DEPARTEMENT INFORMATIQUE
DE LA FACULTE DES SCIENCES

Ahmed Kaci, Guilhèm Blanchard et Yanis Allouch

Rapport du TER : Création d'un comparateur de codes sources annotés pour lignes de produits logiciels

HMIN201 — Travail d'Etude et de Recherche

Référent: Nicolas Hlad

2021

Table des matières

Remerciement

Nous tenons à remercier toutes les personnes qui ont contribué à la réalisation de nôtre projet et qui nous ont aidée lors de la rédaction de ce mémoire.

Nous voudrions dans un premier temps remercier, notre responsable M.Hlad, doctorant de l'équipe MAREL du LIRMM (UMR 5506 (CNRS)), pour sa patience, sa disponibilité et surtout ses judicieux conseils, qui ont contribué à alimenter notre réflexion.

Nous remercions également toute l'équipe pédagogique de l'université de Montpellier, ainsi que les intervenants professionnels responsables de notre formation, pour avoir assuré la partie théorique et pratique de celle-ci et transmis les concepts-clés fondamentales autour des lignes de produits, des designs patterns et des bad smells qui nous ont permit d'aborder ce projet de recherche et de programmation.

Résumé

Introduction générale

Première partie

Préambule

Introduction

Les domaines de recherche très célèbre tels que la réutilisation et l'analyse de domaine ont donné naissance aux lignes de produits logiciels (LPL). Et ceux afin de pouvoir construire des logiciels de manière analogue avec la construction d'objet matériels, en d'autre terme au lieu de développer chaque nouvelle application en recodant toutes les lignes de codes, il serait plus judicieux de créer un socle commun permettant de dériver chaque produit.

Ainsi, les lignes de produits logiciel ont été créé en se basant sur le fait que, d'une part, dans un domaine d'application, tout système partage des point en commun avec d'autres systèmes. D'autre part, il est possible de définir une structure de base des systèmes d'un même domaine pour l'utiliser dans une développement plus rapide de nouveaux systèmes qui seront de meilleurs qualités et garantis.

Ayant pris une place importante dans le développement logiciel, des problèmes de maintenabilité de LPL apparaissent au fur et à mesure de l'évolution des lignes de produits, d'où la nécessité d'évaluer le code de ces dernières afin de l'améliorer.

Dans ce chapitre, nous allons commencer par introduire le sujet de notre projet de travail d'étude et de recherche, qui consiste en l'implémentation d'un analyseur de variabilité pour un code annoté, puis nous énumérerons ses objectifs et quelques techniques et méthodes utilisée dans ce domaine.

Ensuite, nous présenterons les lignes de produits logiciels,leurs caractéristiques et leurs paradigmes de conception, en particulier l'approche par annotation.

Enfin, nous exposerons les problèmes de maintenabilité liée au LPLs et nous terminerons par une conclusion.

1 Présentation du sujet

Dans ce stage, nous nous intéressons à la notion de qualité de code annoté. Nous souhaitons créer un outil permettant d'évaluer la qualité de ce type de code par des métriques, afin d'émettre un score clair sur la qualité du code produit.
En outre, cela permettrait aussi de comparer deux codes annotés et améliorer les analyseurs existant. Par ailleurs la récolte d'informations sur le code compris entre les annotations permettrait d'identifier un certain nombre de métriques tels que les similarités de code entre elles.

Afin de mieux illustrer notre problématique, considérons le code Java suivant qui est un extrait d'un code annoté complet disponible en annexe 5.9, p24. Ce code sera utilisé comme exemple dans la suite du rapport, il est standard et tiré des benchmark [Mei+17], cette exemple n'est pas difficile a comprendre et bien réfléchis. En outre, nous observons quand même l'enfer des //#ifdef voir [Dei15] et [Pre19] ainsi que d'autres traits caractéristiques des LPL.

Cette exemple illustratif comporte notamment des annotations avec les traits suivants :

1. Copié / Collé du code variant (code java) pour exprimer la variabilité ;
2. Beaucoup de code annotés et donc très peu de code en commun ;
3. Annotations successives avec la même condition, à la ligne 3 et 8 ;
4. Des annotations Imbriquées qui sont :
   - Équivalentes, à la ligne 8 et 11 ;
   - Redondantes, à la ligne 8 et 14.

Généralement, les développeurs sont libre d'interpréter ces informations et peuvent effectuer eux-mêmes des changements sur le code annoté, afin d'améliorer sa maintenabilité ou compréhension.

Extrait de l'exemple du code annoté

```
...
public class MainWindow implements ITickListener,
//#if CallButtons
    ActionListener
//#endif
{
    ...
//#if CallButtons
    @Override
    public void onRequestFinished(Elevator elevator, Request request) {
//#if CallButtons
        ...
//#endif
//#if DirectedCall && CallButtons
        switch (request.getDirection()) {
            case MOVING_UP:
//#if DirectedCall && CallButtons
                listFloorComposites.get(request.getFloor()).resetUp();
//#endif
//#if DirectedCall3
                break;
//#endif
            case MOVING_DOWN:
//#if DirectedCall && CallButtons
                listFloorComposites.get(request.getFloor()).resetDown();
//#endif
//#if DirectedCall
                break;
//#endif
            default:
//#if DirectedCall
                break;
//#endif
        }
//#if DirectedCall
        ...
//#endif
//#endif
//#if UndirectedCall
        listFloorComposites.get(request.getFloor()).resetFloorRequest();
//#endif
    }
//#endif
    private void createPanelControlsContent(int maxFloors) {
        ...
//#if (( DirectedCall  &&  FloorPermission ) || ( ShortestPath  &&  UndirectedCall  &&
    FloorPermission ) || ( FIFO  &&  Service )) && ! Sabbath
        gbc_btnService.insets = new Insets(0, 0, 0, 10);
//#endif
//#if (( Service  &&  Sabbath ) || ( FloorPermission  &&  Sabbath )) && ! DirectedCall  && ! FIFO
    && ! ShortestPath  && ! UndirectedCall  && ! CallButtons
        gbc_btnService.insets = new Insets(0, 0, 0, 0);
//#endif
//#if (( Service  &&  Sabbath ) || ( FloorPermission  &&  Sabbath )) && ! DirectedCall  && ! FIFO
    && ! ShortestPath  && ! UndirectedCall  && ! CallButtons
        gbc_btnService.gridwidth = 4;
//#endif
    ...
}
```

Les objectifs

Nous nous rendons compte que, même si ce code annoté peut être totalement ou partiellement compris de manière intuitive, il est une augmentation d'un code considéré traditionnel pour faire apparaître la variabilité de la LPL. A cette égard, ce code devient donc plus complexe à manipuler pour un développeur, ce dernier, doit prendre en compte la variabilité de la ligne de produit pour comprendre le code annoté. La variabilité amène donc une nouvelle dimension à la compréhension logicielle et par conséquent à la maintenance. De ce fait, contrairement au code "traditionnel", il faut se doter de nouveau outils pour évaluer la maintenabilité et la compréhension de ce type de code augmenté.

Ainsi, les deux principaux objectifs de notre TER s'énumèrent comme suit :

1. L'évaluation de la qualité d'un code annoté, par l'utilisation de métrique ;
2. La comparaison de deux code annoté pour déterminer lequel est le plus compréhensible.

Dans le deuxième cas, cette comparaison est appliqué au travaux d'isiSPL.

Dans ce stage, nous nous intéressons à la notion de qualité d'un code annoté. Nous souhaitons créer un outil permettant d'évaluer la qualité d'un code annoté, et ainsi émettre un score clair sur la qualité du code produit. En outre, nous souhaitons également améliorer les analyseurs existant. De plus, en récoltant des informations sur le code compris entre les annotations nous pourrions identifier par exemple les similarités de code entre deux annotations.

Pour cela, nous proposons de :

1. produire un interpréteur (parseur) de code annoté ;
2. identifier ou créer des critères de qualité pour un code annoté ;
3. implémenter ces critères sous la forme de métriques ;
4. analyser le résultat de l'interpréteur vis-à-vis de ses métriques afin d'évaluer la qualité d'un code annoté ;
5. permettre la comparaison de deux codes annotés vis-à-vis de ces critères.

Techniques et méthodes : VITAL

La maintenance de lignes de produits évolué a vu naître un ensemble de problèmes, car au fur et à mesure qu'elles évoluent dans le temps, la réalisation de la variabilité devient de plus en plus complexe et contraignante. Pour remédier à ces problèmes l'outils VITAL [Zha15] (Variability ImprovemenT AnaLysis) qui permet l'extraction automatique d'un modèle de réflexion [ZB12] à partir d'un code de variabilité, et la réalisation d'analyse complémentaire [ZB14] a été développé.

Avant de décrire les différentes étapes suivit dans l'approche VITAL, nous allons définir les principaux éléments qui contribuent à la réalisation de la variabilité dans les lignes de produits logiciels.

— La variabilité (Var) : représente une caractéristique variable dans l'espace du problème ;
— Le point de variation (VP) : espace de solutions où est réalisé la variabilité pour sélectionner les codes variants (CV) ;
— Le code variant (CV) : un code variant représente le code à l'intérieur d'un point de variation ;
— Le groupe de points de variation (VPG) : ensemble de point de variation ayant un prédicat équivalent pour la sélection de fragments de code (variants). Les VPGs seront instancier par la même caractéristique variable (Var) avec une proposition équivalente.

La figure suivante illustre ces différents concepts :

Pour faire faces au problème de maintenance de code annoté, l'outils VITAL effectue une :

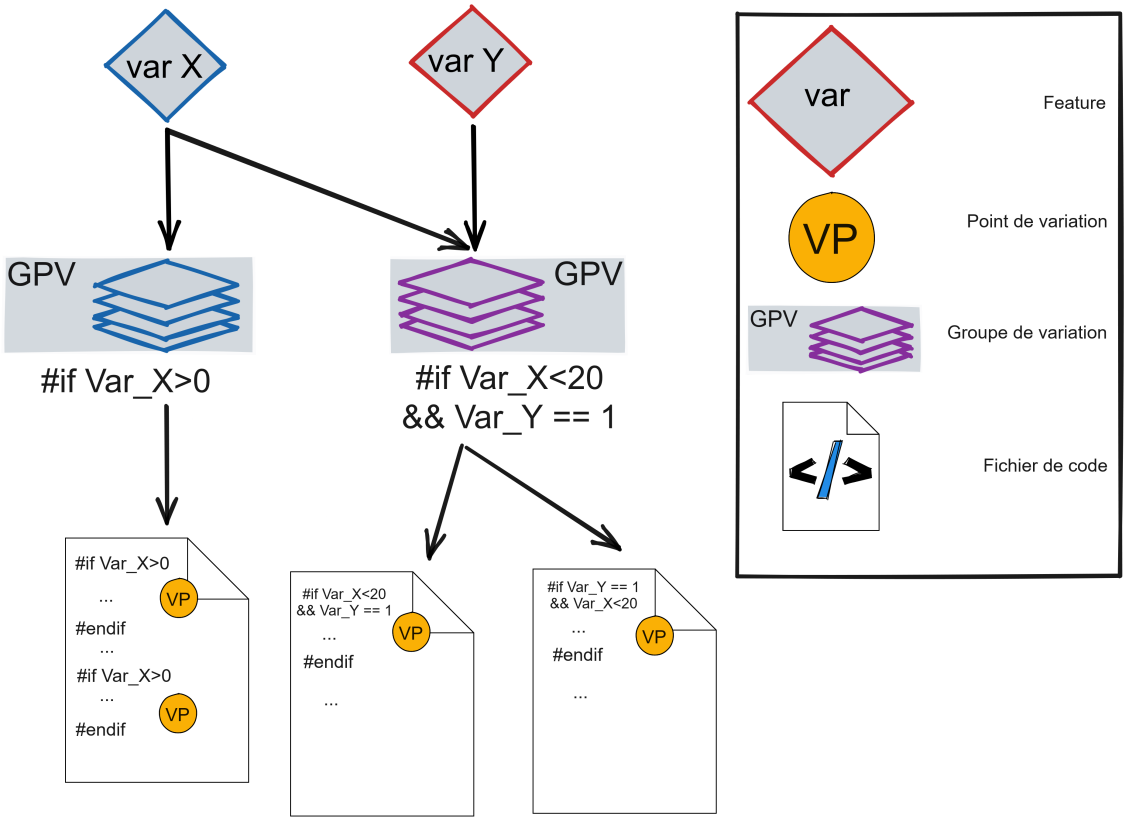


FIGURE 1 – Eléments de variabilité en Compilation Conditionnelle

1. Analyse de variabilité qui utilise différents mécanismes d'implémentation, tels que le parsing et filtrage au niveau syntaxique ;
2. Extraction d'un modèle de réflexion de variabilité qui se base, d'une part, sur l'expression des caractéristiques variables et des points de variation, d'autre part, sur l'extraction des dépendances de variabilités ;
3. Réalisation et la visualisation de mesures effectuées sur le code variabilité.

Nous devons faire remarquer la chose suivante pour terminer.

L'analyseur VITAL [Zha15] permet d'offrir des métriques sur du code C/C++ annoté selon la syntaxe C pré-processeur définissant des inclusions et des macros (*Conditional Compilation*) #ifdef, #define, etc. en analysant notamment le nombre d'annotations, leurs tailles (nombre de lignes de code qu'elles entourent) et leurs niveaux d'imbrications (s'il elles sont contenu dans une autre annotation), etc.

Il est important de noter ce détail.

En conséquence, nous avons re-conceptualiser VITAL pour les LPL. Notamment une variabilité (VAR) se retrouve exprimé par une *feature* et une *feature* est soit vrai ou fausse et ne peux pas etre tester supérieur ou égal a des entiers (cela ne fait pas de sens).
Ces métrique peuvent être améliorer en faisant des études empiriques sur des produit industriels et des extensions du modèle d'extraction, via d'autres mécanisme que la CC.

2 Présentation des lignes de produits logiciel

L'ingénierie des lignes de produits logiciels, (LPL) regroupe un ensemble de méthodes pour la conception et l'exploitation de logiciels partageant des caractéristiques communes. Cette ingénierie prône une conception des logiciels par la réutilisation, ce qui permet d'accélérer leurs développements ainsi que leurs maintenabilité.

Définition

Définition de l'article [Anq+08] :

> Les lignes de produits logiciels se basent sur les idées de personnalisation de masse (mass customization) telles qu'appliquées à la production de biens matériels, c'est-à-dire la production à grande échelle de biens conçus pour des marchés spécifiques ou des individus. La personnalisation de masse est appliquée dans plusieurs marchés comme les voitures, les appareils photos, les téléphones portables, etc. Elle fonctionne à partir d'une base commune à tous les produits qui autorise l'addition ou la modification de petites variations permettant de différencier les produits. Par exemple, dans le cas de l'industrie automobile, la base peut-être le châssis, la suspension et la transmission. A partir de cette base, on peut construire des voitures très différentes selon la carrosserie et le moteur que l'on adapte. Normalement, cette plate-forme de base est la partie la plus chère de l'ensemble ce qui offre de meilleurs bénéfices quand on peut la réutiliser dans de nombreux produits.

Les caractéristiques des lignes de produits

Les logiciels résultant d'une ligne de produit sont souvent décrit en terme de caractéristiques (features), qui décrivent des fonctionnalités logiciels perceptible par l'utilisateur.

Les logiciels d'une LPL peuvent, d'une part, partager des caractéristiques communes. D'autre part, ils différent les uns des autres en implémentant des caractéristiques qui leurs sont propres. Ainsi, deux logiciels d'une même ligne de produit sont différents, si leurs ensemble de caractéristiques est différents.

La conception des logiciels de ligne de produit repose sur la conception d'une architecture logiciels communes. Cette implémentation de la LPL, regroupe l'implémentation de l'ensemble des caractéristiques spécifiques ou communes de ses logiciels. La spécificités de l'implémentation logicielle d'une ligne de produit est qu'elle se doit d'être *variable*, afin de rendre compte de la spécificité de chaque logiciels, tout en maximisant leurs partie communes (ce qui facilitera la maintenabilité).

Les paradigmes de conception des lignes de produits logiciel

Il existe différent paradigme de conception d'une telle implémentation, notamment l'approche par annotation et l'approche par composition. Mais des deux, l'approche par annotation reste la plus appliqué de nos jours [KA08][Ale+12].

M. Hlad et son équipe développent l'outil nommé isiSPL. C'est un outil de génération de lignes de produits logiciels par incrément (de produit, voir la figure 2), qui produit des LPL *Feature-Oriented*.

Son besoin étant de pouvoir analyser les performances de son outil vis-à-vis d'une implémentation par un développeur lambda.

De ce fait, Notre recherche n'a considéré que l'approche par annotation d'une ligne de produits logicielles.

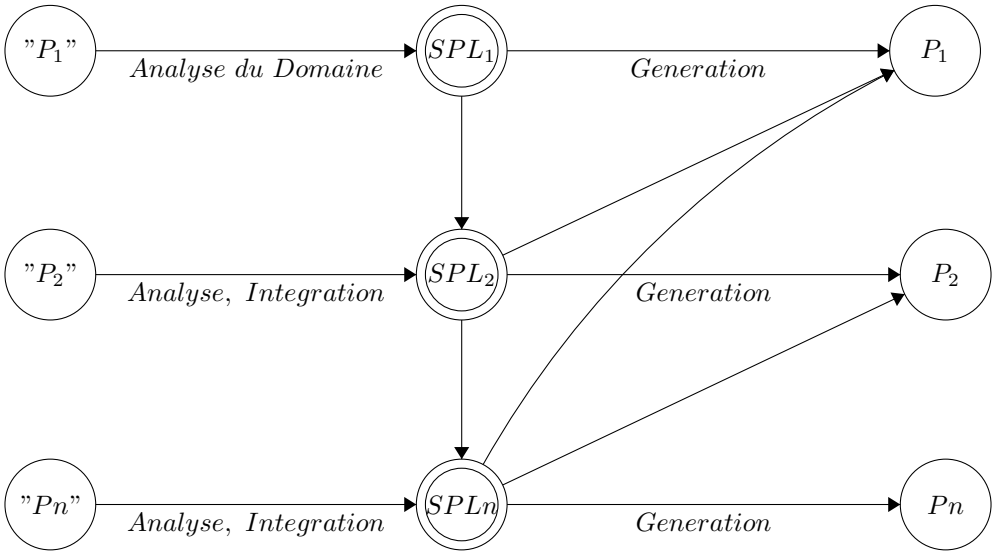


FIGURE 2 – Schématisation de l'approche de génération d'une LPL par isiSPL

Dans cette approche, il est possible d'implémenter la variabilité du système à l'intérieur même du code. En utilisant des annotations et un pré-processeur, les développeurs peuvent créer ce que nous appelons un code annoté.
Ce code peut rapidement devenir difficile à comprendre plus le nombre d'annotations augmentent. Cette difficulté est le produit de l'ajout d'une couche de compréhension qui ne fait pas partie du langage de programmation de base (par exemple Java) mais appartient au langage du preprocesseur (par exemple CPP [1]).

Présentation de l'approche par annotation

Une annotation se compose d'au moins deux parties distinctes :

1. L'annotation
   a) débute par un commentaire *//#if*
      i. Le premier commentaire est suivit d'une proposition logique syntaxiquement correcte éventuellement parenthésé. Cette proposition est composer :
         A. de variables (vu comme des propositions logiques), faisant référence au noms définis de chaque caractéristique dans le feature model d'une LPL ;
         B. des opérateurs logique traditionnelle accessible en programmation ; and, or, not .
   b) ce termine par un commentaire *//#endif*.
2. Le code variant
   a) est un code syntaxiquement correcte dans le langage de programmation cible (java dans note cas) ;
   b) peut posséder plusieurs annotations.

Le code annoté présenté en introduction page 7 présente les points précédents.
Ce code exprime la variabilité des features suivantes :

— DirectedCall
— CallButtons
— DirectedCall3
— UndirectedCall

1. C Pre-Processor. A ne pas confondre avec l'extension homologue désignant le langage C++

— FloorPermission
— ShortestPath
— FIFO
— Sabbath

Chacune caractérisant une fonctionnalité final voulue d'un produit.
Chaque partie de code commençant par *//#if* et se terminant par *//#endif* délimite le début et la fin d'une annotation. Exemple :

```
1 //#if CallButtons
2     if(btnFloor.isSelected()) {
3         btnFloor.setSelected(false);
4         btnFloor.setEnabled(true);
5     }
6 //#endif
```

Elle peut contenir zéro ou plusieurs annotations. Exemple :

```
//#if CallButtons
    @Override
    public void onRequestFinished(Elevator elevator, Request request) {
//#if DirectedCall
        switch (request.getDirection()) {
            case MOVING_UP:
//#if DirectedCall
                listFloorComposites.get(request.getFloor()).resetUp();
//#endif
...
//#endif
    }
//#endif
```

Ainsi que zéro ou plusieurs ligne de code. Exemple :

```
//#if DirectedCall
//#endif
```

Un preprocesseur génère un produit suivant les valeurs logiques de chaque variables composant la LPL.

Admettons que nous souhaitons un produit composé d'aucune feature. Alors, il faudra donné dans un fichier de configuration «produit_1.xml» [2] la valeur logique *false* à toutes les variables de la LPL.

Nous avons illustré en page 16, un extrait du code généré par FeatureIDE de la configuration précédente.

Cette génération du code source est automatique et ne nécessite pas d'action de la part du développeur.

Dans le cadre des preprocesseurs implémentés dans FeatureIDE (Munge/Antenna), ils peuvent aussi prendre en compte des contraintes sur les features liés a l'analyse du domaine.
Ces contraintes figure dans le *feature model*.
Une contrainte peut-être :

1. déclarant la nécessité ;
2. une implication ;
3. une co-existence entre features : A OR B, c'est à dire que A et B peuvent exister ensemble ;
4. une exclusion mutuelle entre features : A XOR B, c'est à dire que A et B ne peuvent pas co-exister ensemble ;
5. une dépendance logique : B => A, c'est à dire que B dépend de A .

En figure 3 page 14 le feature model complet de la LPL dont notre exemple de code est tiré. Vous remarquez qu'il n'est pas possible de créer un produit ne disposant pas de feature. Enfin, une caractéristique intéressantes des annotations est qu'elle est peut-être présentes partout dans une LPL et a plusieurs niveau d'abstractions, il n'y a aucune restriction de ce point de vue là.
Par exemple, la figure 4 schématise une LPL sur 4 fichiers et une représentation disparate des features dans chacune.

2. Remarque : nous nous intéressons pas de savoir où est stocké cette valeur ni comment parce qu'elle ne fait pas partie des fichiers résultat produit par isiSPL a ce jour.

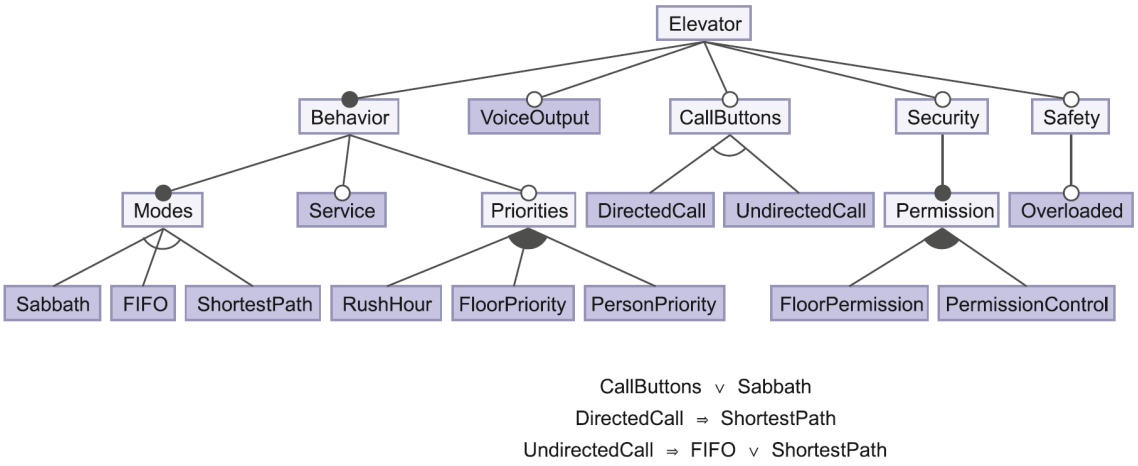


FIGURE 3 – Feature model complet de la LPL Elevator tiré de [Mei+17]

Avec peu de contrainte, cette approche bénéficie d'une forte capacité d'expression de la variabilité.
Cependant, comme dirait Peter Parker, un grand pouvoir implique de grandes responsabilités.

Par ailleurs, comme il est expliqué dans cette article de recherche [Dei15], [Mei+17] ainsi que [Mar09] (heuristique G6), l'imbrication des *//#if* est un *enfer* et l'entrelacement des niveaux d'abstraction est à éviter autant que possible !

Pour le moment, notre hypothèse sur ce que pourrait-être un bon code annoté est basé sur les critères suivants :

1. Réduire le nombre d'annotation, c'est-à-dire augmenter le code commun ;
2. Éviter de les imbriquer ;
3. Réduire leur éparpillement sur la LPL ;
4. Simplifier les propositions logiques (réduire le nombre de variables utilisés).

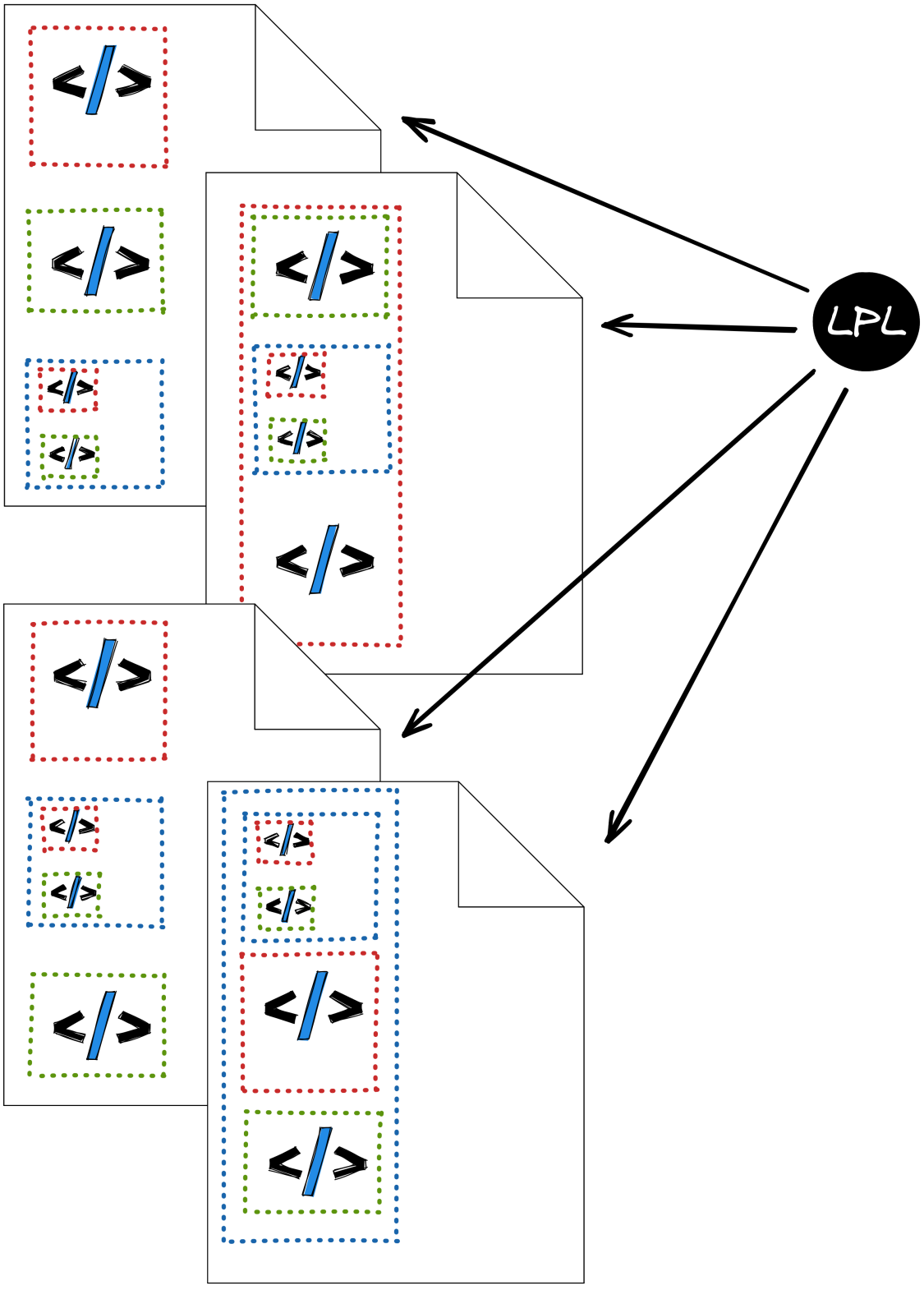


FIGURE 4 – Modélisation sur plusieurs fichiers d'une LPL quelconque

```
public class MainWindow implements ITickListener

{
 ...
    private void clearPresent() {
        for (FloorComposite fl : listFloorComposites) {
            fl.showElevatorNotPresent();
        }
    }

    public void onTick(Elevator elevator) {
        ElevatorState state = elevator.getCurrentState();
        int currentFloor = elevator.getCurrentFloor();
        switch (state) {
        case MOVING_UP:
                    this.listFloorComposites.get(currentFloor - 1).showImageClose();
            break;
        case MOVING_DOWN:
                    this.listFloorComposites.get(currentFloor + 1).showImageClose();
            break;
        case FLOORING:
                    this.listFloorComposites.get(currentFloor).showImageOpen();
            break;
        }
        this.clearPresent();
        this.listFloorComposites.get(currentFloor).showElevatorIsPresent();
    }

    public void initialize(int maxFloors) {
        if(frmElevatorSample != null) {
            return;
        }
        frmElevatorSample = new JFrame();
        frmElevatorSample.setTitle("Elevator Sample");
        frmElevatorSample.setBounds(100, 50, 900, 650);
        frmElevatorSample.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        createBaseStructure();
        createPanelControlsContent(maxFloors);
        addBuilding(maxFloors);
        frmElevatorSample.setVisible(true);
    }
    ...
}
```

3 Problème de maintenabilité du code annoté

La maintenabilité d'un code annoté peut vite devenir complexe.
En effet, les annotations liées aux différentes fonctionnalités peuvent présenter un très haut niveau d'interdépendance.
De plus, celles liés à une fonctionnalité donnée peuvent se trouver distribuées dans un grand nombre de fichiers. Ainsi, le besoin de mettre à jour une méthode pour une fonctionnalité ne devrait pas impacter le code du reste de la LPL. Au contraire, nous devons être certains qu'une modification sur une section commune soit bien mise à jour pour tous les produits.
Celas nécessite donc d'avoir une LPL bien conçue, en évitant notamment les redondances de code inutile. Notre socle commun devrait se trouver hors de toute annotation et non copiée-collée à différents emplacements. Au contraire, une section de code avec un niveau d'annotation imbriqué très élevé nécessitera plus d'effort de réflexion de la part du développeur qui parcourt le code afin de déterminer quel seras la condition d'utilisation de son code.
Par conséquent, nous cherchons à remédier à ces problèmes en fournissant différentes métriques d'évaluation de LPL qui permettrai de détecter ces situations problématiques systématiquement.

Par exemple, duplication de code, fort niveau d'imbrication, etc... pour permettre au développeur de remanié [3] si cela est possible.

Conclusion

Dans ce premier chapitre, nous avons commencé par une brève introduction des LPLs, puis présenté le sujet de notre TER, les objectifs visé dans ce dernier et la méthode VITAL qui fait partie de son état de l'art.

En suite, nous avons effectué une présentation des lignes de produits logiciels dans laquelle nous avons défini cette technique du génie logiciel, présenté ses caractéristiques et ses paradigmes de conception, où nous avons développé de manière très explicite l'approche par annotation qui est l'approche suivi dans notre projet.

Enfin, nous avons exposer les problèmes de maintenabilité des LPL qui constituent la source de la problématique du sujet traité dans ce Travail d'Etude et de Recherche.

3. En lieu et place du terme "refactor".

Deuxième partie

Développement

4 Analyse et conception

4.1 Introduction

4.2 Explication des données

Données en entrées

Données en sorties

4.3 Diagrammes

De classe de l'application

De classe de l'application

4.4 Conclusion

5 Réalisation

5.1 Introduction

5.2 Présentation des technologies utilisées

Lorem ipsum dolor est

5.3 Nos résultats

5.4 Captures d'écran

5.5 Conclusion

Conclusion générale, limites et perspectives

lorem ipsum dolor est

Troisième partie

Annexe

5.6 Configuration

5.7 Tutoriel

5.8 Le bestiaires des abréviations

5.9 Code annoté complet

```
1  //#if FloorPermission
2  import java.util.Arrays;
3  //#endif
4  //#if Service
5  import java.awt.Dimension;
6  //#endif
7  //#if CallButtons
8  import de.ovgu.featureide.examples.elevator.core.controller.Request;
9  //#endif
10 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
       Sabbath ))
11 import javax.swing.JToggleButton;
12 //#endif
13 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
       Sabbath ))
14 import java.awt.event.ActionEvent;
15 //#endif
16 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
       Sabbath ))
17 import java.awt.event.ActionListener;
18 //#endif
19 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
       Sabbath ))
20 import java.awt.GridLayout;
21 //#endif
22 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
       Sabbath ))
23 import de.ovgu.featureide.examples.elevator.sim.SimulationUnit;
24 //#endif
25 public class MainWindow implements ITickListener,
26 //#if CallButtons
27     ActionListener
28 //#endif
29 {
30     private JFrame frmElevatorSample;
31     private JSplitPane splitPane;
32     private JLabel lblEvent;
33     private List<FloorComposite> listFloorComposites = new ArrayList<>();
34 //#if CallButtons
35     private List<JToggleButton> listInnerElevatorControls = new ArrayList<>();
36 //#endif
37 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
       Sabbath ))
38     private SimulationUnit sim;
39 //#endif
40 //#if (( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) ||
       DirectedCall || FIFO || ( FloorPermission  &&  Sabbath ) || ( Service  &&
```

```
    Sabbath ))
41      public MainWindow(SimulationUnit sim) {
42          this.sim = sim;
43      }
44  //#endif
45      private void clearPresent() {
46          for (FloorComposite fl : listFloorComposites) {
47              fl.showElevatorNotPresent();
48          }
49      }
50  //#if CallButtons
51      @Override
52      public void onRequestFinished(Elevator elevator, Request request) {
53  //#if DirectedCall
54          switch (request.getDirection()) {
55              case MOVING_UP:
56  //#if DirectedCall
57                  listFloorComposites.get(request.getFloor()).resetUp();
58  //#endif
59  //#if DirectedCall3
60                  break;
61  //#endif
62              case MOVING_DOWN:
63  //#if DirectedCall
64                  listFloorComposites.get(request.getFloor()).resetDown();
65  //#endif
66  //#if DirectedCall
67                  break;
68  //#endif
69              default:
70  //#if DirectedCall
71                  break;
72  //#endif
73          }
74  //#endif
75  //#if UndirectedCall
76          listFloorComposites.get(request.getFloor()).resetFloorRequest();
77  //#endif
78      }
79  //#endif
80      private void createPanelControlsContent(int maxFloors) {
81          JPanel panel_control = new JPanel();
82          try {
83              panel_control = new JBackgroundPanel(MainWindow.class.
    getResourceAsStream("/elevator_inside2.png"));
84          } catch (IOException e) {
85              e.printStackTrace();
86          }
87          splitPane.setRightComponent(panel_control);
88          GridBagLayout gbl_panel_control = new GridBagLayout();
89          panel_control.setLayout(gbl_panel_control);
90          lblEvent = new JLabel("");
91          lblEvent.setFont(new Font("Tahoma", Font.BOLD, 15));
92          lblEvent.setForeground(Color.WHITE);
93          lblEvent.setHorizontalAlignment(SwingConstants.CENTER);
94          GridBagConstraints gbc_lbl = new GridBagConstraints();
```

```
95          gbc_lbl.gridwidth = 4;
96          gbc_lbl.insets = new Insets(0, 0, 185, 0);
97          gbc_lbl.fill = GridBagConstraints.HORIZONTAL;
98          gbc_lbl.gridx = 0;
99          gbc_lbl.gridy = 0;
100         panel_control.add(lblEvent, gbc_lbl);
101 //#if Service
102         JToggleButton btnService = new JToggleButton("Service");
103 //#endif
104 //#if Service
105         btnService.setMinimumSize(new Dimension(80, 30));
106 //#endif
107 //#if Service
108         btnService.setPreferredSize(new Dimension(80, 30));
109 //#endif
110 //#if Service
111         btnService.setMaximumSize(new Dimension(80, 30));
112 //#endif
113 //#if Service
114         GridBagConstraints gbc_btnService = new GridBagConstraints();
115 //#endif
116 //#if (( DirectedCall  &&  FloorPermission ) || ( ShortestPath  &&
        UndirectedCall  &&  FloorPermission ) || ( FIFO  &&  Service )) && !
        Sabbath
117         gbc_btnService.insets = new Insets(0, 0, 0, 10);
118 //#endif
119 //#if (( Service  &&  Sabbath ) || ( FloorPermission  &&  Sabbath )) && !
        DirectedCall  && ! FIFO  && ! ShortestPath  && ! UndirectedCall  && !
        CallButtons
120         gbc_btnService.insets = new Insets(0, 0, 0, 0);
121 //#endif
122 //#if (( Service  &&  Sabbath ) || ( FloorPermission  &&  Sabbath )) && !
        DirectedCall  && ! FIFO  && ! ShortestPath  && ! UndirectedCall  && !
        CallButtons
123         gbc_btnService.gridwidth = 4;
124 //#endif
125 //#if Service
126         gbc_btnService.fill = GridBagConstraints.HORIZONTAL;
127 //#endif
128 //#if Service
129         gbc_btnService.gridx = 0;
130 //#endif
131 //#if Service
132         gbc_btnService.gridy = 4;
133 //#endif
134 //#if Service
135         panel_control.add(btnService, gbc_btnService);
136 //#endif
137 //#if Service
138         btnService.addActionListener(new ActionListener() {
139             @Override
140             public void actionPerformed(ActionEvent e) {
141                 sim.toggleService();
142                 if (sim.isInService()) {
143                     setEventLabel("Service-Mode!", Color.ORANGE);
144                 } else {
```

```
145                     setEventLabel("", Color.WHITE);
146                 }
147             }
148         });
149 //#endif
150 //#if CallButtons
151         JPanel panel_floors = new JPanel(new GridLayout(0,3));
152 //#endif
153 //#if CallButtons
154         panel_floors.setBackground(Color.GRAY);
155 //#endif
156 //#if CallButtons
157         JToggleButton btnFloor;
158 //#endif
159 //#if CallButtons
160         for (int i = maxFloors; i >= 0; i--) {
161             btnFloor = new JToggleButton(String.valueOf(i));
162             btnFloor.setActionCommand(String.valueOf(i));
163             btnFloor.addActionListener(this);
164 //#if (( DirectedCall  &&  FloorPermission ) || ( ShortestPath  &&
    UndirectedCall  &&  FloorPermission )) && ! FIFO  && ! Sabbath
165             btnFloor.setEnabled(sim.isDisabledFloor(i));
166 //#endif
167             panel_floors.add(btnFloor);
168             listInnerElevatorControls.add(0, btnFloor);
169         }
170 //#endif
171 //#if CallButtons
172         GridBagConstraints gbc_btnFloor = new GridBagConstraints();
173 //#endif
174 //#if CallButtons
175         gbc_btnFloor.insets = new Insets(0, 0, 0, 0);
176 //#endif
177 //#if CallButtons
178         gbc_btnFloor.fill = GridBagConstraints.BOTH;
179 //#endif
180 //#if CallButtons
181         gbc_btnFloor.gridwidth = 4;
182 //#endif
183 //#if CallButtons
184         gbc_btnFloor.gridx = 2;
185 //#endif
186 //#if CallButtons
187         gbc_btnFloor.gridy = 4;
188 //#endif
189 //#if CallButtons
190         panel_control.add(panel_floors, gbc_btnFloor);
191 //#endif
192     }
193     public void onTick(Elevator elevator) {
194         ElevatorState state = elevator.getCurrentState();
195         int currentFloor = elevator.getCurrentFloor();
196         switch (state) {
197         case MOVING_UP:
198                     this.listFloorComposites.get(currentFloor - 1).
    showImageClose();
```

```
199             break;
200         case MOVING_DOWN:
201                     this.listFloorComposites.get(currentFloor + 1).
    showImageClose();
202             break;
203         case FLOORING:
204                     this.listFloorComposites.get(currentFloor).showImageOpen();

205 //#if CallButtons
206             JToggleButton btnFloor = listInnerElevatorControls.get(
    currentFloor);
207 //#endif
208 //#if CallButtons
209             if(btnFloor.isSelected()) {
210                 btnFloor.setSelected(false);
211                 btnFloor.setEnabled(true);
212             }
213 //#endif
214             break;
215         }
216                 this.clearPresent();
217         this.listFloorComposites.get(currentFloor).showElevatorIsPresent();
218     }
219 //#if CallButtons
220     @Override
221     public void actionPerformed(ActionEvent e) {
222 //#if DirectedCall
223         sim.floorRequest(new Request(Integer.valueOf(e.getActionCommand()),
    ElevatorState.FLOORING));
224 //#endif
225 //#if UndirectedCall
226         sim.floorRequest(new Request(Integer.valueOf(e.getActionCommand())));
227 //#endif
228         listInnerElevatorControls.get(Integer.valueOf(e.getActionCommand())).
    setEnabled(false);
229     }
230 //#endif
231     public void initialize(int maxFloors) {
232         if(frmElevatorSample != null) {
233             return;
234         }
235         frmElevatorSample = new JFrame();
236         frmElevatorSample.setTitle("Elevator Sample");
237         frmElevatorSample.setBounds(100, 50, 900, 650);
238         frmElevatorSample.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
239 //#if FloorPermission
240         FloorChooseDialog permissionDialog = new FloorChooseDialog(maxFloors,
    Arrays.asList(0), "Choose disabled floors");
241 //#endif
242 //#if FloorPermission
243         List<Integer> disabledFloors = permissionDialog.getSelectedFloors();
244 //#endif
245 //#if FloorPermission
246         sim.setDisabledFloors(disabledFloors);
247 //#endif
248 //#if FloorPermission
```

```
249         permissionDialog.dispose();
250 //#endif
251         createBaseStructure();
252         createPanelControlsContent(maxFloors);
253         addBuilding(maxFloors);
254         frmElevatorSample.setVisible(true);
255     }
256     private void addBuilding(int maxFloors) {
257         JPanel panel_building = new JPanel();
258         GridBagLayout layout = new GridBagLayout();
259         panel_building.setLayout(layout);
260         JScrollPane scrollPane = new JScrollPane(panel_building);
261         scrollPane.setVerticalScrollBarPolicy(ScrollPaneConstants.
    VERTICAL_SCROLLBAR_ALWAYS);
262         scrollPane.setHorizontalScrollBarPolicy(ScrollPaneConstants.
    HORIZONTAL_SCROLLBAR_NEVER);
263         scrollPane.getVerticalScrollBar().setUnitIncrement(10);
264         GridBagConstraints gbc = new GridBagConstraints();
265         gbc.insets = new Insets(0, 0, 0, 0);
266         gbc.fill = GridBagConstraints.BOTH;
267         gbc.gridx = 2;
268         gbc.gridy = 0;
269         gbc.anchor = GridBagConstraints.SOUTH;
270         for (int i = maxFloors; i >= 0; i--) {
271 //#if DirectedCall
272             FloorComposite floor = new FloorComposite(i == 0, i, sim, i ==
    maxFloors);
273 //#endif
274 //#if (FIFO || ( ShortestPath  &&  UndirectedCall  &&  FloorPermission ) || (
    FloorPermission  &&  Sabbath )) && ! DirectedCall
275             FloorComposite floor = new FloorComposite(i == 0, i, sim);
276 //#endif
277 //#if Sabbath && ! DirectedCall  && ! FIFO  && ! FloorPermission  && !
    ShortestPath  && ! UndirectedCall  && ! CallButtons
278             FloorComposite floor = new FloorComposite(i == 0, i);
279 //#endif
280             layout.setConstraints(floor, gbc);
281             gbc.gridy += 1;
282             panel_building.add(floor);
283             listFloorComposites.add(0, floor);
284         }
285                 splitPane.setLeftComponent(scrollPane);
286     }
287     public void setEventLabel(String text, Color color) {
288         if(lblEvent != null) { //1
289             lblEvent.setText(text);
290             lblEvent.setForeground(color);
291         }
292     }
293     private void createBaseStructure() {
294         JPanel contentPane = new JPanel();
295         contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
296         contentPane.setLayout(new BorderLayout(0, 0));
297         frmElevatorSample.setContentPane(contentPane);
298         splitPane = new JSplitPane();
299         splitPane.setResizeWeight(0.5);
```

```
300         contentPane.add(splitPane, BorderLayout.CENTER);
301     }
302 }
```

5.10 Feuille de route

Le Contexte

M. Hlad, doctorant de l'équipe MAREL du LIRMM (UMR 5506 (CNRS)), développe l'outil **isiSPL** qui génère une ligne de produit par incrément. Sous sa supervision et collaboration, nous allons développé un outil qui évaluera la qualité des annotations générées par **isiSPL** pour la comparer avec une annotation développée a la main. Nous avons été introduits la méthode **VITAL** publiée en 2014.

Le nécessaire

- Nom du groupe : TçaPöt
- Noms, prénoms, mails des membres :
  1. Allouch, Yanis, yanis.allouch@etu.umontpellier.fr
  2. Blanchard, Guilhèm, guilhem.blanchard@etu.umontpellier.fr
  3. Kaci, Ahmed, ahmed.kaci@etu.umontpellier.fr
- Listes des documents a lire :
  - [Mei+17]
  - [Zha15]
  - [ZB14]
  - [Ale+12]
  - [ZB12]
  - [KA08]
  - [WB09]
- **Gantt**, voir la pièce jointe [Planning_TER] à ouvrir avec **ProjectLibre**.

Les objectifs

- Phase **Analyse et conception** du projet.
  - Identification des bonnes pratiques utilisées pour la réalisation de code annoté.
  - Élaboration de métriques dont le but est d'identifier des annotations de qualité et qui nous aideront pour classer des projets annotés.
  - Diagrammes UML pour la phase de développement.
- Phase **Développement** du projet.
  - Implémentation des critères d'évaluation et de comparaison de l'approche VITAL afin de lui ajouter de nouvelles métriques, plus spécifique à l'équipe MAREL.
  - Implémentation d'un logiciel basé sur la réalisation précédente en Java ou en Python (à étudier). Dans le but d'avoir un outil portable et exécutable sur toutes plates-formes.
  - Nous testerons notre outil sur des lignes de produits réalisé à la main et généré par **isiSPL**. Nous réaliserons ces tests progressivement, en premier lieu sur la première métrique implémentée, puis à chaque métrique ajoutée pour mesurer notamment leur complexité temporelle.
- Étudier d'éventuelles utilisations des designs patterns dont la mise en œuvre serait intéressante lors de la création d'un code avec des annotations dans le cadre de la construction d'une ligne de produits logiciels. (Perspective)

5.11 Synthèse et Analyse de la méthode VITAL et de ses métrique

Avec l'évolution des lignes de produits, des problèmes de maintenance sont apparus, car au fur sont à mesure qu'elles évoluent dans le temps la réalisation de la variabilité deviennent de plus en plus complexes.

L'outil VITAL (Variability ImprovemenT AnaLysis) qui permet d'analyser ce problème, est développé pour extraire automatiquement un modèle de réflexion de variabilité à partir d'un code annoté et mené des analyses complémentaires automatiques. Ainsi, cet outil :

1. Analyse le code de variabilité qui utilise différents mécanismes d'implémentation (par exemple l'algorithme naïf de parsing CPP figure 3 et 4, voir TypeChef pour un outil de parsing plus évolué) et extrait un modèle de réflexion de variabilité dans une base de données (Voir figure 5 : méta modèle de réflexion.).
2. Ensuite, différentes mesures de code de variabilité peuvent être effectuées en exécutant des requêtes de base de données (quelles requêtes ?).
3. En outre, en se basant sur la BDD MS Access, une analyse et une visualisation supplémentaires du code sont effectuées pour pouvoir être utilisé par des outils d'analyse standard tels que MS Excel, R et Treeviz.

Nous présentons les concepts de base sur lesquels se repose la méthode.

— La variabilité (VAR), représente une caractéristique variable dans l'espace du problème, la variable de l'annotation #if.

— Le point de variation (PV) : espace de solutions où est réalisé VAR pour sélectionner les variantes de code (CV).

— Le code variant (CV), le code définit dans un bloc annoté appartenant à un PV.

— Groupe de points de variation (GPV) : ensemble de PV ayant un prédicat logiquement équivalent pour la sélection de fragments de codes.

— Les GPVs seront instanciés par la même caractéristique variable (VAR) avec la même logique.

Voici l'équivalence entre le vocabulaire français et Anglais pour des références futures.

Concept	Anglais	Raccourçi FR	Raccourçi EN
Variabilité	Variability	VAR	VAR
Point de Variation	Variation Point	PV	VP
Code Variant	Code Variant	CV	CV
Groupe de Points de Variation	Variation Point Group	GPV	VPG

Nous avons pu extraire les métriques de l'article suivant [ZB14].

— VP Nesting Degree ➔ Degré/Niveau d'imbrication d'un Point de Variation. (V.P. <=> P.V.)

— Var Tangling Degree ➔ Plus une variable est utilisé dans un VP, plus elle est emmêlée dans le code (à éviter ?)

— Var Fan-out on VPG ➔ Il s'intéresse à la présence d'une variable (ou bien de l'ensemble des variables ?) dans un VPG et VITAL fait la moyenne de la dispersion, de la présence, de la répartition a travers le VPG.
➔ autre façon de voir ?

Var Fan-out on VPG ➔ Il s'intéresse à la présence d'une variable (ou bien de l'ensemble des variables ?) dans un VPG et VITAL fait la moyenne du nombre de variable apparu par apport au nombre de VP du VPG.

— Var Fan-out on File ➔ Il s'intéresse à la même métrique que ci dessus mais sur un fichier.

— Var Fan-in on File ➔ Il répertorie le nombre de Vars dans un fichier (le faire sur tous les fichiers ?)

— VP Fan-in on File ➔ Il compte le nombre de VP dans un fichier donné et il sauvegarde dans la BDD pour pouvoir faire des rquêtes. (Deux VP équivalents dans un même fichier feraient augmenté sa présence ?)

— Le nombre fichiers possédants des points de variations.

— Nombre de ligne (LOC) de code annoté.

Question pour N. Hlad

— Qu'est ce qu'un modèle de réflexion ?

— Est ce que les macros telles que #if, #ifndef et #elif fonctionne aussi sur FeatureIDE ?

— Dans un code **filtrer** peut-on proposer l'outil de transformation de code source en code filtré avec une syntaxe suivante

```
#if
// @LOC 7
// @paramXYZ valueFooBar
#if X > 10
// @func true
// @funcArgsNumber 5
// @md5 @line X == d465fg44s654465465496874fdg9684h65
// @md5 @line X+1 == 4984q324d65tujh796584796eryt7i98uk
// (par exemple pour faire la metrique correspondance de code entre deux VP)
// imaginez une syntaxe/structure/standard pour analyser la metrique sur du code filtre.
#endif
#endif
```

— Problème de collusion avec le md5 ? de sécurité ? voir d'autre protocole.

— Quels convention de nommage est utilisé par isiSPL ? Y 'a t-il une spécification pour la génération de nom de feature ?

— Si le tangling diminue est ce que cela veut dire que le nombre de variable diminue (?) Dans ce cas est ce que le code est plus facile à lire/prendre en main, ou alors le code possède toujours autant de variable mais moins éparpillé et emmêlée entre elle et donc même conclusion ?.

— Pouvons-nous appliquer le tangling aux VPG ? une VPG = ensemble de VP Équivalent qui contient déjà la Var ?

Idées

— B. Zhang introduit un outil plus évolué pour parser du code source : TypeChef.
Une rapide recherche nous fait penser que nous pouvons récupérer les recommandations de leur analyse de type checking variabilité comme métriques (parce qu'un code faux, est un mauvais code) ?

5.12 Explication du prototype du 14 février 2021

Annotation

Dans notre code nous avons créé une classe *Annotation* qui contient comme attributs :

— Le chemin vers le fichier de l'annotation.

— Le prédicat de l'annotation.

— Les variables du prédicat.

— La ligne du ficher où commence l'annotation.

— Le degré d'imbrication.

— Le nombre de ligne et de caractères protégé par l'annotation.

Application

La classe principale *Application* possèdes des attributs statiques :

— Le classpath pour connaître son chemin d'exécution.

— Une string pour cibler le fichier d'input, un second pour le fichier d'output (à défaut "out/output").

— Une liste de string qui contient le code parser ligne par ligne.

— Une liste d'annotation qui accueil les annotations de l'analyse.

— L'annotation courante.

— Une pile d'annotation utilisé par l'algorithme récursif de parcours du code parser.

La classe principal de l'application possède deux méthodes principales en plus de la méthode Main().

— *readFile()* lis le contenu d'un fichier et insère chaque ligne dans un élément d'une liste.

— *creerArborescenceDesAnnotations()* méthode récursive qui permet de créer l'arborescence des annotations en instanciant des annotations au fur et a mesure de leur lecture.

Analyse

L'invariant de notre algorithme de parcours est le suivant :

— L'indice du curseur de parcours de la liste croit avec chaque appel récursif. Après N appel récursif notre algorithme s'arrête.

5.13 Bilan à mi-parcours

Le Contexte

M. Hlad, représentant l'équipe MAREL du LIRMM (UMR 5506 (CNRS)), développe l'outil **IsiSPL** qui génère une ligne de produit par incrément. Sous sa supervision et collaboration, nous allons développer un outil qui évaluera la qualité des annotations générées par **IsiSPL** pour la comparer avec une annotation développée a la main. Nous avons été introduits la méthode **VITAL** publiée en 2014.

Le nécessaire

— Nom du groupe : TçaPöt

— Noms, prénoms, mails des membres :

1. Allouch, Yanis, yanis.allouch@etu.umontpellier.fr
2. Blanchard, Guilhèm, guilhem.blanchard@etu.umontpellier.fr
3. Kaci, Ahmed, ahmed.kaci@etu.umontpellier.fr

— Listes des documents lus :

- [Mei+17]
- [Zha15]
- [ZB14]
- [ZB12]
- [WB09]
- [Mar09]

Le bilan

Elles ont été faites ensembles, a raison 4h à 6h par jour, deux fois par semaines depuis le 5 février.

— Tâches effectuées avec succès.

- Maintien d'un carnet de bord à jour jusqu'à la réunion précédent ce bilan.
- Lecture total ou partiels des articles cités ci-dessus.
- Identification des bonnes pratiques utilisées pour la réalisation de code annoté.
- Diagrammes UML pour la phase de développement.
- Prototype de traitement des annotations d'un dossier à traiter.

— Problèmes rencontré.

- Comment concevoir une métrique adaptée a nos besoins ?

— Tâches à effectuer avant la fin du TER.

- Implémentation des critères d'évaluation et de comparaison de l'approche VITAL afin de lui ajouter de nouvelles métriques, plus spécifique à l'équipe MAREL.
- Élaboration de métriques dont le but est d'identifier des annotations de qualité et qui nous aideront pour classer des projets annotés.

5.14 Rendez-vous

Le premier contact

➔ Le 25 novembre 2020 a 17h30-19h.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Guilhèm Blanchard, Yanis Allouch.

Objectifs

— Début du TER après la période d'examen, fin Janvier 2021.

Nous, Allouch Yanis et Blanchard Guilhèm avons échangés pour la première fois pour poser les bases avec Hlad Nicolad, alors doctorant au LIRMM au sein de l'équipe MAREL [4]. Ce premier contact consiste à convenir des termes du sujet par la suite soumis à Lafourcade Mathieu, responsable de l'organisation sous-jacente du module.

https://www.lirmm.fr/users/utilisateurs-lirmm/nicolas-hlad http://www.lirmm.fr/~lafourcade/TERM1/index.php

Nous avons noté quelques points important concernant les objectifs qui sont les suivants :

— La dualité optimal recherché est la suivante : maximiser le code annoté commun, tout en minimisant le code entre annotation.

— Un outil développé par Nicolas H. : izySPL promet avec comme entrée différentes variations d'un même code, une sortie etant un code rassemblant les features de chacune des variations en utilisant les annotations.

— Définir des critères de qualités. Qu'est ce qu'une bonne annotations ?

— Qu'elle limite nous pouvons poser sur la granularité ?

— Quel métrique pouvons-nous associer au critère de qualités ?

— Quelques techniques ou heuristiques pouvons-nous appliquer pour calculer les métriques ?

— Analyse de la variabilité dite *embarquée* : état de l'art conclue ➔ 0.

— Peut-etre le parser n'est pas la solution mais l'utilisation de reconnaissance de mot via des algorithmes de deeplearning, apprentissage etc ? Le problème : trouver, générer ces codes.

— Le problème sur l'absence d'AST-variable [5]

— **Le corps du TER** : consiste en l'implémentation de méthode **VITAL** (voir bibliographie).

Pré-rendu de la feuille de route du 10 février 2021

➔ Le 27 janvier à 9h30-11h30.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

— Faire la feuille de route.

---

4. Models And Reuse Engineering, Languages
5. Nicolas devait nous fournir un papier la dessus.

— Pour le 10 février, rendre 1 page présentant les objectifs (être exhaustif), le planning prévisionnelle (**Gantt**), l'état de l'art.

— Rendu du code le 15 mai (provisoire), pas de présentation en juin [6], fin-avril a début-mail consacré au rapport.

— Les familles de produit, constitué de produit variants, utilisent des features modèles (arbres de caractéristiques) qui servent à générer des configurations valides == produit.

— Feature Oriented Domain Analyses (**FODA**), Feature Oriented Développement (**FOD**), Feature Oriented Programming (**FOP**).

— Une Feature peut être répartie sur plusieurs classe.

— FeatureIDE possèdent des plugins : **A-HEAD**, **FeatureHouse**, **FSTComposer** qui sont différentes approches par composition pour l'implémentation des LPL différentes de l'approche par annotation vu en cours et appliqué pour le projet.

— Différentes méthodes de conception : par annotation, par composition, par aspect, par trait.

— Nous retrouvons des outils pour l'aide a la conception des LPL, **FeatureIDE**, **PureVariant**, **Gears**. Cela dis FeatureIDE est dominant parce qu'il est majoritairement utilisé par les chercheurs. Les deux autres sont résilient du a leur utilisation historique par des géants de l'industrie comme **Toyota** ou **Volkswagen** mais risque de mourir s'ils ne se renouvellent pas [7].

— Vital : approche pour évaluer le code annoté d'une ligne de produits.

— Regarder quels sont les bonnes pratiques pour faire des codes annotés.

— Le pourcentage de code en commun entre 2 codes annotés. (métrique)

— Le Lirmm a développé un outil d'extraction de Feature-Model, nommé isiSPL, a partir d'un corpus de configuration valides. Ce Feature-Model n'est un qu'un simulacre et potentiellement biaisé.

— Les codes annotés ne possèdent pas d'assistance syntaxique.

— Faire un petit logiciel qui prends un code annoté et applique la méthode **VITAL** (compare les lignes de produits faite à la main et celle extraite du logiciel). Nicolas (Le Lirmm) à besoin d'évaluer la qualité des annotations (extraites par leur outil), une façon de faire c'est d'implémenter les critères de la méthode **VITAL** qui date de 8 ans.

— Vu le temps écouler depuis le début de la recherche, il faut chercher et proposer d'autres critères de notation pour les annotations d'un modèle et ce qu'il en est devenu de **VITAL**.

— Implémentation en Java (Python à été évoquer).

— Nicolas nous fournis des données a analyser par exemple. https://gite.lirmm.fr/hlad/isispl-experimentation/-/tree/master/SPLs.

— Application de Design Pattern quant a la conception d'annotation.

— Référence au livre : Coder proprement [8] par Robert C. Martin, Michael C. Feathers, Tim Ottinger, Jeff Langr qui propose des métriques de bonnes pratiques de développement en général, pourquoi ne pas essayer de les appliquer aux annotations ?

— les métriques doivent reflétés : les bonnes pratiques, un code facilement maintenables.

— 1er objectif : lister ces pratiques du coté annotation (pourcentage de code commun, taille, annotation frère fusionable, ...) comparer une ligne de produit crée à la main et un automatiquement -> ajout de métrique, approche VITAL (littérature : microsoft academic)

— Les lignes de produit doivent être spécifique à un domaine métier, -> bien analyser le domaine pour savoir quoi implémenter.

— créer un Trello / Diagramme de gant.

Question Post-Réunion :

6. Suite à l'épidémie de Covid-19, il n'y a plus de soutenances prévues - vous serez évalués sur travaux (votre rapport + 10 slides)
7. Référence à la Loi de Newman
8. Titre original, Clean Code : A Handbook of Agile Software Craftsmanship

— Quel est le lien avec le lambda-calculus analysis ?

— Pourquoi l'équipe du Lirmm c'est appuyé sur l'approche annotation ?

— Est ce que l'équipe du Lirmm a recherché dans d'autres approches ?

— L'arbre des caractéristiques développé pour le TP du module HMIN102, des LPL par *Annotation* est aussi utilisé dans l'approche par *Composition* ?

— Il est demandé pour la feuille de route du 10 février 2021 de fournir une liste des documents à lire dans le cadre du TER. Plus précisément _au moins un article de recherche par étudiant pour les TER demandant de la programmation.

Mise en bouche sur la conception d'une ligne de produit d'hier à aujourd'hui, présentation de la place d'IsiSPL

➔ Le 5 Février à 9h30-11h.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

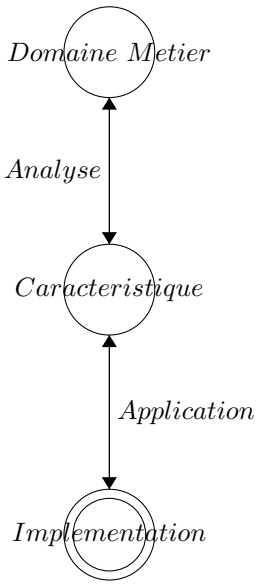
Objectifs

— Faire le tri des articles a lire pour ne garder que ceux qui permettent de produire un rapport avec les métriques de la méthode **VITAL** (*V*ariability *I*mprovemen*T* *A*na*L*ysis).

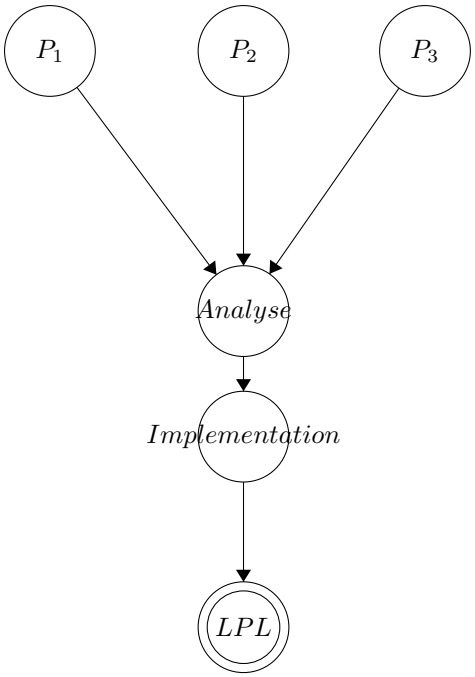
— Élaborer un pseudo-algorithme d'évaluation de métrique.

— Il existe **trois** approche pour concevoir une ligne de produit logicielle.

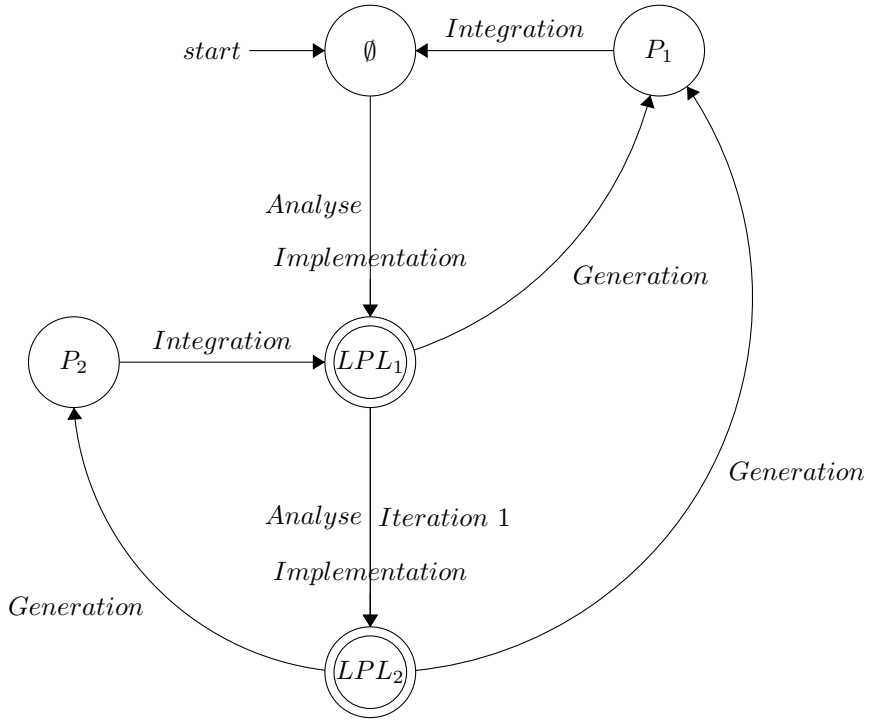
1. Approche **Pro-active** : Les connaissances métier d'un domaine (auto, aviation, médecine, ...) permet de son analyse d'extraire un ensemble de caractéristiques le plus exhaustif possible pour répondre aux besoins de l'entreprise immédiat et futur (boucle de rétro-action sur l'analyse). La dernière étape consiste en l'implémentation des caractéristiques et leur maintenance (boucle de rétro-action sur l'implémentation). Nous omettons la génération d'un produit qui serait la dernière étape. Cela consiste a fournir une configuration conforme et qui produit automatiquement le produit désiré.



2. Approche **Extractive** : nous partons d'un ensemble de produits, nous extrayons des connaissances pour créer la ligne de produit. Aussi appelée l'étape d'***I*nformation *R*etrieval** (IR). Puis implémentation des caractéristiques produits par l'analyse. Malgré que l'étape d'analyse est optimise l'investissement de l'entreprise, en pratique cette approche ne dépasse pas cette phase et n'est donc pas utilisable.



3. Approche **Réactive** : Nous partons d'un premier produit qui permet de générer un modèle, puis le modèle s'enrichit par une étape d'intégration continue au fur et a mesure de l'ajout de nouveau produits. Au début l'intégration est tenable par un humain sur le facteur temps, investissement, complexité d'intégration. Puis avec le nombre croissant d'intégration, la faisabilité du projet semble décroître exponentiellement.
   Cela rends l'approche inutilisable dès un très grand nombre d'intégration.



— Les deux premières sont coûteuses respectivement en temps et argent/ROI [9] pour l'analyse du domaine et des caractéristiques, respectivement en temps et argent/ROI sur chaque itération et due a un turnover important dans le domaine de l'ingénierie logicielle plus généralement du développement logicielle.

— La troisième le devient avec le temps et les ambitions du projet.

— C'est ce dernier problème que ***I*siSPL** viens élégamment résoudre en proposant de s'occuper de crée de façon itérative et par intégration la nouvelle ligne de produit logicielle avec des caractéristiques communes. Il faut remarquer que l'ajout de produit aux caractéristiques non communes est possibles mais produit alors des features non corrélés avec le reste du code et potentiellement du mauvais code.
Il est donc a la croisé des trois approches mentionnées précédemment.

— **Nous rappelons que l'évaluation de LPL par composition n'est pas le sujet de ce TER, nous nous concentrerons uniquement sur l'évaluation de LPL par annotation** [10]. La composition consiste a produire des caractéristiques dans des boites puis par lambda-calculus, de "composer" les configurations.

— Nous avons revues et corriger quelques détailles en rapport sur aux diags UML (activité > séquence) mentionnées sur la *Feuille de route du 10 février 2021* proposition N°2 et le Gantt.

— Nous avons proposé de contacter B. Zhang et al, pour essayer de récupéré le code source ou d'autre information mais M. Hlad nous la déconseillé (perte : temps de compréhension, intérêt privé de l'auteur, etc.) et recommander de le faire qu'en fin de projet pour une étude comparative (gains : citation, idées, résultats, etc.).

— Nous mentionnions Microsoft Academics pour chercher d'autre articles sur le même sujet autre B. Zhang. Effectivement, nous pouvons déjà étudier les sources que Zhang lui même cite, nous y retrouvons des noms fréquents sur les auteurs (S. Apel pour n'en citer que un).

— M. N. Hlad nous a présenté Software Heritage pour parcourir une bibliothèque de code source.

— Guilhèm s'était posé la question "En quoi consistent les expérimentations ?" et ce que Nicolas voulait dire dans son commentaire fait sur la proposition N°1 de notre feuille de route. A cela il nous éclaircis qu'il nous suggéraient de mettre en place une fois notre outil développé, une phase d'expérimentation via une étude comportementale de l'efficacité de nos métriques. Nous pouvons par exemple récupérer les LPL produites en TP du module HMIN102 - Ingénierie Logicielle et demander a nos collègues de juger eux-mêmes les codes et enfin comparer avec notre propre outil d'évaluation.

— Par ailleurs la complexité en temps de l'évaluation d'une métrique n'est pas important, nous ne somme pas dans un contexte critique, cette évaluation n'est effectué qu'une fois pour le projet et non pas en continue lors d'une mise en production.

— Très bonne remarque de M. Hlad, sur l'implémentation des chemins du FS [11]. Étant donnée que l'outil doit être cross-platform et que l'utilisation de Java est bien adapté. Nous devons utiliser les outils de Java pour construire des paths relatifs/absolues non dépendant de l'os cible. Par exemple le slash et backslash pour délimiter un chemin.

— Il va sans dire que le code doit être au plus propre possible, documenté, avec un format d'exportation des métriques standardisé, comme CSV pour un ajout rapide a un tableur du style LibreOffice Calc ou tout autre logiciel non cité a ce jour.

— Yanis a mentionner Munge mais c'est bien pour confirmer que ce ne sera pas utilisé dans ce projet. Il est utilisé comme pré-processeur d'annotation. Munge est un moteur de pre-processing utilisé par FeatureIDE pour la génération de configuration. Il est par ailleurs maintenu principalement par eux.

---

9. Return On Investment
10. M. N. Hlad nous dis quand même que le TER précédent le notre c'était concentrer dessus et qu'ils savent qu'il est possible de changer d'une LPL par composition, vers une LPL par annotation. Il y a donc une équivalence.
11. File System

Réunion pré-prototype du parser Naif

➔ Le 12 Février à 9h30-11h.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

— Faire un parser naïf sur un code simplement annoté.

— Pour le 5 mars (après les vacances du 19 Fev 21) un fichier output de l'arbre XML produit par l'analyse.

— Nous pouvons voir les annotations comme un AST.

— Dans note cas nous n'auront pas besoin de base de donné, elles amènent des problèmes d'inter-opérabilité, de temps de calcul des query vu que nous serons amené a faire tourné cela plusieurs fois par jour (nuancé parce l'écriture d'une query SQL, son traitement et analyse en comparaison avec une méthode sur notre implémentation), l'alternative c'est de sauvegarder les analyses dans un fichier XML ou JSON. Cela sauvegarde l'arborescence induite par le format et aussi c'est facilement transportable, analysable, etc.

— B. Zhang propose TypeChef comme parseur plus évolué d'annotations (plus puissant ?) cependant ça prise en main est compliqué, longue et potentiellement hors sujet étant donnée qu'il se spécialise dans du type checking pour les annotations, ce qui n'est pas notre TER.

— Car notre objectif est de comparer automatiquement deux résultat produit par l'analyse de code annoté (a terme par IsiSPL ou non).

— Pour le prototype nous pouvons trouver de nouvelles métriques ou implémenter VITAL, ne pas s'attarder sur l'implémentation de classes ou métriques complexes. Nous devons produire un petit programme fonctionnel rapidement avec une ou deux métriques simples.

— Dans tous les cas, nous allons commencer par implémenter les métriques proposées par VITAL, il y en a 6 (+ deux autres métriques implicites a l'analyse). Parce qu'elles n'apporte pas d'évaluation sur la comparaison entres les annotations mais se contente de les énumérer puis faire de l'agrégation de donnée via un SGBD (MS Access). C'est pourquoi nous devons proposer nos métriques et comparer les résultats dans un deuxième temps ! C'est à dire implémenter le code de cette méthode avec une représentation arborescente. Un arbre est plus adapté, optimisé pour analyser rapidement qu'une BDD avec un schéma.

— Évidemment se concentrer sur l'analyse de code open source en remarque a notre proposition de hashé avec un protocole, par exemple md5, des lignes de codes annoté pour pouvoir les comparer dans le cadre de code filtré.

— Nicolas nous fait remarquer que le hachage des lignes de codes amène des problèmes comme lorsqu'un caractère passe d'une majuscule a une minuscule etc.

— Il existe JavaParser et la JDT Parser (Java Developpement Toolkit) cependant chacune des deux méthode présente des inconvénients tels que :javaParser évalue une annotation par rapport au code qui est juste en dessous ? Avec l'utilisation de JDT nous ne prenons pas en compte les commentaires, soit une annotation.

— Nous pouvons trouver un PV dans une instruction (inline). Ce probleme est mal-compris mais Guilhèm a alors proposé une solution par rapport au problème que nous pouvont rencontré face au annotations inline d'une expressions du langage : Si un commentaire est reconnu alors introduire un CR (carriage return) ce qui montre que ce problème est réduit a celui du parsage d'une simple annotation. Par ailleurs, nous ne chercherons pas a résoudre des problèmes d'annotations dans un LITERAL (pour l'instant) étant donné que Nicolas se l'est interdis dans son outil ! A la place il duplique la variable qui reçoit le LITERAL.

— Nous allons vite ce poser la question de géré l'imbrication d'annotation, par exemple avec compteur d'ouverture et fermeture d'annotations. Nous avons réglé le problème de l'imbrication : par un algorithme récursif (voir **Réunion post-prototype du paser Naif**).

— Le problème des annotations en C++ est connu, les annotations et code bien formé ne sont pas bien compatible mais c'est hors sujet.

— Yanis propose ANTLR pour construire la grammaire des annotations. Mais enfaîte Nicolas à déjà un peu cherché la dessus avec un collègue et nous a déconseiller d'emprunter ce chemin là qui semble très ardu. Le problème réside dans le faite que au sein même de la déclaration d'une annotation (de son prédicat/var) elle pourrait contenir une annotation ! Et d'autre excentricité autorisé par les commentaires.

  Yanis : je remarque que ce qu'il manque aux annotations c'est une spécification, un formalisme qui délimite et permettent de rendre déterministe (c'est pas déjà le cas ?) son analyse. D'ailleurs Nicolas nous dis que nous pouvons le voir comme un AST (est ce que nous avons pas des analyses d'AST déterministe/indéterministe ?)

— La dispersion d'une annotation, ça correspond a son nombre d'apparition a travers le code source. C'est une précision sur les métriques tangling/fan-out. C'est mieux qu'une annotation soit concentré dans un seul fichier (meilleur maintenabilité).

— Ahmed a proposé une solution pour parser les fichier avec les I/O de java pour générer un arbre XMl et Yanis a complété avec la possibilité d'utiliser les patrons de conceptions, exemple Décorateur pour décoré différents types d'annotations.

Réunion post-prototype du parser Naif

➔ Le 19 Février à 9h30-10h30.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

— Avoir un fichier résultat a la racine du dossier input.

— Étudier des métriques sur la comparaison de deux arbres.

— Optionnelle : permettre l'analyse d'autre types de langages et formats.

— Mettre a jour le code existant pour analyser plusieurs fichiers java a partir d'un dossier d'input. Il faut prendre en charge les fichiers dans dossiers imbriqués.

— Écrire un fichier résultat (output) a la racine du dossier d'input.

— Le fichier résultat doit contenir toutes les annotations et une synthèse de l'analyse (partiellement fait au 19 février 2021, voir commit **9e886a4d**).

— La structure du fichier résultat est la suivante : le listing de nos annotations, puis une synthèse de nos annotations (avec métriques VITAL implémentés jusque là).

— Nous peut nommer le fichier résultat : nom-du-projet+timestamp.

— Produire une comparaison de 2 arbres, c'est a dire pouvoir répondre aux questions : plus grand sous arbre commun, pénalité par branche manquantes/supplémentaires, la couvertures, englobant, la profondeur/degré, le nombre de noeuds.

— Nicolas a pour intuition que si nous comparons deux arbres et que l'un est moins profonds, et possède moins de noeud (d'annotations) il est alors meilleur.

— Il n'y a pas d'annotation elif produite par isiSPL, cela va changer.

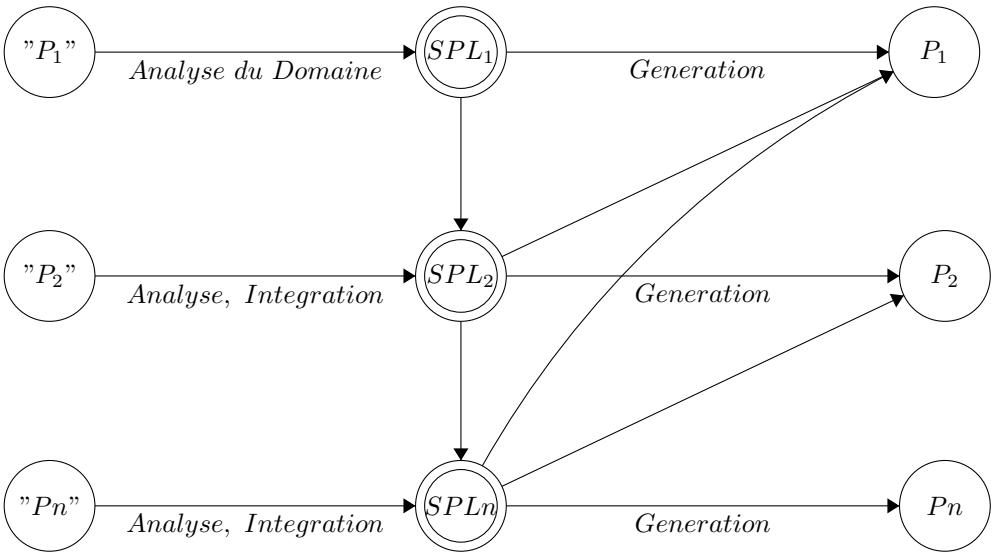
— Les variables des prédicats d'annotations produites par IsiSPL sont séparées par des virgules.

— Le " ;" est ambiguë il signifie soit un "Et" ou bien un "Ou" logique. Un TER de M2 travaille a résoudre ce problème avec Nicolas.

— Pour vérifié si deux annotations sont équivalents nous avons deux possibilités :

- Augmenter la regex existante pour améliorer la comparaison syntaxique.
- Ajouter une comparaison sémantique (logique) si le temps le permet.

— Refaire le schéma de l'approche Réactive :



— Remarque sur l'approche Pro-active : il n'y a pas de retour : **utopique**

— Remarque sur l'approche Extractive : aligné P1, P2 et les LPLs pour les distingue (P1,P2,..Pn) n'existe pas (c'est un cahier des charges)

— Rappel : Antenna c'est le moteur de configuration de FeatureIDE pour les propositions logiques, qui parcourt le modèle et qui va sélectionner le prédicat.

— Étudiez des métriques sur la comparaison de deux arbres. Par exemple quand nous avons beaucoup d'annotations dans le projet. Devons nous favoriser le même niveau d'imbrication ou non ? Quelle est la meilleure pratique/compromis entre un code pas trop différent d'un autre mais pas trop difficile a comprendre pour un humain ?

Réunion pré-implémentation UML

➔ Le 5 Mars à 9h30-10h25.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

— Faire l'output XML de l'arborescence des annotations.

— Chercher des outils pour faire la comparaison de 2 codes dans le but de chercher les similarités de codes.

— Nous avons réussi un objectif sur les deux précédemment obligatoires.

— Nous avons réaliser un UML le 3 Mars soumis a Nicolas qui la validé.

— Étude des heuristiques provenenant de [Mar09] : des bonnes pratiques de développement, des «bad smells» cependants pas toutes adaptés au LPL.

— Nous avons réglé la confusion entre Prédicat et Proposition.

— Nicolas utilise une représentation des annotations par graphes dont les arêtes sont les contraintes et les noeuds des features. les croix encerclés sur les arêtes sont des mutex, nous pouvons faires des groupes de noeud avec un cadre.

— Nicolas nous a expliqué le processus d'intégration de isiSPL avec la mise en arbre d'artefact puis avec une étape de merge.

— Une métrique c'est un objectif. Quel est le but d'une métrique ? La maintenabilité ? lisibilité, compréhension ? Les métriques permette de savoir si un code annoté est bon.

— Nous retrouvons la maintenabilité qui passe par l'imbrication des annotations, la compréhension du code qui passe parle nombre de d'annotations.

— Si deux codes sont similaire, la maintenabilité est plus difficile, la redondance introduit des bugs, les design pattern essayent de l'éradiquer depuis 15 ans.

— Nous pouvons étudié la similarité de code en comparant 2 codes en utilisant un AST.

— Il faut chercher des outils pour faire la comparaison de 2 codes.

— Nous pouvons aussi résoudre des cas triviaux comme deux Features équivalentes successives, deux if ayant des propositions logiques complémentaires par un else.

— Nous pouvons étudié la présence de commentaires en lien direct avec la feature.

— GumTree est un outil d'analyse d'AST. A partir de 2 fichiers A et B différents contenant le même code et il affiche les actions à faire pour passé du fichier A au fichier B.

— Il faut étudié comme intégré l'utilisation de GumTree pour notre analyse des annotations. Nicolas nous a parlé de mocker le code pour le faire passer dans gumtree.

— Le " ;" est désambiguïsé. Le prochain objectif de Nicolas c'est de faires des propositions logiques les plus "vrais".

— Rappel : Le but du TER est de comparer 2 ligne de produit une faite à la main, et l'autre faite avec isiSPL.

— Si deux features sont toujours active en même temps, ça peut être soit du à une mauvaise conception (ça serait la même) ou au fait qu'il manque encore des produit dans la LPL pour les différencier.

— Le nombre de prédicat d'une annotation est un score de difficulté intéressant.

— Deux proposition logique équivalente peuvent avoir une écriture différente, exemple : A ou B est équivalente à : B ou A.

Réunion post-implémentation UML

➔ Le 12 Mars à 9h30-11h05.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

— Implémentation de VITAL + le calcul du degré par le LOC sur notre implémentation.

— Faire ressortir les dépendances logiques (implications) entre annotations sans connaissance du Feature Model.

— Tester le code sur les annotations que Nicolas va nous envoyer.

— Implémenter le calcul du score de comparaison en se basant sur les métriques .

— Après discussion notre output XML est ok.

— Les différentes métriques d'un code normal.

— Sur le code général : métriques de couplage (en fonction du paradigme), factorisation, similarité du code, LOC, documentation (si il y a des commentaires, par rapport au code annoté : sémantique) .

— Métrique de couplage : Référence entre les classes A et B (sans la classe B, la classe A ne peut pas fonctionné) Solution : nous ne pouvons pas réduire le couplage en utilisant les annotations.

— Est ce que les annotations peuvent être factorisés ? les annotations imbriqués, par exemple une annotation A dans une annotation A n'est pas nécessaire, qui se suivent (seulement les triviaux), les dépendances temporelles (slicing avec Spoon). Combien il y a d'annotations successives avec le même prédicat (factoriser ses annotations) (en ces 2 annotations nous ne devons avoir aucune ligne de code même java). Nous retrouvons le cas du "{" est non trivial, donc des que nous avons un caractères entre 2 annotations identiques. Nous pouvons alors trier les annotations successives en trivial et non trivial.

— Spoon, proposé par l'INRIA est une api Java qui manipule des CT-Elements pour faire du slicing.

— Métrique de similarité de code. Comment mesurer la similarité ? Avec une distance en caractère ou similarité AST ? Nous pourrions sortir le code, l'inclure dans une classe mock pour le faire passé sur gumtree et avoir le résultat mais gumtree ne sais pas calculer la similarité. Nicolas nous recommandes de faire la distance de Hamming entre les mots.

— En résumé pour déterminer la la similarité de code nous pouvons faire :
  — Calculer la distance entre deux codes avec la distance de Hamming.
  — Utiliser des outils de comparaison d'arbres.
  — GumTREE te donne la différence entre 2 codes mais pas les similarités.

— Nous devons choisir quels annotations analysés sinon il y en a beaucoup trop a analyser, Nicolas recommande de choisir celles qui sont différentes, nous pouvons retrouver une nuance faible entre 2 features différentes.

— Nous pouvons calculer le nombre de prédicats imbriqués inutilement qui peuvent être simplifier. Par ce qu'ils viennent parasité la maintenabilité du code. Exemple une feature A qui imbrique A qui imbrique C. La feature est simplifiable en A imbrique C.

— Cela revient chercher les annotations redondantes dans le cas d'une annotation fille directe.

— Remarque, il faut réfléchir la métrique : qu'est ce que nous voulons observer ? non pas qu'est ce que nous voulons changer ?

— Remarque 2, nous avons un biais **énorme**, le code que nous utilisons est peu erroné, plutôt bien formé (hypothèse de base), sur de bonne pratique. Pour contre balancé, nous pouvons essayé de voir le code des M1 et M2 ayant suivi le cours sur les LPL. (Yanis contacté la promo le 14 mars a 9h45).

Réunion sur les métriques

➔ Le 12 Mars à 9h30-10h45.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

—

—

Réunion sur les métriques

- ➔ Le 19 Mars à 9h30-10h45.
- ➔ Liste des enseignants présents Nicolas Hlad.
- ➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

—

—

Réunion sur les métriques

- ➔ Le 26 Mars à 9h30-10h45.
- ➔ Liste des enseignants présents Nicolas Hlad.
- ➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

—

—

Réunion sur les métriques

- ➔ Le 02 Avril à 9h30-10h45.
- ➔ Liste des enseignants présents Nicolas Hlad.
- ➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

  pour le rapport : 2 partie 1. métrics(pour reporduire se qu'a fait vital et allez plus loin (la redondance, le couplage, et la similarité) et comparaison 2. nous sommes cappable de comparer 2 codes annotées faire la comparaison du code de 2 implémentations (faire la comparaison entre 2 codes 1 fait à la mains, l'autre par isiSPL).

  Si la version à la main et plus redondantes que celle de isiSPL alors celle de isiSPL est meilleur.

  Suivre une méthode en mode distance de hamming pour comparer les codes variants pour indiquer à l'utilisateur quelles sont les annotations qui peuvents être réfractoré.

  1.le couplage entre 2 features 2. la similarité de codes

Objectifs

—

—

Réunion sur les métriques

➔ Le 09 Avril à 9h30-10h45.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

dans 2 semaines pouvoir faire une présentation global de tout ce qui a été fait, toute les métriques + exemples + execution (si pas d'exemple, montré dans le code ce qu'il se passe) semaine prochaine : inventaire des metriques + maquette dessein des résultats (basé sur DrawSPL ou Elevator) voulu pour faire le tri de ce que nous implémentons Pour les soutenance des TER M1, la semaine est fixée mais les créneaux exacts pour chaque groupe sont susceptibles d'être modifiée en fonction des disponibilités des encadrants et des rapporteur. Concernant les soutenances des CMI elle aura lieu sans doute la semaine d'avant le jour reste à préciser. Les rapports devront être envoyés au rapporteur le lundi précédent la semaine des soutenance. Lundi 24 mai Si les soutenances ont lieu en présentiel elle se dérouleront dans la salle projet du bâtiment 16 de la fds. Si les soutenances ont lieu en distanciel elles se feront aux créneaux indiqués. Dans tous les cas, les étudiants ainsi que les enseignants peuvent assister à tout ou partie des soutenances.
//————————————————————————————————————-
Shémas des mértriques faire une matrice des moyennes faire un exemple de sur A=>B faire des main différents d'ici 1 semaine faire des diapos de toute les métriques (inventaire de tous se quue nous avons faits) les résultat sur la ligne de produit drawSPL.

Objectifs

—

—

Réunion sur les métriques

➔ Le 16 Avril à 9h30-10h45.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

—

---

1. représentation XML de l'arbre des annotations. contenu d'une annotation
2. représentation XML des annotations synthèse contenu d'une annotation
3. implémentation des metriques de vital
4. implémentation du score d'un fichier
5. implémentation de nos métriques Représentation XML des annotations simplifiables et éliminables calcule du pourcentage des annotations simplifiable calcule du pourcentage des annotations éliminable calcule du nombre d'annotation fusionnable
6. création du fichier contenant une représentation légère des annotations
7. création du fichier contenant les implications d'annotations
8. création de la matrice de similarité de code et son affichage sous forme d'image.

[camembert ou histogramme]

faire un tableau pour les metriques vital préparer une petite mise en scène Implémentation des métriques : ok Nous devrons faire l'exploitation des métrique Faire le dot (a la main ou généré)

des implications par exemple : moins il y a des dépendances, mieux c'est nous ne pouvons pas commencer apres 5 min de présentation : la nous avons calculer la similiraite = NON ce qu'il faut faire pour la présentation : * il ya ce probleme : c'est un fait, exemple : il existe des c/C ce qui a des conséquences sur la maintenance * nous proposons ça : détécté le c/C d'annotations et nous avons fait comme ça

exposer les problèmes en fonction de l'étude du problème sur 1 ligne de produit et sur la comparaison des lignes de produit faire une démo où nous présentons chaque métrique quel est le problème qui la rends nécéssaire que veut dire la métrique dans le code (être précis sur ce que dit la métrique) si je veut exploiter cette matrice qu'est ce que cette dernère peut m'apporter

Ce qu'il faut chercher dans une LPL : * moins possible d'annotation * moins possible de dépendances entres les implémentations des features et que le code soit petit * moins de redondance trouvé des noms plus explicites sauter des lignes dans output_lite afficher le total #annotations - # LOC nous pouvons l'afficher dans la diapo, nous pouvons montrer le calcul du score, le comparer a 2 implémentations Pourquoi le pixelmap est intéréssant ?

Problème -> va expliquer la nécessité de créer la métrique le sens que nous donnons à la métrique (sa sémantique) comment est-elle implémenté comment l'exploiter

---

Réunion sur les métriques

➔ Le 23 Avril à 9h30-10h45.

➔ Liste des enseignants présents Nicolas Hlad.

➔ Liste des étudiants présents Ahmed Kaci, Guilhèm Blanchard, Yanis Allouch.

Objectifs

—

—

Références

[Ale+12] Aleixo, F. A. et al. « A Comparative Study of Compositional and Annotative Modelling Approaches for Software Process Lines ». In : *2012 26th Brazilian Symposium on Software Engineering*. IEEE, sept. 2012.

[Anq+08] Anquetil, N. et al. « Lignes de produits logiciels et usines logicielles ». In : *L'objet* t. 14, n° 3 (sept. 2008), p. 15-31.

[Dei15] Deissenböck, D. F. *Living in the #ifdef Hell*. 2015. url : https://www.cqse.eu/en/news/blog/living-in-the-ifdef-hell/.

[KA08] Kästner, C. et Apel, S. « Integrating Compositional and Annotative Approaches for Product Line Engineering ». In : (2008), p. 91-98.

[Mar09] Martin, R. C. *Coder proprement*. Pearson Education France, 2009.

[Mei+17] Meinicke, J. et al. *Mastering Software Variability with FeatureIDE*. Springer International Publishing, 2017.

[Pre19] Preschern, C. « Patterns to escape the #ifdef hell ». In : (2019).

[WB09] Wilson, J. et Ball, T. *Peprocessing .java with Munge*. 2009. url : https://publicobject.com/2009/02/preprocessing-java-with-munge.html.

[ZB12] Zhang, B. et Becker, M. « Code-based variability model extraction for software product line improvement ». In : *Proceedings of the 16th International Software Product Line Conference on - SPLC '12 -volume 1*. ACM Press, 2012.

[ZB14] Zhang, B. et Becker, M. « Variability code analysis using the VITAL tool ». In : *Proceedings of the 6th International Workshop on Feature-Oriented Software Development - FOSD '14*. ACM Press, 2014.

[Zha15] Zhang, B. *VITAL - Reengineering Variability Specifications and Realizations in Software Product Lines*. T. 53. Fraunhofer IESE, Kaiserslautern : Fraunhofer Verlag, 2015.