

L'ingénierie des lignes de produits logiciels

Concepts et représentation de la variabilité avec les *feature models*

Jessie Galasso Carbonnel

Université de Montpellier, LIRMM & CNRS

06 novembre 2018

Méthodes de production 1/2

○ Artisanat

→ *produit unique issu du savoir-faire d'un artisan*

- prendre en compte les souhaits/besoins d'un client
- processus chronophage

○ Production de masse

→ *produits identiques issus d'éléments standardisés et réutilisables, combinés dans une chaîne de production*

- augmente la productivité (+ de produits, - de temps)
- au détriment de la "spécificité"
- **standardisation**

Méthodes de production 2/2

- Allier une production **rapide** et **spécifique** ?

→ éléments standardisés et réutilisables

→ combinés de manières **différentes**

- **Ligne de produits**

→ *produit configuré avant sa production*

- production de masse
- **personnalisation** de masse

Application au développement logiciel

○ Logiciels spécialisés / individuels (\approx artisanat)

- créés par des experts
- pour une problématique et un client spécifiques
- processus chronophage

○ Logiciels standardisés (\approx production de masse)

- solution logicielle standard
- répond aux besoins du plus grand nombre d'utilisateurs possible
- *"one-size-fits-all solution"*

○ Lignes de produits logiciels

- logiciels similaires différents
- personnalisables avant production
- mêmes moyens de production

Définitions

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 (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

L'ingénierie des lignes de produits logiciels

Paradigme de développement :

- pour créer un **ensemble de logiciels similaires**
- ... partageant une **architecture logicielle commune** ...
- ... ainsi qu'un **ensemble d'artefacts communs et réutilisables** ...

Basé sur :

- ... la **réutilisation systématique** de ces artefacts ...
- ... et la **personnalisation de masses** des logiciels produits.

Ensemble de logiciels similaires

Famille de produits (Parnas 1976)

Ensemble de systèmes logiciels assez similaires pour qu'il soit plus avantageux et pertinent de considérer d'abord leurs propriétés communes avant d'étudier celles qui sont spécifiques à chacun de ces produits.

Logiciels similaires

- Partager un **même domaine**

→ Sécurité, gestion, commerce en ligne, système d'exploitation, ...

- Satisfaire un **même besoin**

✗ logiciel de gestion de calendrier & logiciel de gestion de matériel

✓ deux antivirus

- Partager assez d'**éléments communs**

→ Code, exigences, architecture, documentation ...

Logiciels similaires : exemple

Niveaux de jeu vidéo similaires bien que différents



Architecture générique + exemple

- regroupe les **éléments communs** à tous les logiciels similaires
- possibilité d'ajouter des éléments plus spécifiques
- en fonction de **points de variabilité prédéfinis**

Points communs dans chaque niveau :

- taille de la fenêtre
- commandes de jeu
- plateformes
- ennemis
- pièges



Ensemble d'artefacts communs et réutilisables + exemple

- Éléments fonctionnels / non-fonctionnels
- Pas présents dans dans tous les logiciels
- Différents **niveaux de granularité**
→ depuis des morceaux de code de bas niveau vers des fonctionnalités de haut niveau

Artefacts variables

- types de plateformes
- types et pouvoirs des ennemis
- taille des pièges
- ...



Réutilisation systématique

■ Différent du **clone-and-own**

- copier/coller le code d'un logiciel pour en faire un nouveau
- adapter le clone logiciel, *repeat*
- difficile à maintenir, baisse la qualité globale des logiciels

■ **Systématisation de la réutilisation des différents artefacts**

- ensemble d'artefacts communs développés individuellement
- pouvant être combinés avec l'architecture générique
- pour produire une nouvelle variante logicielle

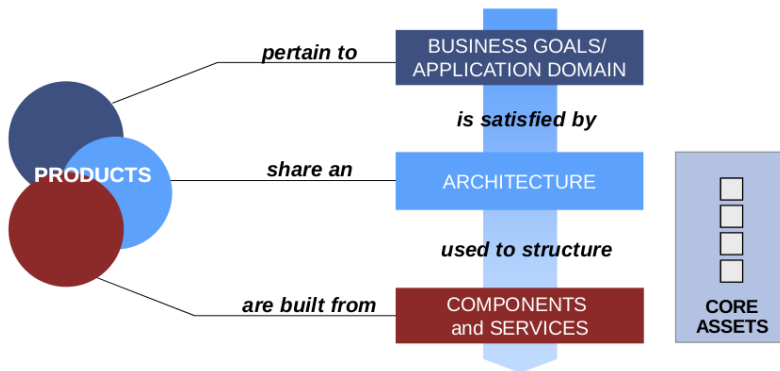
Factorisation et exploitation des différents artefacts

Personnalisation de masse

- Sélection des fonctionnalités désirées par l'utilisateur
 - identification du sous-ensemble d'artefacts correspondant
 - combinés à l'architecture générique
 - artefacts composant la variante = **configuration**
- Délimiter les configurations possibles
 - définir la façon dont les artefacts peuvent varier/interagir dans une variante
 - contraindre la sélection = **modéliser la variabilité**

Délimiter le scope de la ligne de produits

Synthèse



Carnegie Mellon Software Engineering Institute - Patrick Donohoe

Dériver plusieurs logiciels différents à partir d'un ensemble d'artefacts logiciels pouvant être combinés de plusieurs façons sur une architecture logicielle commune

Représentation de la variabilité

○ **Variabilité** :

- Quels sont les éléments communs ?
- Quels sont ceux qui peuvent varier ?
- Comment peuvent-ils varier ?

→ **concept clé des lignes de produits**

○ Définie par des **modèles de variabilité**

Deux principales approches de modélisation de la variabilité

- En termes de **décisions**
- En termes de **caractéristiques**

Modélisation en termes de décisions

- Liste les différentes décisions pouvant être prises par l'utilisateur
- Concentrée sur la **configuration d'un produit**

decision name	description	type	Range	cardinality/constraint	visible/relevant if
GSM_Protocol_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_Resolution	Required camera resolution?	Enum	2.1MP 3.1MP 5MP	1:1	Camera == true
MP3_Recording	Support for recording MP3 audio?	Boolean	true false	{Selected Audio_Formats.MP3 = true	

GSM_Protocol_1900: one of (GSM_1900, NO_GSM_1900)

Audio: list of (WAV, MP3)

Camera: composed of

Presence: one of (Camera, NO_Camera)

Resolution: one of (2.1MP, 3.1MP, 5MP)

MP3_Recording: one of (MP3, NO_MP3)

Constraints

Resolution is available only if Presence has the value Camera

MP3_Recording requires that also MP3 Audio is supported

{indicates whether support for making and receiving calls using GSM 1900 is available}

{indicates the types of supported audio formats}

{indicates whether camera support is available}

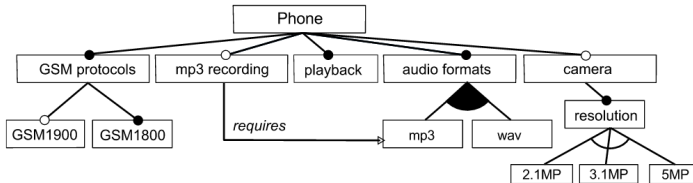
{resolution of the camera}

{indicates whether MP3 recording is available}

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

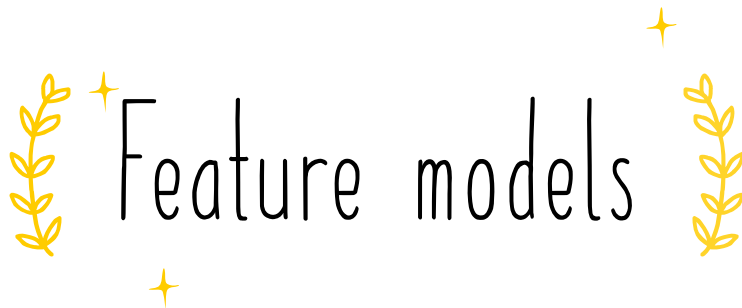
Modélisation en termes de caractéristiques

- **Caractéristiques** : fonctionnalités ou comportements visibles d'un logiciel
- Permet de comparer et de distinguer les logiciels
- Concentrée sur la **représentation du domaine modélisé**



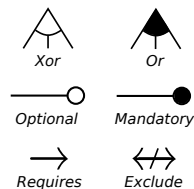
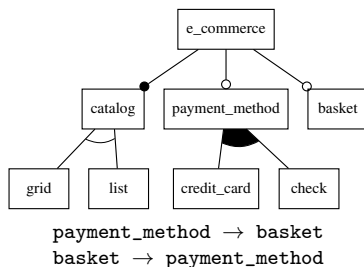
Czarnecki, Krzysztof, et al. "Cool features and tough decisions : a comparison of variability modeling approaches."

How to model SPL variability in terms of features ?

The title 'Feature models' is centered in a black, handwritten-style font. It is flanked by two yellow laurel wreaths, one on the left and one on the right. There are three yellow four-pointed stars: one above the left wreath, one above the right wreath, and one centered below the text.

Feature models

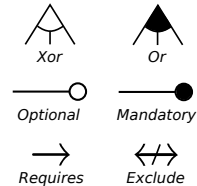
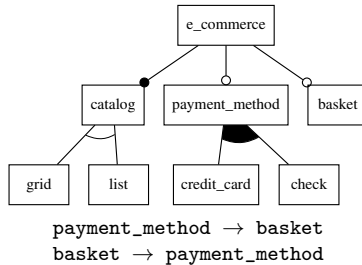
Feature models



Feature models : family of visual description languages

- permit to describe a finite set of features and dependencies between them
- ⇒ depict a finite set of valid combinations of features = **configurations**
(1 configuration = 1 software system of the family)

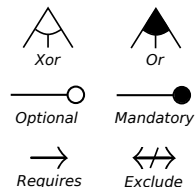
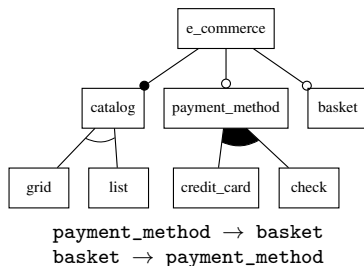
Feature tree



→ 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



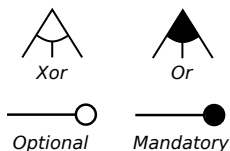
→ Software selection

- 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

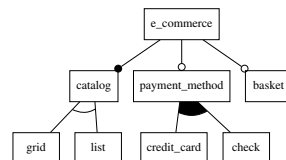
- graphical constraints expressed in the feature tree



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



Feature tree :



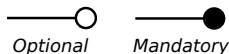
Cross-tree constraints :

payment_method → basket
basket → payment_method

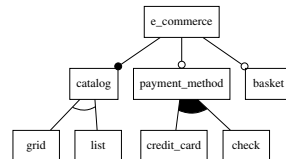
Graphical constraints (1)

→ 4 different “graphical constraints” (1/2)

- between a parent feature and its child feature :



- *Optional* : 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

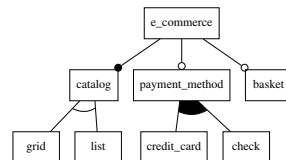
Graphical constraints (2)

→ 4 different “graphical constraints” (2/2)

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



- *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)

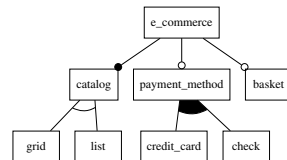
→ 2 different “textual constraints”

- between two independant features :

→
Requires

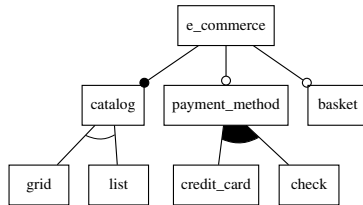
↔
Exclude

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

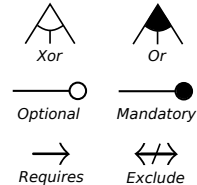


payment_method → basket
basket → payment_method

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 :

→ *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

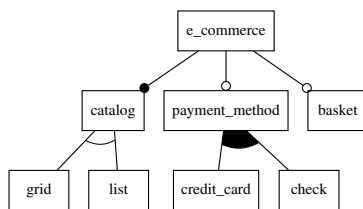
Ontological semantics

→ *Domain knowledge depicted by the feature model*

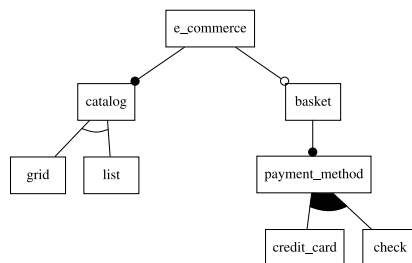
Example.

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

Non-canonical representation



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

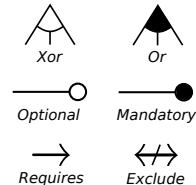
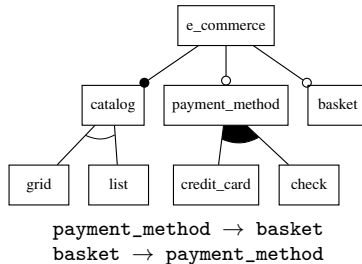


→ Same *configuration semantics*, but different *ontological semantics*

→ describe different domain knowledge

⇒ **Non-canonical** representation

Feature models



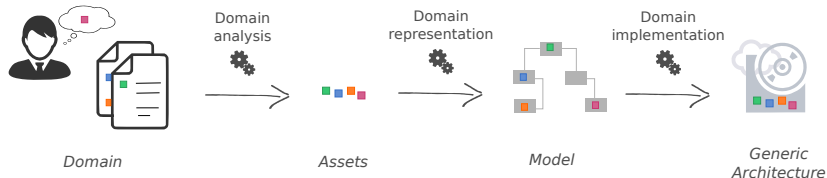
- **understandable** and **compact** way to express variability
- *combinatorial explosion* of the possible software variants
 - potentially large number of represented software systems
 (Example : Linux SPL = 41 features = 2×10^7 configurations)
- ⇒ **enlarge the selection** of products offered

Domain engineering

Software product line engineering - phase 1

■ Domain engineering

- Domain analysis
- Domain representation
- Domain implementation
- ⇒ **Development FOR reuse**



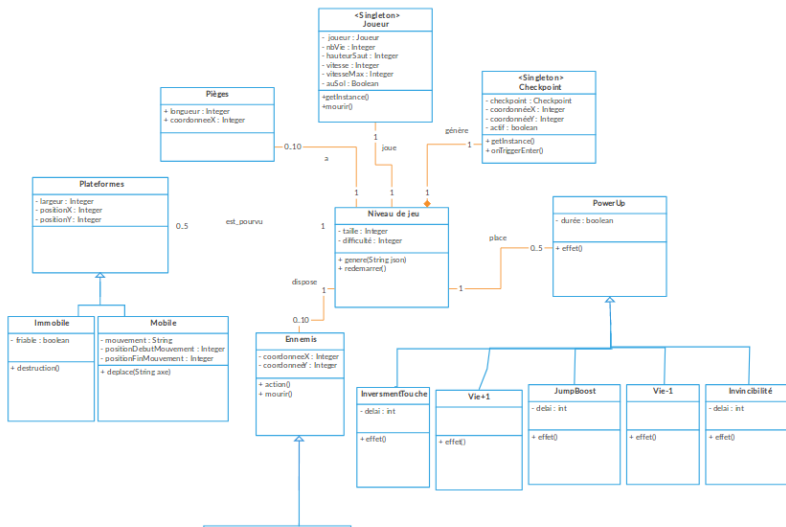
Domain analysis

What do we want in our future video game levels ?

- Des **pièges**, ou plutôt des trous où le joueur pourrait tomber et perdre de la vie.
- Des **plateformes**, des morceaux de terre au dessus du sol ou des pièges, avec une longueur variable, pouvant être mobiles ou immobiles. De plus certaines pourraient être friables et se détruire au bout de quelques secondes dès que le joueur monte dessus.
- Des **ennemis**, de deux types, faisant perdre de la vie au joueur lorsqu'ils le touchent, mais pouvant mourir lorsque le joueur leur saute dessus. L'un s'attaquant au joueur en courant (appelé Bumper dans le projet) et l'autre en lançant des projectiles (appelé Tireur dans le projet).
- Des **objets**, appelés "power-up" dans le reste du projet, ils modifieraient certaines caractéristiques dans le niveau comme donner ou enlever une vie au joueur, inverser ses commandes pour jouer, sauter plus haut ou encore devenir invincible face aux ennemis ; tout ça pendant un certain laps de temps.
- Des **difficultés**, afin d'appuyer la variabilité des niveaux générés. C'est-à-dire que plus la difficulté du niveau est élevé, plus il y aura d'ennemis et de pièges par exemple.

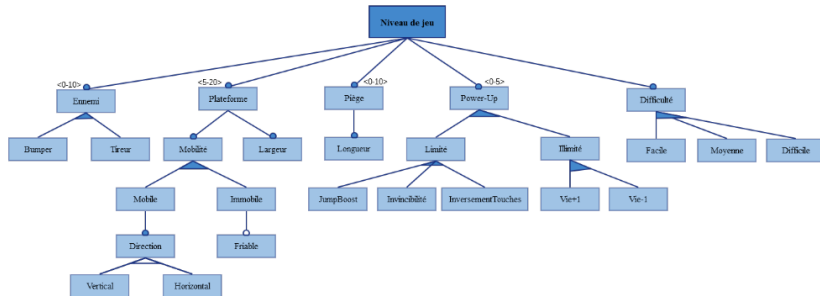
Domain representation

How these different assets interact ?



Domain representation

How these different assets interact ?



Domain implementation

Implementing the core architecture and the different assets

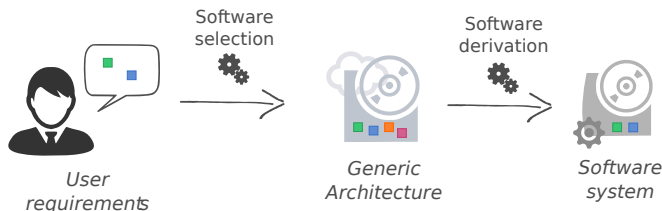
```
1 public void chargerNiveau(){
2
3     /*Instanciation des powerups*/
4     niveau.powerups.ForEach(delegate(PowerUp obj) {
5         GameObject tmp = Instantiate(Resources.Load ("powerup") as GameObject, new Vector2
6             (obj.getX(), obj.getY()), Quaternion.identity);
7         tmp.GetComponent<SpriteRenderer> ().sprite = Resources.Load<Sprite>("powerup/"+obj.
8             GetType());
9         tmp.name = obj.GetType().ToString();
10    });
11
12    /*Instanciation des Ennemis*/
13    foreach(var obj in niveau.ennemis){
14        bool isBumper = (obj.GetType() == typeof(Bumper));
15        GameObject tmp = Resources.Load("ennemis") as GameObject;
16        if (isBumper) {
17            Instantiate (tmp, new Vector2 (obj.x + 0.5f, obj.y - 0.4f), Quaternion.identity))
18                .GetComponent<ennemis_script> ().init (isBumper, niveau);
19        } else {
20            Instantiate(tmp,new Vector2(obj.x+0.5f,obj.y-0.5f),Quaternion.identity)).
21                GetComponent<ennemis_script>().init(isBumper, niveau);
22        }
23    }
24
25    /*Instanciation des Pieces*/
26    niveau.pieces.ForEach(delegate(Piece p) {
27        GameObject coin = Instantiate(Resources.Load ("coin") as GameObject, new Vector2(p.
28            positionX, p.positionY), Quaternion.identity);
29        coin.name = "coin";
30    });
31 }
```

Application engineering

Software product line engineering - phase 2

■ Application engineering

- Product selection
- Product derivation
- ⇒ **Development BY reuse**



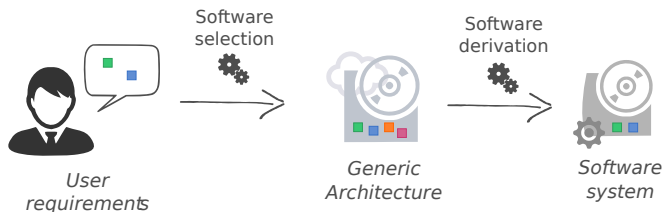
Software selection and software derivation

Software selection / configuration

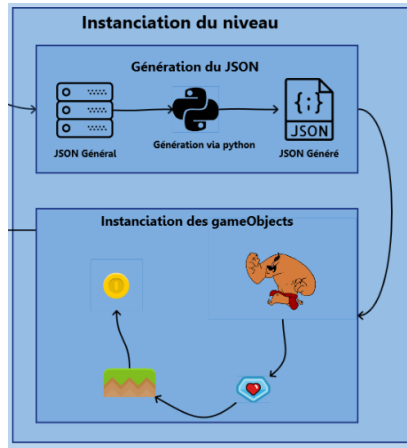
- A user specifies its requirements = **configures** the architecture
 - designate a product configuration
 - which has to comply with the architecture

Software derivation

- Implementation of the designated configuration
 - ⇒ Leads to **(semi-)automated source code generation**



Level selection and derivation



Benefits

What are the benefits of software product line engineering ?

- Improved productivity by as much as 10×
- Increased quality by as much as 10×
- 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

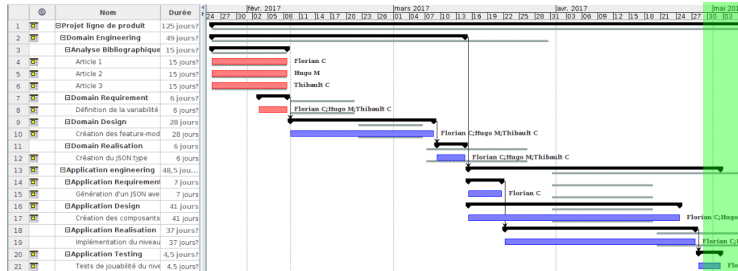
Carnegie Mellon Software Engineering Institute - <http://www.sei.cmu.edu/productlines/>

Benefits



But ...

A lot of time and efforts before producing the first level ...



But² ...

Adding novel features in few hours !



Stratégie d'adoption proactive

- **Création de la ligne de produits *from scratch***

- Depuis l'analyse du domaine
- Jusqu'à la dérivation des variantes logicielle

Dans la vraie vie :

- Beaucoup trop de temps / efforts avant de pouvoir vendre un produit
- Culture de la ligne de produits peu répandue
- Développement de produits indépendants / *clone-and-own*

Autres stratégies d'adoption

○ Adoption extractive

- Développement de variantes indépendantes
- Difficiles à maintenir et à faire évoluer
- **migration vers une approche ligne de produits**
 - basée sur les produits logiciels existants
 - grâce à une culture de la réutilisation

- Extraire la variabilité depuis des variantes existantes
- Identifier les caractéristiques / artefacts
- Extraire l'architecture

○ Adoption réactive

- Créer une ligne de produits à partir d'un produit
- Faire évoluer la ligne de produits en fonction des nouvelles exigences