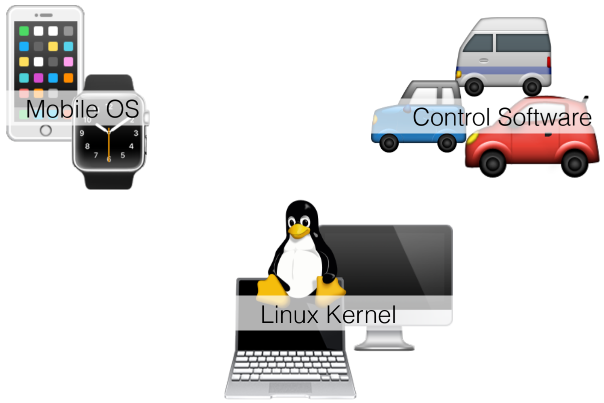
Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Software Product Line Engineering

based on slides created by Sarah Nadi

Examples of Software Product Lines



Software product lines are ubiquitous!

Resources

Software Product Lines

Software Engineering Institute
Carnegie Mellon University

"A software product line (SPL) 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."

Advantages of SPLs

- Tailor-made software
- Reduced cost
- Improved quality
- Reduced time to market

SPLs are ubiquitous

Challenges of SPLs

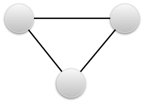
- Upfront cost for preparing reusable parts
- Deciding which products you can produce early on
- Thinking about multiple products at the same time
- Managing/testing/analyzing multiple products

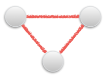
Feature-oriented SPLs

Thinking of your product line in terms of the features offered.

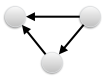
Examples of a Feature

(Graph Product Line)

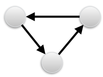




feature:
edge color



feature:
edge type
(directed vs. undirected)



feature:
cycle detection

Examples of a Feature
(*Collections* Product Line)

- Serializable
- Cloneable
- Growable/Shrinkable/Subtractable/Clearable
- Traversable/Iterable
- Supports parallel processing

Feature

A feature is a *characteristic or end-user-visible behavior of a software system*. 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 features would a Smartphone SPL contain?

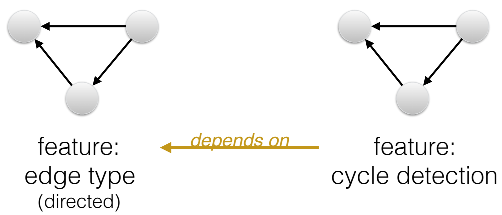
Discussion

- Integrated hardware (e.g., size and resolution of the display, network connections support (Bluetooth 4.x), Wireless 802.11abcg…, amount of memory, storage capacity)

- Integrated software
  (Product differentiation in the smartphone market is (also) done purely based on software features.)

Feature Dependencies

Constraints on the possible feature selections!

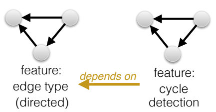


Product

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.

Valid Products

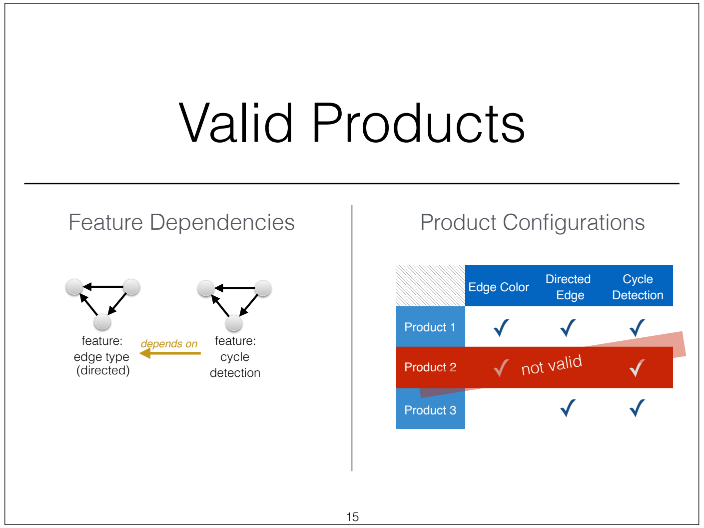
Feature Dependencies



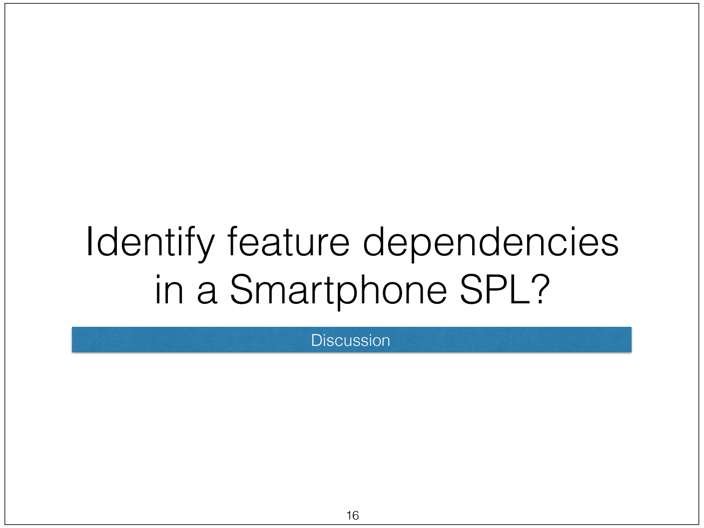
Product Configurations

	Edge Color	Directed Edge	Cycle Detection
Product 1	√	√	√
Product 2	√		√
Product 3		√	√

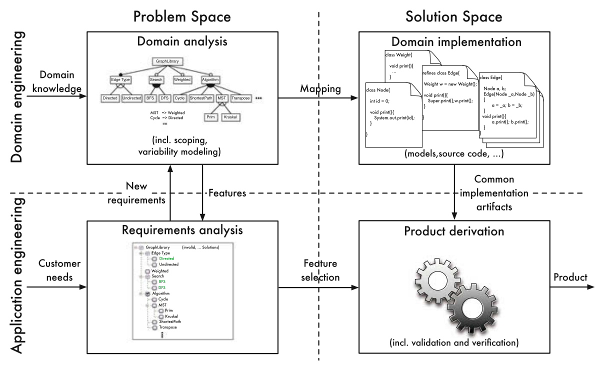
Ask yourself which product is (in)valid?



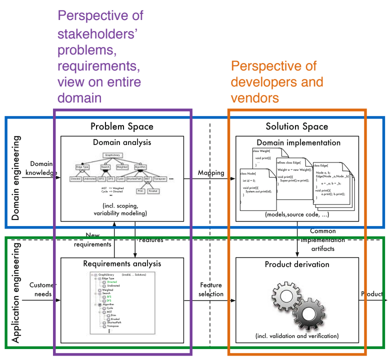
The dependency constraint is not satisfied by product 2.



Software Product Line Engineering



Software Product Line Engineering



Development for reuse

- Analyze domain & develop reusable artifacts
- Does not result in a specific product
- Prepares artifacts to be used in various products

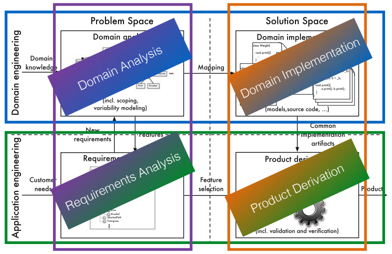
Development with reuse

- Develop specific product for needs of a particular customer
- Repeated for every derived product

Software Product Line Engineering

Perspective of stakeholders' problems, requirements, view on entire domain

Perspective of developers and vendors



Development for reuse

- Analyze domain & develop reusable artifacts
- Does not result in a specific product
- Prepares artifacts to be used in various products

Development with reuse

- Develop specific product for needs of a particular customer
- Repeated for every derived product

Domain Analysis

Domain Analysis

- Domain scoping
  Deciding on product line's extent or range
- Domain modeling
  - Captures & documents the commonalities & variabilities of the scoped domain
  - Often captured in a feature model

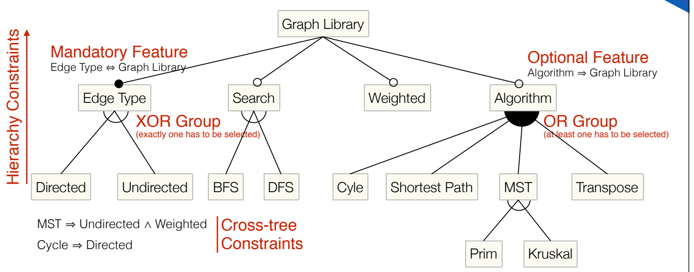
Feature Model

Domain Analysis

- Document the features of a product line & their relationships
- Can be translated into propositional logic

Graph Library
Feature Model

Domain Analysis



MST = Minimum Spanning Tree

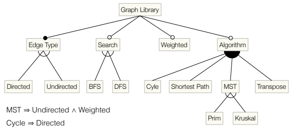
Hierarchal Relationships: Parent/child relationship - Child cannot be selected unless parent is selected

Graph Library

Feature Model in Propositional Logic

Domain Analysis

$\quad$ root(GraphLibrary)
∧ mandatory(GraphLibrary,EdgeType)
∧ optional(GraphLibrary,Search)
∧ optional(GraphLibrary,Weighted)
∧ or(Search,{BFS,DFS})
…
∧ alternative(MST,{Prim,Kruskal})
∧ (MST ⇒ Weighted)
∧ (Cycle ⇒ Directed)
∧ (· · ·)

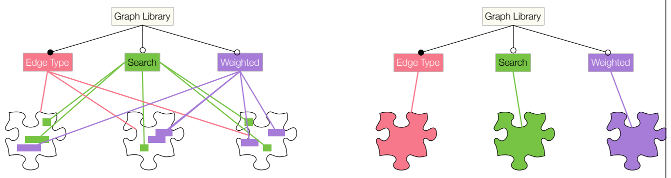


Domain Implementation

Domain Implementation

- Underlying code must be variable
- Dimensions of implementation techniques
  - Binding times: compile-time binding, load-time binding and run-time binding.
  - Representation: annotation vs composition

Domain Implementation
(Representation)



Annotation-based Approach

Composition-based Approach

Variability Implementation

- Parameters
- Design patterns
- Build systems
- Preprocessors
- *Feature-oriented programming*

- Parameters - binding time: runtime
- Design Patterns - binding time: compile-time/run-time

Variability Implementation Parameters

★ simple

★ flexible

★ language support

- code bloat
- computing overhead
- non-modular solution

annotation-based; binding time: run time

Domain Implementation

Variability Implementation Design Patterns

★ well established

★ easy to communicate design decisions

- architecture overhead
- need to preplan extensions

composition; binding time: run time

Variability Implementation Build Systems

Domain Implementation

- ★ simple if features can be mapped into files
- ★ can control other types of parameters
- \- code duplication if finer level of granularity needed
- \- hard to analyze

(Here, the build-script is extended to model the variability!)
annotation (in the build-script); binding time: compile time

Variability Implementation Preprocessors

Domain Implementation

- ★ Easy to use, well-known
- ★ Compile-time customization removes unnecessary code
- ★ Supports arbitrary levels of granularity
- \- No separation of concerns (lots of scattering & tangling)
- \- Can be used in an undisciplined fashion
- \- Prone to simple (syntactic) errors

annotation; binding time: compile time

Variability Implementation Feature-Oriented Programming

Domain Implementation

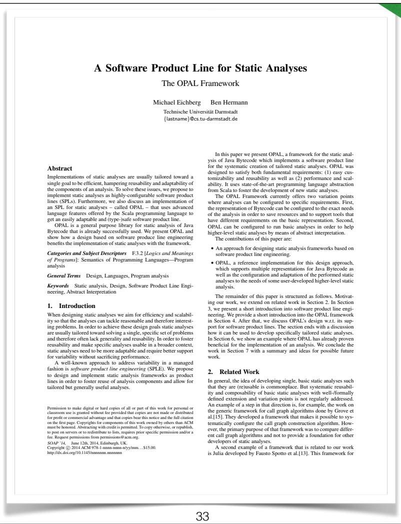
- ★ easy-to-use language mechanism, requiring minimal language extensions
- ★ compile-time customization of source code
- ★ direct feature traceability from a feature to its implementation
- requires composition tools
- granularity at level of methods
- *only academic tools so far, little experience in practice*

composition; binding time: compile time

Research Topics

- feature-model reengineering/extraction from existing code
- detecting inconsistencies between the feature-model and its "implementation"
- feature interactions - intended vs. unintended?

composition; binding time: compile time



A Software Product Line for Static Analyses

The OPAL Framework

Michael Eichberg Ben Hermann
Technische Universität Darmstadt
{lastname}@cs.tu-darmstadt.de

Abstract

Implementations of static analyses are usually tailored toward a single goal to be efficient, hampering reusability and adaptability of the components of an analysis. To solve these issues, we propose to implement static analyses as highly-configurable software product lines (SPLs). Furthermore, we also discuss an implementation of an SPL for static analyses – called OPAL – that uses advanced language features offered by the Scala programming language to get an easily adaptable and (type-)safe software product line.

OPAL is a general purpose library for static analysis of Java Bytecode that is already successfully used. We present OPAL and show how a design based on software produce line engineering benefits the implementation of static analyses with the framework.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Design, Languages, Program analysis

Keywords Static analysis, Design, Software Product Line Engineering, Abstract Interpretation

1. Introduction

When designing static analyses we aim for efficiency and scalability so that the analyses can tackle reasonable and therefore interesting problems. In order to achieve these design goals static analyses are usually tailored toward solving a single, specific set of problems and therefore often lack generality and reusability. In order to foster reusability and make specific analyses usable in a broader context, static analyses need to be more adaptable and require better support for variability without sacrificing performance.

A well-known approach to address variability in a managed fashion is *software product line engineering* (SPLE). We propose to design and implement static analysis frameworks as product lines in order to foster reuse of analysis components and allow for tailored but generally useful analyses.

SOAP '14, June 12th, 2014, Edinburgh, UK.
Copyright © 2014 ACM 978-1-nnnn-nnnn-n/yy/mm…$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper we present OPAL, a framework for the static analysis of Java Bytecode which implements a software product line for the systematic creation of tailored static analyses. OPAL was designed to satisfy both fundamental requirements: (1) easy customizability and reusability as well as (2) performance and scalability. It uses state-of-the-art programming language abstraction from Scala to foster the development of new static analyses.

The OPAL Framework currently offers two variation points where analyses can be configured to specific requirements. First, the representation of Bytecode can be configured to the exact needs of the analysis in order to save resources and to support tools that have different requirements on the basic representation. Second, OPAL can be configured to run basic analyses in order to help higher-level static analyses by means of abstract interpretation.

The contributions of this paper are:

- An approach for designing static analysis frameworks based on software product line engineering.
- OPAL, a reference implementation for this design approach, which supports multiple representations for Java Bytecode as well as the configuration and adaptation of the performed static analyses to the needs of some user-developed higher-level static analysis.

The remainder of this paper is structured as follows. Motivating our work, we extend on related work in Section 2. In Section 3, we present a short introduction into software product line engineering. We provide a short introduction into the OPAL framework in Section 4. After that, we discuss OPAL's design w.r.t. its support for software product lines. The section ends with a discussion how it can be used to develop specifically tailored static analyses. In Section 6, we show an example where OPAL has already proven beneficial for the implementation of an analysis. We conclude the work in Section 7 with a summary and ideas for possible future work.

2. Related Work

In general, the idea of developing single, basic static analyses such that they are (re)usable is commonplace. But systematic reusability and composability of basic static analyses with well-/formally defined extension and variation points is not regularly addressed. An example of a step in that direction is, for example, the work on the generic framework for call graph algorithms done by Grove et al.[15]. They developed a framework that makes it possible to systematically configure the call graph construction algorithm. However, the primary purpose of that framework was to compare different call graph algorithms and not to provide a foundation for other developers of static analyses.

A second example of a framework that is related to our work is Julia developed by Fausto Spotto et al.[13]. This framework for

Case Study

Michael Eichberg and Ben Hermann
SOAP'14; Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis
ACM 2014

A Software Product Line for Static Analyses

Case Study

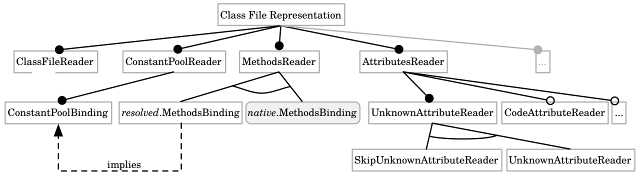
- Commonalities
  - we need to be able to process .class files
- Variability
  - enable different representation for .class files (e.g., if you want to write a disassembler a 1:1 representation is needed; for most static analyses a more abstract representation is required.)
  - only reify those parts that are needed

Requirement: composition based approach

binding time: compile-time

Case Study

Processing Java .class Files

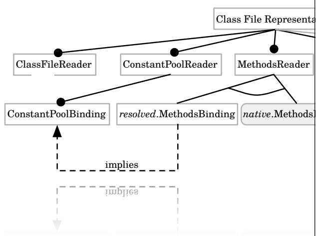


Case Study

Processing Java.class Files

Base Trait which defines the general infrastructure.

```
trait ClassFileReader{
  /* Abstract over the representation of the ... */
  type ClassFile
  type Constant_Pool
  type Fields
  type Methods
  type Attributes
  ...
  /* Methods to read in the respective data structures. */
  def Constant_Pool(in: DataInputStream): Constant_Pool
  def Fields(in: DataInputStream, cp: Constant_Pool): Fields
  def Methods(in: DataInputStream, cp: Constant_Pool): Methods
   ...
  /* Factory method to create a representation of a Class File. */
  def ClassFile(
   ... // Version information, defined type, etc.
   fields: Fields,
   methods: Methods,
   attributes: Attributes)(implicit cp: Constant_Pool): ClassFile

  def ClassFile(in: DataInputStream): ClassFile = {
   // read magic and version information
   val cp = Constant_Pool(in)
   val fields = Fields(in,cp)
   val methods = Methods(in,cp)
   val attributes = Attributes(in,cp)
   // call factory method
   ClassFile(...,fields,methods,attributes)(cp)
  }
}
```



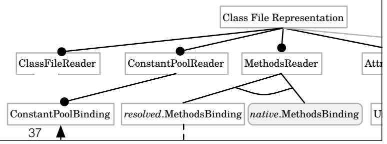
Processing Java.`class` Files

Case Study

Trait which implements the MethodsReader feature!

```
trait MethodsBinding extends MethodsReader {
  this: ConstantPoolBinding with AttributeBinding =>

  type Method_Info = de.tud.cs.st.bat.resolved.Method
  def Method_Info(
      accessFlags: Int,
      name: Int,
      descriptor: Int,
      attributes: Attributes
  )(implicit cp: Constant_Pool): Method_Info =
      create Method representation
}
```

reified cross-tree constraint

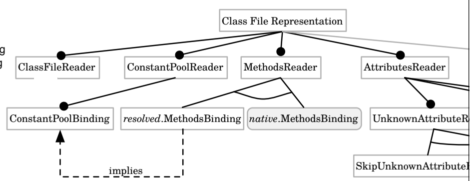


Processing Java.`class` Files

Case Study

Product configurations

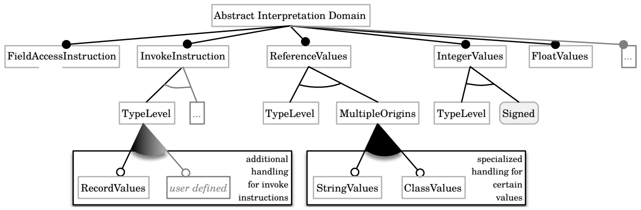
```
class Java7ClassFilesPublicInterface
    extends ClassFileBinding
    with ConstantPoolBinding
    with FieldsBinding
    with MethodsBinding
    with AttributesReader
    with SkipUnknown_attributeReader
    with AnnotationsBinding
    with InnerClasses_attributeBinding
    with InterfacesBinding
    // further attributes related to a class' public interface

class Java7ClassFiles
    extends Java7ClassFilesPublicInterface
    with CodeAttributeBinding
    with StackMapTable_attributeBinding
    with LineNumberTable_attributeBinding
    with LocalVariableTable_attributeBinding
    with BootstrapMethods_attributeBinding
    // further code related attributes
```

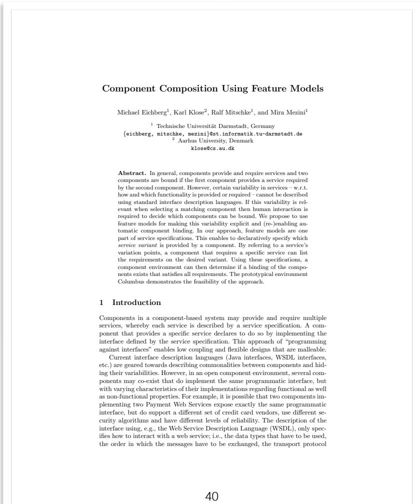


Case Study

Analyzing Methods

(Implemented using a second product line; which supports several products of the first product line.)





Component Composition Using Feature Models

Michael Eichberg[1], Karl Klose[2], Ralf Mitschke[1], and Mira Mezini[1]

[1] Technische Universität Darmstadt, Germany
{eichberg, mitschke, mezini}@st.informatik.tu-darmstadt.de
[2] Aarhus University, Denmark
klose@cs.au.dk

Abstract. In general, components provide and require services and two components are bound if the first component provides a service required by the second component. However, certain variability in services – w.r.t. how and which functionality is provided or required – cannot be described using standard interface description languages. If this variability is relevant when selecting a matching component then human interaction is required to decide which components can be bound. We propose to use feature models for making this variability explicit and (re-)enabling automatic component binding. In our approach, feature models are one part of service specifications. This enables to declaratively specify which *service variant* is provided by a component. By referring to a service's variation points, a component that requires a specific service can list the requirements on the desired variant. Using these specifications, a component environment can then determine if a binding of the components exists that satisfies all requirements. The prototypical environment Columbus demonstrates the feasibility of the approach.

1 Introduction

Components in a component-based system may provide and require multiple services, whereby each service is described by a service specification. A component that provides a specific service declares to do so by implementing the interface defined by the service specification. This approach of "programming against interfaces" enables low coupling and flexible designs that are malleable.

Current interface description languages (Java interfaces, WSDL interfaces, etc.) are geared towards describing commonalities between components and hiding their variabilities. However, in an open component environment, several components may co-exist that do implement the same programmatic interface, but with varying characteristics of their implementations regarding functional as well as non-functional properties. For example, it is possible that two components implementing two Payment Web Services expose exactly the same programmatic interface, but do support a different set of credit card vendors, use different security algorithms and have different levels of reliability. The description of the interface using, e.g., the Web Service Description Language (WSDL), only specifies how to interact with a web service; i.e., the data types that have to be used, the order in which the messages have to be exchanged, the transport protocol

Michael Eichberg, Karl Klose, Ralf Mitschke and Mira Mezini
13th International Symposium on Component Based Software Engineering
Springer; 2010