COST Action CA22137

2023-2027

ROAR-NET

Software Development for Reuse

1st ROAR-NET Training School – Lecture

Sandra Greiner
greiner@imada.sdu.dk

COST EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Funded by the European Union

Preclaimer

This lecture...

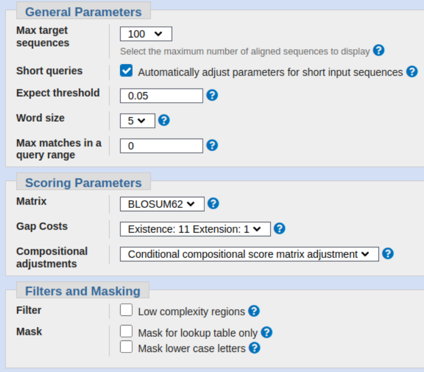
does **NOT** involve mathematics

reuses some material from an open-source course on SPLE by Thomas Thüm, Elias Kuiter, and Timo Kehrer

please be interactive, asks questions throughout the lecture

⇒ learn concepts how to reuse software systematically

Motivation: Complex, Configurable Software



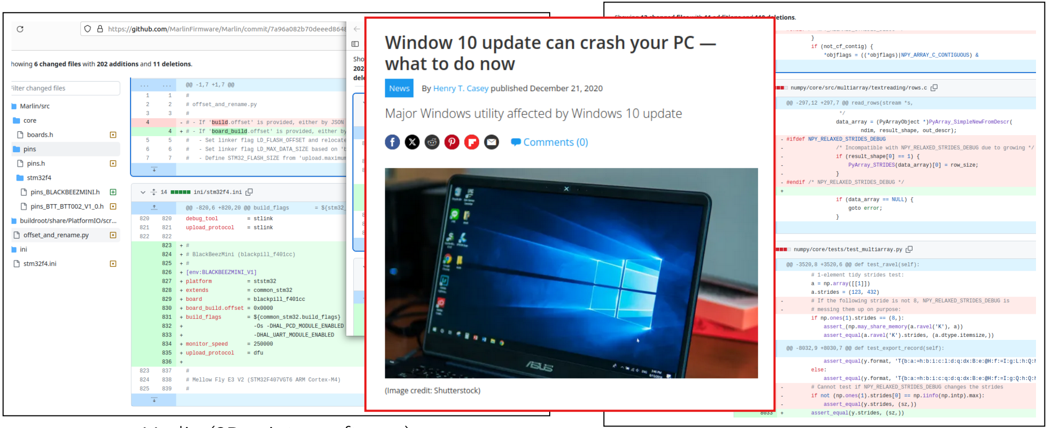
Configuring F´

This guide is a first attempt to describe the various configuration settings in F´. Most users can operate with the default settings, but as the system design is finalized, some of these options may need to be changed such that the system is most efficient.

This guide includes:

- How to Configure F´
- AcConstants.ini
- FpConfig.hpp
  - Type Settings
  - Object Settings
  - Asserts
  - Port Tracing
  - Port Serialization
  - Serialization Type ID
  - Buffer Sizes
  - Text Logging
  - Misc Configuration Settings
- Component Configuration
- Conclusion

Complex, Configurable Software which EVOLVES

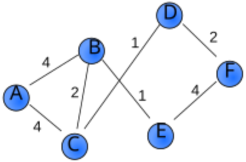


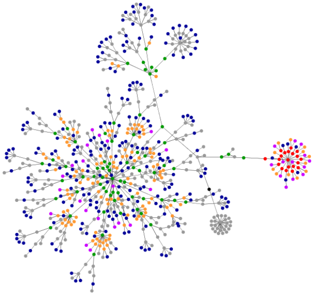
Marlin (3D printer software)

numpy-lib commit: markdown, tests, C-file, …

Graph Lib - Design & Implementation Sketch

Let's build a **Graph** library:





[weighted] [graph2]

What constitutes a Graph?

```
public class Graph { ... }

public class Edge {
  private double weight;
  private Node source;
  private Node target;

  public Edge(...) { ...}
}

public class Node { ... }
public class Color { ... }
```

Copy and Adapting Programs (= Software Variants)

```
public class Graph { ... }

public class Edge {
  private double weight;
  private Node source;
  private Node target;

  public Edge(...) { ...}
}

public class Node { ... }
public class Color { ... }
```

Weighted, Directed, Colored Graph

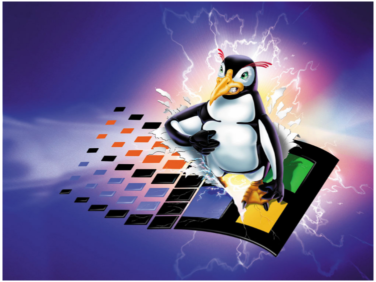
```
public class Graph { ... }
public class Edge {
  private Node[] nodes = new Node[2];

  public Edge(...) { ...}
}
```

```
ublic class Graph { ... }
ublic class Edge {
 private Node source;
 private Node target;

 public Edge(...) { ...}
}

public class Node { ... }
```

Directed Graph

works and pays off for few variants
increasingly complex with evolution
how can we do better?

Complex Configurable Software, which EVOLVES



use **abstractions and organized reuse**

[linux] [wrangling]

Concepts to support organized reuse

(and introduce software product line engineering)

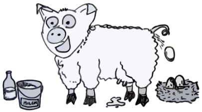
Mass Production

Mass Production

- result of industrialization
- goods produced from standardized parts
- → reduced cost, increased productivity, improved quality
- but: (almost) no individualized product

Example Principle: One Size Fits All

- e.g., swiss-army knife



Mass Produced Software?

→ software satisfying the needs of most customers

→ but then, customers

- miss desired functionality
- overwhelmed with not needed functionality (e.g., contemporary office or graphics programs).

Often this generality makes software complex, slow, and buggy.

[wollmilchsau] [army knife]

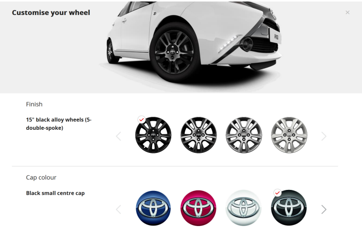
Mass Customization

Mass Customization

= mass production + customization

individual, customized goods at cost similar to mass production

Example: Car Configuration



Car Production

Other Domains

computers and laptops, electronics, food, medicine, clothing, bikes ..., software?

Mass Customization for Software?

Mass Customization for Software?

- customization: individually developed software products
- 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, or laws

Goal:

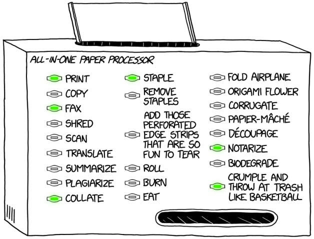
avoid expensive customization

how is software normally developed?

What is a Feature?

Feature

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



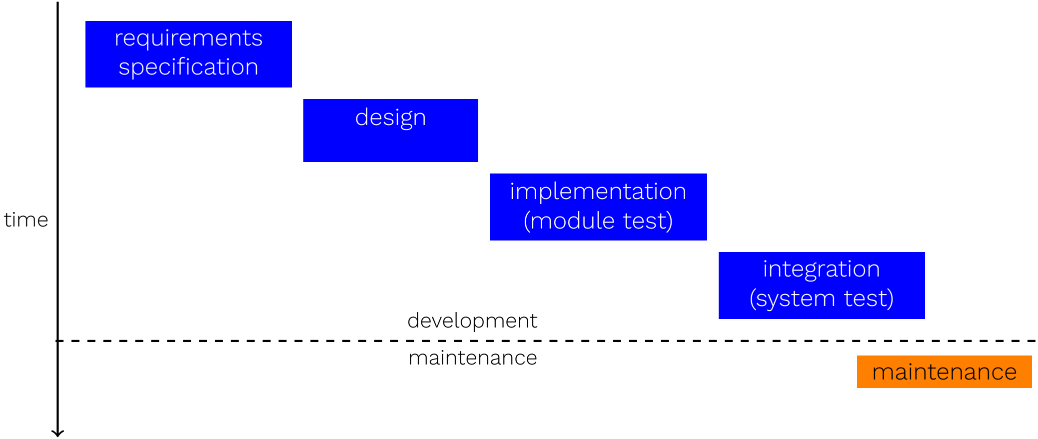
Features in Software Product Lines

In SPLE, features

specify and communicate commonalities and differences of products between stakeholders

guide the structure, reuse, and variation throughout software life cycle.

Recap: The Software Life Cycle



What is a Product?

Product

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.

Terminology

here: *product* == *product variant* == *variant*

software product: a product only of software

Note
software is more than source code: e.g., requirements, models, source code, tests, documentation; we focus on source code

What is a Domain?

ROAR NET

Domain

A domain is an area of knowledge:

- scoped to maximally satisfy its stakeholders' requirements,
- including a set of concepts and terminology understood by practitioners in that area
- including the knowledge of how to build (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 Lines

A **software product line** is

a set of software-intensive systems that (aka. products or variants)

share a common, managed set of features
(common set; not all products have all features in common)

satisfy specific needs of a particular market segment or mission (aka. domain)

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

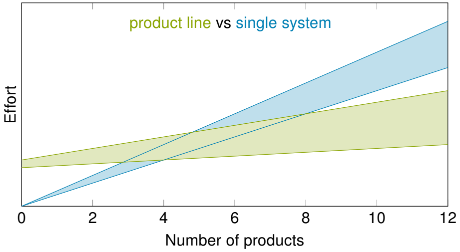
Software product-line engineering is a paradigm to develop software applications (software-intensive systems and software products) using software platforms and mass customization.

Promises of Product Lines

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

Idea of Product-Line Engineering

Reduced effort per product resulting from up-front investment into a product line:



Single-System Engineering

classical software development (of single product);
not considered as product-line engineering

Summary - Introduction

Lessons Learned

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

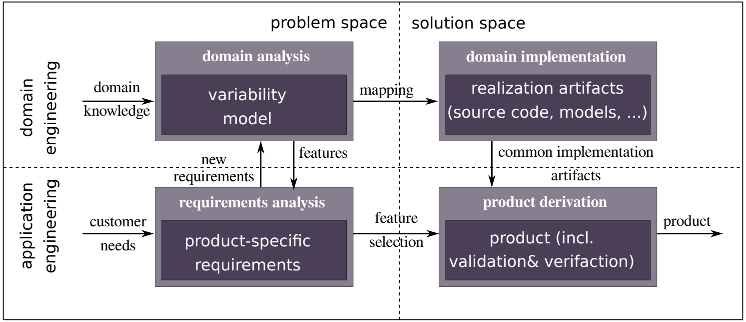
Interaction

- Which examples of (software) product lines are you aware of?
- Where have you experienced copying and pasting source code and thought does that work more systematically?
- Exemplify the difference between feature, product, domain, and product line

ROAR NET

Software Product Line Engineering Process

Software Product Line Engineering Process



[Greiner 2022] based on [Apel 2013]

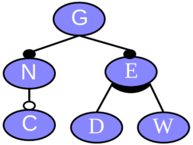
principles of *organized reuse* and *variability*;
implemented in a shared *platform*

feature model to describe the configuration space
variability mechanism to realize the platform

SPLE Process: Graph Library contd.

```
public class Graph { ... }

public class Edge {
  private double weight;
  private Node source;
  private Node target;

  public Edge(...) { ...}
}

public class Node { ... }
public class Color { ... }
```

superimposition of variants



feature model

```
public class Graph { ... }
public class Edge {
  // #ifdef Weighted
  private double weight;
  // #endif
  // #ifdef Directed
  private Node source;
  private Node target;
  // #elif
  private Node[] nodes = new Node[2];
  // #endif

  public Edge(...) { ...}
}

public class Node { ... }
// #ifdef Colored
public class Color { ... }
// #endif
```

annotated superimposition of variants

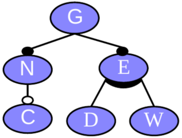
Product Derivation

```
public class Graph { ... }
public class Edge {
  // #ifdef Weighted
  private double weight;
  // #endif
  // #ifdef Directed
  private Node source;
  private Node target;
  // #elif
  private Node[] nodes = new Node[2];
  // #endif

  public Edge(...) { ...}
}

public class Node { ... }
// #ifdef Colored
public class Color { ... }
// #endif
```

annotated superimposition

```
G=true
N=true
E=true
C=false
W=true
D=false
```

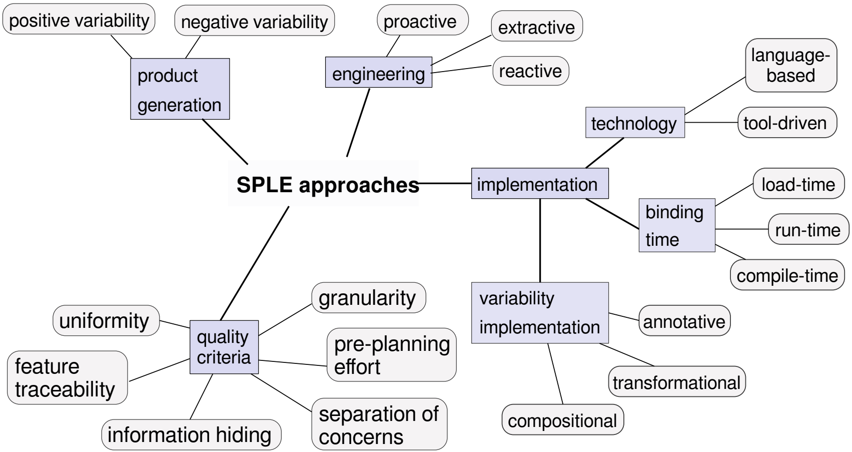
filter




```
public class Graph { ... }
public class Edge {
  private double weight;
  private Node[] nodes = new Node[2];

  public Edge(...) { ...}
}

public class Node { ... }
```

single variant (product)

Classification



based on [Greiner2022]

Classification

Implementation Criteria:

binding time: runtime vs compile-time vs load-time
technology: tool-driven, language-based
technique: annotative, transformational, compositional

Quality criteria:

granularity
uniformity
feature traceability
information hiding
separation of concerns
pre-planning effort

Runtime Variability

(Boolean) Variables

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

```
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;
  }
  ...
}
```

Idea:
parameters control configuration options

global parameters vs. immutable global parameters

if immutable, no possibility to change after delivery

'modern' approach: feature toggles (e.g. with Kubernetes)

Method Parameters

```
public class Graph {
  boolean weighted;
  boolean colored;
  ...
  Graph(boolean _weighted, boolean _colored) {
    weighted = _weighted;
    colored = _colored;
  }

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

```
public class Edge {
  boolean weighted;
  Weight weight;
  ...
  Edge(Node _a, Node _b, boolean weighted) {
    a = _a; b = _b;
    if (weighted) { weight = new Weight(); }
  }
  ...
}
```

expose parameters in methods of interface

parameter values passed through method invocations

benefit different instances within same code (e.g., graph with and without colors)

drawback code smell (methods with many parameters)

(Object-Oriented Design Patterns)

ideal for reuse and composition:

`Template method`, `Decorator`, `Abstract Factory`

Extension through delegation vs. inheritance

Limitations and drawbacks w.r.t. feature combinations

Compile-time/Build-time Variability
–
coarse-grained

Build Systems - Example of KConfig

Feature Model with KConfig [linux/arch/x86/Kconfig]

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

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

KBuild [kernel.org]

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

Feature Mapping with KBuild [linux/arch/x86/Kbuild]

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

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

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

Recurse into Subsystems [linux/arch/x86/ia32/Makefile]

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

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

[SPL lecture, chapter 5, slide 13]

Build Systems for SPLs

How to?

1. model variability in feature model
2. in- and exclude files based on feature selection
3. provide feature selection at build time

Benefit

compile-time variability
→ fast, small binaries

free feature selection
→ products generated automatically

in- and exclusion of entire files or subsystems
→ coarse-grained modularity

Drawback

hard to reconfigure at load- or runtime

complexity of build scripts
→ hard to comprehend and analyze

coarse-grained: no line-wise in- or exclusion of source code

Software Modularization

Modularization:
application of information hiding and data encapsulation to enable

strong, logical connection of inner parts

precisely defined interfaces

Cohesion and Coupling

Cohesion: measures how strong parts of a module work together (intra communication)

Coupling: measures the complexity of communication between modules

→ aim for low coupling and strong cohesion

Reasons for Modules

Promises of Modular Implementations

- independent development of other modules
- easier maintenance (locality of changes)
- hide complexity and ease comprehension
- stability and reliability through data encapsulation

Typical techniques:

- components
- **(micro-)services**
- **frameworks**

(Micro)services

(Micro-)Services are “implemented and operated as a *small yet independent* system, offering access to its internal logic and data through a *well-defined network interface*.” [Jamshidi2018]

- architectural pattern based on distributed programming
- ”cohesive, independent process interacting via messages” [Dragoni2017]
- essential: inter-process communication (e.g. through REST API)
- inside a microservice different technology stack possible
- each service must be manageable by a small team

Promises of Microservices

Scalability: small enough to be developed by a small, agile team.

Continuous integration/deployment: deployed independently of each other.

Heterogeneity: each implemented using its own technology stack.

Fault tolerance: crash of single microservice should not lead to crash of entire system.

Efficiency: Optimized configuration of execution environment for each microservice.

Modernization: easy replacement by alternative microservice (even re-implemented from scratch).

Microservices in Action -- Jolie

Tailored for microservices and APIs

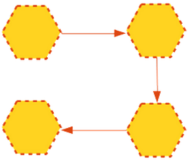
Jolie is a **contract-first** programming language, which puts API design at the forefront. It supports both synchronous and asynchronous communication. Data models are defined by types that support **refinement** (in red on the right), and DTO (Data Transfer Objects) transformations are transparently managed by the interpreter.

```
type GetProfileRequestType {
        id:int
}
type GetProfileResponseType {
        name:string
        surname:string
        email:string( regex(".*@.*\\..*") )
        accounts[0,*] {
                nickname:string
                service_url:string
                enabled:bool
        }
        ranking:int( ranges( [1,5] ) )
}

type SendMessageRequestType {
        id:int
        message:string( length( [0,250] ) )
}

interface ProfileInterface {
requestResponse: // Synchronous RPC
        getProfile( GetProfileRequestType )( GetProfileResponseType )

oneWay: // Asynchronous
        sendMessage( SendMessageRequestType )
}
```

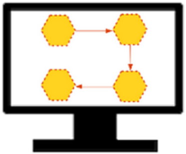
[Jolie Website]

Microservices in Action -- Jolie

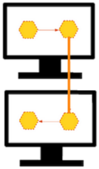
Program your microservice system



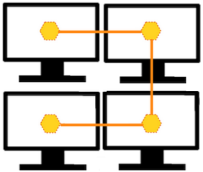
Deploy it in a single machine



Deploy it in two different machines



Deploy it in four different machines



[Jolie Website]

Microservices for SPLs

Idea:

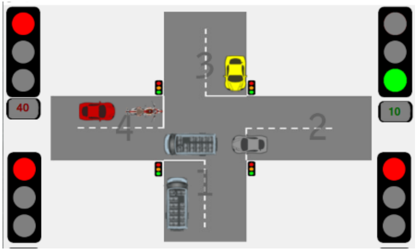
each feature is a service

feature selection determines how to compose the services

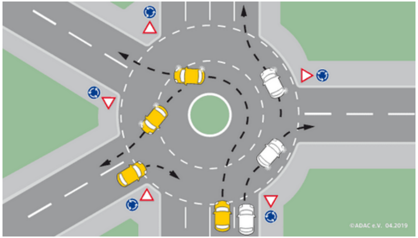
benefit: 'standardized' service composition (does not need glue code like for library composition)

Microservices Composition

Orchestration describes an executable (centralized) process combining services



Choreography each service describes own task within composition



Framework Principles

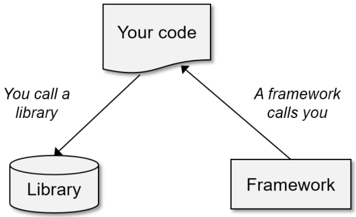
Framework and Hotspots

framework = set of classes with an abstract design
support reuse beyond class-level
open for extension at hotspots (=interface or abstract class)

Plugin:

extends hot spots with custom behavior
compiled and deployed separately

Inversion of Control



→ hollywood principle (we call you)
⇒ pre-planning required

Plug-In Loading and Management

Simple Example vs. Reality

typical requirements in practice:

- arbitrarily many plug-ins to be registered at one extension point
- single plug-in may extend several extension points
- plug-in may add new extension points to the framework (framework of frameworks)
- plug-in implementation provided by third parties

Plug-In Loader

- searches for plugin files in dedicated directory
- tests whether file implements a plugin
- checks dependencies
- initializes plug-ins

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

Frameworks for SPL

Idea

each plugin implements a feature

feature selection determines which plugins to load and register

Benefit

no glue code needed

no service composition needed

Drawback

preplanning necessary

coarse-grained

Frameworks in Action

```java
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);
    }
  }
  // ...
}
```

[SPL lecture, Chapter 6, slide 39]

```java
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);
}
```

```java
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
  }
}
```

Points to Consider

many empty methods in `ColorPlugin`
all plugins considered by registry before printing

General challenge: cross-cutting concerns
if implemented as plugin

- large interfaces with many irrelevant parts
- large communication overhead between plugins and framework

Preplanning problem

- if not familiar with domain: do we know color and weight needed in plugin interface?
- hard identification and anticipation of relevant hot spots
- → excellent domain knowledge and expertise required

Summary

Services:

small, exchangeable units

dedicated composition mechanism required

information hiding

coarse-grained, but adhoc variability

Frameworks

large units

composition mechanism out-of-the box

information hiding

coarse-grained

pre-planning (no ad-hoc variability)

Reflection

If you had to implement a flexible API for randomized optimization algorithms which variability mechanism would you use?

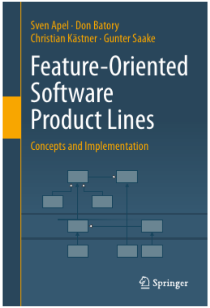
What are optional and what are mandatory features?

Summary of Approaches

variability approach	binding time: compile time	binding time: loadtime / runtime	technology: language	technology: tool	representation: annotation	representation: composition
runtime parameter		x	x		x	
design patterns		x	x			x
build systems	x			x		x
services	x	x	x			x

binding time: when is the variability fixed to realize a specific product?
technology: is the variability mechanism enabled by the language or only by the tool?
representation: how are the features represented, where are they located?

If you'd like to learn more about SPLE or SE



[Apel2013]

check out the

- entire opensource SPL lecture
- SPL community activities @ SPLC net
- FOSD community activities @ FOSD website

reach out:
greiner@imada.sdu.dk

References

[Apel2013] Sven Apel, Don S. Batory, Christian Kästner, Gunter Saake, 2013
Feature-Oriented Software Product Lines - Concepts and Implementation. Springer 2013, ISBN 978-3-642-37520-0, pp. I-XVI, 1-315

[Greiner2022] Sandra Greiner, 2022
Reuse of Model Transformations for Propagating Variability Annotations in Annotative Software Product Lines. University of Bayreuth, Germany

[Jamshidi2018] P Jamshidi, C Pahl, NC Mendonça, J Lewis, S Tilkov
Microservices: The journey so far and challenges ahead. IEEE Software 35 (3), 24-35

[Dragoni2017] N Dragoni, S Giallorenzo, AL Lafuente, M Mazzara, F Montesi, R Mustafin, L. Safina
Microservices: Yesterday, Today, and Tomorrow.

Acknowledgments

This presentation is based upon work from COST Action Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, supported by COST (European Cooperation in Science and Technology).

COST (European Cooperation in Science and Technology) is a funding agency for research and innovation networks. Our Actions help connect research initiatives across Europe and enable scientists to grow their ideas by sharing them with their peers. This boosts their research, career and innovation.

cost
EUROPEAN COOPERATION
IN SCIENCE & TECHNOLOGY

Funded by
the European Union