Patterson et al. 2022]

# FAQ – 10. Product-Line Analyses

## Lecture 10a

- How to find variability bugs?
- What is a program analysis? What are examples?
- What is a product-line analysis?
- What are principal strategies to analyze product lines? What are (dis-)advantages?
- Given a specific algorithm, classify its analysis strategy!

## Lecture 10b

- How to analyze feature mappings?
- What are potential problems in feature mappings?
- What are presence conditions, dead code, superfluous annotations?
- Shall we incorporate the feature model when analyzing feature mappings?
- Shall product-line analyses analyze problem and solution space separately?
- What is special when analyzing the feature mapping of feature modules?
- What are limitations of analyzing feature mappings?
- Given CPP source code, determine its presence conditions, dead code, and superfluous annotations!

## Lecture 10c

- What are (examples of) type errors?
- Why are type errors challenging to detect in product lines?
- What is a type-safe product line, reachability condition, conflict condition?
- How does the analysis complexity differ for real-world product lines?
- What are analyses for problem and solution space?
- Give examples for easy and difficult product lines in terms of analysis effort!

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. **Product-Line Testing**
12. Evolution and Maintenance

### 11a. Challenges of Product-Line Testing

- Recap: Software Testing
- Test-Case Design in Single-System Engineering
- Testing All Configurations
- Testing One Configuration
- Sample-Based Testing
- Summary

### 11b. Combinatorial Interaction Testing

- Pairwise Interaction Testing
- T-Wise Interaction Testing
- Algorithms for Combinatorial Interaction Testing
- Efficiency of Combinatorial Interaction Testing
- Effectiveness of Combinatorial Interaction Testing
- Summary

### 11c. Solution-Space Sampling

- Coverage in Single-System Engineering
- Coverage of Ifdef Blocks
- Presence-Condition Coverage
- Overview on Coverage Criteria
- Input for Sampling Algorithms
- Summary
- FAQ

# 11. Product-Line Testing – Handout

Software Product Lines | Thomas Thüm, Sebastian Krieter, Timo Kehrer, Elias Kuiter | June 21, 2023



# Recap: Quality Assurance

[Ludewig and Licher 2013]



## Lectures on Quality Assurance

how to **avoid** variability bugs  
(esp. feature interactions) ...

- with processes  
(e.g., domain scoping)

[Lecture 8]

- with guidelines

[Lecture 9]

how to **find** variability bugs ...

- statically

[Lecture 10]

- dynamically

[Lecture 11]

- challenges of product-line testing in Part a
- black-box testing in Part b
- white-box testing in Part c

# 11. Product-Line Testing

## 11a. Challenges of Product-Line Testing

Recap: Software Testing

Test-Case Design in Single-System Engineering

Testing All Configurations

Testing One Configuration

Sample-Based Testing

Summary

## 11b. Combinatorial Interaction Testing

## 11c. Solution-Space Sampling

# Recap: Software Testing

## Software Testing

[Sommerville]

“Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.”

## Validation Testing

[Sommerville]

“Demonstrate to the developer and the customer that the software meets its requirements.”

## Defect Testing

[Sommerville]

“Find inputs or input sequences where the behavior of the software is incorrect, undesirable, or does not conform to its specification.”

## Stages of Testing

[Sommerville]

1. “**Development testing**, where the system is tested during development to discover bugs and defects”
2. “**Release testing**, where a separate testing team tests a complete version of the system before it is released to users”
3. “**User testing**, where users or potential users of a system test the system in their own environment”

## Manual vs Automated Testing

[Sommerville]

“In **manual testing**, a tester runs the program with some test data and compares the results to their expectations. [...] In **automated testing**, the tests are encoded in a program that is run each time the system under development is to be tested.”

# Recap: Test-Case Design

## Test Case

[Ludewig and Licher 2013]

In a test, a number of test cases are executed, whereas each test case consists **input values** for a single execution and **expected outputs**. An **exhaustive test** refers a test in which the test cases exercise all the possible inputs.

## Systematic Test

[Ludewig and Licher 2013]

A systematic test is a test, in which

1. the **setup** is defined,
2. the **inputs** are chosen systematically,
3. the **results** are documented and evaluated by criteria being defined prior to the test.

## Goal

[Ludewig and Licher 2013]

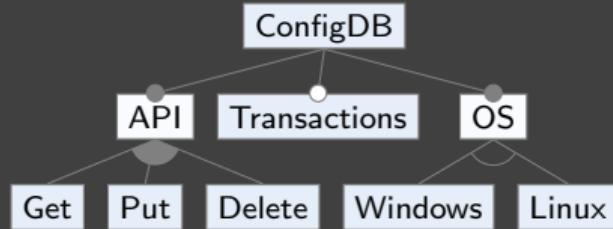
Detect a large number of failures with a low number of test cases. A test case (execution) is **positive**, if it detects a failure, and **successful** if it detects an unknown failure.

## An ideal test case is ...

[Ludewig and Licher 2013]

- representative: represents a large number of feasible test cases
- failure sensitive: has a high probability to detect a failure
- non-redundant: does not check what other test cases already check

# Testing All Configurations



*Transactions → Put ∨ Delete*

## Recap: 26 Valid Configurations

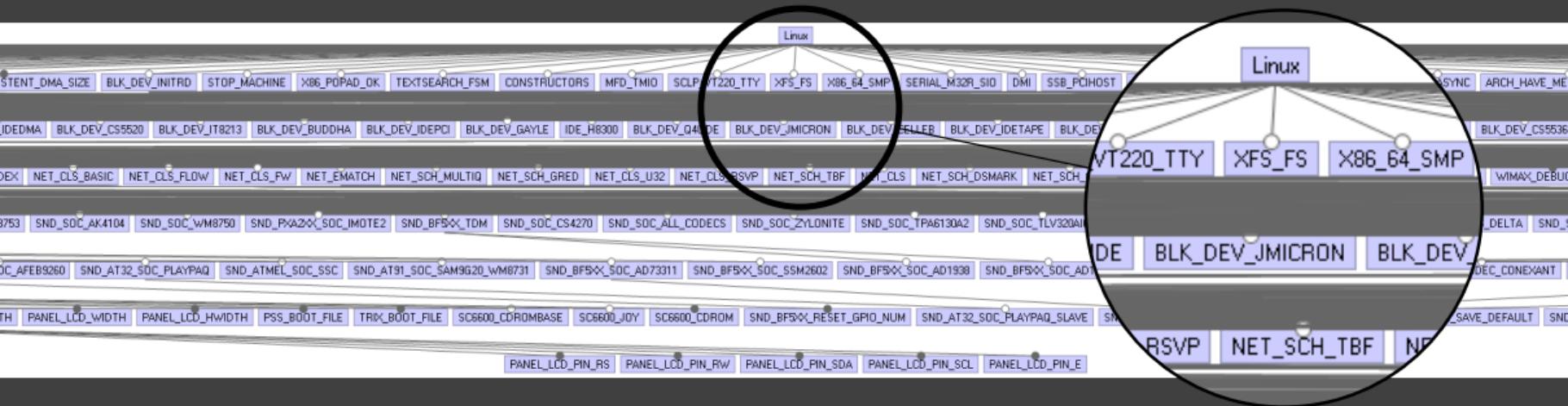
[Lecture 4]

{C, G, W}	{C, G, L}
{C, P, W}	{C, P, L}
{C, G, P, W}	{C, G, P, L}
{C, D, W}	{C, D, L}
{C, G, D, W}	{C, G, D, L}
{C, P, D, W}	{C, P, D, L}
{C, G, P, D, W}	{C, G, P, D, L}
{C, P, T, W}	{C, P, T, L}
{C, G, P, T, W}	{C, G, P, T, L}
{C, D, T, W}	{C, D, T, L}
{C, G, D, T, W}	{C, G, D, T, L}
{C, P, D, T, W}	{C, P, D, T, L}
{C, G, P, D, T, W}	{C, G, P, D, T, L}

## Discussion

- only feasible for **small** product lines (few valid configurations)
- redundant test effort
- large product lines: not feasible to generate and compile all configurations
  - (some) large product lines: even number of valid configurations is unknown

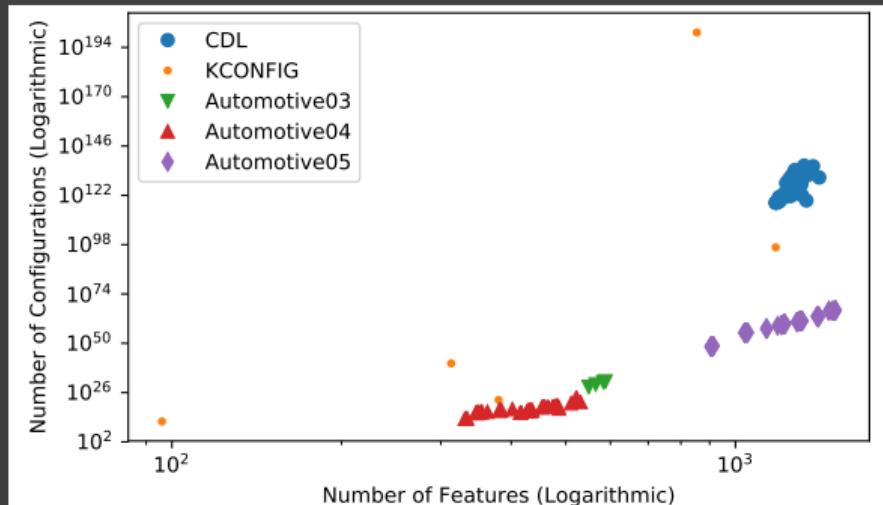
# Recap: Feature Model of the Linux Kernel



# Testing All Configurations

## Recap: Industrial Configuration Spaces

[Lecture 1]



Why being complete on the configurations then?

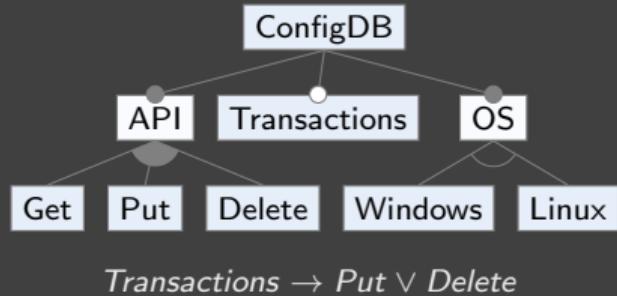


Edsger W. Dijkstra (1972)

"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

[The Humble Programmer]

# Testing One Configuration



## Recap: 26 Valid Configurations

[Lecture 4]

{C, G, W}	{C, G, L}
{C, P, W}	{C, P, L}
{C, G, P, W}	{C, G, P, L}
{C, D, W}	{C, D, L}
{C, G, D, W}	{C, G, D, L}
{C, P, D, W}	{C, P, D, L}
{C, G, P, D, W}	{C, G, P, D, L}
{C, P, T, W}	{C, P, T, L}
{C, G, P, T, W}	{C, G, P, T, L}
{C, D, T, W}	{C, D, T, L}
{C, G, D, T, W}	{C, G, D, T, L}
{C, P, D, T, W}	{C, P, D, T, L}
{C, G, P, D, T, W}	{C, G, P, D, T, L}

## Discussion

- applicable to large product lines
- no redundant test effort (from configurations)
- strategy in practice: all-yes-config (configuration with many features selected)
- often unfeasible to test all features with a single configuration (e.g., Windows and Linux)  
⇒ unnoticed feature interactions

[Lecture 9]

## What about interactions with missing features?



# Sample-Based Testing

## Intuition

- to analyze the product line, just analyze **some products**
- sample refers to a subset of all valid configurations
- common technique to test a product line
- sample configurations chosen by experts, randomly, or systematically

## Advantages and Challenges

- + lower effort than testing all configurations
- + higher chance to detect defects than testing one configuration
- how many configurations to test?  
which configurations to test?



# Challenges of Product-Line Testing – Summary

## Lessons Learned

- recap on software testing and test-case design
- testing all configurations
- testing one configuration
- sample-based testing

## Further Reading

- Varshosaz et al. 2018: overview on sampling literature
- Sampling Database: database on sampling algorithms and evaluations

## Practice

Recap on feature interactions: What are examples of interactions that cannot be detected statically (cf. Lecture 10) and could be missed when testing a single configuration only?

# 11. Product-Line Testing

## 11a. Challenges of Product-Line Testing

## 11b. Combinatorial Interaction Testing

Pairwise Interaction Testing

T-Wise Interaction Testing

Algorithms for Combinatorial Interaction Testing

Efficiency of Combinatorial Interaction Testing

Effectiveness of Combinatorial Interaction Testing

Summary

## 11c. Solution-Space Sampling

# Recap: Black-Box Testing

## Motivation

[Ludewig and Licher 2013]

- source code not always available (e.g., outsourced components, obfuscated code)
- errors are not equally distributed

## Black-Box Testing

[Ludewig and Licher 2013]

- test-case design based on specification
- source code and its inner structure is ignored (assumed as a black-box)

## Sample Configuration $\neq$ Test Case

- test case: concrete inputs and expected outputs for a program
- sample configuration: selection of features to derive the program
- both needed when testing product lines
- often confused in the literature
- test case derivation
  - out of scope here
  - global tests (i.e., identical for all configurations)
  - product-line implementation technique used to automatically derive configuration-specific tests
- on next slides: idea of black-box testing applied to derive sample configuration

[Lecture 8]

# Pairwise Interaction Testing

## Configurations with the Interaction Get $\wedge$ Put

$\{C, G, W\}$	$\{C, G, L\}$
$\{C, P, W\}$	$\{C, P, L\}$
$\{C, G, P, W\}$	$\{C, G, P, L\}$
$\{C, D, W\}$	$\{C, D, L\}$
$\{C, G, D, W\}$	$\{C, G, D, L\}$
$\{C, P, D, W\}$	$\{C, P, D, L\}$
$\{C, G, P, D, W\}$	$\{C, G, P, D, L\}$
$\{C, P, T, W\}$	$\{C, P, T, L\}$
$\{C, G, P, T, W\}$	$\{C, G, P, T, L\}$
$\{C, D, T, W\}$	$\{C, D, T, L\}$
$\{C, G, D, T, W\}$	$\{C, G, D, T, L\}$
$\{C, P, D, T, W\}$	$\{C, P, D, T, L\}$
$\{C, G, P, D, T, W\}$	$\{C, G, P, D, T, L\}$

## Pairwise Interaction Testing

- create a sample  $S \subseteq C$ , in which every pairwise interaction is covered by at least one configuration
- test every configuration in  $S$

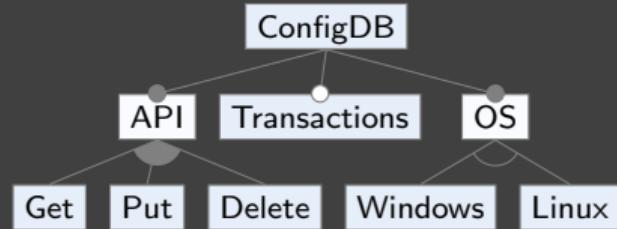
## Pairwise Combinations

- four combinations between  $A$  and  $B$ 
  - both selected:  $A \wedge B$
  - one selected:  $\neg A \wedge B$  and  $A \wedge \neg B$
  - none selected:  $\neg A \wedge \neg B$

## Discussion

- applicable to large product lines
- reduced redundant effort compared to testing all configurations
- full coverage guarantee  
(opposed to random configurations)
- still requires good test cases (program inputs)
- hard to compute small sample sets

# Pairwise Coverage



*Transactions*  $\rightarrow$  *Put*  $\vee$  *Delete*

## Interactions to Cover

- exclude abstract features (e.g., *API*, *OS*)
- exclude features contained in every configuration (e.g., *C*)
- exclude invalid combinations (e.g.,  $W \wedge L$ )

## Pairwise Interactions

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

## Pairwise Coverage with Six Configurations

{C, P, D, T, W}  
{C, G, D, L}  
{C, G, P, T, L}  
{C, G, W}  
{C, P, W}  
{C, D, T, L}

# T-Wise Interaction Testing

## T-Wise Interaction Testing

- generalization of pairwise interaction testing
- t-wise coverage: every t-wise interaction is covered by at least one configuration in the sample
- $t = 1$ : every feature is selected and also deselected
- $t = 2$ : pairwise interaction coverage
- $t = 3$ : every valid combination of three features covered

## $t = 3$ Interactions

for the features  $G$ ,  $P$ , and  $D$ :

$$\begin{array}{ll} G \wedge P \wedge D & \neg G \wedge P \wedge D \\ G \wedge P \wedge \neg D & \neg G \wedge P \wedge \neg D \\ G \wedge \neg P \wedge D & \neg G \wedge \neg P \wedge D \\ G \wedge \neg P \wedge \neg D & \neg G \wedge \neg P \wedge \neg D \end{array}$$

# Algorithms for Combinatorial Interaction Testing

## A Greedy Algorithm

idea: select configuration that cover most missing interactions in each step

1. randomly choose first configuration
2. find next optimal configuration
3. repeat step 2 until all interactions are covered

## Challenges and Optimizations

- non-deterministic: different sample for each run (cf. Step 1)
  - starting with all-yes-config?  $\Rightarrow$  covers more code
- iterating all valid configurations does not scale (cf. Step 2)
- greedy strategy: optimal configuration in each step does not guarantee optimal sample

## ICPL

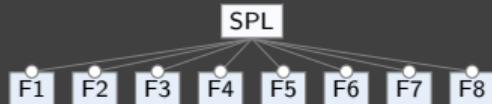
[Johansen et al. 2012]

- widespread greedy algorithm
- iterates over all interactions
  - identifies core and dead features early
  - identifies invalid and already covered interactions
  - utilizes parallelization
- incrementally increases  $t$  up to desired value
- performance shown on next slides

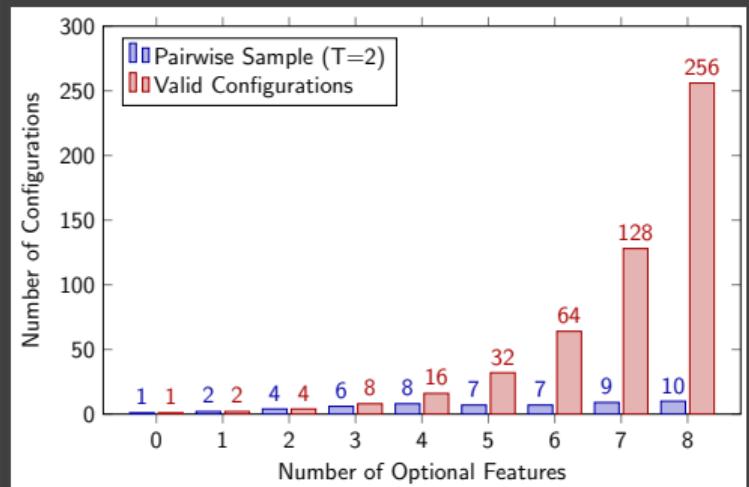
# Efficiency of Combinatorial Interaction Testing

[Johansen et al. 2012]

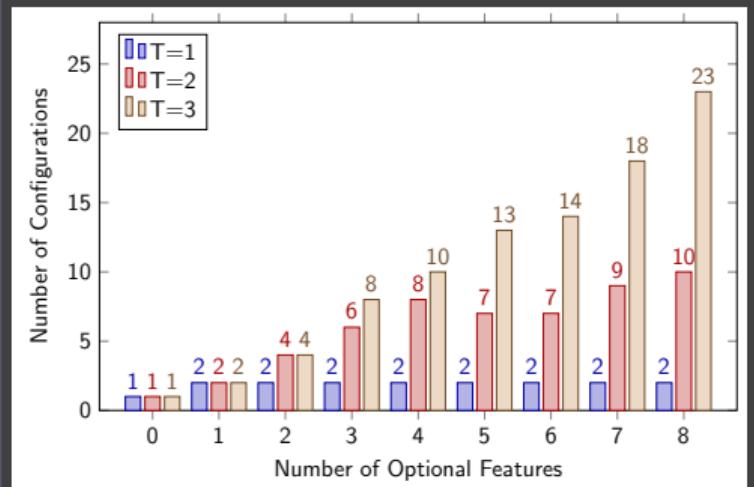
Assumption: All Features are Optional



Number of Configurations in Pairwise Sample



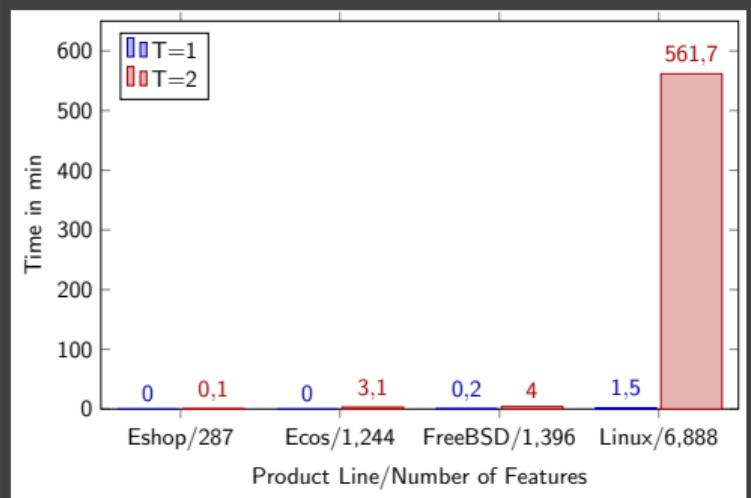
Number of Configurations in T-Wise Sample



# Efficiency of Combinatorial Interaction Testing

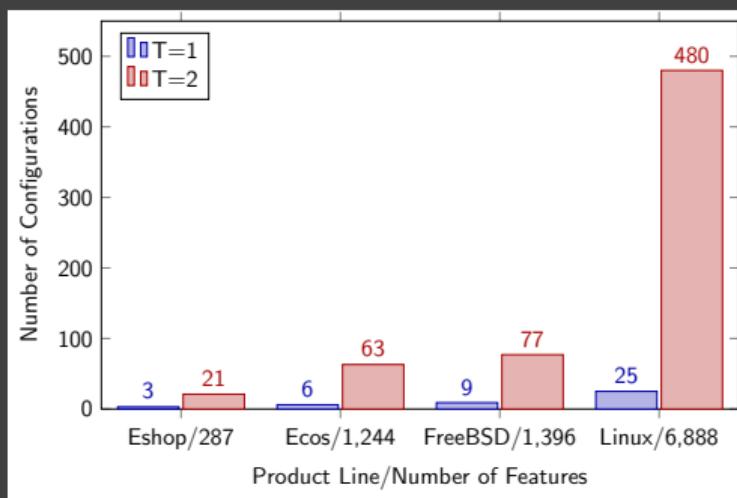
[Johansen et al. 2012]

Time in Minutes to Compute Sample



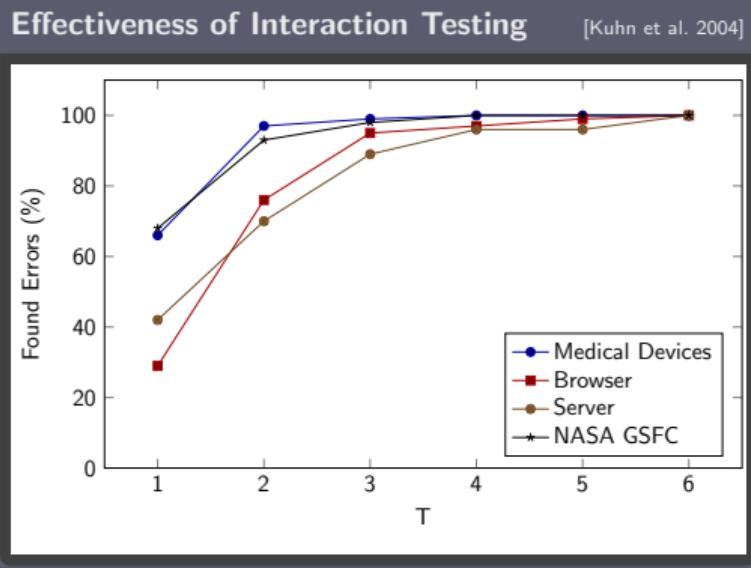
- about 9h for Linux
- 480 configuration in pairwise sample

Number of Configurations in Sample



- Linux kernel v2.6.28.6 (February 2009)
- 6,888 features
- 187,193 clauses in conjunctive normal form

# Effectiveness of Combinatorial Interaction Testing



## Trade-Off

large  $t$ : high coverage (more effective)  
small  $t$ : low testing effort (more efficient)

# Combinatorial Interaction Testing – Summary

## Lessons Learned

- recap on black-box testing
- combinatorial interaction testing: pairwise testing, t-wise testing
- efficiency: number of configurations, time to compute sample
- effectiveness: percentage of found defects

## Further Reading

- Johansen et al. 2012 – popular t-wise sampling algorithm ICPL
- Krieter et al. 2020 – alternative t-wise sampling algorithm YASA

## Practice

Why is it hard to find a good trade-off between efficiency and effectiveness?

# 11. Product-Line Testing

## 11a. Challenges of Product-Line Testing

## 11b. Combinatorial Interaction Testing

## 11c. Solution-Space Sampling

Coverage in Single-System Engineering

Coverage of Ifdef Blocks

Presence-Condition Coverage

Overview on Coverage Criteria

Input for Sampling Algorithms

Summary

FAQ

# Recap: Coverage in White-Box Testing

[Ludewig and Licher 2013]

## White-Box Testing

- inner structure of test object is used
- idea: coverage of structural elements
  - code translated into control flow graph
  - specific test case (concrete inputs) derived from logical test case (conditions)
  - derived from path in control flow graph

## Coverage Criteria

1. **statement coverage** : all statements are executed for at least one test case
2. **branching coverage** : statement coverage and all branches of branching statement are executed
3. **term coverage** : branching coverage and all terms used in a branching statement ( $n$ ) are combined exhaustively ( $2^n$ )  
(simplified)

## Can You Spot Problems in the Elevator Product Line?

[Varshosaz et al. 2018]

```
1 class ControlUnit {           16    ##if DirectedCall          31  }
2   Elevator elevator;         17    sortQueue();            32  //##if Service
3   ElevatorState state, nextState; 18    ##endif               33  boolean serviceNextState() {...}
4   ##if FIFO                  19    }                      34  ##endif
5   Req req = new Req();       20  }                      35  //##if Sabbath
6   ##elif DirectedCall        21  void calculateNextState() { 36  boolean sabbathNextState() {...}
7   Req req = new UndReq(this); 22    ##if Sabbath            37  ##endif
8   Req dreq = new DirReq(null); 23    if (sabbathNextState()) return; 38  ##if DirectedCall
9   ##else                      24    ##endif               39  void callButtonsNextState(Req d)
0   Req req = new Req(this);   25    ##if Service            40  void sortQueue() {...}
1   ##endif                     26    if (serviceNextState()) return; 41  ##endif
2   void run() {              27    ##endif               42  }
3     while (true) {           28    ##if FIFO
4       calculateNextState();  29    callButtonsNextState(dreq); 30  ##endif
5       setDirection(nextState); }
```

- Line 29: compiler error (i.e., field `dreq` undefined) when  $FIFO \wedge \neg DirectedCall$
- Line 8: null pointer exception when  $DirectedCall \wedge \neg FIFO$
- both problems detectable with pairwise coverage, but presence conditions are more complicated in practice
- also: pairwise coverage often too much effort for large configuration spaces / continuous integration

## Coverage of Ifdef Blocks

[Tartler et al. 2012]

- every block selected for at least one configuration in the sample (cf. statement coverage)

# Presence-Condition Coverage

## Presence-Condition Coverage

[Krieter et al. 2022]

- application of t-wise interaction testing to presence conditions
- recap presence condition: formula specifying exactly those configurations under which a block is present
- t-wise presence condition coverage: every t-wise interaction of presence conditions is covered by at least one configuration in the sample
- $t = 1$ : every block is selected and also deselected (i.e., more than Tartler's coverage of ifdef blocks)
- $t = 2$ : every combination of two blocks covered
- $t = 3$ : every combination of three blocks covered

## T=3 Presence-Condition Interactions

for the blocks  $a$ ,  $b$ , and  $c$  with presence conditions

$A$ ,  $B$ , and  $C$ :

$A \wedge B \wedge C$

$A \wedge B \wedge \neg C$

$A \wedge \neg B \wedge C$

$A \wedge \neg B \wedge \neg C$

$\neg A \wedge B \wedge C$

$\neg A \wedge B \wedge \neg C$

$\neg A \wedge \neg B \wedge C$

$\neg A \wedge \neg B \wedge \neg C$

## Presence-Condition Coverage

[Krieter et al. 2022]

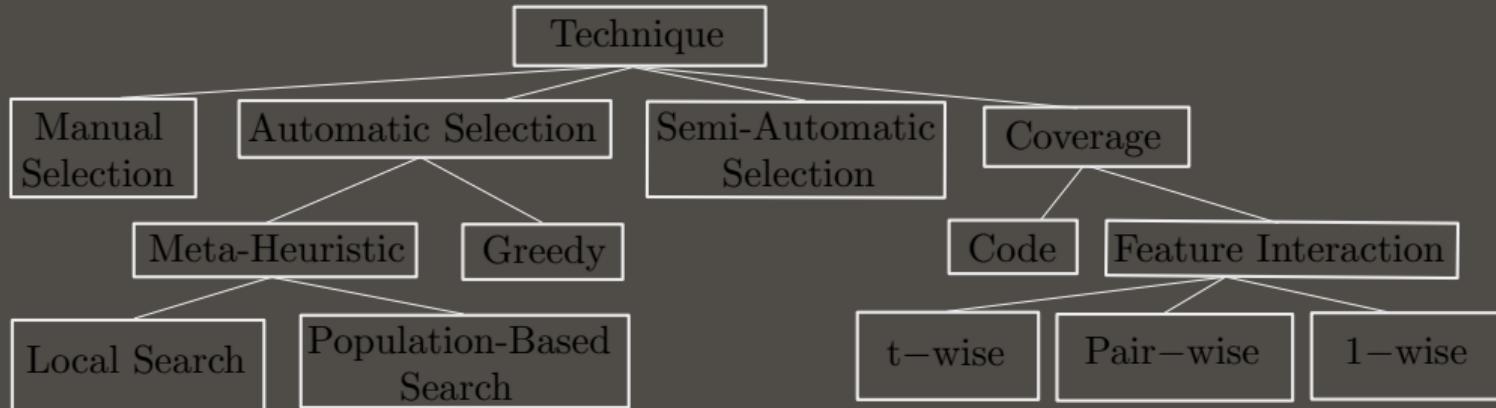
- coverage of solution space (not problem space)
- aka. solution-space sampling
- for same  $t$ : often fewer configurations and similar effectiveness than feature interaction coverage
- also feasible by translating presence conditions into feature model

[Hentze et al. 2022]

# Overview on Coverage Criteria

## Techniques & Coverage Criteria

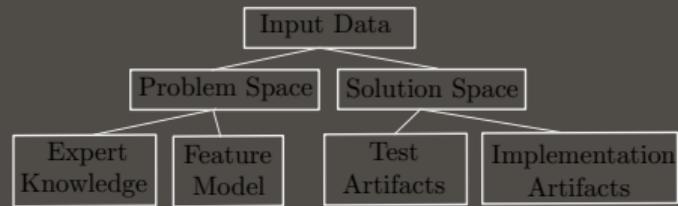
[Varshosaz et al. 2018]



# Input for Sampling Algorithms

## Input Data

[Varshosaz et al. 2018]



## Further Domain Knowledge

[Varshosaz et al. 2018]

- in addition to feature model
- e.g., configurations chosen by experts
- e.g., specialized feature model for sampling

## Part 2: Combinatorial Interaction Testing

- (Problem-Space Sampling)
- **feature model** used to consider only valid configurations

## Part 3: Solution-Space Sampling

- mapping from features to **implementation artifacts**
- **feature model** used to consider only valid configurations

## Combinatorial Reduction of Tests

[Kim et al. 2011]

- which configurations matter for each test?
- analyze **unit tests** and **impl. artifacts**
- **feature model** used to consider only valid configurations

# Solution-Space Sampling – Summary

## Lessons Learned

- recap on white-box testing and coverage criteria
- coverage of ifdef blocks
- t-wise presence condition coverage
- overview on techniques, coverage criteria, input data for sampling

## Further Reading

- Tartler et al. 2012: covering every ifdef block (but not their absence)
- Krieter et al. 2022: solution-space sampling as discussed in this part
- Hentze et al. 2022: translation of presence conditions into feature model + reuse of problem-space sampling

## Practice

Does the order of configurations matter during testing?

[Al-Hajjaji et al. 2019]

# FAQ – 11. Product-Line Testing

## Lecture 11a

- What is the goal of quality assurance for software product lines?
- How can product lines be tested?
- Why is testing product lines challenging?
- What are (dis-)advantages of testing all configurations?
- What are (dis-)advantages of testing only one configuration?
- What is sample-based testing?
- What is a sample?
- How can a sample be computed?

## Lecture 11b

- How is black-box testing used for testing product lines?
- What is the difference between a test configuration and a test case?
- What are (dis-)advantages of combinatorial interaction testing?
- What is pairwise interaction testing?
- What is t-wise interaction testing?
- When does a sample achieve 100% pairwise coverage?
- How can a t-wise sample be computed?

## Lecture 11c

- How can white-box testing be used for testing product lines?
- What are potential problems with t-wise interaction testing?
- What is presence condition coverage?
- What are different techniques for t-wise sampling?
- Which additional inputs can be used for sampling algorithms?
- How efficient and how effective are sampling algorithms?

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. Evolution and Maintenance

### 12a. Product-Line Evolution

- Recap and Motivation
- Recap: Evolution of the Linux Kernel
- Evolution of Feature Models
  - Refactorings
  - Generalizations
- Co-Evolution of Product Lines
- Summary

### 12b. Product-Line Maintenance

- Recap on Software Maintenance
- Kinds of Maintenance
- Recap on Feature Model Transformations
- Reengineering Tasks
- Summary

### 12c. Course Summary

- The Vision of Product Lines
- Modeling Features
- Implementing Features
- Analyzing Features
- Summary
- FAQ

# 12. Evolution and Maintenance – Handout

Software Product Lines | Thomas Thüm, Elias Kuiter, Timo Kehrer | June 28, 2023



# 12. Evolution and Maintenance

## 12a. Product-Line Evolution

Recap and Motivation

Recap: Evolution of the Linux Kernel

Evolution of Feature Models

Refactorings

Generalizations

Co-Evolution of Product Lines

Summary

## 12b. Product-Line Maintenance

## 12c. Course Summary

# Recap and Motivation

Jimmy Koppel, 2019

[corecursive.com]

“Software maintenance is important because the world runs on software, and changing the world means changing the software.”

# Recap and Motivation

## Lehman's Laws of Software Evolution (excerpt)

[Lehman et al. 1997]

- Continuing Change: systems must be continually adapted to stay satisfactory
- Increasing Complexity: complexity increases during evolution unless work is done to maintain or reduce it
- Continuing Growth: functionality must be continually increased to maintain user satisfaction
- Declining Quality: quality will decline unless rigorously maintained and adapted to operational environment changes

## Essence of the Laws

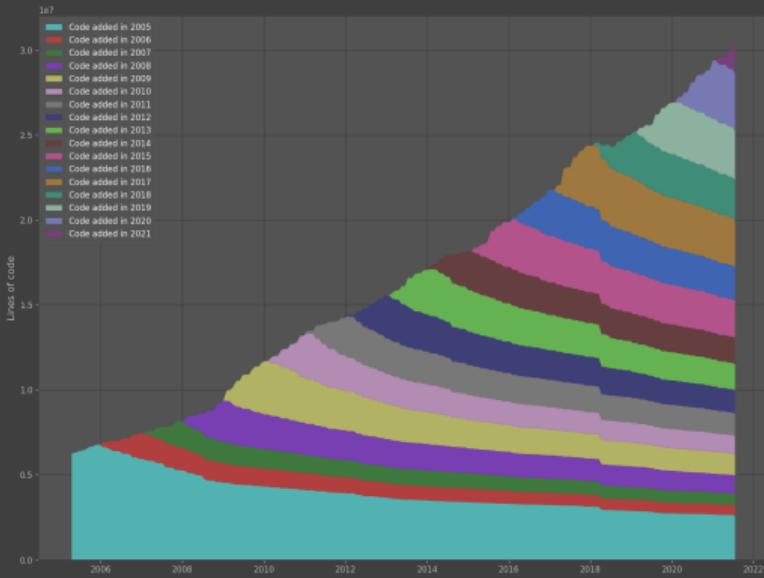
- software that is used will be modified
- when modified, its complexity will increase (unless one does actively work against it)

## Consequences for Product Lines

- number of features and size of implementation increases over time
- discussed challenges increase over time
  - more software clones
  - harder to trace features
  - automated generation more urgent
  - increasing combinatorial explosion
  - more feature interactions

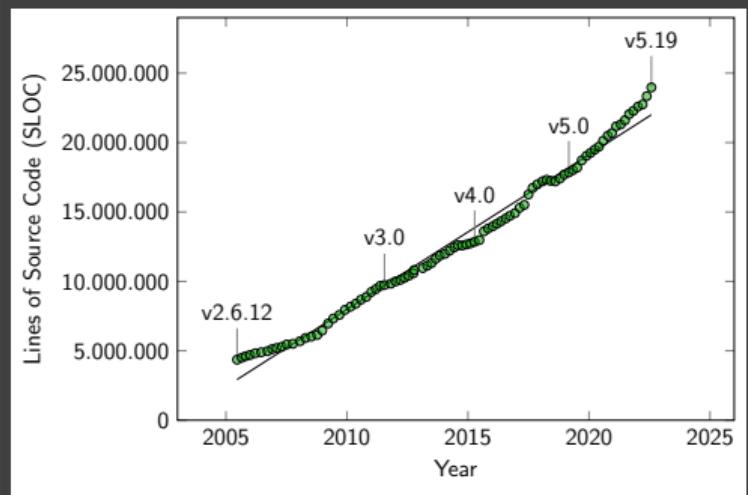
# Evolution of the Linux Kernel

- about 60,000 commits per year
- in peak weeks: new commit every 5 minutes
- in average weeks: every 9 minutes



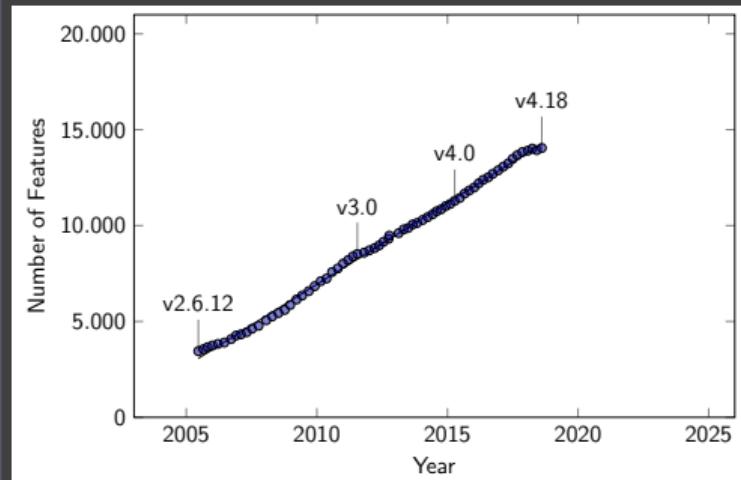
# Evolution of the Linux Kernel

Size of the Code Base



- from 4 to 24 millions in 17 years
- about one million LOC added every year
- about 3,000 LOC per day

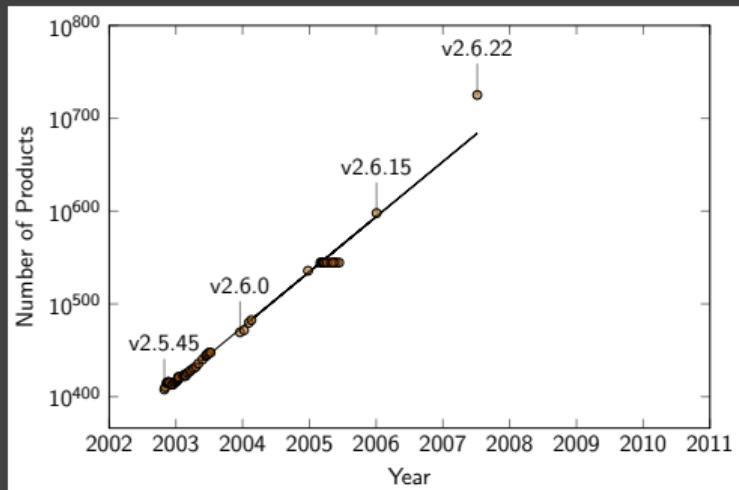
Number of Features



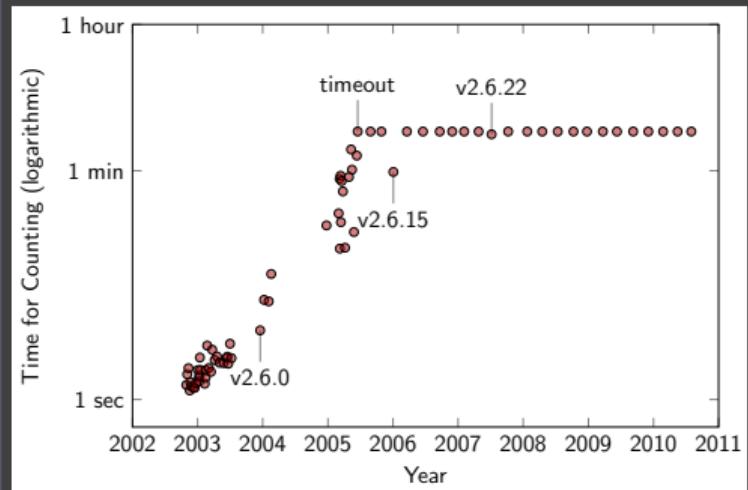
- about 800 new features every year
- about 15 new features every week
- in 2018 four times more features than in 2005

# Evolution of the Linux Kernel

Number of Products



Time to Count Products



- number of products grows by factor 100.000 each month
- the current kernel is likely to have more than  $10^{1500}$  products

- most kernel versions before 2006 can be computed within 1 minute
- most kernel versions after 2006 cannot be computed within 1 hour

# Evolution of Feature Models [Thüm et al. 2009]

## Classification of Changes

	No Products Added	Products Added
No Products Deleted	Refactoring	Generalization
Products Deleted	Specialization	Arbitrary Edit

## Automated Classification

classification can be reduced to SAT problems

## Advantages for Quality Assurance

assumption: only feature model has changed

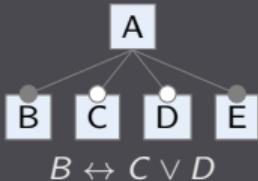
- refactoring: no retest needed
- specialization: cannot produce new faults
- generalization: cannot fix known faults
- arbitrary edit: retest needed

## Advantages for Modelers

did the configuration space change as intended?

# Evolution of Feature Models

Version 1



$$B \leftrightarrow C \vee D$$

Version 2



## Refactoring

1 to 2, 2 to 1

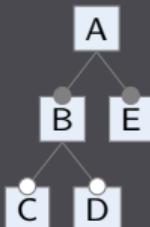
## Generalization

1/2 to 3/4

Version 3



Version 4



## Specialization

3/4 to 1/2

## Arbitrary Edit

3 to 4, 4 to 3

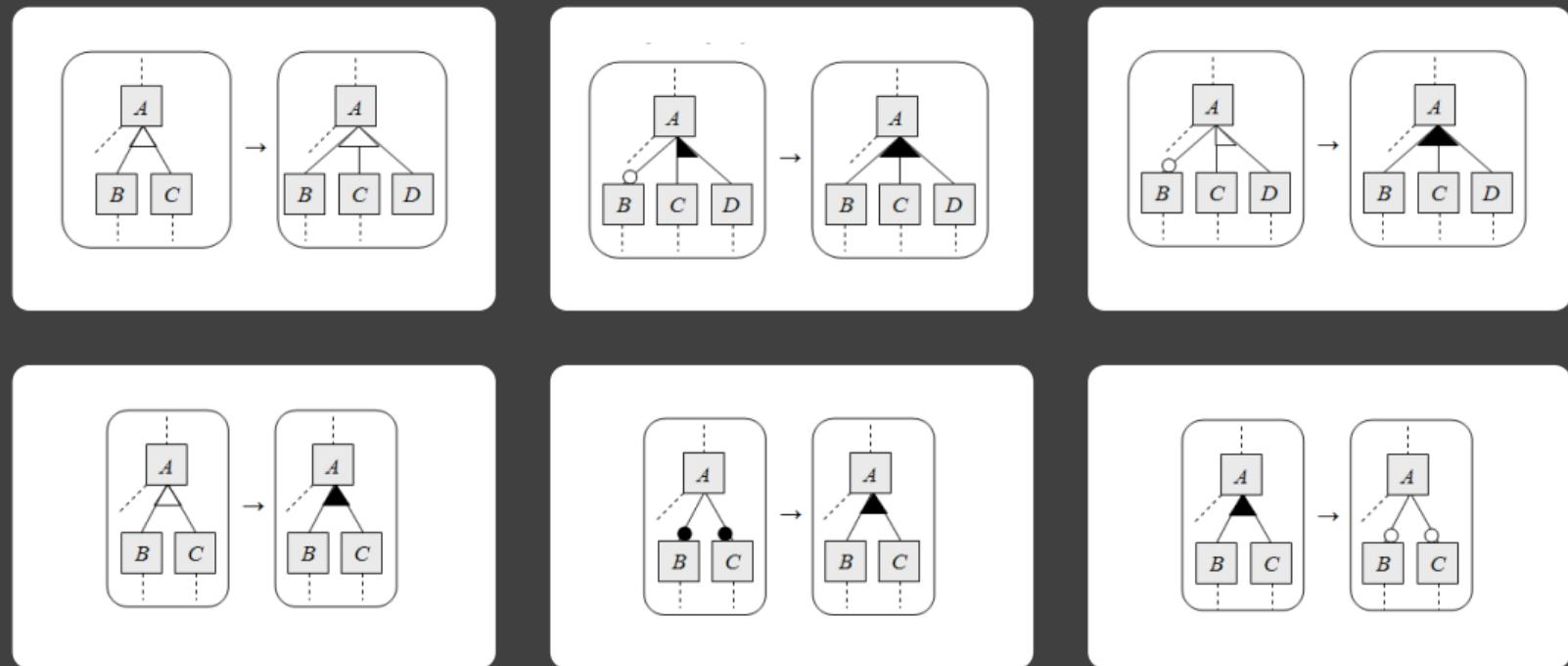
# Evolution of Feature Models – Refactorings

[Alves et al. 2006]



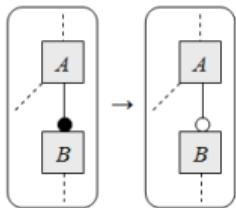
# Evolution of Feature Models – Generalizations

[Alves et al. 2006]

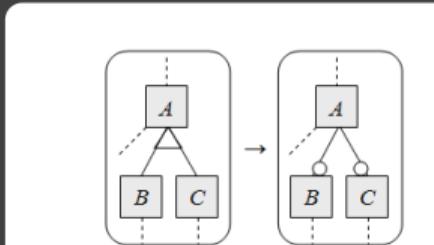
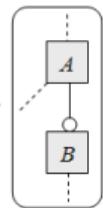


# Evolution of Feature Models – Generalizations

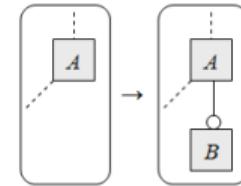
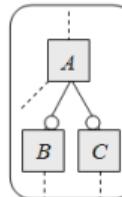
[Alves et al. 2006]



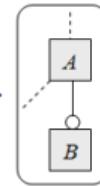
→



→



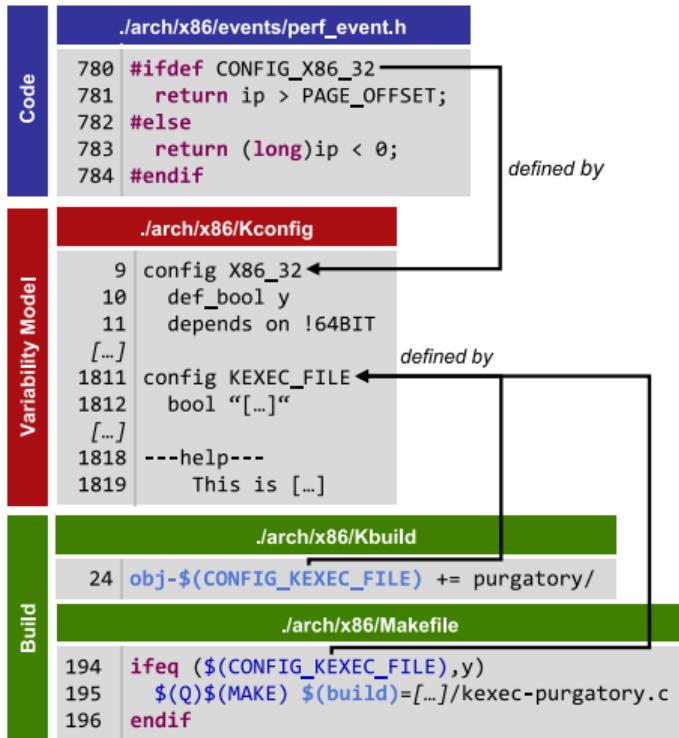
→



$forms \wedge f$  →  $forms$

⋮ ⋮

# Co-Evolution of Product Lines [Kröher et al. 2023]



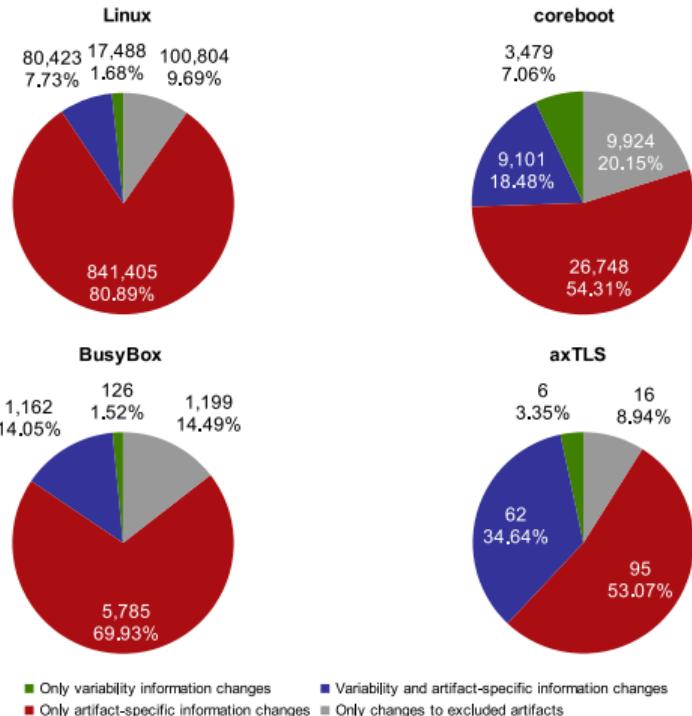
- not only feature models evolve
- but also source code
- and mapping from features to source code
- aka. **co-evolution**
- for conditional compilation:
  1. feature model
  2. build scripts with features
  3. source code with preprocessor statements
- how frequent are those changes?

# Co-Evolution of Product Lines [Kröher et al. 2023]

Code	Commit Excerpt
	<pre>780 #ifdef CONFIG_X86_32 781     return ip &gt; PAGE_OFFSET; 782 -#else 783 -    return (long)ip &lt; 0; 784 #endif</pre> <p>Variability information</p> <p>Artifact-specific information</p>
Variability Model	Commit Excerpt
	<pre>1811 +config KEXEC_FILE 1812 +  bool "[...]" 1813 [...] 1818 +---help--- 1819 +  This is [...]</pre> <p>Variability information</p> <p>Artifact-specific information</p>
Build	Commit Excerpt
	<pre>194 -ifeq (\$CONFIG_KEXEC_FILE),y) 195 -  \$(Q)\$(MAKE) \$(build)=... 196 -endif</pre> <p>Variability information</p> <p>Artifact-specific information</p> <p>Variability information</p>

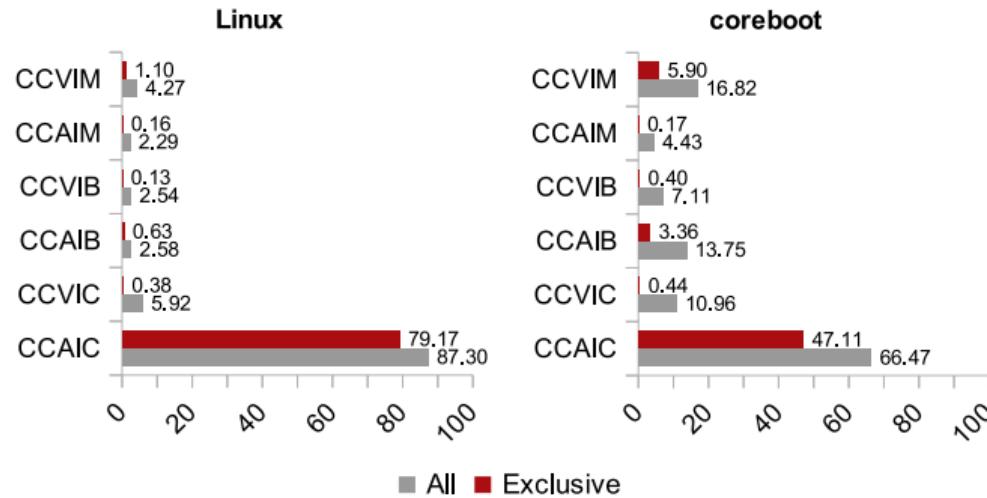
- variability information evolves
- artifact-specific information evolves
- how frequent are those changes?

# Co-Evolution of Product Lines [Kröher et al. 2023]



- most changes only affect artifact-specific information
- fewest changes only affect variability information

# Co-Evolution of Product Lines [Kröher et al. 2023]



CCVIM = Commits Changing Variability Information in Model artifacts  
CCAIM = Commits Changing Artifact-specific Information in Model artifacts  
CCVIB = Commits Changing Variability Information in Build artifacts  
CCAIB = Commits Changing Artifact-specific Information in Build artifacts  
CCVIC = Commits Changing Variability Information in Code artifacts  
CCAIC = Commits Changing Artifact-specific Information in Code artifacts

- 1 % of commits in Linux **only** change feature model
- 4 % of commits in Linux change feature model
- <1 % of commits in Linux **only** change build scripts
- 3 % of commits in Linux change build scripts
- 79 % of commits in Linux **only** change source code
- 87 % of commits in Linux change source code
- different percentages for other systems

# Product-Line Evolution – Summary

## Lessons Learned

- changes are inevitable and occur frequently
- product lines typically grow over time
- different kinds of changes to feature models (e.g., refactorings)
- co-evolution of feature model, feature mapping, artifacts

## Further Reading

- Thüm et al. 2009
  - SAT-based classification of feature-model changes
- Alves et al. 2006
  - patterns for feature-model refactorings
- Kröher et al. 2023
  - product-line evolution

## Practice

After which changes do we need to analyze and test the product line again?

Do we always need to analyze and test the complete product line again?

# **12. Evolution and Maintenance**

## **12a. Product-Line Evolution**

### **12b. Product-Line Maintenance**

Recap on Software Maintenance

Kinds of Maintenance

Recap on Feature Model Transformations

Reengineering Tasks

Summary

## **12c. Course Summary**

# Recap on Software Maintenance

[Ludewig and Licher 2013]

## Motivation

- for software: no compensation of deterioration, repair, spare parts
- corrections (especially shortly after first delivery)
- modification and reconstruction

## Operation and Maintenance Phase

[IEEE Std 610.12]

“The period of time in the software life cycle during which a software product is employed in its operational environment, monitored for satisfactory performance, and modified as necessary to correct problems or to respond to changing requirements.”

## Maintenance

[IEEE Std 610.12]

“The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.”

# Recap on Software Maintenance

[Ludewig and Licher 2013]

## Evolution

- new or removed functionality
- larger changes
- often foreseen changes
- results in upgrades, service packs, or cumulative updates

## Maintenance

- mostly corrections
- smaller changes
- often unforeseen changes
- results in patches and hot fixes

## Minor Release

new minor version: 2.3.1 ⇒ 2.4.0

## Patch Release

new patch version: 2.3.1 ⇒ 2.3.2

## Major Release

new major version: 2.3.1 ⇒ 3.0.0

not easy to distinguish – there is a continuum between evolution and maintenance

# Kinds of Maintenance

[Ludewig and Licher 2013]

## Adaptive Maintenance

[Ludewig and Licher 2013]

“Software maintenance performed to make a computer program usable in a changed environment.”

desktop application for a new version of an operating system (e.g., from Windows 10 to 11)

## Corrective Maintenance

[Ludewig and Licher 2013]

“Maintenance performed to correct faults in software.”

feature interaction of Lenovo products fixed with BIOS update

## Perfective Maintenance

[Ludewig and Licher 2013]

“Software maintenance performed to improve the performance, maintainability, or other attributes of a computer program.”

improve start-up time of the Linux kernel

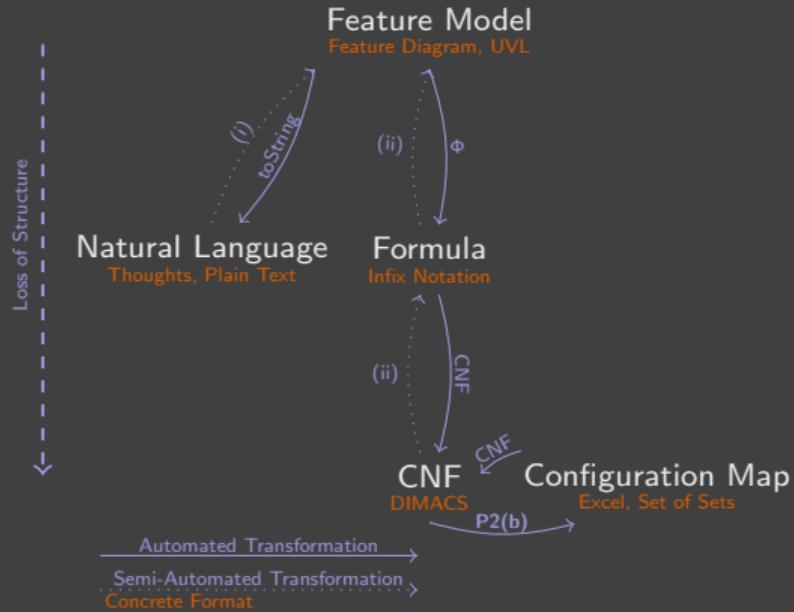
## Preventive Maintenance

[Ludewig and Licher 2013]

“Maintenance performed for the purpose of preventing problems before they occur.”

code audit of car software before next leap second

# Recap on Feature Model Transformations



## Problems

- P1 How to express feature models **textually**?
- P2 How to
- validate configurations and
  - get all valid configurations
- automatically**?
- P3 (How to reverse engineer feature models?)

## Solutions

- P1 Universal Variability Language  $\Rightarrow$  **Syntax**
- P2 Propositional Formulas  $\Rightarrow$  **Semantics**
- evaluate feature-model formula
  - Lecture 4c
- P3 (i) e.g., Bakar et al. 2015  
(ii) e.g., Czarnecki and Wasowski 2007

# Reengineering Tasks

[Ludewig and Licher 2013]

## Reverse Engineering

[Chikofsky und Cross]

“Reverse engineering is the process of analyzing a system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction.”

create feature model from existing configurations

## Forward Engineering

[Chikofsky und Cross]

“Forward engineering is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system.”

domain engineering

[Lecture 8a]

## Refactoring

[Chikofsky und Cross]

“Refactoring is a transformation from one form of representation to another at the same relative level of abstraction. The new representation is meant to preserve the semantics and external behavior of the original.”

feature-model refactorings

[Lecture 12a]

## Reengineering

[Chikofsky und Cross]

“Reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”

combination of reverse engineering, refactoring, and forward engineering

# Product-Line Maintenance – Summary

## Lessons Learned

- what is maintenance?
- what are examples for product-line maintenance?
- what is reengineering?
- what are examples of product-line reengineering?

## Further Reading

- Ludewig and Licher 2013  
— maintenance and reengineering

## Practice

How do **product-line adoption strategies** (proactive, reactive, extractive) correlate with **reengineering tasks** (reengineering, refactoring, forward/reverse engineering)?

# **12. Evolution and Maintenance**

## **12a. Product-Line Evolution**

## **12b. Product-Line Maintenance**

## **12c. Course Summary**

The Vision of Product Lines

Modeling Features

Implementing Features

Analyzing Features

Summary

FAQ

# Software Product Lines

## Part I: Ad-Hoc Approaches for Variability

1. Introduction
2. Runtime Variability and Design Patterns
3. Compile-Time Variability with Clone-and-Own

## Part II: Modeling & Implementing Features

4. Feature Modeling
5. Conditional Compilation
6. Modular Features
7. Languages for Features
8. Development Process

## Part III: Quality Assurance and Outlook

9. Feature Interactions
10. Product-Line Analyses
11. Product-Line Testing
12. **Evolution and Maintenance**

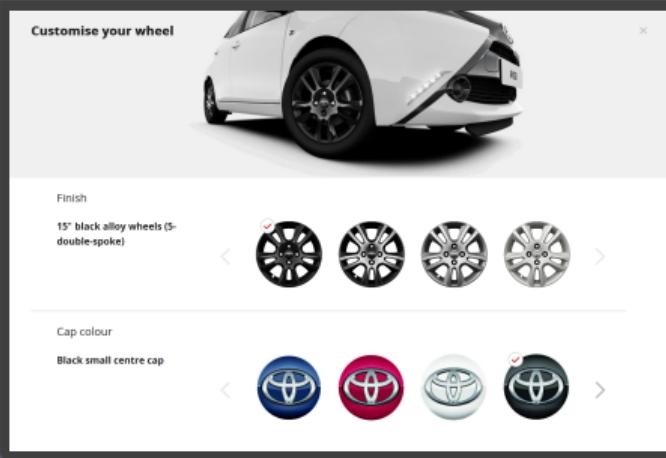
# The Vision of Product Lines

## Mass Customization

[Apel et al. 2013, p. 4]

- = mass production + customization
- customized, individual goods at costs similar to mass production

## Car Configuration



## Car Production



## Other Domains

bikes, computers, electronics, tools, medicine, clothing, food, financial services, . . . , software?

# The Vision of Product Lines

## Software Product Line

[Northrop et al. 2012, p. 5]

"A **software product line** is

- a set of software-intensive systems

aka. products or variants

- that share a common, managed set of features

common set, but not all products have all features in common

- satisfying the specific needs of a particular market segment or mission

aka. domain

- and that are developed from a common set of core assets in a prescribed way."

aka. planned, structured reuse

[Software Engineering Institute, Carnegie Mellon University]

# The Vision of Product Lines

## Product-Line Engineering

[Pohl et al. 2005, p. 14]

“Software product-line engineering is a paradigm to develop software applications (software-intensive systems and software products) using software platforms and mass customization.”

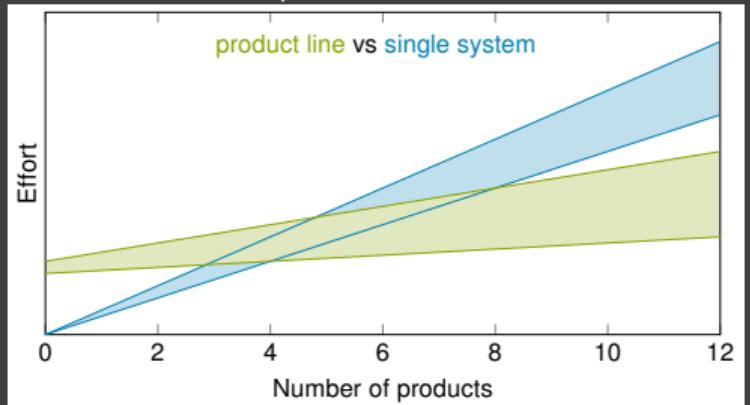
## Promises of Product Lines

[Apel et al. 2013, pp. 9–10]

- tailor-made
- reduced costs
- improved quality
- reduced time-to-market

## Idea of Product-Line Engineering

Reduce effort per product by means of an up-front investment for the product line:



## Single-System Engineering

classical software development that is not considered as product-line engineering

# Modeling Features

## Natural Language

"A configurable database has an API that allows for at least one of the request types Get, Put, or Delete. Optionally, the database can support transactions, provided that the API allows for Put or Delete requests. Also, the database targets a supported operating system, which is either Windows or Linux."

## Configuration Map

$\{C, G, W\}$

:

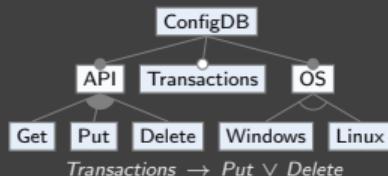
$\{C, G, P, D, T, W\}$

$\{C, G, L\}$

:

$\{C, G, P, D, T, L\}$

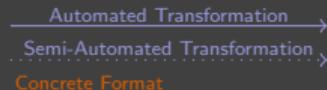
## Feature Diagram (Graphical Feature Model)



## Feature Model Feature Diagram, P1

## Natural Language Thoughts, Plain Text

## Configuration Map Excel, Set of Sets



## Problems

- P1 How to express feature models **textually**?
- P2 How to (a) validate configurations and (b) get all valid configurations **automatically**?
- P3 (How to reverse engineer feature models?)

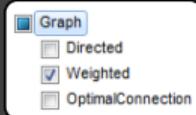
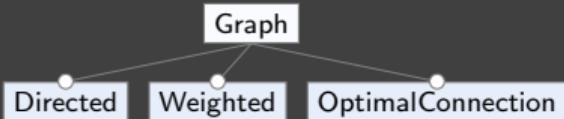
# Modeling Features

## Problem Space

[Apel et al. 2013, p. 21]

"The **problem space** takes the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. Features are, in fact, domain abstractions that characterize the problem space."

[Lecture 4]



## Solution Space

[Apel et al. 2013, p. 21]

"The **solution space** represents the developer's and vendor's perspectives. It is characterized by the terminology of the developer, which includes names of functions, classes, and program parameters. The solution space covers the design, implementation, and validation and verification of features and their combinations in suitable ways to facilitate systematic reuse."

[Lecture 2, Lecture 3, Lecture 5, Lecture 6, Lecture 7]

```
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));  
    } }
```



Product-Line Requirements

## Domain Engineering



Domain Analysis



Domain Design



Domain Implementation



Domain Testing



Product Requirements

## Application Engineering



Application Analysis



Application Design



Application Implementation



Application Testing



Product

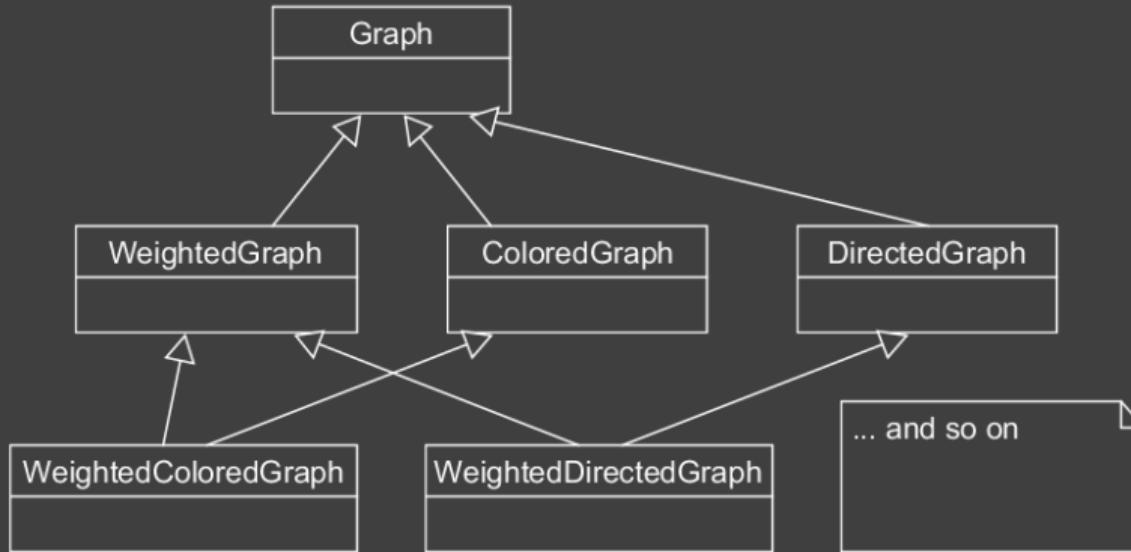
# Implementing Features

	Compile-Time Variability	Features	Product Generation	Feature Traceability
Runtime Variability	no (very limited for immutable global variables)	only fine grained	yes (except for preference dialogs)	no
Clone-and-Own	yes (only for implemented products)	no	no (limited generation for clone-and-own with build systems)	no
Build Systems (for Conditional Compilation)	yes	only coarse grained	yes	with tool support
Preprocessors (for Conditional Compilation)	yes	only fine grained	yes	with tool support
Components/Services	yes	only coarse grained	no (except pure exchange)	only coarse grained
Frameworks with Plug-Ins	yes	only coarse grained	yes	only coarse grained
Feature Modules/Aspects	yes	yes	yes	yes

## Further Criteria

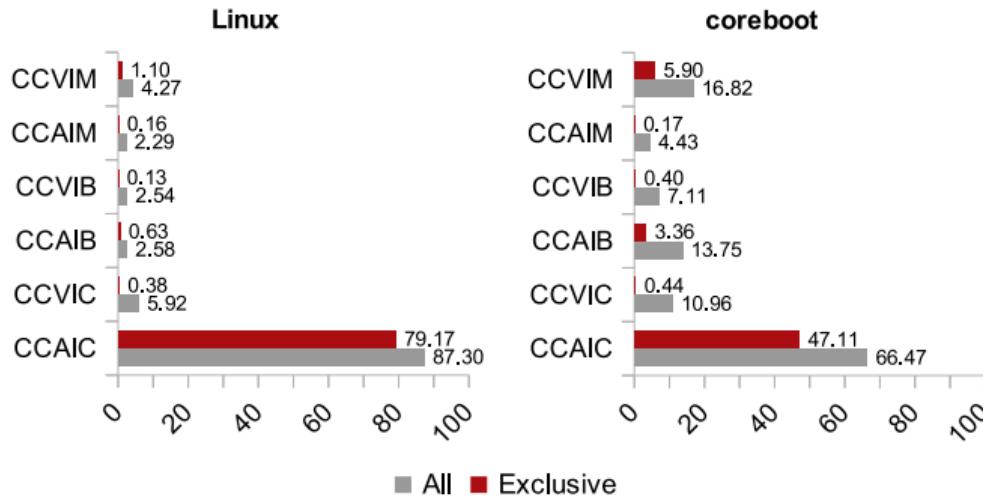
interfaces between features? code duplication necessary? modularization of cross-cutting concerns? ...

# Static Modeling of Feature Combinations



Even if multiple inheritance is supported, statically combining features through inheritance is tedious (or infeasible).

# Implementing Features [Kröher et al. 2023]

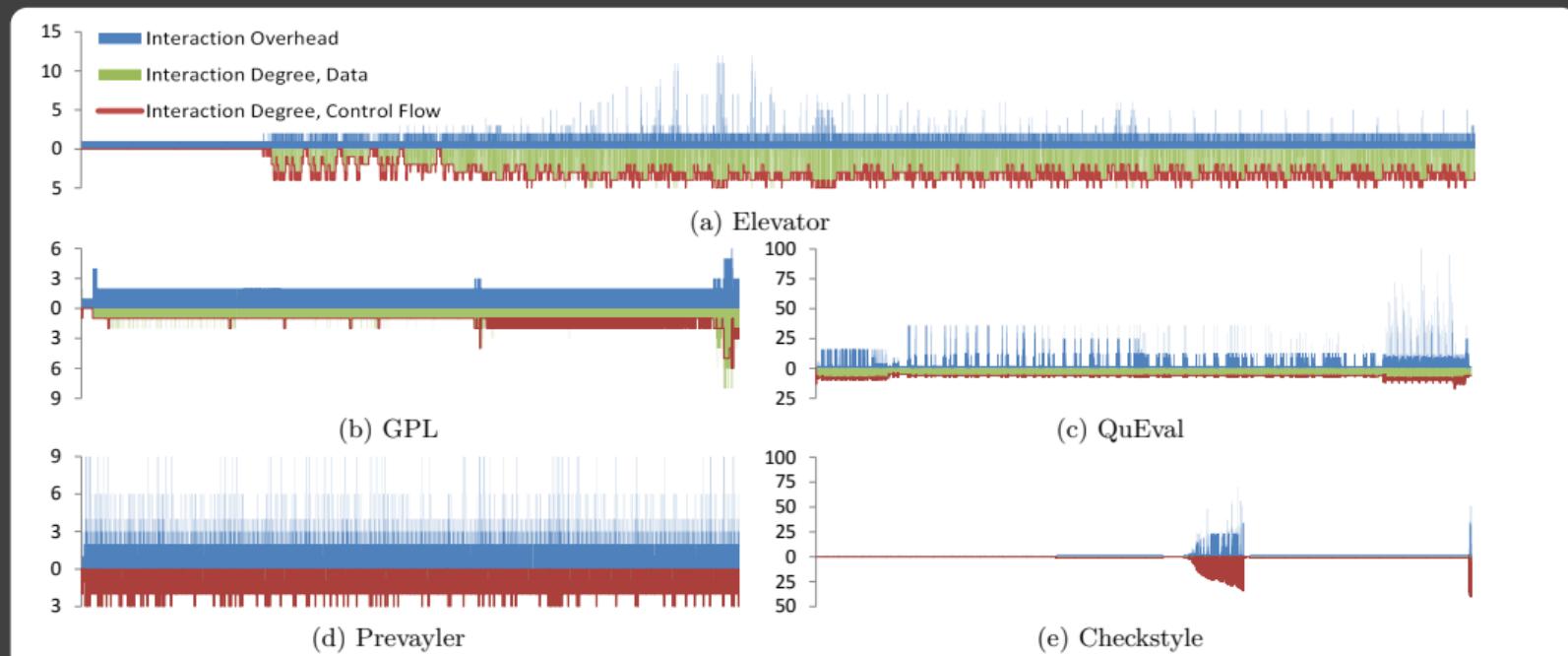


CCVIM = Commits Changing Variability Information in Model artifacts  
CCAIM = Commits Changing Artifact-specific Information in Model artifacts  
CCVIB = Commits Changing Variability Information in Build artifacts  
CCAIB = Commits Changing Artifact-specific Information in Build artifacts  
CCVIC = Commits Changing Variability Information in Code artifacts  
CCAIC = Commits Changing Artifact-specific Information in Code artifacts

- 1 % of commits in Linux **only** change feature model
- 4 % of commits in Linux change feature model
- <1 % of commits in Linux **only** change build scripts
- 3 % of commits in Linux change build scripts
- 79 % of commits in Linux **only** change source code
- 87 % of commits in Linux change source code
- different percentages for other systems

# Execution Traces in Configurable Systems

[Meinicke et al. 2016]



insights: not all features interact. some statements may lead to higher interactions than others.

# Analyzing Features



## Product-Based Strategy

- analyze individual **products**
- + sound, complete
- + uses off-the-shelf generator  $\gamma$  and analysis  $\alpha$
- redundant effort
- does not scale well

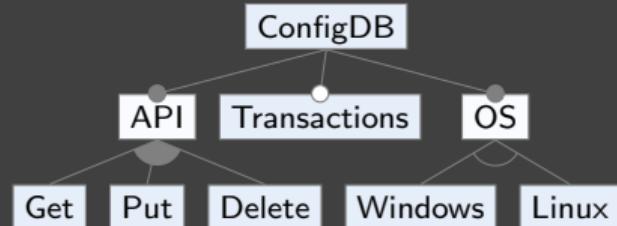
## Feature-Based Strategy

- analyze individual **features**
- + sound, efficient
- analysis  $\alpha$  requires features with interfaces
- incomplete: misses all feature interactions

## Family-Based Strategy

- analyze the **product line**
- + sound, complete, efficient
- requires careful, hand-crafted analysis  $\alpha$

# Pairwise Coverage



*Transactions*  $\rightarrow$  *Put*  $\vee$  *Delete*

## Interactions to Cover

- exclude abstract features (e.g., *API*, *OS*)
- exclude features contained in every configuration (e.g., *C*)
- exclude invalid combinations (e.g., *W*  $\wedge$  *L*)

## Pairwise Interactions

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

## Pairwise Coverage with Six Configurations

{C, P, D, T, W}  
{C, G, D, L}  
{C, G, P, T, L}  
{C, G, W}  
{C, P, W}  
{C, D, T, L}

# Handling Feature Interactions

Solution	Variability	Effort	Size & performance	Quality
S1: Adapt Feature Model	⚡	✓	✓	✓
S2: Orthogonal implementation	?	✓	?	✓
S3: Duplicate implementations	✓	⚡	✓	⚡
S4: Move source code	✓	✓	⚡	⚡
S5: Conditional compilation	✓	✓	✓	⚡⚡
S6: Derivative modules	✓	⚡	✓	✓

# Course Summary – Summary

## Lessons Learned

- how to implement features and variability?
- how to model valid combinations and reason about those?
- how to do quality assurance?
- (how to evolve and maintain a product line?)

## Further Reading

see earlier parts

## Practice

Questions? Feedback?

# FAQ – 12. Evolution and Maintenance

## Lecture 12a

- Why do we need to change product lines?
- How does the Linux kernel evolve over time?
- How can changes to feature models be classified?
- What are advantages of classifying changes to feature models?
- Give an example for a refactoring/-generalization/specialization!
- What is referred to as co-evolution of product lines?
- How do feature model, build scripts, and source code co-evolve?

## Lecture 12b

- What is the difference between evolution and maintenance?
- Which kinds of maintenance and reengineering are there?
- What are examples for product-line maintenance?
- What are examples for adaptive, corrective, perfective, preventive maintenance?
- What are examples for reverse engineering, forward engineering, reengineering?

## Lecture 12c

- What is a software product line?
- When to use which implementation technique for variability?
- How to perform quality assurance for product lines?