

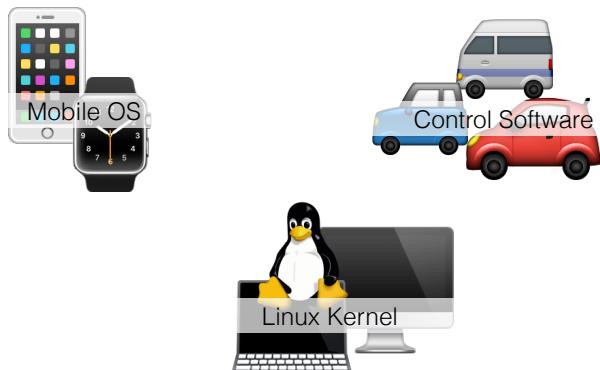
Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Software Product Line Engineering

based on slides created by Sarah Nadi

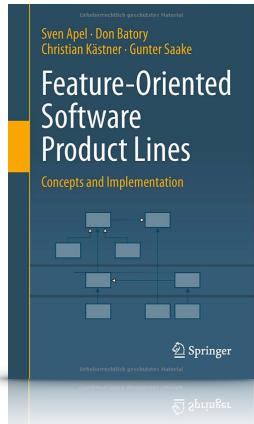
Examples of Software Product Lines



2

Software product lines are ubiquitous!

Resources



3

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."

4

Advantages of SPLs

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

5

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

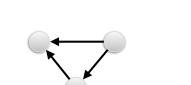
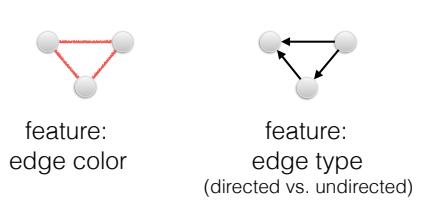
6

Feature-oriented SPLs

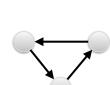
Thinking of your product line in terms of the **features** offered.

7

Examples of a Feature (Graph Product Line)



feature:
edge type
(directed vs. undirected)



feature:
cycle detection

8

Examples of a Feature

(Collections Product Line)

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

9

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.

10

What features would a Smartphone SPL contain?

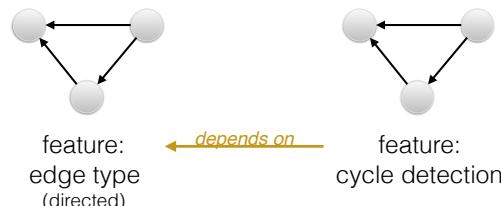
Discussion

11

- Integrated hardware (e.g., size and resolution of the display, network connections support (Bluetooth 4.x), Wireless 802.11abg..., 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!



12

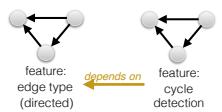
Product

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.

13

Valid Products

Feature Dependencies



Product Configurations

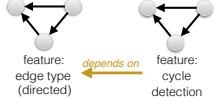
	Edge Color	Directed Edge	Cycle Detection
Product 1	✓	✓	✓
Product 2	✓		✓
Product 3		✓	✓

Ask yourself which product is (in)valid?

14

Valid Products

Feature Dependencies



Product Configurations

	Edge Color	Directed Edge	Cycle Detection
Product 1	✓	✓	✓
Product 2	✓	not valid	✓
Product 3		✓	✓

15

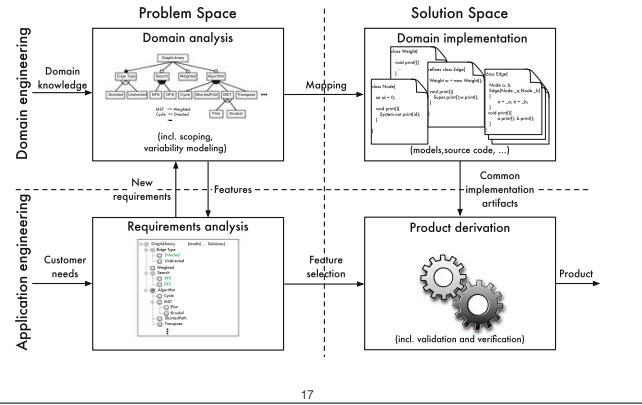
Identify feature dependencies
in a Smartphone SPL?

Discussion

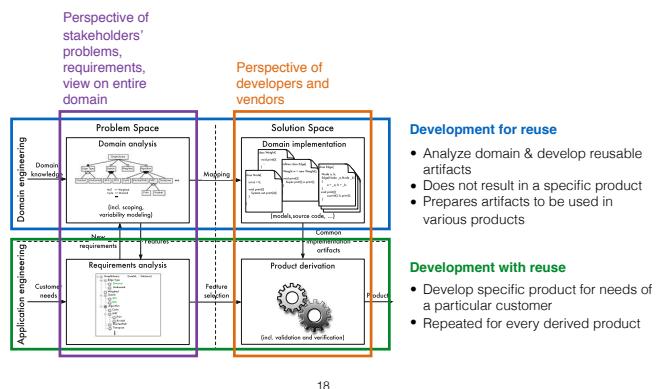
16

The dependency constraint is not satisfied by product 2.

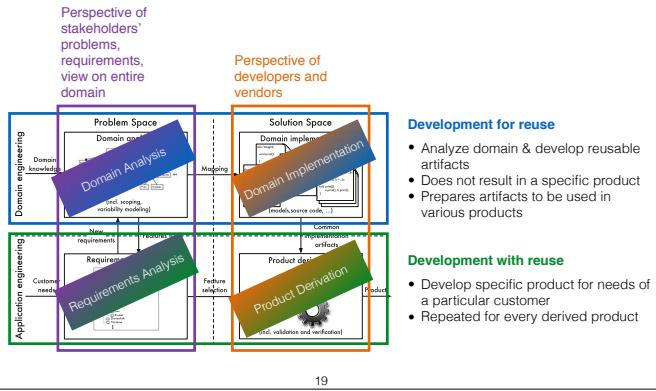
Software Product Line Engineering



Software Product Line Engineering



Software Product Line Engineering



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

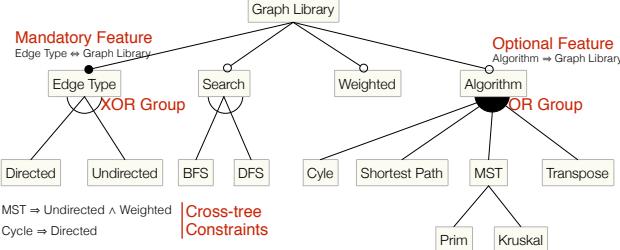
- Document the features of a product line & their relationships
- Can be translated into propositional logic

21

Domain Analysis

Graph Library Feature Model

Hierarchy Constraints



22

MST = Minimum Spanning Tree

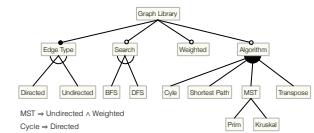
Hierarchal Relationships: Parent/child relationship - Child cannot be selected unless parent is selected

Domain Analysis

Graph Library

Feature Model in Propositional Logic

```
root(GraphLibrary)
  ^ mandatory(GraphLibrary,EdgeType)
  ^ optional(GraphLibrary,Search)
  ^ optional(GraphLibrary,Weighted)
  ^ or(Search,{BFS,DFS})
  ...
  ^ alternative(MST,{Prim,Kruskal})
  ^ (MST => Weighted)
  ^ (Cycle => Directed)
  ^ (...)
```



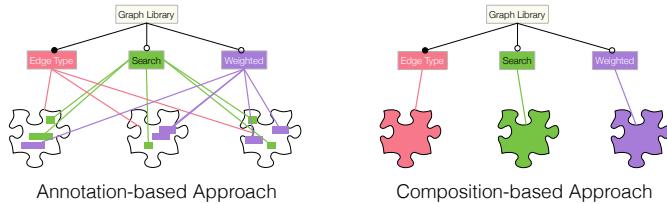
23

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

24

Domain Implementation (Representation)



25

Variability Implementation

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

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

26

Variability Implementation Parameters

- ★ simple
- ★ flexible
- ★ language support
 - code bloat
 - computing overhead
 - non-modular solution

27

annotation-based; binding time: run time

Variability Implementation Design Patterns

- ★ well established
- ★ easy to communicate design decisions
 - architecture overhead
 - need to preplan extensions

28

composition; binding time: run time

Variability Implementation Build Systems

- ★ 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

29

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

Variability Implementation Preprocessors

- ★ 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

30

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*

31

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?

32

composition; binding time: compile time



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

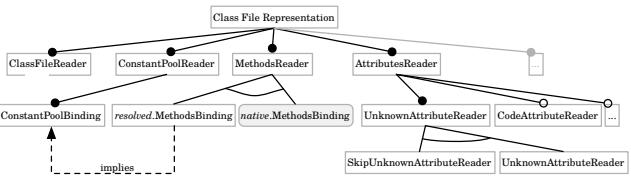
binding time: compile-time

- 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

Processing Java .class Files

Case Study



35

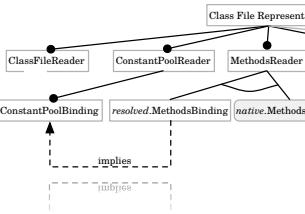
Processing Java.class Files

Case Study

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)  
    }  
}
```

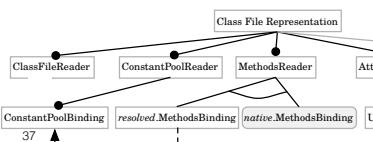
36



Processing Java.class Files

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
}
```

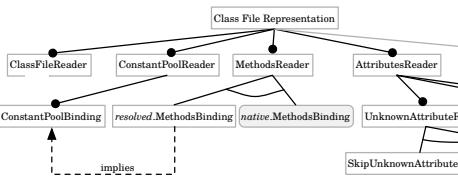


Processing Java.class Files

Product configurations

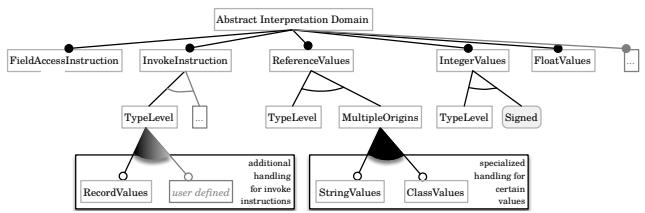
```
class Java7ClassFilesPublicInterface  
extends ClassFileBinding  
with ConstantPoolBinding  
with FieldsBinding  
with MethodsBinding  
with AttributesReader  
with SkipUnknown_attributeReader  
with AnnotationBinding  
with InnerClasses_attributeBinding  
with InterfacesBinding  
// further attributes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

```
class Java7ClassFiles  
extends Java7ClassFilesPublicInterface  
with CodeAttributeBinding  
with StackMapTable_attributeBinding  
with LineNumberTable_attributeBinding  
with LocalVariableTable_attributeBinding  
with BootstrapMethods_attributeBinding
```



Analyzing Methods

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



39

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

Component Composition Using Feature Models

Michael Eichberg¹, Karl Klose², Ralf Mitschke³, and Mira Mezini²
¹ Technische Universität Darmstadt, Germany
 eichberg, mitschke, mezini}@infrastruktur.tu-darmstadt.de
² Achmea Nederland
 klose@achmea.nl

Abstract. In general, components provide and require services and thus can be composed. However, certain variability in a service may be required by the second component. However, certain variability in a service – w.r.t. its interface – is often not described in the service specification, but rather using standard interface description languages. If this variability is not described, it is not possible to automatically determine which components are required to decide which components can be bound. We propose to use feature models to describe the variability in a service and to use them for semantic component binding. In our approach, feature models are one part of a component's interface. This allows to define what kind of service which service variant is provided by a component. By referring to a service's requirements, a component can then bind to another component and fulfill the requirements on the desired variants. Using these specifications, a component can be composed with other components such that all requirements are satisfied that satisfies all requirements. The potential environment consists of components that satisfy the requirements 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 may also specify a different set of services than the services provided by the interface defined by the service specification. This approach of "programming against interfaces" is well known from component-based systems and is called *open composition*. Current interface description languages (Java interfaces, WSDL interfaces, etc.) are not able to describe the variability in a service, i.e., the variability in providing their variabilities. However, in an open composed environment, several components may co-exist that implement the same programmatic interface, but with very different semantics. For example, two components may implement the same Java interface, but provide different functionality. They may implement the same WSDL interface, but do support a different set of credit card vendors, or different security mechanisms. In general, the variability in a service is not described by 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

40