VICCI

Visual and Interactive Cyber-Physical Systems Control and Integration

Exercise Academic Skills for Software Engineers

# Feature-based Software Product Lines

## …and their Application

Christoph Seidl

Georg Püschel

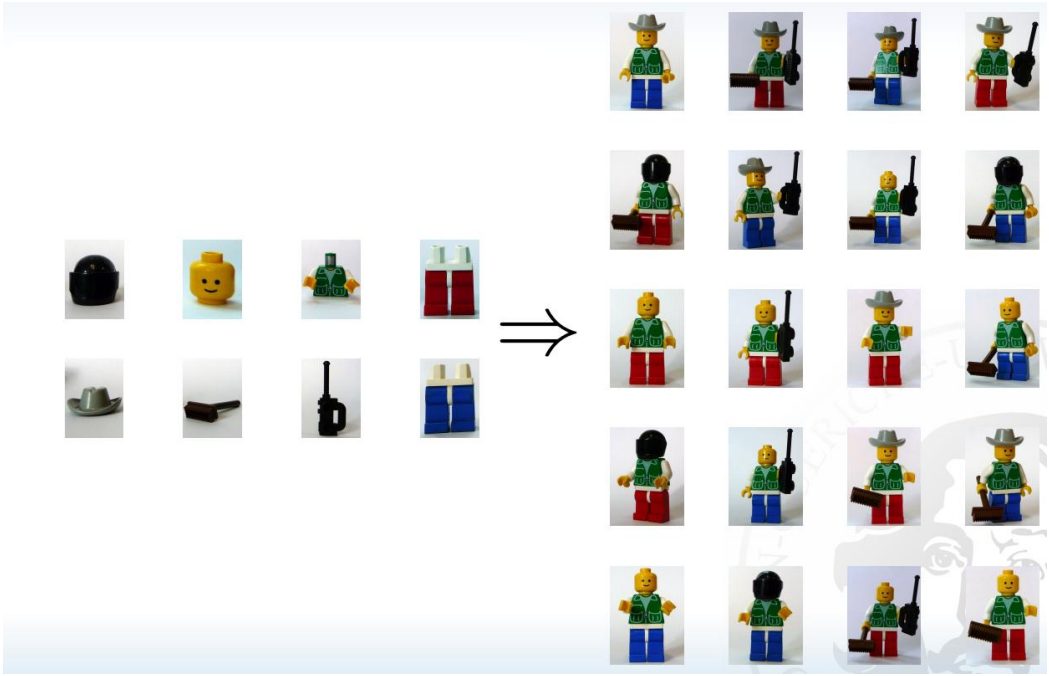
Julia Schroeter


# Configurable Products (1)

- Real world configurable product: Lego Manikin

Image courtesy Thomas Thüm, used by permission.

# Configurable Products (2)



- Software Product Lines
  - Something similar for software
  - Approach for software reuse in the large
  - Build individual software programs by combining reusable blocks

Image courtesy Thomas Thüm, used by permission.

# Developer/Vendor View

- Customers want similar (but not equal!) software products
- Making modifications to individual applications causes problems
  - Hard to maintain, update, fix
  - Hard to reuse similar functionality
- Solution
  - Variability management in the large scale
  - **Software Product Lines!**

# Software Product Lines (1)

- Intent
  - Define common functionality
  - Define variable parts
  - Define how variable parts can be combined with common functionality to create products
- -> All possible products are (theoretically) known in advance (**closed variant space**)

- Terms
  - Program Family: the set of all possible programs created by the SPL
  - Product/Variant: one program out of the program family
  - Realization Asset: part directly related to implementing a particular program, e.g., source code, UML models, documentation etc.

# Software Product Lines (2)

- Challenge: Express variablity and configuration options

- Pragmatic solution: ifdefs in C/C++
  - Only in implementation!? (code, design models, documentation etc.)
  - Problem
    - Configuration knowledge distributed over implementation
    - Hard to see configuration options for non-technicians (management, customers)
  - Solution
    - Model variability explicitly and connect it to the implementation (**variability model**)

# Variability Model

- Use separate model to capture variability

- Intent
  - Express configuration options and configuration logic
  - Use domain language (non-technical)
  - Describe all possible products without iterating them (too many)
  - At this point: No regard to implementation of individual products

- Possibilities
  - **Feature Models**
  - Decision Models
  - Orthogonal Variability Models
  - …

# Feature Models (1)

- Feature
  - Set of requirements describing user visible functionality of a software product
  - Variable unit of functionality that can be reused in multiple products
  - Use terms of domain (non-technical) language
  - Examples: CreditCardPayment, SearchFunction

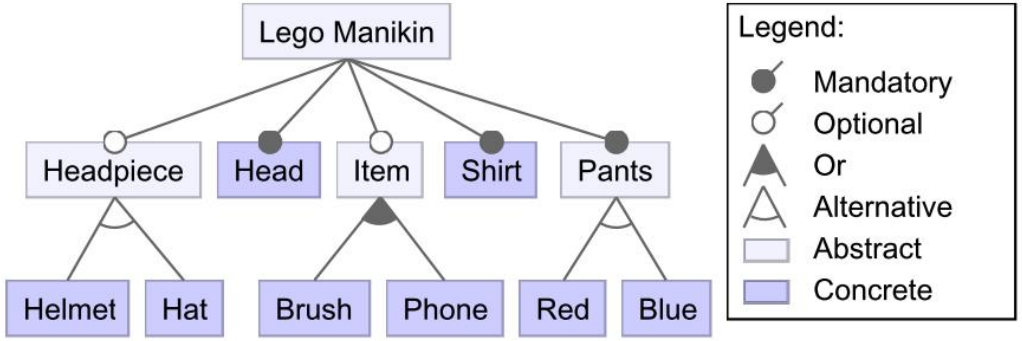
# Feature Models (2)

- Feature Model
  - Capture commonality and variability of SPL
  - Use features
  - Often represented as tree, cross-tree constraints make it a graph
  - Describes variant space

- Variant Configuration
  - A subset of features
  - Must be consistent regarding feature model constraints
  - All variability is bound
  - Used to derive a product

# FODA Notation for Feature Models

- FODA: Feature-oriented Domain Analysis [KCH+90]
  - Optional/Mandatory features
  - Alternative/Or groups




- Pros
  - Good as graphical representation
  - Graphical representation supports (simple) constraints (requires, excludes)
- Cons
  - Limitations regarding selections in groups (e.g., 2 out of 3 possible options?)

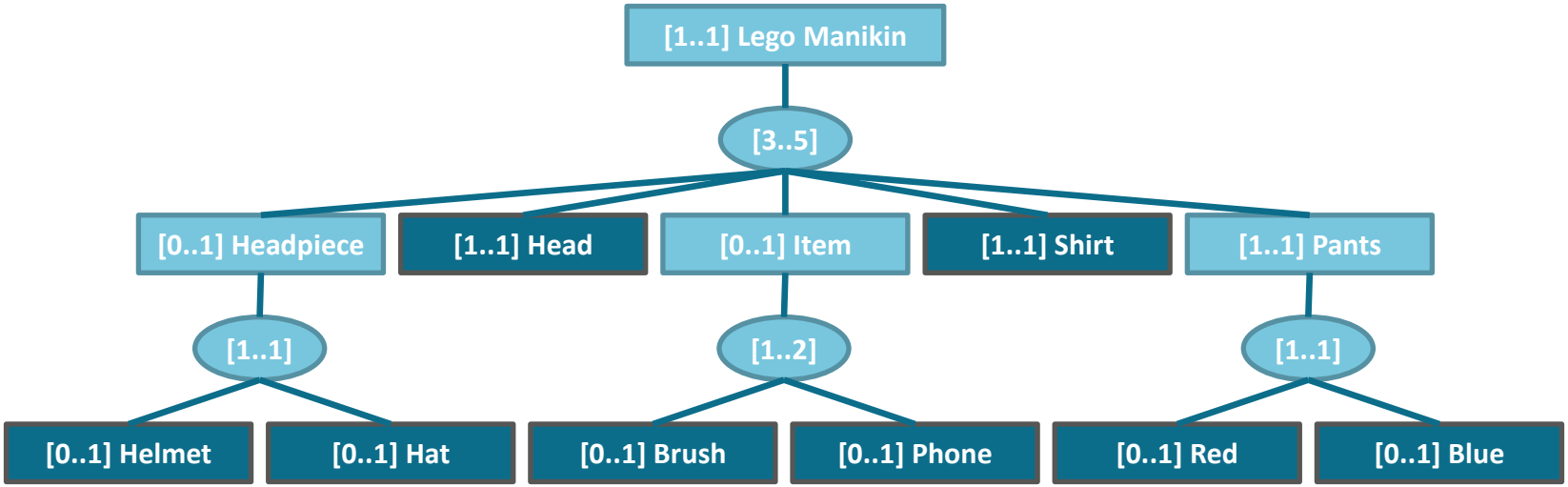
Image courtesy Thomas Thüm, used by permission.

# Cardinality-based Feature Models (1)

- Distinguish between features and groups

- Use min and max cardinality for features and groups
  - Features
    - optional: [0..1]
    - mandatory: [1..1]
    - ~~cloned features [0..n]~~
  - Groups (n child features, m mandatory child features)
    - alternative group: [1..1]
    - or group: [1..n]
    - and group: [m..n]
    - arbitrary cardinality: [i..j] (i <= j, i >= m, j <= n)

# Cardinality-based Feature Models (2)




- Pros
  - More powerful expressiveness (e.g., 2 out of 3 no problem)
  - Easier to evaluate and transform (only numbers not different structures for optional/mandatory, alternative/or etc.)
- Cons
  - Not so intuitive visualization

Image courtesy Thomas Thüm, used by permission.

# Cross-tree Constraints

- Tree structure of feature model is primary dimension of configuration options
- Additional configuration constraints may exist
- -> Cross-tree constraints
- Graphical/textual notation for constraints
- **Feature Expression**: logical formula containing references to features (describing their presence in configuration)
- Example: *Helmet => not Phone*

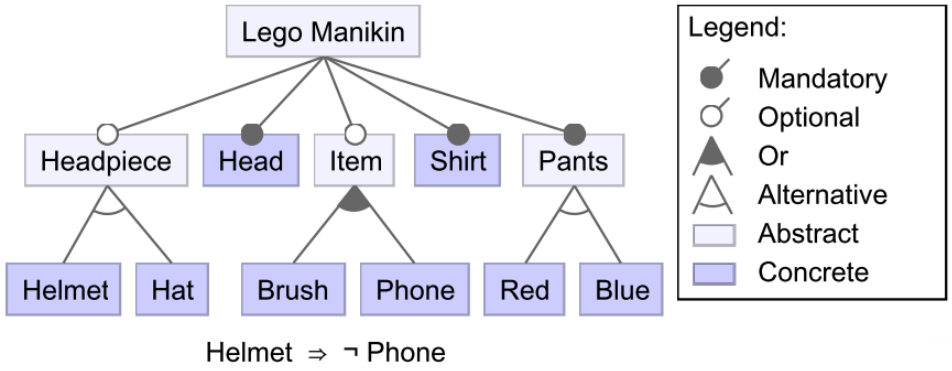



Image courtesy Thomas Thüm, used by permission.

# Implementation

- Feature model describes variability but not how products are implemented
- Challenge: Not all parts of implementation are required for all configurations
  - A feature may require parts of multiple assets (e.g., UML design and implementing classes)
  - A feature may only require parts of an asset (e.g., only a few methods of a class)
- -> Need to modify assets/resources to include them in a particular product
- Two basic procedures:
  - **Positive/Additive Variability**
  - **Negative/Subtractive Variability**

# Positive Variability

- Also known as: Additive Variability
- Create an asset as multiple small parts and combine them

- Pros
  - Parts of asset can be modeled in same granularity as features
- Cons
  - High maintenance effort because hard to deal with small fragments
  - Standard tools may not be useable (partial artifacts not always allowed!)
  - Requires composition approach

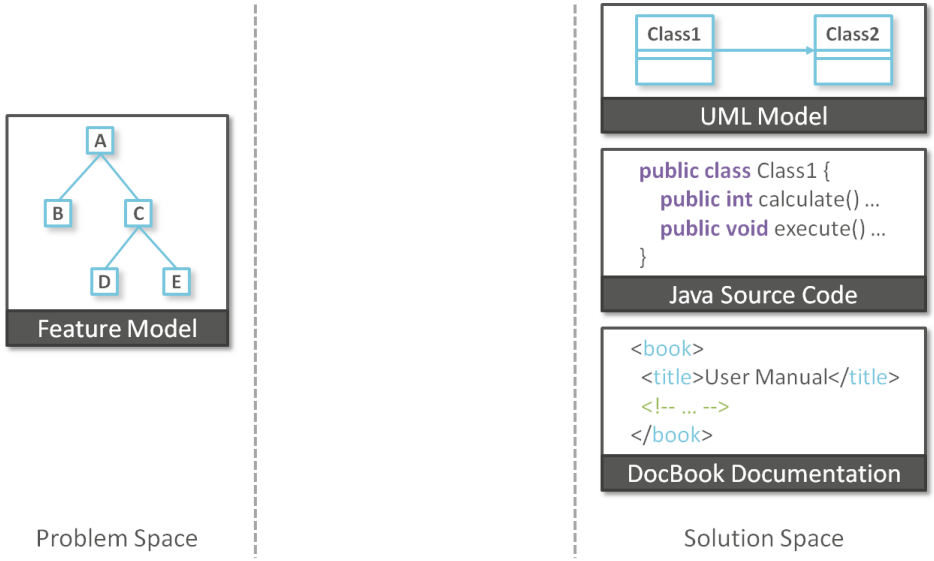
# Negative Variability

- Also known as: Subtractive Variability
- Create one large asset for all features and remove what is not needed in configuration
- Model based: „150% model“

- Pros
  - Standard tools (widely) useable (just a regular model)
  - Composition through removal of parts
- Cons
  - Conflicting information for single asset hard to express (e.g., in UML model, one feature multiplicity „*“ other feature has „1“?)

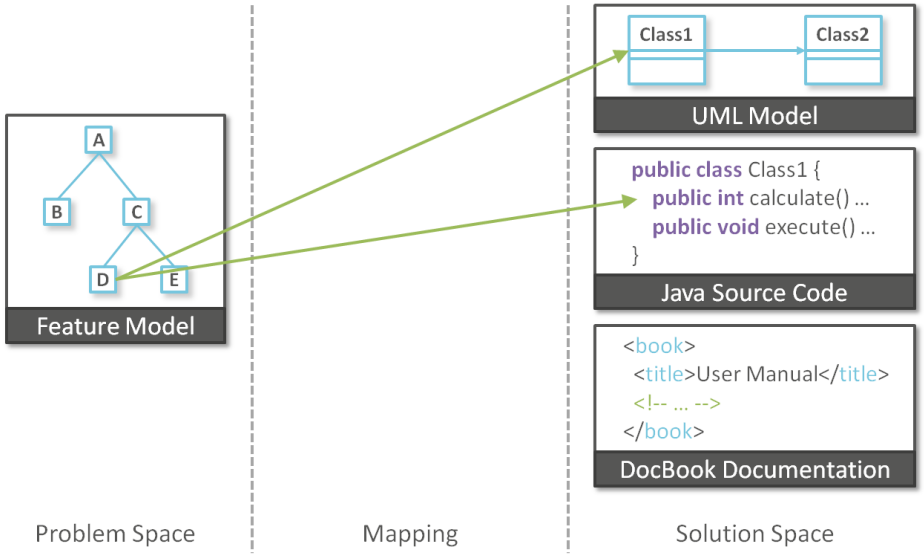
# Problem Space/Solution Space

- Problem Space [PBL05]
  - Conceptual modeling of variability
  - Variability model, cross-tree constraints etc.
- Solution Space [PBL05]
  - Realization/implementation assets
  - Source code, documentation, UML models/diagrams, configuration files etc.



# Deriving Products from the Software Product Line

- Configure products in problem space
- Create implementation from solution space
- Assemble relevant assets for products
- Needs connection from problem space to solution space



# Creating/Maintaining Software Product Lines

- **Domain Engineering**: deals with the development and maintenance of reusbale core or domain assets, which typically are reusable pieces of software, but can also be requirements, design, documentation, etc. [Han10]

- **Application Engineering**: deals with the development of software products, or applications, using the core assets for rapid and efficient composition of software products adjusted to the need of the customers [Han10]

# Process of Domain/Application Engineering

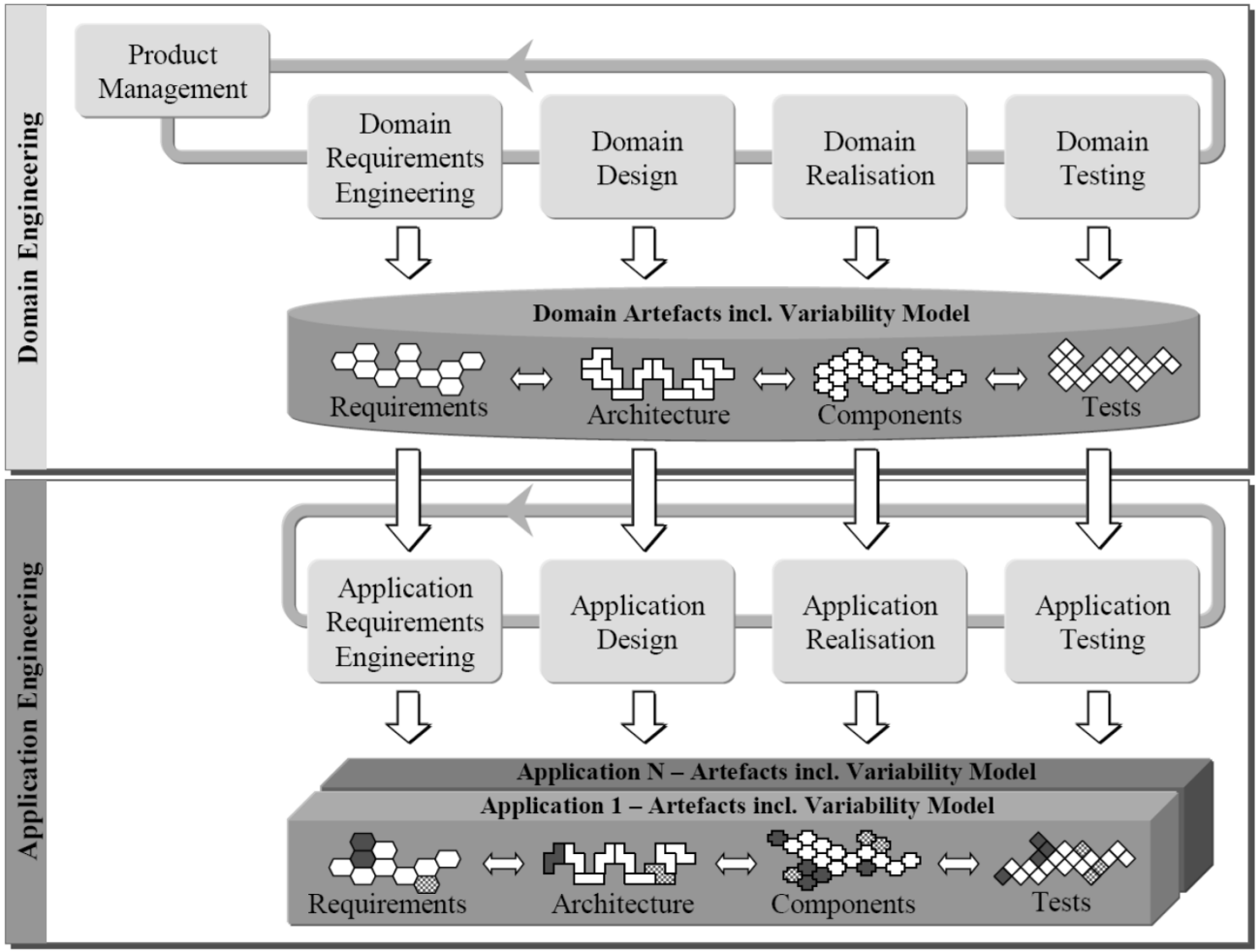



Image source: [PBL05]

# SPLs vs. other Software Reuse Mechanisms

- SPL
  - prescribes application logic
  - one vendor of products
  - explicit variability model
  - variant space is closed
- Class Library (e.g., Swing)
  - does not prescribe application logic
  - one/multiple vendors of products
  - no variability model
  - variant space is not closed
- Framework (e.g., Salespoint, Spring)
  - prescribes application logic
  - one vendor of products
  - no variability model
  - variant space is not closed
- Software Ecosystem (e.g., Eclipse, Android)
  - prescribes application logic
  - multiple vendors of products
  - implicit variability model
  - variant space is not closed

# Open Challenges

- Dynamic Staged Configuration (Julia)
  - Domain of multi-tenant aware applications in the cloud
  - Multiple stakeholders with different concerns involved in variant configuration
  - Ensure that configuration decisions do not contradict each other
  - Add stakeholders dynamically and allow for reconfiguration
  - -> Use consistent perspectives and configuration workflows

- Testing Dynamically Variable Software Product Lines (Georg)
  - Context-adaptive software
  - Too many variations (functional, temporal)
  - -> Build test models for dynamically variable systems

- Configurability in Software Ecosystems (Christoph)
  - Systematically handle variability in open systems such as Eclipse
  - Hard to model/manage variability because systems are evolving constantly and multiple vendors have independent release cycles
  - -> Extend variability models to allow extension, evolution, multiple contributors etc.

# References


- **[Han10]** Hanssen: *Opening Up Software Product Line Engineering* (2010)
- **[PBL05]** Pohl, Böckle, Linden: *Software Product Line Engineering - Foundations, Principles and Techniques* (2005)
- **[KCH+90]** Kang, Cohen, Hess, Nowak, Peterson: *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (1990)