Software Product Line Engineering
Variability Representation with Feature Models

Jessie Carbonnel

Université de Montpellier, LIRMM & CNRS

8 November, 2016

Definitions

Software product line engineering

*"Software product line engineering (SPLE) refers to software engineering methods, tools and techniques for creating a **collection of similar software systems** from a **shared set of software assets** using a **common means of production**."*

Van Vliet et al. - *Software engineering : principles and practice*, 1993

Software product line

*"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**."*

Carnegie Mellon Software Engineering Institute - *http ://www.sei.cmu.edu/productlines/*

Software product line engineering

Software product line engineering

- Development paradigm to efficiently create and manage a **collection** of related software systems

    $\rightarrow$ Opposed to *single system development*

- Application of *mass customization* in the software engineering domain

3 important concepts

- Similar software systems
- Sharing software assets
- In a prescribed way

SPLE concepts (1/3)

Similar software systems

- Software systems from a **same domain** ...
  - $\rightarrow$ Security, management, e-commerce, operating system, ...

- ... satisfying a **specific need** ...
  - ✘ A schedule management software & a supplies management software
  - ✔ Two antivirus softwares

- ... and sharing **commonalities**
  - $\rightarrow$ Code, requirement, architecture...

SPLE concepts (2/3)

Sharing software assets

4 software systems and their common set of assets : ▪▪▪▪

- Terminology : Asset / **feature** / functionality

  $\rightarrow$ an important caracteristic defined by domain experts to distinguish systems from one another

  $\rightarrow$ functional and non-functional aspects of a system

- Different **levels of granularity**

  $\rightarrow$ from low level code chunks to high level software functionalities

Asset's levels of abstraction

Different levels of abstraction of the shared assets

Example : A collection of e-commerce applications

- **High level** functionalities understandable by **final users**
  - → payment methods, basket, newsletter, wishlist, ...

- **Low level** methods/algorithms implemented by **developers**
  - → paypal authentication, connection to databases, form validation, ...

Why ?

⇒ The basket functionality is implemented by several assets of lower level, but it is hidden from the user for understandability sake

⇒ A low level asset can be used in different high level functionalities (e.g., database connection)

SPLE concepts (2/3)

Sharing software assets

- **Commonalities** / **variabilities**

4 software systems and their common set of assets : ▪▪▪▪

→ The *green* asset is present in all softwares

→ The *blue* and *orange* asset are shared by several softwares

→ The *purple* asset is specific to the fourth software

SPLE concepts (3/3)

In a prescribed way

- Documentation of what is common and what varies between the software systems

    $\rightarrow$ Defining the way assets vary/interact in the software systems

    Example. A collection of e-commerce applications

    - $\rightarrow$ All e-commerce application have a *catalog*
    - $\rightarrow$ Some can optionally have a *wishlist* or a *newsletter* functionality
    - $\rightarrow$ The *wishlist* functionality requires a *user account* management

    $\Rightarrow$ **Organisation** of the set of assets in a **generic architecture**

SPLE concepts (3/3)

Generic architecture
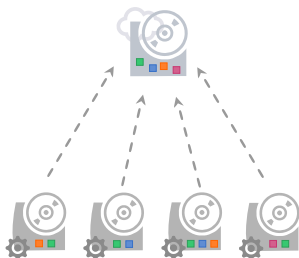
$\rightarrow$ The generic architecture permit to describe several related software systems depending on a set of assets

$\rightarrow$ The set of software systems comply with the generic architecture

Why ?

$\Rightarrow$ **Factorisation and exploitation** of common assets

$\Rightarrow$ **Delimits the scope of a software family**

Synthesis
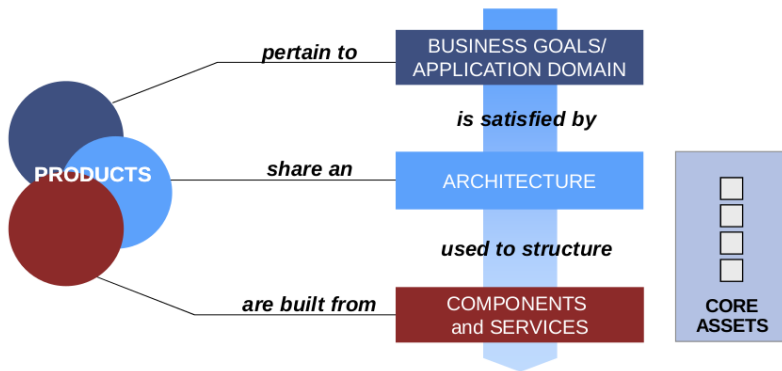

Carnegie Mellon Software Engineering Institute - *Patrick Donohoe*

⇒ Permits to **derive** several different software systems from the generic architecture

Domain engineering

Software produt line engineering - phase 1

- **Domain engineering**
    - $\rightarrow$ Domain analysis
    - $\rightarrow$ Domain representation
    - $\rightarrow$ Domain implementation
    - $\Rightarrow$ **Development FOR reuse**

	Domain analysis		Domain representation		Domain implementation	
Domain		*Assets*		*Model*		*Generic Architecture*
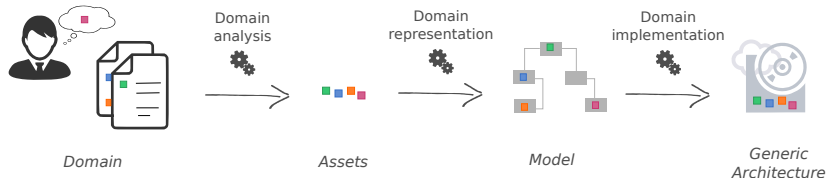
Application engineering

Software produt line engineering - phase 2

- **Application engineering**
    - $\rightarrow$ Product selection
    - $\rightarrow$ Product derivation
    - $\Rightarrow$ **Development BY reuse**

User requirements → Software selection → *Generic Architecture* → Software derivation → *Software system*
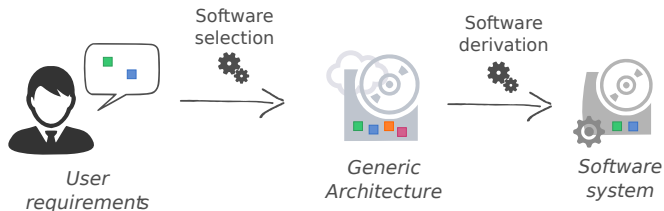
Software selection and software derivation

Software selection / configuration

- A user specifies its requirements = **configures** the architecture
- $\rightarrow$ designate a product configuration
- $\rightarrow$ which has to comply with the architecture

Software derivation

- Implementation of the designated configuration
- $\Rightarrow$ Leads to **(semi-)automated source code generation**
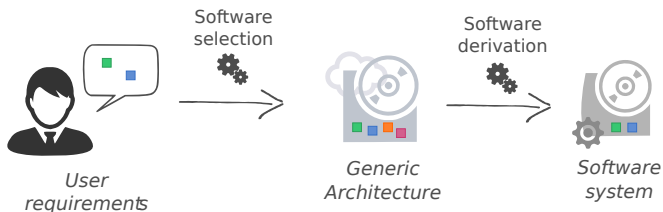
Benefits

What are the benefits of software product line engineering ?

- Improved productivity by as much as $10\times$
- Increased quality by as much as $10\times$
- Decreased cost by as much as 60%
- Decreased labor needs by as much as 87%
- Decreased time to market (to field, to launch) by as much as 98%
- Ability to move into new markets in months, not years

Carnergie Mellon Software Engineering Institure - *http ://www.sei.cmu.edu/productlines/*

Benefits

Variability representation
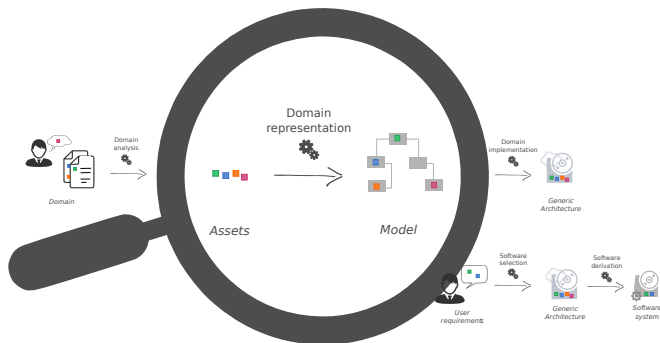
Central point of SPLE

Modelisation of the common parts and variants contained in the software systems

= **variability** of the software product line

Variability models

Software product line variability representation

⇒ **Variability models**

Basis for SPLE operations and SPL management :

- Software selection
- Software derivation
- Product line evolution
- Information retrieval
- ...

How to represent the variability of a software product line ?

Variability modelling approaches

Several **variability modelling approaches** exist in the literature.

Two prevalent ones

- Decision modelling

- Feature modelling

Decision modelling

- **Decision modelling**

    $\rightarrow$ List of possible decisions a user can make

    $\rightarrow$ *Focus on product selection/derivation*

decision name	description	type	Range	cardinality/constraint	visible/relevant if
GSM_Proto-col_1900	Support GSM 1900 protocol?	Boolean	true | false		
Audio_Formats	Which audio formats shall be supported?	Enum	WAV | MP3	1:2	
Camera	Support for taking photos?	Boolean	true | false		
Camera_Resolu-tion	Required camera resolution?	Enum	2.1MP | 3.1MP | 5MP	1:1	Camera == true
MP3_Recording	Support for recording MP3 audio?	Boolean	true | false	ifSelected Audio_Formats.MP3 = true	

GSM_Protocol_1900: one of (GSM_1900, NO_GSM_1900)   {indicates whether support for making and receiving calls using
                                                          GSM 1900 is available}
Audio: list of (WAV, MP3)   {indicates the types of supported audio formats}
Camera: composed of
  Presence: one of (Camera, NO_Camera)   {indicates whether camera support is available}
  Resolution: one of (2.1MP, 3.1MP, 5MP)   {resolution of the camera}
MP3_Recording: one of (MP3, NO_MP3)   {indicates whether MP3 recording is available}

Constraints
Resolution is available only if Presence has the value Camera
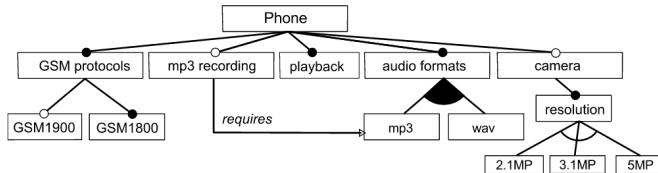MP3_Recording requires that also MP3 Audio is supported

Czarnecki, Krzysztof, et al. "Cool features and tough decisions : a comparison of variability modeling approaches."

Feature modelling
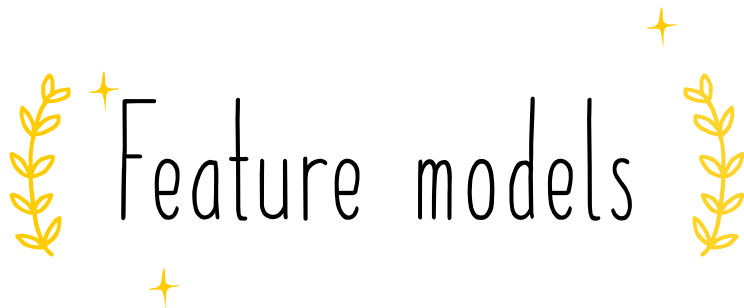
- **Feature modelling** (most prevalent one)

    $\rightarrow$ Distinguishable characteristics, dependencies
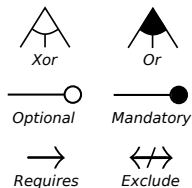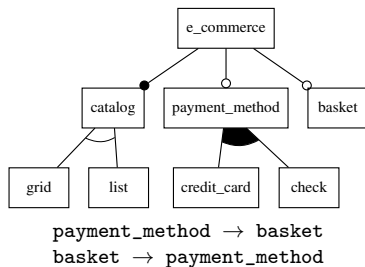
    $\rightarrow$ *Focus on domain representation*

Czarnecki, Krzysztof, et al. "Cool features and tough decisions : a comparison of variability modeling approaches."

How to model SPL variability in terms of features ?

Feature models

Feature models

payment_method → basket
basket → payment_method

Feature models : family of visual **description languages**

$\rightarrow$ permit to describe a finite set of features and dependencies between them

$\Rightarrow$ depict a finite set of valid combinations of features = **configurations**
 (*1 configuration = 1 software system of the family*)

Feature tree

payment_method → basket
basket → payment_method
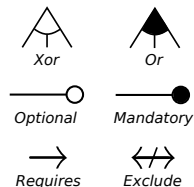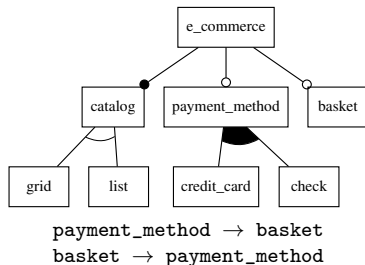
→ structure hierarchically the set of features in a tree = **feature tree**

- root feature = name of the modelised system
- (top to bottom) from most generalised features to most specialised one
- describe the system in several level of increasing details
- express **refinement relationships**

Software selection

payment_method → basket
basket → payment_method
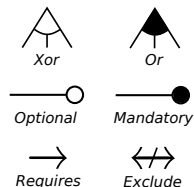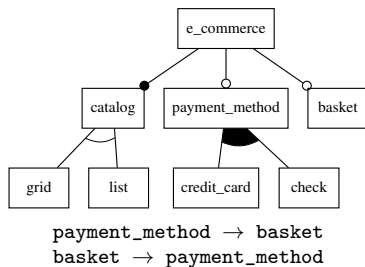
→ **Software selection**

- start from the root feature
- select feature from more generalised to more specialised ones (graph search)
- while respecting the expressed constraints

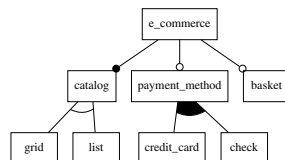## Constraints

→ **2 types of constraints**

- graphical constraints expressed in the feature tree

Xor   *Or*

Optional   *Mandatory*
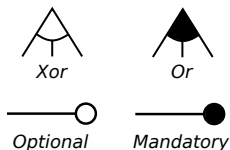
- textual constraints which cannot be expressed in the tree : **cross-tree constraints**

$$\longrightarrow \qquad \longleftrightarrow$$
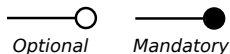
Requires   *Exclude*

Feature tree :

Cross-tree constraints :
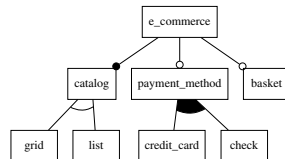
payment_method → basket
basket → payment_method

Graphical constraints (1)

→ **4 different "graphical constraints"** (1/2)

■ between a parent feature and its child feature :

Optional   *Mandatory*

- *Optional* : if the parent feature is selected, the child feature can be selected, or not

- *Mandatory* : if the parent feature is selected, the child feature is necessarily selected


payment_method → basket
basket → payment_method

Graphical constraints (2)

→ **4 different "graphical constraints"** (2/2)

- between a parent feature and several of its
  child features :

Xor   *Or*

  - *Or-group* : if the parent feature is selected, at
    least one feature involved in the group has to be
    selected

  - *Xor-group* : if the parent feature is selected,
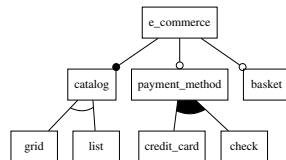   exactly one feature involved in the group has to
    be selected

`payment_method → basket`
`basket → payment_method`
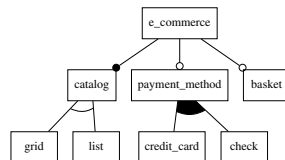
Textual constraints (2)

→ **2 different "textual constraints"**

- between two independant features :

$$\longrightarrow \qquad \longleftrightarrow$$
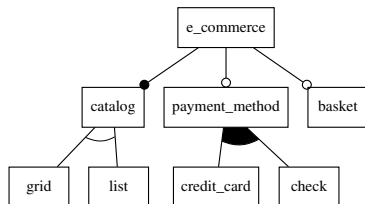
Requires   *Exclude*

  - *Requires* : if the premise is selected, the conclusion is also selected
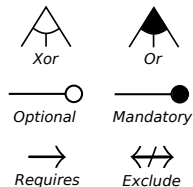
  - *Exclude* : the two features are mutually exclusive


```
payment_method → basket
basket → payment_method
```

Feature models

payment_method → basket
basket → payment_method

→ an e-commerce application necessarily possesses a catalog

→ this catalog can be displayed in a grid or in a list, but not both

→ it can eventually possess payment methods (credit card, check, or both)

→ it can also optionally have a basket

→ if the basket feature is selected, the application must possess at least one payment method (and conversely)

Feature model semantics

2 types of semantics

→ *what do feature models define ?*

- a *configuration* semantics / *logical* semantics

- an *ontological* semantics

Configuration/logical semantics

Configuration semantics :

$\rightarrow$ *The list of valid configurations depicted by the feature model*

1. {*e_commerce*, *catalog*, *grid*}
2. {*e_commerce*, *catalog*, *list*}
3. {*e_commerce*, *catalog*, *grid*, *payment_method*, *credit_card*, *basket*}
4. {*e_commerce*, *grid*, *payment_method*, *check*, *basket*}
5. {*e_commerce*, *catalog*, *grid*, *payment_method*, *credit_card*, *check*, *basket*}
6. {*e_commerce*, *catalog*, *list*, *payment_method*, *credit_card*, *basket*}
7. {*e_commerce*, *list*, *payment_method*, *check*, *basket*}
8. {*e_commerce*, *catalog*, *list*, *payment_method*, *credit_card*, *check*, *basket*}
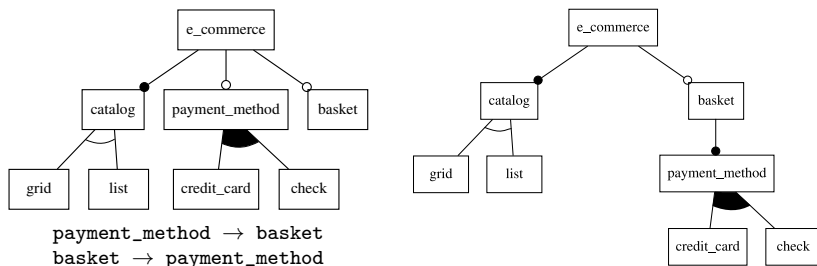
Ontological semantics

Ontological semantics

$\rightarrow$ *Domain knowledge depicted by the feature model*

Example.

- *grid* and *list* refine *catalog*
- *catalog* and *payment_method* are two independent features
- *credit_card* and *check* are independent but can cohexist

Non-canonical representation
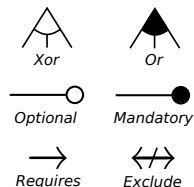

payment_method → basket
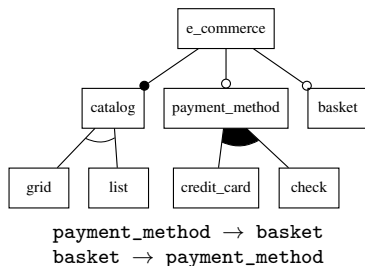basket → payment_method

→ Same *configuration semantics*, but different *ontological semantics*
    → describe different domain knowledge
⇒ **Non-canonical** representation

Feature models

payment_method → basket
basket → payment_method

→ **understandable** and **compact** way to express variability
- *combinatorial explosion* of the possible software variants
- potentially large number of represented software systems
 (*Example* : Linux SPL = 41 features = 2×10^7 configurations)
 ⇒ **enlarge the selection** of products offered