8. Development Process – Handout

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

Technische Universität Braunschweig

UNIVERSITÄT BERN

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

8. Development Process

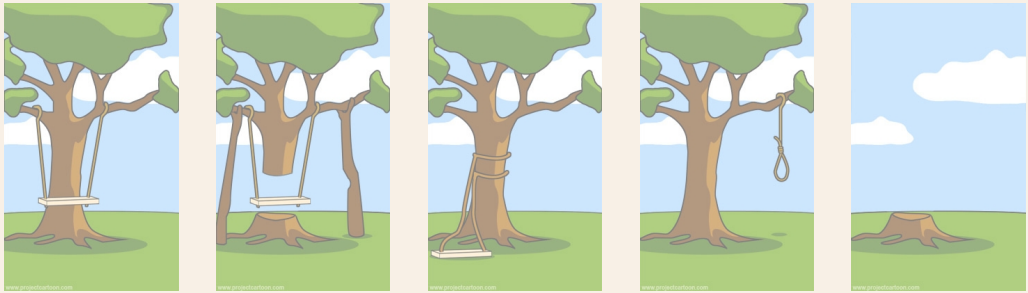
8a. Process Model for Product Lines

8b. Implementation of Product Lines

8c. Adoption of Product Lines

Recap: Process Models

Recap: The Software Life Cycle



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

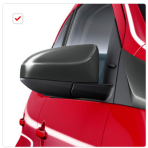
Mirror cover, Cyan Splash, left side
£22.00
More info

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)

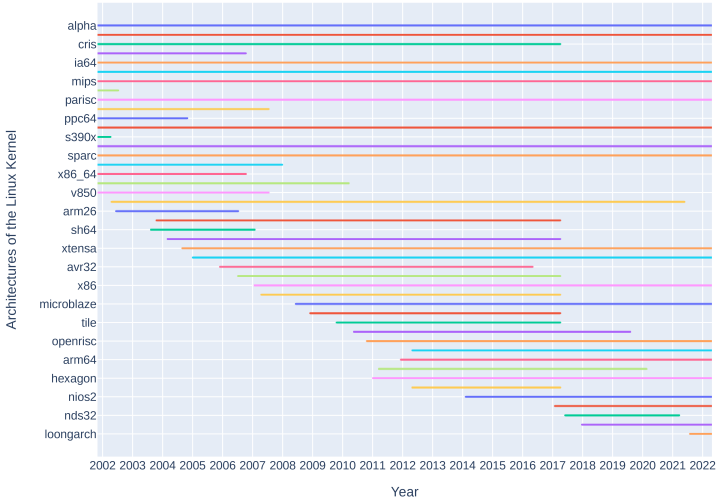
Retail price	**£10,910.00**	
Red Pop		
Gleam fabric		
15" steel wheels with wheel caps (6-spoke)		
Mirror cover, Bold Black, right side	£22.00	×
Mirror cover, Bold Black, left side	£22.00	×
Mirror cover, Cyan Splash, left side	£22.00	×
Mirror cover, Cyan Splash, right side	£22.00	×
Mirror cover, Dark 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



Linux Kernel Architectures

- in Linux, $\approx$ 20 CPU architectures are carefully maintained in 2023
- to keep it manageable, obsolete architectures are regurlarly 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]

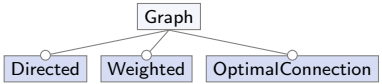
"The **problem space** takes the perspective of stakeholders and their problems, requirements, and views of the entire domain and individual products. Features are, in fact, domain abstractions that characterize the problem space."

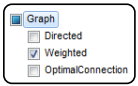
[Lecture 4]

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
//#endif
    ) && 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

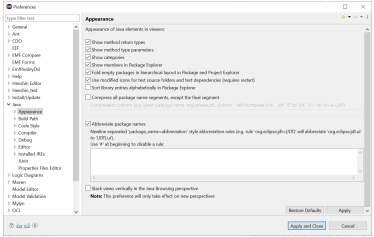
```
public class Config {
  public static boolean COLORED = true;
  public static boolean WEIGHTED = false;
}
```

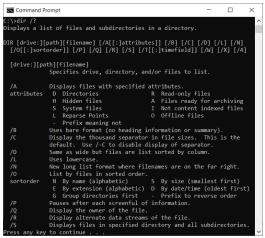
```
class Graph {
  ...
  Edge add(Node n, Node m) {
    Edge e = new Edge(n, m);
    if (Config.WEIGHTED) { e.weight = new Weight(); }
    nodes.add(n); nodes.add(m); edges.add(e);
    return e;
  }
  Edge add(Node n, Node m, Weight w) {
    if (!Config.WEIGHTED) {
      throw new RuntimeException();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

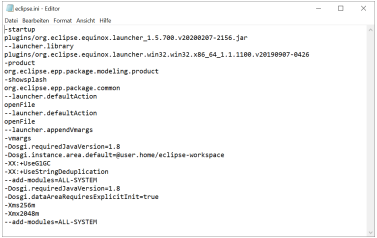
```
class Node {
  Color color; ...
  Node() {
    if (Config.COLORED) { color = new Color(); }
  }
  void print() {
    if (Config.COLORED) {
      Color.setDisplayColor(color);
    }
    System.out.print(id);
  }
}
```

```
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







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:

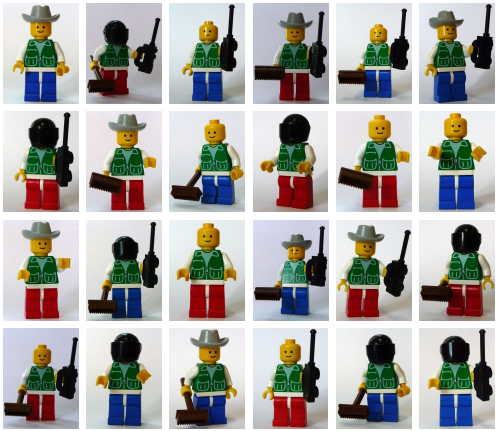
- Variable parts are always delivered
- Not well-suited for compile-time binding

Recap: Clone-and-Own

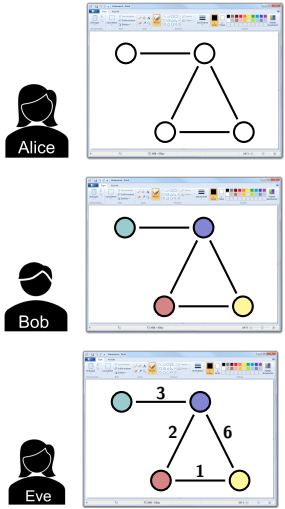
Clone-and-Own

- 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



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?

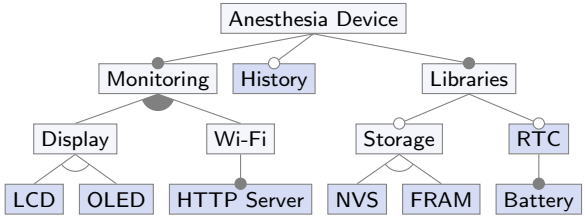
- compile-time variability only for implemented products
- no automated generation: for clone-and-own (with version control systems)
- automated generation based on build script and extra files: for clone-and-own with build systems
- no free feature selection (i.e., configuration)

Recap: Conditional Compilation – Build Systems [Kuiter et al. 2021]

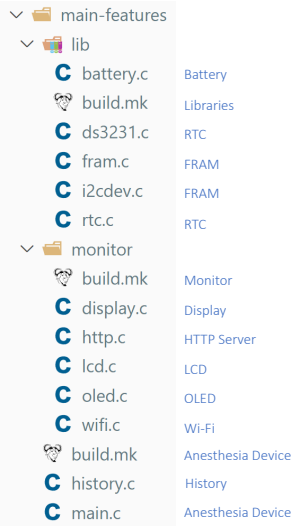
How to Implement Features with Build Systems?

- 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

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



$History \rightarrow Storage \wedge RTC$



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 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

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

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

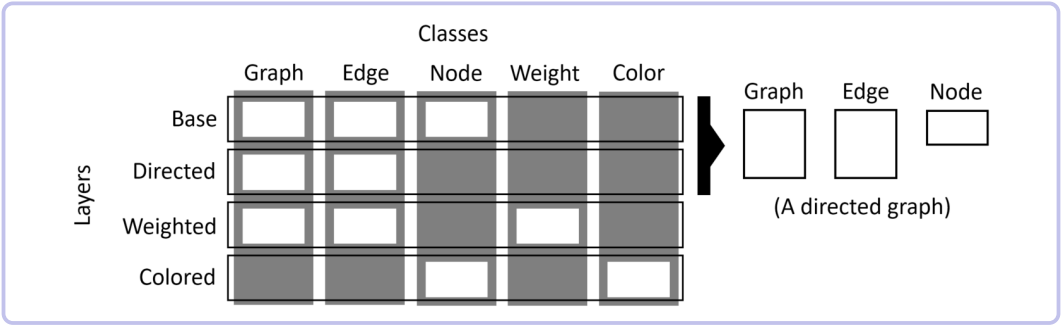
Recap: Languages for Features – Feature-Oriented Programming

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

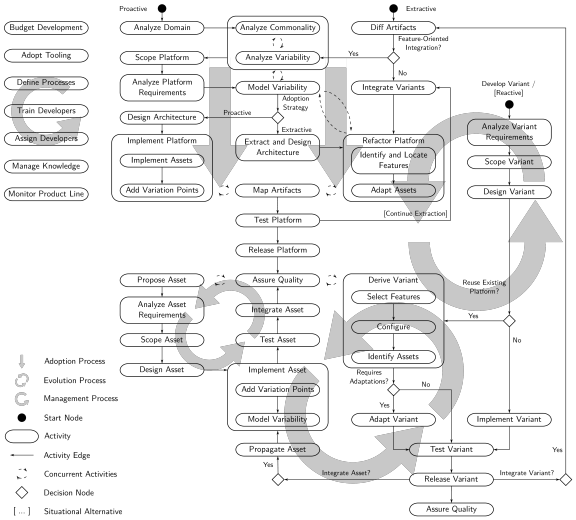
Advantages

- less up-front investment that 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?