UCLouvain

epl

École polytechnique de Louvain

Exploration de patterns sur du code source C#

Auteurs: **Jean Bosco Rwibutso, Roman Skubiszewski**
Promoteurs: **Céline Deknop, Kim Mens**
Lecteur: **Siegfried Nijssen**
Année académique 2019–2020
Master [60] en sciences informatiques

Abstract

L'exploration de patterns sur du code source est un vaste sujet dans le monde de la recherche informatique. L'ingénierie logicielle porte un grand intérêt à la découverte de régularités dans du code source, tant dans le milieu universitaire que dans le milieu industriel. En effet, les régularités peuvent fournir des informations utiles pour aider le développeur dans une variété de tâches telles que la compréhension ou la réécriture de codes, la détection de mauvaises pratiques, etc.

Malheureusement, les algorithmes d'exploration de patterns traditionnels ne sont pas toujours efficaces et ne découvrent que peu de régularités riches et intéressantes. Ce mémoire se base sur un framework d'exploration de patterns qui essaie de surmonter ces limitations. Ce framework paramétrable permet la découverte de patterns dans du code source, indépendamment du langage de programmation de ce code source. L'algorithme est basé sur un algorithme d'exploration d'arbres fréquents. Les recherches concernant ce framework considèrent du code source écrit en Java et en COBOL.
A travers ce travail, nous analyserons comment cette technique s'applique sur du code source C#. En effet, dans ce document seront expliqués le fonctionnement de ce framework ainsi que les configurations à mettre en place pour l'exploration de code source C#. La découverte de patterns riches et intéressants suite à cette exploration sera exposée lors de ce mémoire.

Remerciements

Tout d'abord, nous aimerions remercier l'ensemble de l'équipe INTiMALS pour sa disponibilité lors des questions que nous pouvions avoir à leur poser.

Plus particulièrement, nous remercions nos promoteurs Kim Mens et Céline Deknop pour leur temps et leur écoute durant nos échanges hebdomadaires ainsi pour que leurs précieux conseils. Merci également à Johan Fabry qui nous a permis d'avoir accès au code d'un composant important du framework INTiMALS et sans qui notre mémoire n'aurait pas été possible. Merci pour la confiance et la considération accordées par chacun d'eux et pour la disponibilité dont ils ont fait preuve dès qu'une question nous taraudait.

Ensuite, un grand merci à notre lecteur Siegfried Nijssen pour l'attention portée à notre travail, à tous nos relecteurs et plus particulièrement Nathalie Legrain, mère de Roman Skubiszewski, et Evase Nizeyimana, oncle de Jean Bosco Rwibutso, pour leur aide et le temps passé sur notre mémoire.

Enfin, merci à tous ceux, amis et familles, qui nous ont soutenus et encouragés durant tout notre cursus.

Table des matières

Chapitre 0

Introduction

Dans ce chapitre, nous allons brièvement introduire le contexte, le problème, la motivation ainsi que l'objectif de cette recherche.

Contexte Ce travail de recherche s'inscrit dans le cadre du projet INTiMALS (INTelligent Modernization Assistance for Legacy Software) [1]. Ce projet est une collaboration entre l'UCLouvain, la VUB et la société Raincode Labs [2] et financé par Innoviris dans le cadre de l'appel à projets Team Up 2017 [23].

INTiMALS est un projet de recherche d'extraction de régularités dans des systèmes hérités. Un système hérité est, selon Wikipédia, "*un logiciel continuant d'être utilisé dans une organisation (entreprise ou administration), alors qu'il est supplanté par des systèmes plus modernes. L'obsolescence de ces systèmes et leur criticité les rendent difficilement remplaçables sans engendrer des projets coûteux et risqués.*" [12].

Les applications bancaires sont des exemples de systèmes hérités, leurs codes datent d'il y a plusieurs années, utilisant les technologies de l'époque, pour la plupart du COBOL ou du Fortran. Après avoir constaté que ces applications exigeaient beaucoup de sécurité, il a été privilégié de garder ces anciennes technologies. Aujourd'hui, ces technologies désuètes ne sont plus enseignées et très pratiquées par les développeurs.
La finalité de cette recherche est d'aider à la compréhension d'un système hérité afin de, par la suite, pouvoir le mettre à jour, ou le migrer vers un autre langage avec des technologies plus récentes [18].

Problème Exploration des patterns afin de trouver des régularités dans un

code source C#.

Les régularités du code C# ou autre langage, peuvent être définies comme différents idiomes, conventions, syntaxes et structures utilisés dans un langage de programmation. Un bout de code riche ou intéressant comprend généralement l'utilisation de patrons de conception (design patterns), de conventions de nomenclature, de conventions de syntaxe, d'utilisation de frameworks, de librairies, d'API externes, etc.

Motivation La motivation du projet de recherche INTiMALS est d'avoir un framework indépendant du langage de programmation, d'exploration des patterns et adapté dans le but de trouver des régularités dans du code source.

Il a été observé que d'autres algorithmes traditionnels pouvant être utilisés pour explorer les régularités dans des fichiers et ayant un domaine d'application particulier ne sont pas totalement adaptés à notre problème.
Par exemple, *"Diff tool"* [15], gamme d'outils de comparaison de textes entre les fichiers, peut être utilisé pour explorer les différences entre plusieurs fichiers. Il est donc possible de récupérer les similitudes dans ces fichiers puisqu'il s'agira de repérer ce qui n'a pas été retourné comme étant une différence par l'outil. Dans ce cas, les similitudes dans les fichiers sont apparentées à des patterns pouvant contenir des régularités de code. De par sa nature, ces similitudes de code, du texte copié-collé, peuvent être très nombreuses et le rapport du nombre de régularités intéressantes et de la quantité de similitudes retournées est très minime. Par conséquent, cette technique traditionnelle ou une adaptation de celle-ci ne concorde pas avec la finalité recherchée.

INTiMALS propose une solution, basée sur un algorithme d'exploitation des arbres de structures de données, FreqT [7], apte à trouver des régularités en étant indépendante du langage de programmation. Les recherches sur l'adaptabilité de cette solution sur le langage Java et COBOL ont été menées dans le passé.

Objectif Ce mémoire s'intéresse à l'analyse de l'adaptabilité du framework existant, INTiMALS, sur un langage autre que Java et COBOL. Dans le cadre de ce document, des expériences ont été menées sur des code sources C#. L'objectif est donc de trouver une bonne configuration du framework afin de récupérer des patterns au contenu riche et intéressant pour un code source C#.

Contributions Grâce à ce mémoire, nous avons exposé d'éventuelles limitations du framework INTiMALS lors de l'exploration de patterns. Nous avons mis en place des adaptations pour ce dernier afin d'améliorer la découverte de patterns dans un code source C# ainsi que d'autres langages de programmation. Il a donc été observé par les expériences menées que le framework trouve des résultats intéressants et riches dans des codes sources C#.

Plan Dans ce document, nous exposerons tout d'abord dans le chapitre 1, les concepts et le framework INTiMALS, les connaissances requises pour comprendre le contexte dans lequel nous avons commencé notre mémoire. Ces concepts seront utilisés tout au long de ce document. Ensuite, lors du chapitre 2, nous présenterons les problèmes rencontrés suivis par les solutions mises en place pour y remédier, en détaillant les raisons qui nous ont poussés à ajouter un nouveau fichier de configuration. Par après, le long du chapitre 3, nous énoncerons les implémentations que nous avons effectuées durant ce travail. Pour finir, nous validerons nos solutions dans le chapitre 4 à l'aide de différentes expériences menées sur divers projets C# ainsi qu'au moyen d'une analyse de l'apport de nos solutions sur une expérience déjà réalisée sur un projet Java.

Chapitre 1

Analyse de l'existant

Le travail présenté dans ce mémoire se base sur un projet de recherche existant, il est important de mettre au clair certains points de ce dernier. Dans ce chapitre, nous allons premièrement expliquer plusieurs concepts liés à l'exploration de code source sur lesquels nous nous baserons tout au long de ce document. Ensuite, les tenants et aboutissants du framework INTiMALS d'exploration de code source seront exposés. De cette manière, nous aurons situé le contexte du travail que nous avons fourni.

1.1 L'exploration de code source

1.1.1 L'Arbre de syntaxe abstraite : représentation des données

Un arbre de syntaxe abstraite, "Abstract Syntax Tree" en anglais, AST en abrégé, est un arbre représentant la structure syntaxique d'un programme informatique. Un arbre est une structure de données récursive où un élément est appelé nœud ; cet élément peut avoir 0, 1 ou plusieurs enfants, qui seront également des nœuds. Un nœud sans enfant est appelé une feuille. La figure 1.1 donne l'exemple d'une représentation d'un AST pour de simples opérations "x = 1 ;", "y = 2 ;" et "3 * (x + y) ;".

De manière à pouvoir comparer plusieurs ASTs entre eux, il est important de mettre en place un méta-modèle afin de garder une certaine cohérence entre les ASTs. A titre d'exemple, le méta-modèle, dans le cas de l'exemple cité précédemment, pourrait être un AST dans lequel, les opérandes se trouvent sur les feuilles, tandis que les nœuds sur les branches représentent les opérateurs.

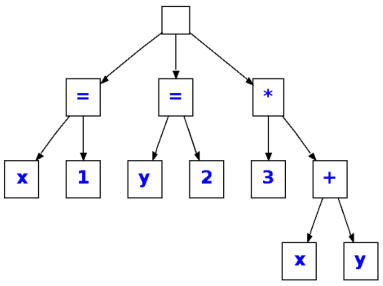


FIGURE 1.1 – Exemple d'Arbre de syntaxe abstraite (AST) [9]

1.1.2 Algorithme d'exploration de données : FreqT

La technique appelée **FreqT** [7], dédiée à l'exploration des nœuds dans un AST est décrite dans cette section.

L'algorithme FreqT [7] a été développé pour découvrir les sous-arbres induits [1] dans un arbre de données ordonnés dont les nœuds sont étiquetés. L'arbre de syntaxe abstraite (AST) expliqué ci-dessus correspond à cette définition.
Cet algorithme reçoit en entrée une base de données d'arbres correspondant à l'ensemble des données dans lesquelles les patterns sont recherchés, et un support minimal [2].
Le but de cet algorithme est de trouver tous les patterns dont le support est supérieur ou égal au support minimal donné par l'utilisateur. Le support d'un pattern est le nombre d'arbres dans la base de données dans lesquels le pattern apparaît. Le pattern est donc un sous-arbre présent dans plusieurs de ces arbres.
Un pattern recherché est un sous-arbre induit, FreqT cherche des patterns en parcourant les arbres ordonnés ; pour cela, il utilise la recherche en profondeur (Depth-First Search).

L'algorithme **Depth-First Search** commence au nœud racine d'un arbre et l'explore autant que possible le long de chaque branche avant de revenir en arrière. L'idée de base est donc de partir de la racine ou de n'importe quel nœud arbitrairement, de marquer ce nœud, de se déplacer vers un nœud non marqué enfant et de continuer cette boucle jusqu'à ce qu'il n'y ait plus de nœud enfant non marqué. Ensuite, revenir en arrière, vérifier les autres nœuds non marqués et les parcourir en suivant la même logique que précédemment.

1. Un arbre T' est un sous-arbre induit de l'arbre T si et seulement si les nœuds et les arêtes de l'arbre T' sont inclus dans les nœuds et les arêtes de l'arbre T et si pour tous les nœuds v' de T' à l'exception du nœud racine et pour tous les nœuds correspondants v de T, le parent de v' est égal au parent de v. [13]
2. Défini dans la section 1.2.1, sur les fichiers de configuration.

1.1.3 Patrons de conception

Durant l'exploration d'ASTs, l'algorithme FreqT trouve des régularités dans le code qui sont, par la suite, analysées. Certaines couvrent une structure de code faisant partie de pratiques connues en programmation, comme par exemple, les patrons de conception. Cette notion prendra toute son importance lors de l'analyse de nos résultats [3].

Les patrons de conception [16], design patterns en anglais, ont une place importante dans le monde du développement informatique. Un patron de conception est une solution générale à un problème commun lors de l'écriture de code. Il ne s'agit pas d'un bout de code pouvant être copié-collé pour répondre à ce problème, mais la description d'un modèle pouvant être utilisé dans différentes situations. Leur écriture est considérée comme une bonne pratique, permettant à tout programmeur de comprendre le code plus facilement, il augmente la lisibilité ainsi que la maintenabilité de ce dernier.

Il existe plusieurs catégories de patrons de conception :

— Les patrons de conception créationnels : il s'agit de patrons de conception dont l'intérêt est l'instantiation des classes.

— Les patrons de conception structuraux : il s'agit de patrons de conceptions concernés par la structure d'un objet, l'héritage entre les différentes classes et interfaces. On y trouve les différentes manières d'agencer les objets pour créer de nouvelles fonctionnalités.

— Les patrons de conception comportementaux : il s'agit de patrons de conception concernés par les communications entre les différents objets.

1.1.4 Précision et rappel

Dans le domaine de la recherche, la précision et le rappel, "precision" et "recall" en anglais, sont des notions importantes. Elles seront utilisées pour mesurer la pertinence des résultats trouvés lors de l'analyse.

3. Voir chapitre 4.

La **précision** est la proportion d'éléments pertinents retrouvés lors d'une recherche parmi l'ensemble des éléments disponibles.
Le **rappel** est la proportion d'éléments pertinents retrouvés lors d'une recherche parmi l'ensemble des éléments pertinents existants.
Un graphique expliquant ces notions est visible sur la figure 1.2, où les éléments positifs sont pertinents et les négatifs ne le sont pas.

Une recherche avec une précision et un rappel tous deux de 100% refléterait donc une recherche parfaite où tous les éléments ont été retrouvés (le rappel) et aucune erreur n'a été commise, aucun élément non pertinent n'a été retrouvé lors de la recherche (la précision). En pratique, il n'est pas évident d'atteindre à la fois une précision et un rappel parfait. De plus, améliorer la précision d'une recherche a régulièrement comme conséquence de diminuer la valeur du rappel de cette recherche, et inversement. Effectivement, augmenter la précision revient, la plupart du temps, à restreindre l'échantillon d'éléments sélectionnés, et de ce fait, éliminer des éléments pertinents du résultat de notre recherche. Inversement, augmenter le rappel cause généralement une augmentation de cet échantillon d'éléments sélectionnés, la précision diminue donc par conséquent.

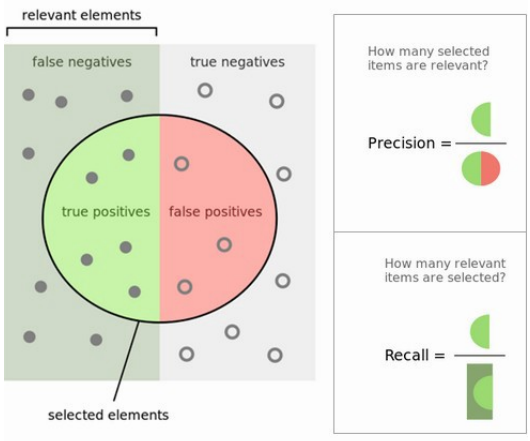


FIGURE 1.2 – Explication graphique de la précision et du rappel [6]

1.2 Framework de code source paramétrable INTiMALS

Comme nous l'avons expliqué précédemment, INTiMALS est un projet de recherche dans le domaine de l'extraction de régularités sur des systèmes hérités. Dans

cette section, nous allons exposer le framework modulaire utilisé par INTiMALS pour l'exploration de codes source. Ensuite, l'utilisation d'une grammaire sera détaillée afin d'optimiser cette exploration.

1.2.1 Framework INTiMALS

Le framework, utilisé pour fouiller les patterns récurrents, se divise en 5 composantes différentes pour former un pipeline comme il peut être vu sur la figure 1.3.

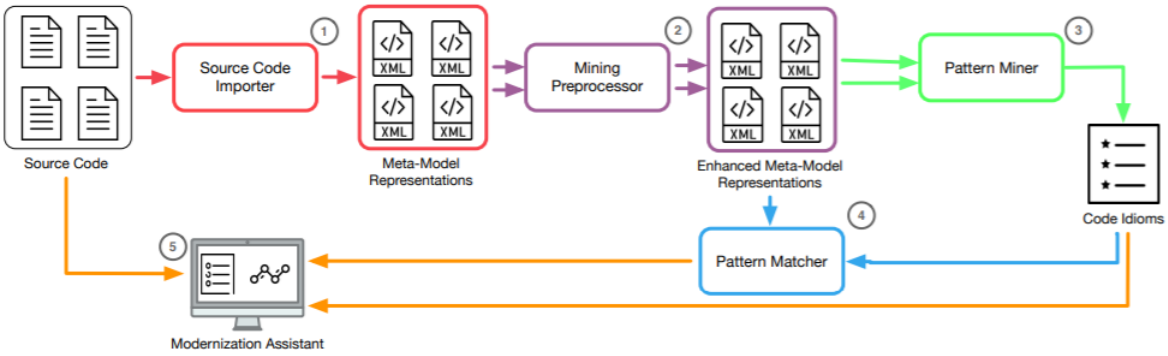


FIGURE 1.3 – Framework modulaire paramétrable pour l'exploration des patterns [3]

Source code importer

Il est important dans le cadre du projet INTiMALS de trouver un méta-modèle commun à n'importe quel langage de programmation. Il ne serait pas concevable de ré-implémenter une nouvelle version du méta-modèle pour chaque langage de programmation, cela compliquerait énormément la suite du fonctionnement du pipeline.

Le *"source code importer"* permet la caractéristique d'être libre de tout langage. Il va permettre de traduire le code source en une structure universelle commune à n'importe quel langage, un AST (expliqué en section 1.1.1). Cet AST est construit et est retranscrit dans un format XML. Un nœud appartenant à un AST est un élément XML dans lequel sont compris tous les enfants de ce nœud.

Cette étape, l'import d'un méta-modèle depuis le code source dans un langage donné, est l'une des seules dépendantes de ce langage. Après la transformation du code source en un fichier XML représentant l'AST correspondant au code source,

le choix des paramètres de configuration [4] est la dernière opération liée au langage de programmation utilisé dans les étapes suivantes.

Mining preprocessor

Le *"mining preprocessor"* est une petite étape permettant d'améliorer l'efficacité du *"pattern miner"* à l'étape suivante. Ce *"mining preprocessor"* va prémâcher le travail devant être effectué par l'explorateur. Les étapes de pré-traitement peuvent être multiples et variées. Il est possible de découper les identifiants de méthodes ou de variables en sous-arbres en fonction des conventions camelcase (utilisée en C#, Java, ...) ou underscore (utilisée en Python, PHP, ...). Il est également possible d'enlever les parties indésirables lors de l'exploration ; par exemple, les commentaires dans le code.

Pattern miner

Le *"pattern miner"* a pour but d'extraire les patterns des ASTs. Il va donc, pour ce faire, prendre ces ASTs prétraités par le *"mining preprocessor"*. Cette extraction est faite grâce à l'utilisation d'un algorithme connu et expliqué précédemment, FreqT (voir section 1.1.2). De FreqT résulte un grand nombre de résultats et un temps d'exécution élevé s'il n'est paramétré qu'avec un support minimal. En effet, un projet informatique étant composé d'un nombre important de fichiers, l'exploration de leurs ASTs est assez conséquente. C'est pourquoi, l'algorithme a été personnalisé en ajoutant diverses contraintes grâce aux fichiers de configuration.

Fichiers de configuration : Nous avons un certain nombre de fichiers de configuration avec lesquels jouer. Tout d'abord, le fichier "config.properties" dans lequel plusieurs contraintes peuvent être ajoutées.

Les attributs les plus importants sont :

- "***minLeaf***" et "***maxLeaf***" correspondant respectivement au nombre minimal et maximal de feuilles dans les ASTs des patterns trouvés. Cette contrainte est très importante afin de pouvoir supprimer les patterns trop petits grâce à notre "minLeaf" et ne pas demander trop de puissance de calcul et de mémoire avec notre "maxLeaf".
- "***minNode***" indiquant le nombre minimal de nœuds contenus dans l'AST d'un pattern. Cet attribut est également utilisé pour supprimer les petits patterns inintéressants.

4. Voir section 1.2.1 sur les fichiers de configuration.

— "***minSupport***" représente le nombre minimal d'ASTs de fichiers dans lequel doit se retrouver un pattern retourné. Chaque pattern se doit donc d'être commun à minimum "minSupport" fichiers.
— "***buildGrammar***", un booléen, mis à faux si l'on donne un fichier correspondant à la grammaire du projet.
La **grammaire** [5] est une structure qui liste l'état de tous les nœuds et leurs enfants présents dans les ASTs des fichiers du projet.
Le booléen est à vrai pour que la grammaire soit construite automatiquement en fonction de ce que retrouve le "*pattern miner*" lors de son traitement.
Ces notions et leur importance seront particulièrement détaillées dans la section dédiée, 1.2.2.

Ensuite, nous avons le fichier "***listRootLabel.txt***". Ce fichier permet d'indiquer les éléments que nous acceptons comme nœud racine des ASTs qui seront trouvés comme étant des patterns.
Parfois, les fichiers contenant le code sont grands et composés de parties peu intéressantes du point de vue d'un programmeur. Il est important de permettre de préciser le type de données que l'on veut explorer afin que les arbres correspondant à ce type soient les seuls à être étendus lors de la phase d'exploration. Par exemple, dans les ASTs exportés du langage C#, si l'on ne veut s'intéresser qu'à ce qui se trouve dans les méthodes de classes, on précisera "MethodDeclaration" dans ce fichier. Le pattern miner ne trouve alors que des patterns dont le nœud racine est un nœud "MethodDeclaration".

De plus, tous les sous-arbres des nœuds racines ne sont pas intéressants à explorer. Grâce à ce second fichier de configuration, "***listWhiteLabel.txt***", il est possible d'énumérer les sous-arbres des nœuds qui seront étendus au cours de la phase d'exploration. L'exemple de la figure 1.4 montre un AST composé de nœuds combiné avec le fichier de configuration "*listWhiteLabel.txt*" contenant les lignes "B E" et "G H I".

Le fichier "*listWhiteLabel.txt*" contient donc sur chaque ligne, un nœud suivi du ou des enfants qui seront étendus lors de la phase d'exploration de patterns. Ce fichier fonctionne finalement comme une blacklist : en effet, on remarque qu'en incluant l'enfant de B, E dans le fichier, l'élément C est alors blacklisté et le sous-arbre dont C est la racine n'est alors pas étendu. De la même manière, l'élément J n'est pas étendu par le pattern miner suite à l'ajout de H et I comme seuls enfants à étendre de G.

5. Voir section 1.2.2.

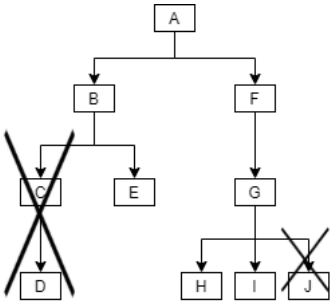


FIGURE 1.4 – Exemple de l'effet du fichier *listWhiteLabel* sur un AST

Pattern matcher

Le "*pattern matcher*" a pour but de faire la correspondance entre les patterns trouvés et le code lisible, plutôt que des éléments sous forme d'AST.
Grâce à cette correspondance, le "*modernization assistant*", décrit ci-dessous, va pouvoir souligner les parties du code que couvre un pattern.

Modernization assistant

Les résultats du pattern miner, les patterns retrouvés, sont sauvegardés dans des fichiers XML difficilement lisibles qui anéantiraient l'interprétation de ces patterns s'ils étaient le seul moyen de les examiner.
Le "*modernization assistant*" est une interface permettant de visionner les patterns retrouvés dans le code tout au long de ce pipeline. Il peut faire le lien entre les parties de code lisible par tout un chacun et les patterns retrouvés sous forme d'arbre grâce au travail du "*pattern matcher*".

Sur la figure 1.5 est exposé le pattern numéro 37 du projet Nunit [6], il a 5 correspondances dans le projet et a une taille de 108, nombre de nœuds dont il est composé. La représentation de son AST est visible sur la fenêtre du bas ; la racine de ce pattern est un nœud "ClassDeclaration".

1.2.2 La construction de la grammaire

Lors de l'explication du "*pattern miner*", la notion de grammaire a été mentionnée, cette section explique plus en détail sa création et son fonctionnement.

6. Projet utilisé dans la section 4.2.1 lors du chapitre Validation.

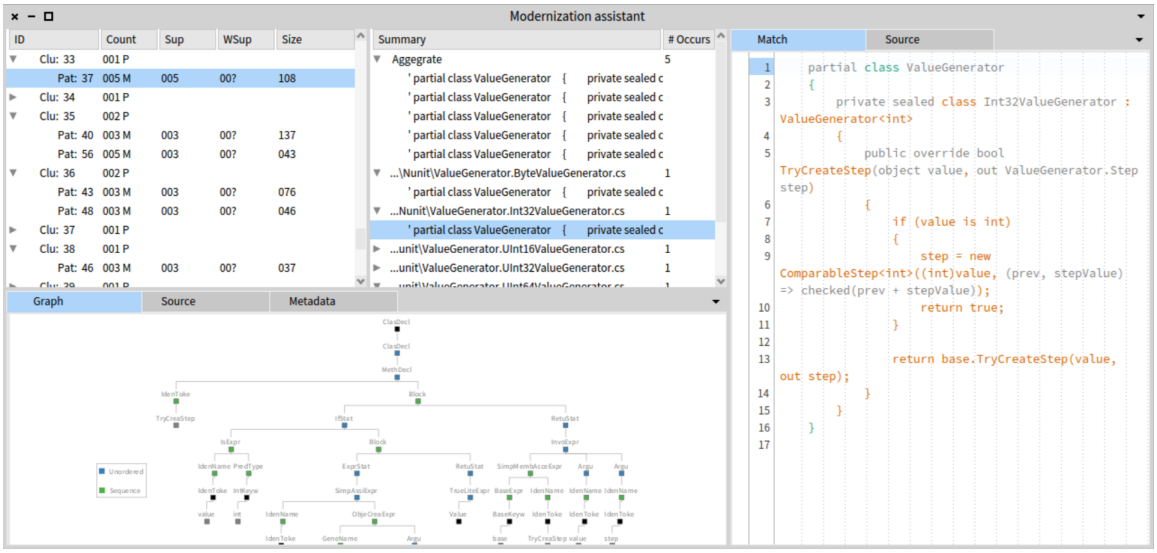


FIGURE 1.5 – Utilisation du "*modernization assistant*" afin d'analyser les patterns

La **grammaire** est une structure qui décrit l'état de tous les nœuds pouvant se présenter dans le méta-modèle d'un projet. Cet état expose la liste des enfants de chaque nœud en précisant lesquels sont obligatoires.

La notion d'enfant obligatoire signifie que, lorsque nous trouvons un pattern avec un nœud X, les enfants obligatoires de ce nœud doivent également appartenir à ce pattern pour être retournés par le "*pattern miner*". Un nœud enfant est donc obligatoire par rapport à un nœud parent.

L'intérêt de cette grammaire est d'exclure un ensemble de patterns inintéressants contenant des nœuds importants dans la construction du méta-modèle mais non pertinents dans le cadre d'une recherche de régularité de code. De surcroît, l'exclusion de sous-arbre lors de l'exploration améliore l'efficacité de celle-ci, les données à traiter étant très importantes pour l'algorithme FreqT.

La grammaire est construite de deux manières différentes. Soit elle est fournie manuellement en passant un fichier en entrée, dans ce cas, le paramètre de configuration *buildGrammar*[7] est mis à faux. Soit elle est construite automatiquement par le "*pattern miner*", le paramètre de configuration *buildGrammar* est alors mis à vrai.

Dans le premier cas, la grammaire étant entièrement fournie manuellement, il est nécessaire de connaître toutes les relations possibles entre tous les nœuds présents

7. Voir la section 1.2.1 sur les fichiers de configuration.

dans les ASTs. Il est donc difficile, pour un développeur, de prendre connaissance de toutes ces relations. Une solution serait d'avoir une seule grammaire spécifique à un langage de programmation qui couvre tous les méta-modèles provenant des tous les programmes de ce langage.
Cependant, un fichier contenant une telle grammaire est laborieusement réalisable de par sa taille et de par la difficulté d'inclusion de tous les idiomes possibles d'un langage. De plus, les syntaxes et pratiques d'un langage évoluant, ce fichier serait laborieusement maintenu.

Dans le deuxième cas, une solution alternative est de laisser le *"pattern miner"* construire lui-même la grammaire en parcourant les ASTs représentant le projet fourni, construction qui a lieu lors de l'exploration de patterns, la phase 3 sur la figure 1.3.
Dans ce cas, la notion de nœud obligatoire se construit comme suit ; lors d'un parcours de ces différents ASTs, si chaque fois que nous trouvons le nœud A, B fait partie de ses fils directs, B sera considéré comme obligatoire pour le nœud A. A l'inverse, un nœud C ne faisant pas partie des enfants de A dans un seul cas de figure, ne sera pas un nœud obligatoire pour le nœud A.

Cette notion est donc fortement dépendante de la richesse du code source donné en entrée dans le cas où nous laissons le *pattern miner* construire la grammaire. En effet, un code source ne comportant que quelques classes avec peu de code chacune ne pourra pas prendre en compte tous les cas de figure, et donc un grand nombre d'éléments risquent d'être obligatoires. Nous reviendrons en détail le problème causé par ce comportement dans le chapitre suivant.

Ce chapitre a donné une brève introduction quant aux techniques et au framework d'exploration de code source que nous utiliserons pour du code C# dans la suite de ce document.
Le chapitre suivant portera sur l'analyse plus détaillée des problèmes rencontrés.

Chapitre 2

Problème rencontré

L'objectif de ce travail est d'adapter la configuration du framework INTiMALS afin de récupérer des patterns avec beaucoup d'occurrences et un contenu riche et intéressant pour un code C#. Des expériences ont donc été menées sur du code source C#. Durant ces recherches, nous avons été confrontés à certaines complications.

Dans ce chapitre, nous allons tout d'abord exposer le problème rencontré lié à la grammaire et à sa construction par le "pattern miner", illustré d'un exemple aidant à la compréhension. Nous recenserons par la suite les solutions utilisées pour répondre à ce problème.

2.1 Problème : notion de nœuds obligatoires

Le framework INTiMALS permet d'explorer et d'analyser les régularités du code sur différents langages de programmation [1]. L'extension de ces langages de programmation sur lesquels le framework INTiMALS fonctionne en ajoutant C# est le centre de notre document.

Pour explorer des patterns d'un code source C#, nous avons besoin d'utiliser tous les composants du framework. Dans ce travail, nous nous concentrerons surtout sur le *"source code importer"* et le *"pattern miner"*.

Le *"source code importer"* est un composant qui est dépendant du langage. Il est donc nécessaire d'en avoir un qui soit spécifique pour C#.
Le *"source code importer"* pour C# a été implémenté par un étudiant externe à ce

1. Des expériences sur du Java et COBOL ont déjà été menées et ne sont pas développées dans le cadre de ce mémoire. [3]

travail, étudiant d'un bachelier en haute école, effectuant un stage dans l'entreprise Raincode Labs [2]. Cependant, quelques modifications y ont été apportées, modifications qui seront expliquées lors du chapitre 3 de ce document.

Quant au *"pattern miner"*, une bonne configuration sera étudiée pour qu'il puisse donner en sortie plusieurs patterns riches mais, avant tout, intéressants.

Comme expliqué dans la section 1.2.2, le nombre de nœuds obligatoires, c'est-à-dire de nœuds devant faire partie du pattern à condition que leur père en fasse partie, dépend de la taille et la richesse du code source donné. Si ce dernier n'est pas assez riche pour couvrir les idiomes les plus intéressants en C#, beaucoup d'éléments seront obligatoires dans la grammaire générée. Ceci implique une diminution du nombre de patterns retournés, étant donné que plus de nœuds se doivent d'être similaires dans différents ASTs pour faire partie du pattern.
Par conséquent, plusieurs patterns intéressants seront perdus à cause d'un nom de variable ou de paramètre différent comme il sera expliqué dans les exemples présentés dans la section 2.2.

La solution que nous proposons est d'ajouter un nouveau fichier de configuration. Ce fichier, *"nonMandatoryLabel"* regroupe des éléments que l'utilisateur aura explicitement choisi de ne pas rendre obligatoires lors de la phase de construction de la grammaire. La structure de ce fichier sera détaillée plus tard dans ce document, dans la section 2.3.2.

Dans la section suivante, à l'aide d'un exemple concret, l'efficacité de cette nouvelle configuration sera mise en contraste avec celle des autres techniques utilisées pour enrichir la qualité des patterns trouvés.

2.2 Exemple de référence

Dans cette section, les expériences seront faites sur un projet très simple [2] que nous avons créé. Le projet est un contrôleur de ventilateur de plafond. Il est composé de sept classes qui résument le design pattern "State". Le patron de conception "State" est un patron de conception structurel (voir section 1.1.3). Celui-ci permet de modifier le comportement d'un objet quand son état change, ici les états sont donc ceux empruntés par un ventilateur de plafond.

Le diagramme de classes UML de ce projet se trouve sur la figure 2.1. Celui-ci

2. Dossier VentilatorState de l'annexe.

sera appelé "VentilatorState" dans la suite de ce travail.

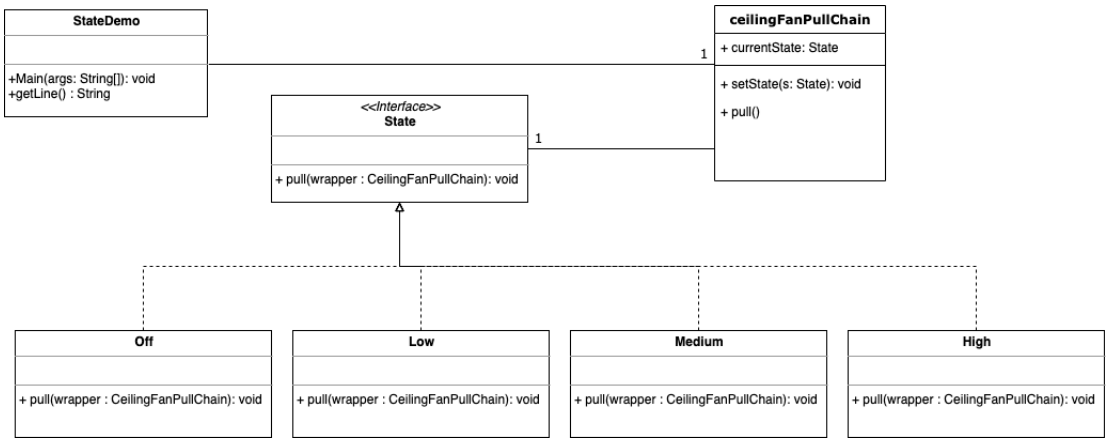


FIGURE 2.1 – Diagramme de classes du projet

Comme il peut être vu dans les images sur la figure 2.2, certains idiomes ressortent de ces classes. Des patterns intéressants devraient être retournés par notre outil, car des similarités dans le code sont visibles, même si ce code n'est pas totalement identique dans chacune des classes.

Différentes configurations sont envisageables pour permettre à l'outil de trouver des patterns grâce à l'utilisation de la *listWhiteLabel*. En effet, nous concentrons nos recherches sur le fichier de configuration *listWhiteLabel*, comme expliqué dans la section 1.2.1, ce fichier permet de blacklister des enfants indésirés dans les ASTs des classes de ce projet, lors de nos explorations de patterns.

Tout d'abord, d'un fichier de configuration *listWhiteLabel* vide résulte peu de patterns intéressants. Les patterns retournés sont présentés sur la figure 2.3 [3]. Effectivement, seuls une variable et un appel de fonction copié-collé dans toutes les classes du projet ont été retrouvés. Cependant, il est visible que certains éléments sont identiques dans les différentes classes mais ne sont pourtant pas inclus dans les patterns, comme les appels de méthodes 'Console.WriteLine()' ou encore la création de la variable de 'DisplayText'.

Cela est dû à la grammaire construite. En effet, une fois la grammaire construite par le "*pattern miner*", celui-ci cherche à trouver des patterns tout en suivant cette grammaire, indiquant l'extension obligatoire de certains nœuds enfants ou non [4]. La

3. Configuration et résultats complets de cette exploration dans le dossier "VentilatorState\importerNotPatched\withoutListWhiteLabel" de l'annexe.

4. Voir section 1.2.2.

```
using System;
1 reference
class Low : State
{

    1 reference
    public void pull(CeilingFanPullChain wrapper)
    {
        State nextState = new Medium();
        String lowStateDisplayText = "[Object] : " + nextState.ToString();

        wrapper.set_state(nextState);

        Console.WriteLine(lowStateDisplayText);
        Console.WriteLine("medium speed");
    }
}
```

(a) Entrevue de la classe Low

```
using System;
1 reference
class Medium : State
{
    1 reference
    public void pull(CeilingFanPullChain wrapper)
    {
        State nextState = new High();

        String mediumStateDisplayText = "[Object] : " + nextState.ToString();

        wrapper.set_state(nextState);

        Console.WriteLine(mediumStateDisplayText);
        Console.WriteLine("high speed");

    }
}
```

(b) Entrevue de la classe Medium

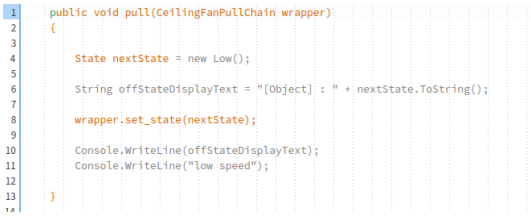
```
using System;
1 reference
class High : State
{

    1 reference
    public void pull(CeilingFanPullChain wrapper)
    {
        State nextState = new Off();

        String HighStateDisplayText = "[Object] : " + nextState.ToString();

        wrapper.set_state(nextState);

        Console.WriteLine(HighStateDisplayText);
        Console.WriteLine("turning off");
    }
}
```

(c) Entrevue de la classe High

```
using System;
2 references
class Off : State
{

    1 reference
    public void pull(CeilingFanPullChain wrapper)
    {

        State nextState = new Low();

        String offStateDisplayText = "[Object] : " + nextState.ToString();

        wrapper.set_state(nextState);

        Console.WriteLine(offStateDisplayText);
        Console.WriteLine("low speed");

    }
}
```

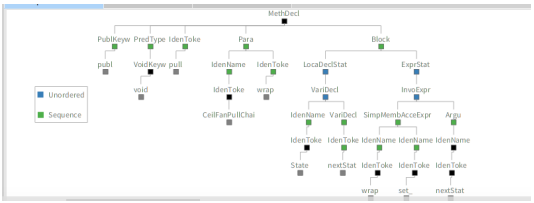
(d) Entrevue de la classe Off

FIGURE 2.2 – Entrevue du code des classes de VentilatorState



```
public void pull(CeilingFanPullChain wrapper)
{

    State nextState = new Low();

    String offStateDisplayText = "[Object] : " + nextState.ToString();

    wrapper.set_state(nextState);

    Console.WriteLine(offStateDisplayText);
    Console.WriteLine("low speed");

}
```

(a) Patterns



(b) AST

FIGURE 2.3 – Pattern et AST correspondant de VentilatorState avec *listWhiteLabel* vide

grammaire construite lors de cette exploration se trouve en annexe dans le dossier "VentilatorState\importerNotPatched\withoutListWhiteLabel\grammarBuilt" et la partie sur laquelle nous allons nous pencher, sur la figure 2.4.

Le nœud "*VariableDeclarator*" a pour enfants "*Identifier*" et "*Initializer*". Le nœud enfant "*Identifier*" est obligatoire ; ce nœud a pour enfant le nœud "*IdentifierToken*" qui est lui aussi obligatoire et qui contient lui-même comme enfant une feuille de l'arbre, "LowStateDisplayText", "MediumStateDisplayText", "HighStateDisplayText" ou "OffStateDisplayText", selon sa classe.

```
VariableDeclarator=[Identifier¥true, Initializer¥false]
Identifier=[IdentifierToken¥true]
IdentifierToken=[Value¥true]

InvocationExpression=[Expression¥true, ArgumentList¥true]
ArgumentList=[Argument¥true]
Argument=[Expression¥true]
Expression=[IdentifierName¥false]
IdentifierName=[Identifier¥true]
```

FIGURE 2.4 – Entrevue des nœuds de la grammaire construite. Le nœud enfant est mis à vrai s'il est obligatoire

Une représentation de l'effet de la grammaire construite par le "*pattern miner*" sur les ASTs créés par le *source code importer* dans le cadre notre exemple est visible sur la figure 2.5.

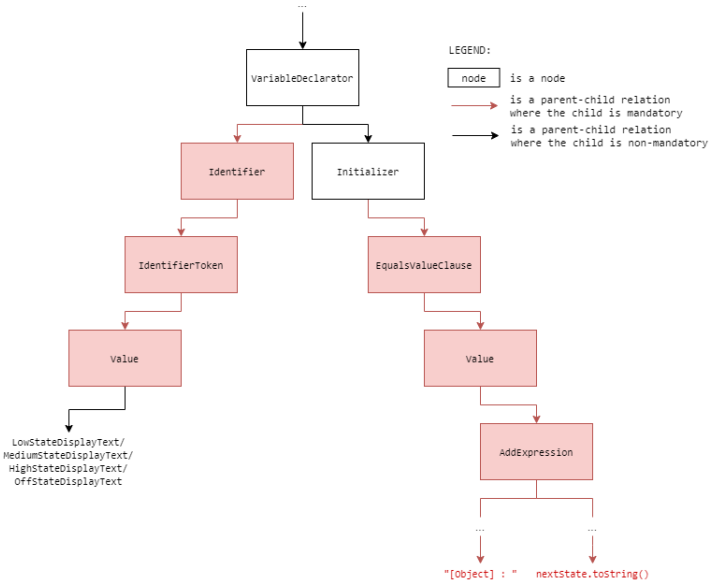


FIGURE 2.5 – Partie de l'AST simplifié des classes du projet VentilatorState

Les nœuds contenant '*String ... = "[Object] : " + nextState.toString() ;*' ne sont alors pas compris dans le pattern trouvé par le miner. En effet, un nœud obligatoire doit faire partie du pattern si son père en fait partie. Pour que le sous-pattern [5] indiqué ci-dessus fasse partie du pattern retourné par le "*pattern miner*", il faut que le nœud "VariableDeclarator" en fasse partie. Ce qui implique que son enfant "Identifier" en fasse partie ainsi que, par récurrence, l'enfant de son enfant "IdentifierToken",

5. Sous-ensemble du pattern.

etc. Jusqu'à en arriver à une feuille de l'arbre, ici, cette feuille est le nœud ayant pour valeur le nom de la variable. Les différents arbres ne sont donc pas similaires, le sous-pattern susdit ne peut faire partie du pattern retourné par le "*pattern miner*".

Plus généralement, les sous-patterns comprenant les déclarations et les assignations de variables mais avec un nom différent ne font pas partie des patterns retournés.

Un autre exemple peut être pris en compte : celui du nœud "*ArgumentList*", enfant de "*InvocationExpression*", le problème est semblable à celui énoncé juste avant. Le sous-pattern problématique ne faisant pas partie du pattern retourné est '*Console.WriteLine(...);*'. Le nœud "*ArgumentList*" est obligatoire pour le nœud "*InvocationExpression*", le nœud "Argument" également pour le nœud "*ArgumentList*", ... Ainsi, les arguments de la méthode '*Console.WriteLine()*' se doivent d'être identiques afin d'appartenir au pattern retourné par le "*pattern miner*".
Ce problème n'arrive pas si l'appel de méthode se fait sans aucun paramètre. De ce fait, un pattern comprenant l'appel est retourné, aucun nœud ne différant parmi les ASTs de ces appels de méthode.

Plus généralement, les sous-patterns comprenant des appels de méthodes mais avec des paramètres différents ne font pas partie des patterns retournés.

Nous nous penchons spécialement sur les nœuds "*VariableDeclarator*" et "*InvocationExpression*", car ils font partie des idiomes les plus fréquents dans les méthodes d'un code écrit en C#, ainsi que d'autres langages de programmation.

2.3 Solution

Dans cette section, nous proposons deux solutions : d'une part, l'utilisation d'un fichier de configuration déjà présent dans le framework INTiMALS, *listWhiteLabel*, et d'autre part, la création d'un autre fichier de configuration *nonMandatoryLabel* qui sera combiné avec le fichier de configuration précédemment cité.

2.3.1 Utilisation du fichier de configuration : *listWhiteLabel*

La configuration de *listWhiteLabel* permet d'améliorer la recherche lors de l'exploration en augmentant la paramétrisation de celle-ci. Le contenu de *listWhiteLabel*

réglant une partie des problèmes est visible sur la figure 2.6. Dans ce fichier, il est stipulé de ne regarder que la partie initialisation de la déclaration d'une variable, blacklistant de cette manière le nom de celle-ci. La deuxième ligne de ce fichier indique de n'étendre que la partie de la fonction lors de l'appel d'une fonction, excluant les paramètres de cet appel.

```
VariableDeclarator Initializer
InvocationExpression Expression
```

FIGURE 2.6 – Entrevue du fichier *listWhiteLabel* complété

(a) Patterns

(b) AST

FIGURE 2.7 – Pattern et AST correspondant de VentilatorState avec *listWhiteLabel* de la figure 2.6

Le résultat [6] de cette exploration avec cette configuration amène une complétion intéressante du pattern visible sur la figure 2.7. En effet, l'ajout de cette configuration permet d'agrandir le pattern trouvé par le "*pattern miner*". La précision de "*VariableDeclarator Initializer*" et "*InvocationExpression Expression*" dans le fichier de configuration *listWhiteLabel* implique donc que le *pattern miner* ne développe que le nœud enfant *Initializer* du nœud parent *VariableDeclarator* et le nœud enfant *Expression* du nœud parent *InvocationExpression*. Les autres enfants des nœuds parents cités ne sont donc pas étendus et ne se retrouveront donc pas dans les patterns potentiels lors de l'utilisation de FreqT, comme nous pouvons le voir sur la figure 2.8.

Cette technique permet d'améliorer les patterns trouvés. Néanmoins, elle entraîne la disparition de certains d'entre eux. Dans le cas de notre exemple, "*State nextState*" disparaît et la taille du sous-pattern "*wrapper.set_state(nextState);*" diminue. En effet, les nœuds enfants *Identifier* et *ArgumentList* de nœuds parents

6. Configuration et résultats complets de cette exploration dans le dossier "VentilatorState\importerNotPatched\withListWhiteLabel" de l'annexe.

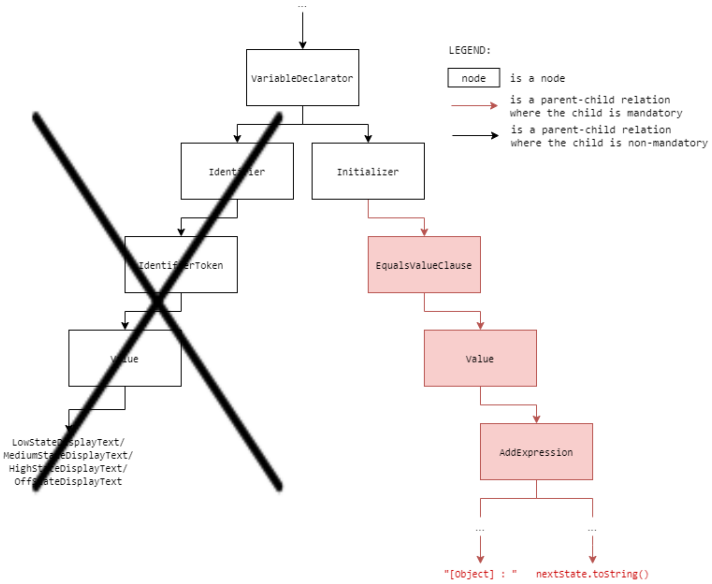


FIGURE 2.8 – Partie de l'AST simplifié des classes du projet VentilatorState avec un fichier *listWhiteLabel* complété blacklistant le nœud Identifier

VariableDeclarator et *InvocationExpression* sont blacklistés. Ceci entraîne que les noms de variables et d'arguments des appels de méthode ne sont plus inclus dans les patterns retournés.
Cependant, cette configuration est utile pour ne pas explorer la partie inintéressante des ASTs. Si ce fichier de configuration est utilisé pour améliorer la quantité ou la richesse des patterns, il exige une certaine connaissance des classes du code source en entrée pour éviter le risque de perdre les patterns comprenant des nœuds inclus dans les nœuds blacklistés.

Un autre problème qui peut empêcher le *pattern miner* de trouver des patterns peut venir de "*source code importer*". Si l'AST généré par ce dernier est mal formé, certains nœuds deviennent obligatoires à tort. Dans le cas de notre exemple, nous constatons que le nœud *ArgumentList* ne devrait pas être obligatoire. Effectivement, dans le code, il se trouve des appels de méthodes qui n'ont pas d'argument ; par exemple 'nextState.toString()'. L'AST généré contient une feuille *ArgumentList* qui est sans valeur. Le code donné en entrée est pourtant assez riche pour construire une grammaire ne contenant pas *ArgumentList* comme étant un enfant obligatoire pour le nœud *InvocationExpression*.

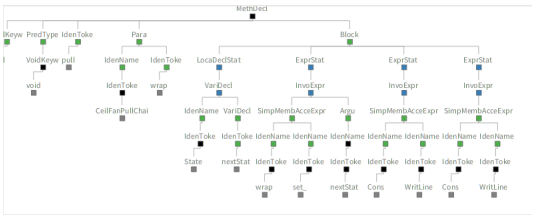
En utilisant un "*source code importer*" patché, qui ne permet pas des feuilles

sans valeur, il en résulte que nous trouvons plus de patterns (voir figure 2.9) [7].

Par conséquent, *ArgumentList* n'est plus un nœud obligatoire, ce qui rend donc les patterns constitués du nœud *InvocationExpression* trouvables.

```
1   public void pull(CeilingFanPullChain wrapper)
2   {
3       State nextState = new Medium();
4       String lowStateDisplayText = "[Object] : " + nextState.ToString();
5
6       wrapper.set_state(nextState);
7
8       Console.WriteLine(lowStateDisplayText);
9       Console.WriteLine("medium speed");
10  }
11
```

(a) Patterns



(b) AST

FIGURE 2.9 – Pattern et AST correspondant de VentilatorState avec *listWhiteLabel* vide et le *source code importer* patché

2.3.2 Ajout d'un fichier de configuration : *nonMandatoryLabel*

La grammaire a une grande importance lors du mining de patterns. Une configuration visant la grammaire peut donc être envisageable pour augmenter la quantité des patterns trouvés. Le fichier "***nonMandatoryLabel***" regroupe les nœuds qui ne seront jamais considérés comme obligatoires dans la grammaire construite, peu importe l'état du code source en entrée.

Le contenu de ce fichier est présenté comme suit, chaque ligne a une structure de la forme : "A B C" où A, B et C sont des nœuds des ASTs construits par le *source code importer*. Ceci signifie que les nœuds enfants B et C du nœud A ne seront pas obligatoires lors du mining.

L'ajout de ce fichier permet une approche plus généraliste que celles mentionnées auparavant ; il s'agit du comportement spécifique d'un langage plutôt que celui dépendant de l'état du code source en entrée. Par exemple, en C#, nous savons que les déclarations de variables sont décomposées en plusieurs parties. La première indique le type de la variable, la seconde, le nom de celle-ci. Ces deux parties obligatoires peuvent être suivies de l'initialisation de la variable. Ce comportement, général en C# ainsi que dans d'autres langage de programmation, entraîne la perte de certains patterns dans le cas où le code source fourni en entrée n'est pas assez

7. Configuration et résultats complets de cette exploration dans le dossier "VentilatorState\importerPatched\withoutNonMandatoryLabel" de l'annexe.

riche. Effectivement, si aucune variable n'est assignée juste après sa déclaration, la grammaire construite par le "*pattern miner*" ne comprend pas que la partie assignation est facultative. Comme expliqué précédemment, les parties interprétées comme étant obligatoires doivent être incluses dans le pattern retourné, il n'est donc pas possible de retrouver des patterns ne comprenant que la partie du type et du nom de la variable lors de la déclaration de celle-ci.

Avec l'utilisation de cette nouvelle configuration, il sera, par exemple, possible d'indiquer que la partie du nom de la variable lors de la déclaration de celle-ci n'est pas obligatoire, ce qui permettra de trouver des patterns de déclarations de variables n'ayant pas nécessairement le même nom.

Nous pouvons mettre en parallèle l'utilisation du fichier *nonMandatoryLabel* et l'utilisation du fichier *listWhiteLabel*, dans lequel sont indiqués les nœuds que le "*pattern miner*" étendra et qui aura pour conséquence de blacklister les nœuds non répertoriés. Remarquons que les patterns contenant les nœuds blacklistés ne sont pas inclus dans les patterns en sortie de l'exploration. L'utilisation du fichier *listWhiteLabel* peut donc être risquée étant donné son caractère de bannissement de certains nœuds.

En utilisant la configuration de *nonMandatoryLabel*, nous allons permettre au *pattern miner* d'étendre les nœuds considérés comme blacklistés auparavant. Ainsi, le nombre des patterns trouvés est plus élevé, ce qui augmente la possibilité d'en trouver qui soient intéressants. Notons que ce fichier de configuration peut aussi être utilisé pour l'exploration d'autres langages de programmation que le C#.

Dans le cas de notre exemple, l'utilisation de cette configuration mène à des résultats plus riches comme il est visible sur la figure 2.12 [8]. En effet, en se concentrant sur la partie de l'exemple que nous avions avec la figure 2.11, nous pouvons voir que, avec l'utilisation du *nonMandatoryLabel*, le nœud "*Identifier*" n'est pas un enfant obligatoire du nœud "*VariableDeclarator*".

Grâce à cette approche, tous les idiomes sont repris dans les résultats. Le contenu du fichier *nonmandatoryLabel* est également visible sur la figure 2.10.

Cependant, l'utilisation de cette configuration exige une connaissance du langage de programmation et surtout des ASTs exportés par le *source code importer*, et ceci pour savoir quels nœuds indiquer dans le fichier *nonMandatoryLabel*.

8. Configuration et résultats complets de cette exploration dans le dossier "VentilatorState\importerPatched\withNonMandatoryLabel" de l'annexe.

```
VariableDeclarator Identifier
InvocationExpression ArgumentList
MethodDeclaration ParameterList
ObjectCreationExpression ArgumentList
```

FIGURE 2.10 – Entrevue du fichier *nonMandatoryLabel* complété

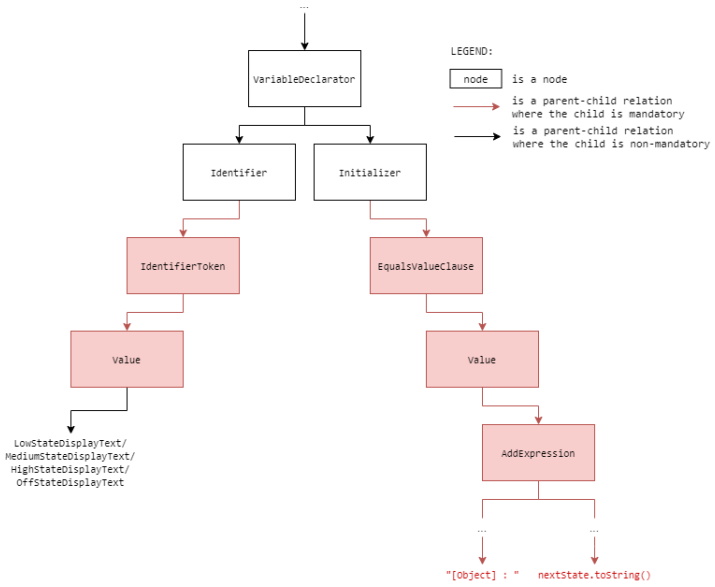
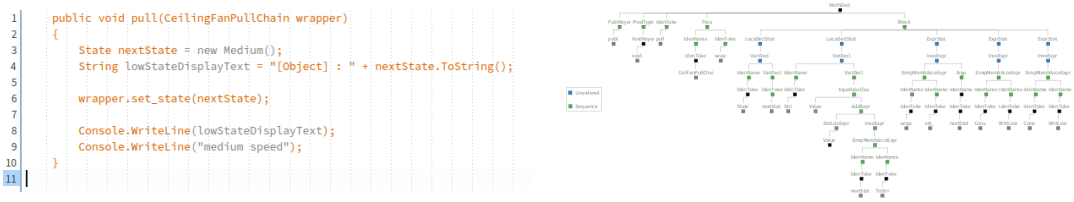


FIGURE 2.11 – Partie de l'AST simplifié des classes du projet VentilatorState avec un fichier *nonMandatoryLabel* complété rendant le nœud *Identifier* non-obligatoire



```
public void pull(CeilingFanPullChain wrapper)
{
    State nextState = new Medium();
    String lowStateDisplayText = "[Object] : " + nextState.ToString();

    wrapper.set_state(nextState);

    Console.WriteLine(lowStateDisplayText);
    Console.WriteLine("medium speed");
}
```

(a) Patterns

(b) AST

FIGURE 2.12 – Pattern et AST correspondant de VentilatorState avec l'utilisation de *nonMandatoryLabel* et le *source code importer* patché

Dans ce chapitre, les différentes techniques pouvant répondre à notre problème lié à la notion d'obligation dans la grammaire d'un projet ont été étudiées. L'utilisa-

tion des différents fichiers de configuration mis à notre disposition par le framework, ainsi que la création d'un nouveau fichier ont été proposés comme solution à ce problème.
Néanmoins d'autres paramètres de configuration peuvent également être envisageables pour augmenter la quantité ou la richesse des patterns. Dans le chapitre 4, ces fichiers vont être combinés avec ces paramètres, sur de plus gros projets, afin d'analyser l'adaptabilité du framework INTiMALS sur un projet C# conséquent.
Avant ça, le prochain chapitre mettra en évidence les implémentations effectuées durant ce travail.

Chapitre 3

Implémentations

Durant cette recherche, les composants du framework INTiMALS mis en évidence dans l'adaptation de ce dernier pour du code C# sont le ***"source code importer"*** *et le* ***"pattern miner"****. Ces composants étant déjà existants, quelques adaptations et corrections ont été nécessaires lors de nos recherches, celles-ci seront développées dans ce chapitre.*

3.1 Adaptation du *source code importer*

Le *source code importer*, comme expliqué dans la section 1.2.1, est un des composants du framework INTiMALS. Il prend en entrée les fichiers du code source d'un projet et retourne une représentation de chacun des fichiers, des ASTs. Il s'agit d'une partie très importante de ce framework : c'est la seule qui est dépendante du langage de programmation du projet pris en entrée. Pour cette raison, il est important de bien définir le méta-modèle utilisé dans ce framework [8] et de s'assurer que les ASTs retournés le respectent.

Ce *source code importer* a été implémenté par un étudiant de haute-école lors d'un stage dans l'entreprise Raincode Labs [2].

Le code source est transformé en AST grâce à une API exposée par un SDK .Net appelé "Roslyn" [11]. Cette API transforme du code source en un arbre syntaxique, qui est ensuite parcouru nœud par nœud pour écrire l'AST sous forme d'un fichier XML respectant le méta-modèle utilisé dans le framework INTiMALS.

Quelques modifications ont été apportées à ce *source code importer*.

Tout d'abord, un problème est apparu quant à la structure des ASTs créés par le *source code importer*, ceux-ci ne respectaient pas complètement le méta-modèle

défini par le framework.

Dans un premier temps, nous avons essayé de faire en sorte que le *source code importer* n'inclue plus de feuilles vides dans les ASTs exportés. En effet, lors de la construction de la grammaire, le *pattern miner* vérifie la présence de nœuds, l'inclusion de ces nœuds vides dans les ASTs augmente le risque d'avoir beaucoup de nœuds obligatoires dans la grammaire. Par exemple, un appel à la méthode ‘foo()‘ était exporté comme sur la figure 3.1 (dans laquelle nous avons enlevé les propriétés des nœuds pour plus de clarté). Par conséquent, le nœud "ArgumentList" devenait obligatoire pour le nœud "InvocationExpression" lors de la construction de la grammaire. Le *pattern miner* retrouvant ce nœud chaque fois qu'il trouvait "InvocationExpression", même quand ce dernier n'avait pas d'argument. Le nœud "ArgumentList" n'aurait pas dû être présent dans les enfants du nœud "InvocationExpression" car il était vide.

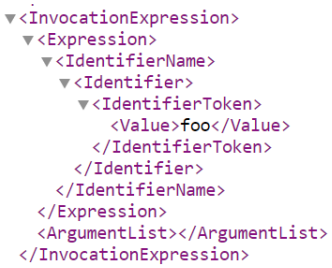


FIGURE 3.1 – Représentation simplifiée de l'AST exporté grâce à l'ancien *source code importer* contenant l'appel de méthode ‘foo()‘

Deuxièmement, plusieurs problèmes empêchaient le *pattern matcher* de faire la correspondance entre le code et la représentation des patterns trouvés sous forme d'AST. En effet, le *pattern matcher*, expliqué en section 1.2.1, s'occupe de mettre en lien les patterns trouvés avec le code lisible, plutôt que des éléments sous forme d'AST.

D'une part, certains nœuds étaient dupliqués par le *source code importer*, lorsque ces nœuds faisaient partie d'un pattern, le *pattern matcher* ne pouvait pas faire son travail. La correction de certains nœuds qui présentaient ce problème a été faite. Par exemple, sur la figure 3.2, est présenté un cas où le nœud "Block" est dupliqué.

D'autre part, sur la même figure 3.2, il est visible que chaque nœud d'un AST contient plusieurs attributs :

— **ID** : Chaque nœud d'un AST a un ID différent, lui servant d'identifiant.

— **LineNr** : Le numéro de ligne du début du code dans le fichier que représente le nœud. Un nœud peut représenter une variable ou un appel de méthode qui s'écrit généralement en une ligne, mais il peut aussi être un "Block", pouvant représenter le contenu d'une méthode, celui-ci s'écrivant sur plusieurs lignes.

— **EndLineNr** : Le numéro de ligne de la fin du code dans le fichier que représente le nœud.

— **ColNr** : Le numéro de colonne du début du code dans le fichier que représente le nœud.

— **EndColNr** : Le numéro de colonne de la fin du code dans le fichier que représente le nœud.

Ces attributs sont aussi utilisés par le *pattern matcher* dans la correspondance, il est donc important que ces numéros soient les bons. Or, ceux-ci ont dû être corrigés.

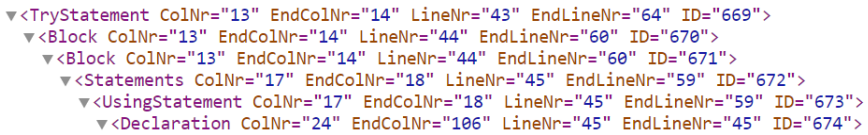


FIGURE 3.2 – Représentation simplifiée de l'AST exporté grâce à l'ancien *source code importer* contenant l'appel de méthode 'foo()'

Cependant, il n'a malheureusement pas été possible de corriger tous les cas des problèmes mentionnés ci-dessus. La correction de ceux-ci pourrait possiblement compléter les patterns retournés dans les expériences faites lors du chapitre 4.

3.2 Adaptation du *pattern miner*

Le *pattern miner* est le composant du framework INTiMALS recherchant les patterns dans les ASTs créés par le *source code importer*, il est plus amplement expliqué dans la section 1.2.1. Pour l'ajout du fichier *nonMandatoryLabel*, ce composant a dû être modifié.

Comme il est expliqué dans la section 1.2.2, quand le paramètre de configuration 'buildGrammar' est à vrai, le *pattern miner* construit lui-même la grammaire sur

base des ASTs reçus en entrée. La grammaire est donc une structure qui définit la relation entre des nœuds et leurs enfants, indiquant lesquels sont obligatoires ou non. Prenons, par exemple, la grammaire montrée sur la figure 3.3, la première ligne *"VariableDeclarator=[Identifier = true, Initializer = false]"* signifie que l'enfant *Identifier* du nœud *VariableDeclarator* est obligatoire mais que l'enfant *Initializer* ne l'est pas.

```
VariableDeclarator=[Identifier¥true, Initializer¥false]
Identifier=[IdentifierToken¥true]
IdentifierToken=[Value¥true]

InvocationExpression=[Expression¥true, ArgumentList¥true]
ArgumentList=[Argument¥true]
Argument=[Expression¥true]
Expression=[IdentifierName¥false]
IdentifierName=[Identifier¥true]
```

FIGURE 3.3 – Aperçu d'un exemple de grammaire.

Une fois que la grammaire est construite, le *pattern miner* itère sur les nœuds de cette grammaire en vérifiant s'ils sont répertoriés dans le fichier *nonMandatoryLabel*. Dans le cas où ils sont présents, les enfants mentionnés, sont rendus non-obligatoires, affectés à faux dans la grammaire.
Par exemple, si nous considérons la grammaire montrée sur la figure 3.3, et le fichier *nonMandatoryLabel* visible à la figure 3.4a ; lorsque le *pattern miner* arrivera au nœud *VariableDeclarator* dans son itération sur la grammaire, la ligne *"VariableDeclarator Identifier"* dans le fichier *nonMandatoryLabel* sera considérée pour affecter *Identifier* à faux et donc le rendre non-obligatoire. Le résultat final de la grammaire est visible sur la figure 3.4b.

Cependant l'utilisation du fichier de configuration *nonMandatoryLabel* n'est pas obligatoire, notre idée n'est pas de l'imposer lors de l'exploration de code source, il s'agit d'une aide supplémentaire à la paramétrisation de celle-ci, c'est donc pourquoi ce fichier est facultatif.

```
VariableDeclarator Identifier
InvocationExpression ArgumentList
```

(a) *nonMandatoryLabel*

```
VariableDeclarator=[Identifier¥false, Initializer¥false]
Identifier=[IdentifierToken¥true]
IdentifierToken=[Value¥true]

InvocationExpression=[Expression¥true, ArgumentList¥false]
ArgumentList=[Argument¥true]
Argument=[Expression¥true]
Expression=[IdentifierName¥false]
IdentifierName=[Identifier¥true]
```

(b) *nouvelle grammaire*

FIGURE 3.4 – Aperçu du fichier *nonMandatoryLabel* et la nouvelle grammaire après son application sur la grammaire de la figure 3.3

Dans ce chapitre nous avons expliqué les implémentations effectuées durant notre travail ainsi que les raisons pour lesquelles ces implémentations ont été faites. Elles ont été documentées et testées avant d'être appliquées.
Lors du prochain chapitre, les solutions proposées aux problèmes fournis lors du chapitre 2, qui ont notamment été mises en place par les implémentations décrites dans ce chapitre, vont être mises en évidence. Différents projets de taille conséquente seront explorés afin d'analyser l'adaptabilité du framework INTiMALS sur un projet C#.

Chapitre 4

Validation

L'adaptabilité du framework INTiMALS sur un code source C# se mesure grâce aux expériences sur de plus gros projets que celui composé de quelques classes utilisé lors du chapitre 2, dont les analyses sont détaillées dans ce chapitre.
De plus, ces analyses mettent en évidence l'utilisation du nouveau fichier de configuration, ***"nonMandatoryLabel"****, sur des projets C# ainsi que sur un projet dans autre langage de programmation, Java, analysé dans le passé lors des recherches antérieures afin d'expérimenter l'adaptabilité du framework INTiMALS sur du code source Java [14].*

4.1 Méthode de validation

Dans cette section, les expériences sont faites sur des projets existants de taille conséquente. Le framework INTiMALS permettra l'exploitation ainsi qu'une analyse du point de vue d'un programmeur de ces projets. L'état des résultats de ces derniers sera mis en évidence afin d'illustrer la qualité et la quantité des patterns qui ressortent.

Les projets, en question, utilisés sont ;

— **Nunit** [4], un framework de tests unitaires, pour tous les langages .Net, écrit en C#. Il est développé et maintenu par .Net Fondation.
— **Notepads** [5], un éditeur de texte léger écrit en C#.
— **JHotDraw** [10], un framework Java pour l'édition de dessins et de graphiques. Il est écrit en Java.

Ces projets ont été choisis car ils sont tous open-source ; i.e. leur code source est

libre d'accès et le développement est participatif, quiconque peut modifier leur code et le redistribuer. L'analyse de chacun sera plus détaillée dans la section ci-après.

Les expériences expliquées lors de ce chapitre sont menées à l'aide d'un ordinateur portable sous Windows 10 avec un processeur Core i3 d'Intel.

Ce mémoire se faisant dans le contexte du projet de recherche INTiMALS, qui est une collaboration entre plusieurs entités dans le domaine informatique, la langue usuelle est par conséquent l'anglais. Certains termes ainsi que les légendes des graphes fournis lors de cette validation seront donc également en anglais.

Lors de la validation, nous allons tout d'abord énoncer la structure que nous utiliserons pour détailler nos différentes expériences. Ensuite, nous analyserons deux projets C#, pour finir avec un projet Java sur lequel nous analyserons l'impact de l'ajout du fichier de configuration *nonMandatoryLabel*.

4.2 Résultats

L'ensemble des solutions que nous avons énoncées plus haut, c'est-à-dire l'utilisation réfléchie des fichiers de configuration *listWhiteLabel* et *nonMandatoryLabel* ainsi que d'un *source code importer* patché, sont appliquées dans chacune des expériences effectuées.

Les résultats des expériences sont présentés de deux manières différentes. Tout d'abord, une analyse quantitative qui résume une comparaison, en matière de chiffres, des patterns intéressants qui ressortent avec différents paramètres de configuration. Ensuite, une analyse qualitative qui met en évidence certains patterns riches considérés comme intéressants à l'égard d'un programmeur.

Les patterns sont divisés en 3 classes de niveau d'intérêt :

— **Interesting**, contient les idiomes utiles pour un programmeur, ainsi que les bouts de codes présentant à un novice du langage de programmation dans lequel le projet est écrit la syntaxe et la sémantique de ce langage. Un pattern jugé intéressant est, par exemple, l'utilisation d'un design pattern, de bonnes ou de mauvaises pratiques de programmation, etc.

— **Intermediate**, regroupe les bouts de code trop petits pour être intéressants mais qui pourraient l'être si le contexte était plus détaillé dans le pattern.

Par exemple, un appel de méthode en plein milieu d'un long bout de code, etc.

— **Not interesting**, contient les bouts de code inintéressants car les idiomes n'apprennent rien au développeur. Par exemple, des déclarations des variables, des méthodes utilitaires comme toString(), etc.

Les paramètres de configuration abordés durant **l'analyse quantitative** sont :

— ***Nodes size*** : le nombre de nœuds que couvre le pattern dans l'AST. Il est utilisé pour estimer la taille des patterns.

— ***Support*** : le nombre des fichiers différents dans lesquels le pattern est présent.

Lors de nos expériences, les paramètres de configuration ci-dessus seront discutés. Le principal élément sur lequel nous allons discuter durant ces expériences sera le niveau d'intérêt des patterns. Grâce à ce niveau d'intérêt, nous pouvons calculer la précision et le rappel d'une expérience, notion expliquée dans la section 1.1.4. Dans notre cas, le rappel est difficilement calculable, il s'agit de la proportion d'éléments pertinents retrouvés lors d'une recherche parmi l'ensemble des éléments pertinents existants. Un élément pertinent sera ici un pattern intéressant. Il est impossible de connaître le nombre de patterns intéressants existant dans un projet de taille conséquente. Nous allons plutôt, pour chaque expérience, nous pencher sur le rappel relatif à une autre expérience. Nous calculerons donc la proportion d'éléments pertinents retrouvés lors d'une recherche parmi l'ensemble des éléments pertinents retrouvés lors d'une autre recherche.

La plupart des structures syntaxiques encapsulant différents idiomes de programmation intéressants se trouvent dans le contenu des méthodes et dans leurs définitions ; par exemple, au sein d'une classe, l'ensemble des méthodes redéfinies dues à l'héritage d'une classe parent ou, dans le contenu d'une méthode, l'utilisation de librairies.
C'est pourquoi les nœuds représentant ces derniers dans un AST sont utilisés dans le fichier de configuration ***listRootLabel*** afin d'augmenter les chances du *pattern miner* de trouver des patterns couvrant le plus d'idiomes intéressants possible.

Quant à **l'analyse qualitative**, il y est présenté certains patterns intéressants et riches du point de vue d'un programmeur. Ces patterns contiennent des idiomes qui pourraient aider l'utilisateur à comprendre les fonctionnalités, la structure ou le langage de programmation du projet en question.

En l'occurrence, les patterns intéressants sont classés dans des groupes sur base des structures syntaxiques et des conventions de programmation s'y trouvant. Ces groupes sont appelés des "*tags*" dans la suite de ce document.

Un même "***tag***" regroupe des patterns ayant directement ou indirectement la même forme syntaxique ou utilisant les mêmes pratiques de codage. Les patterns intéressants trouvés durant nos expériences sont classifiés dans la liste non exhaustive de tags suivante :

— **Design pattern :** Les patrons de conception, comme expliqué dans la section 1.1.3, sont une norme importante du génie logiciel. La présence d'un patron de conception donne de grandes indications au développeur. Grâce aux patrons de conception, la lisibilité du code est beaucoup plus grande, aidant les développeurs à comprendre comment ce code est structuré, et rendant l'extension des fonctionnalités plus aisée.
Sous ce tag se trouvent les implémentations de patrons de conception mais aussi de toute forme de code répondant à la définition d'un patron de conception, sans être obligatoirement référencé comme tel.

— **Bad habit :** L'ensemble des mauvaises habitudes que peut avoir un développeur lors de l'écriture de son code. Le copié-collé de parties de code pouvant être refactoré est l'exemple le plus répandu. Il est intéressant de ressortir ces patterns d'un code, ils permettent au développeur de comprendre comment le code est structuré, et surtout, comment il peut être amélioré. Ce copié-collé est ici considéré comme une mauvaise habitude mais peut avoir certains avantages [19].

— **Library/Framework usage :** Les projets importants font souvent appel à des librairies existantes, ceci permettant de ne pas réimplémenter une fonctionnalité déjà réalisée de manière optimale. Énormément de librairies existent, pouvant être utilisées de nombreuses manières différentes. Les patterns regroupés sous ce tag informent qu'une certaine librairie est employée pour effectuer une tâche et indique la raison pour laquelle elle est utilisée.

— **Programing convention :** Ce tag regroupe l'ensemble des conventions de codage [22] que l'on peut retrouver dans un projet. Ces conventions ne sont pas toujours intéressantes mais il est normal de retrouver la structure de for loop, de try catch, de getter/setter, ... dans de gros projets faisant l'objet de recherche de régularités. Cependant, lors de l'apprentissage du langage

de programmation dans lequel le code est écrit, ces patterns deviennent très intéressants du fait des conventions et idiomes qui y sont démontrés.

— **Coding practice :** Les bouts de code propres à un projet, dont la structure de l'utilisation informe sur la manière dont sont utilisés les différents tenants et aboutissants de ce projet. Ces bouts de code représentent des pratiques de programmations répandues qui peuvent parfois être considérées comme des bonnes pratiques de codage. L'utilisation des interfaces ou la gestion des exceptions sont des exemples de patterns pouvant se retrouver sous ce tag.

Cependant, tous les patterns intéressants ne sont pas présentés dans ce document, mais les plus pertinents d'entre eux seront détaillés afin d'illustrer leur utilité.

Grâce à cette partie, nous mettons en évidence l'approche que pourrait prendre un programmeur pour inspecter les régularités retournées par le framework ; ceci exige parfois une analyse plus poussée qui demande des connaissances en informatique grâce auxquelles la richesse et la qualité du résultat peuvent être mises en évidence.

Dans la section suivante, les résultats analysés dans chaque projet seront développés.

4.2.1 Nunit

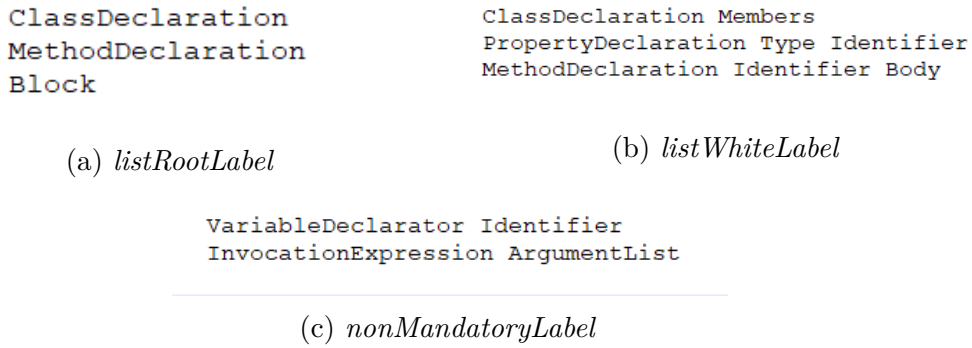
Cette expérience est menée sur un logiciel open-source intitulé "*Nunit*" dans sa troisième version. Nunit [4] est un framework de tests unitaires pour tous les langages .Net écrit en C#.
Cependant, son répertoire étant très volumineux, l'exploration sera faite sur un ensemble de 183 fichiers, tirés du package "*framework*".

Analyse quantitative

A l'aide de l'utilisation réfléchie des fichiers de configuration *listRootlabel*, *listWhiteLabel* et *nonMandatoryLabel* (voir la figure 4.1), **65 patterns** ont été trouvés par le "*pattern miner*" sur l'ensemble de 183 fichiers.
Les déclarations de classes ("ClassDeclaration") et le contenu de méthodes ("MethodDeclaration, Block") seront utilisés comme *listRootlabel* dans le but d'obtenir de riches résultats lors de l'exploration.

Grâce au fichier *listWhiteLabel*, l'exploration s'intéresse au sous-arbre "Members", enfant de "ClassDeclaration". Pour les variables ("PropertyDeclaration"), l'exploration inspecte le type et le nom de celles-ci, blacklistant par conséquent le modificateur d'accès. Enfin, pour les méthodes ("MethodDeclaration"), les modificateurs d'accès et les types de retour de ces dernières sont également blacklistés.
Le fichier *nonMandatoryLabel* est complété suite à la résolution de problèmes que nous avons rencontrés avec les paramètres des appels de méthode ou encore les déclarations et assignations de variables expliqués lors du chapitre 2.

Les différentes contraintes données à l'algorithme FreqT en plus des fichiers de configuration sont visibles sur la figure 4.2.
L'exploration prend environ **26 minutes** à s'exécuter.



(a) *listRootLabel*

(b) *listWhiteLabel*

(c) *nonMandatoryLabel*

FIGURE 4.1 – Aperçu des fichiers de configuration *listRootLabel*, *listWhiteLabel* et *nonMandatoryLabel* pour l'expérience exécutée sur Nunit

```
minNode=20
minLeaf=2
maxLeaf=5
buildGrammar=true
minSupport=3
```

FIGURE 4.2 – Contraintes générales données lors de l'exploration du projet Nunit

Les 65 patterns trouvés lors de l'exploration ont été analysés manuellement, décrivant leur fonction et leur intérêt. Une grande partie a été dépeinte comme étant des patterns intéressants, comme il est visible sur la figure 4.3. La précision de cette expérience peut être calculée, elle est de 65%, en d'autres mots, plus ou moins deux tiers des patterns trouvés ont été considérés comme intéressants. Cette

précision représente le nombre de patterns intéressants sur le nombre total de patterns retournés par notre expérience.

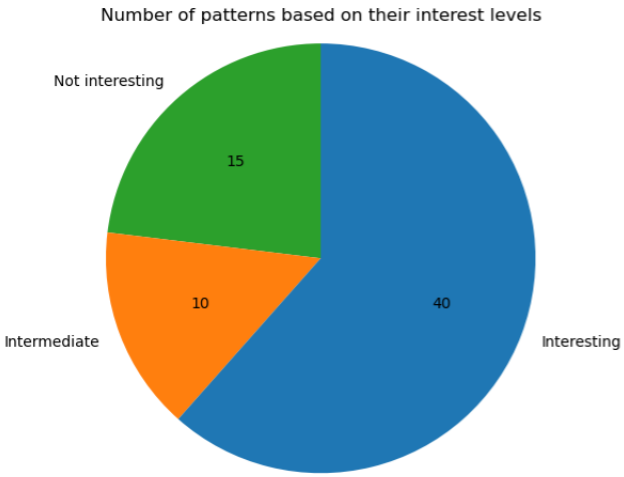


FIGURE 4.3 – Présence des patterns en fonction de leur niveau d'intérêt

Les contraintes ont une grande importance sur l'efficacité de l'exploration. En effet, limiter la recherche de l'algorithme est importante. Le "***minSupport***", nombre minimal d'ASTs de fichiers dans lequel doit se trouver un pattern, a été fixé à une valeur de **3**.
Il est remarquable qu'un "***minSupport***" inférieur à 3 amène à une augmentation exponentielle du temps d'exploration nécessaire. Cette valeur ne pourrait pas être fixée à 2 car les patterns trouvés entre deux fichiers seulement peuvent difficilement être considérés. Effectivement, les régularités entre 2 fichiers dans un projet de plus de 180 fichiers risquent d'être très nombreuses dont la plupart peu représentatives de structures intéressantes à examiner.
Par ailleurs, un "*minSupport*" plus élevé que 3 entraîne une perte en quantité de patterns. De cette manière, ceci diminuerait fortement l'information apprise grâce à l'expérience. Une valeur basse du rappel [1] relatif à l'expérience avec ce "*minSupport*" de 3 en serait donc une conséquence.
Il n'est donc pas intéressant de baser notre expérience avec un "*minSupport*" plus bas ou plus élevé que celui choisi, c'est-à-dire 3. L'aperçu du nombre de patterns en fonction de leur support est visible sur la figure 4.4

Un pourcentage important des patterns sont analysés comme importants. De plus, avec l'histogramme apparent sur la figure 4.5, il est visible que l'augmentation de la contrainte "***minNode***" n'aiderait pas à augmenter la précision[1] de

1. Voir la section 1.1.4.

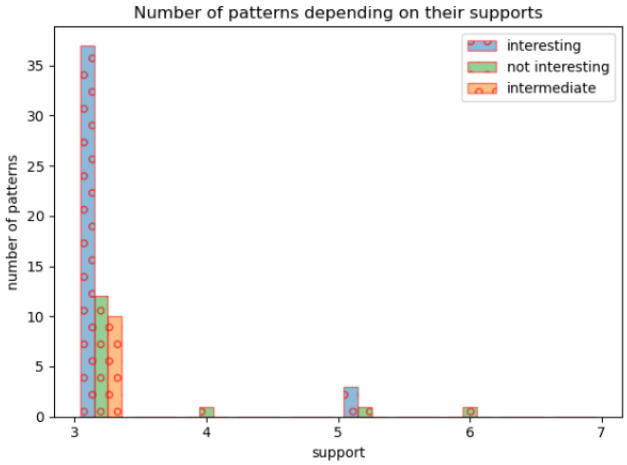


FIGURE 4.4 – Présence des patterns en fonction de leur support

cette exploration. En effet, beaucoup de patterns intéressants ont une taille de 20 nœuds, l'augmentation de cette contrainte expulserait un plus grand nombre de patterns intéressants que de patterns inintéressants, ce qui diminuerait la précision de l'expérience.
La taille moyenne d'un pattern retourné par cette expérience est de 39 nœuds, 56 nœuds pour un pattern intéressant, les quelques patterns intéressants avec une taille de plus de 100 nœuds augmentant cette moyenne.

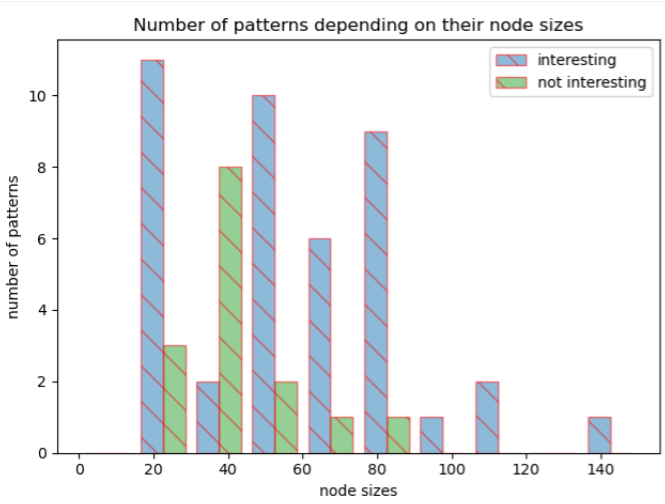


FIGURE 4.5 – Nombre de patterns en fonction du nombre de nœuds dont ils sont composés

Analyse qualitative

En regroupant des patterns intéressants dans différents "***tags***", leur structure syntaxique a été étudiée. Cependant, le code autour, qui n'est pas repris dans le pattern, est aussi analysé afin de mieux comprendre l'utilité et la signification du pattern en question.

L'analyse qualitative nous a amenés à comprendre les différents composants de ce projet. Dans la plupart des cas, le contexte des classes où se trouvent les patterns indique en grande partie leur utilité ; grâce à leur nom, les noms des méthodes ou les conventions de programmation se trouvant dans chacun des patterns. Par exemple, sur le pattern numéro 33, 3 classes, appelées *BeforeAndAfterTestCommand, AfterTestCommand et BeforeTestCommand*, définissant toutes une méthode *execute(...)*, font penser au vu de ces caractéristiques à un patron de conception "commande", ce pattern est expliqué ci-après.

De plus, comme il est visible sur la figure 4.6, nous trouvons également de bonnes pratiques et conventions de programmation ainsi qu'une grande utilisation de librairies.

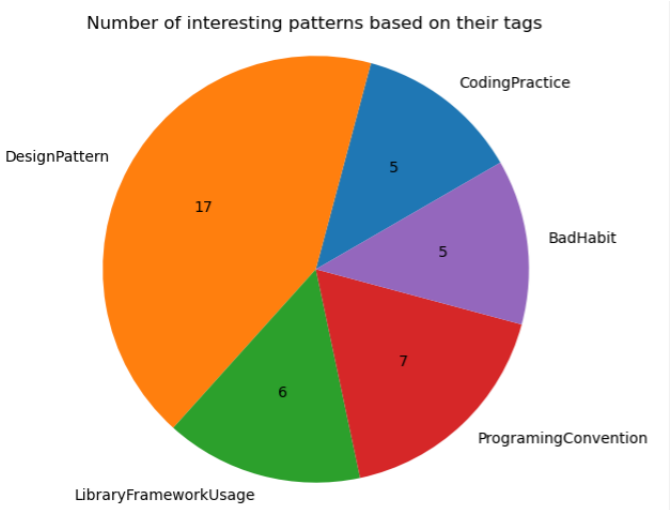


FIGURE 4.6 – Présence des patterns intéressants en fonction du "*tag*" auquel ils ont été assignés

Dans la section suivante sera mise en évidence une liste de quelques patterns qui ont particulièrement attiré notre attention. Un tableau regroupant tous les patterns trouvés lors de cette expérience avec l'ensemble de leurs propriétés est disponible

en annexe dans le dossier "Nunit\analysis". Comme expliqué précédemment, cette analyse demande de disposer de connaissances en informatique.

Patterns intéressants

Pattern numéro 33

Le pattern numéro 33 de notre expérience sur le projet Nunit, nous présente l'implémentation du design pattern *commande*.

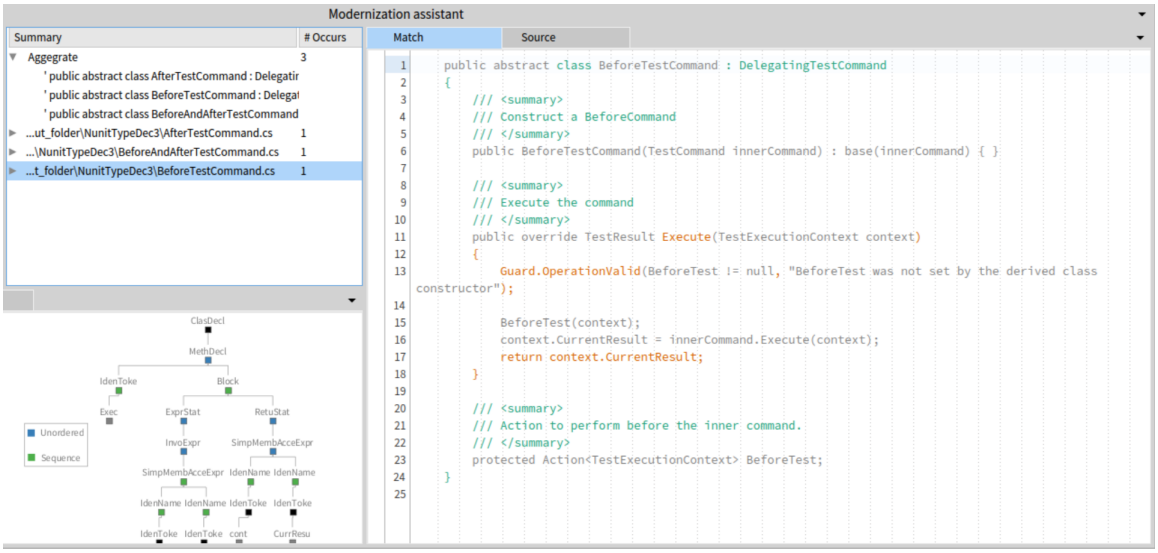


FIGURE 4.7 – **Pattern 33** : Implémentation du design pattern commande

Le design pattern *commande* est un patron de conception comportemental. Comme il peut être vu sur la figure 4.8, ce design pattern commande encapsule une requête sous la forme d'un objet autonome. Cette encapsulation permet de paramétrer la méthode. Ce patron de conception peut aisément isoler une requête.

Pour bien illustrer le lien entre notre pattern, visible sur la figure 4.7, et la structure du patron de conception commande, les composantes de ce patron de conception, apparentes sur la figure 4.8, seront associées à la classe leur correspondant dans le projet.

Les trois classes, *BeforeTestCommand*, *AfterTestCommand* et *BeforeAndAfterTestCommand* correspondent aux **ConcreteCommand**. Elles redéfinissent la méthode *"TestResult Execute(TestExecutionContext context)"* déclarée dans la classe abstraite *DelegatingTestCommand*, cette dernière se référant donc à l'interface **Command**. Le **Receiver**, dans ce cas, est la classe *TestExecutionContext*.

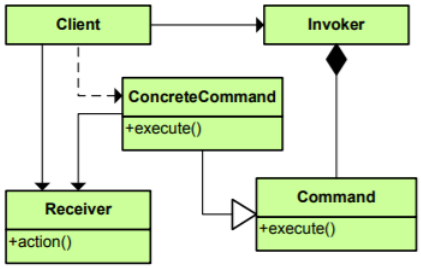


FIGURE 4.8 – Diagramme UML du pattern commande [17]

Quant à l'**Invoker**, il n'est pas visible sur ce pattern, il s'agit de classes faisant appel à la méthode *Execute()* de ces commandes, il s'agira de la classe *SimpleWorkItem* ou *CompositeWorkItem*.

Il est clair désormais que le développeur peut aisément ajouter une commande en créant une classe qui étend *DelegatingTestCommand*, ou encore, créer un Invoker supplémentaire utilisant les commandes citées plus haut dans le but de tester une autre fonctionnalité qui n'est pas prise en compte dans les tests unitaires proposés par Nunit.

Ce pattern est assez enrichissant pour le développeur. En effet, il informe sur la méthode d'implémentation des différents tests. L'observation de l'utilisation d'un design pattern ainsi que sa structure donnent beaucoup d'informations au développeur.

De plus, nous remarquons une bonne pratique dans ce pattern, l'utilisation d'une classe utilitaire *Guard* permettant la vérification des arguments lors de l'exécution d'une méthode. Cette classe *Guard* est composée de méthodes d'assistance permettant la vérification de paramètres.
Grâce à cette classe, nous nous débarrassons des *if statements*[2] redondants souvent utilisés pour vérifier les arguments, leur valeur non nulle, le vide des chaînes de caractères, l'asynchronicité de méthode, ...

Pattern numéro 58

Nous pouvons observer le pattern numéro 58 sur la figure 4.9. Ce pattern présente une régularité connue, l'implémentation d'un patron de conception *stratégie*.

Le design pattern *stratégie* est un patron de conception comportemental permettant de définir une famille d'algorithmes qui peut être interchangée et indépendante

2. Instruction conditionnelle utilisée pour vérifier des conditions booléennes.

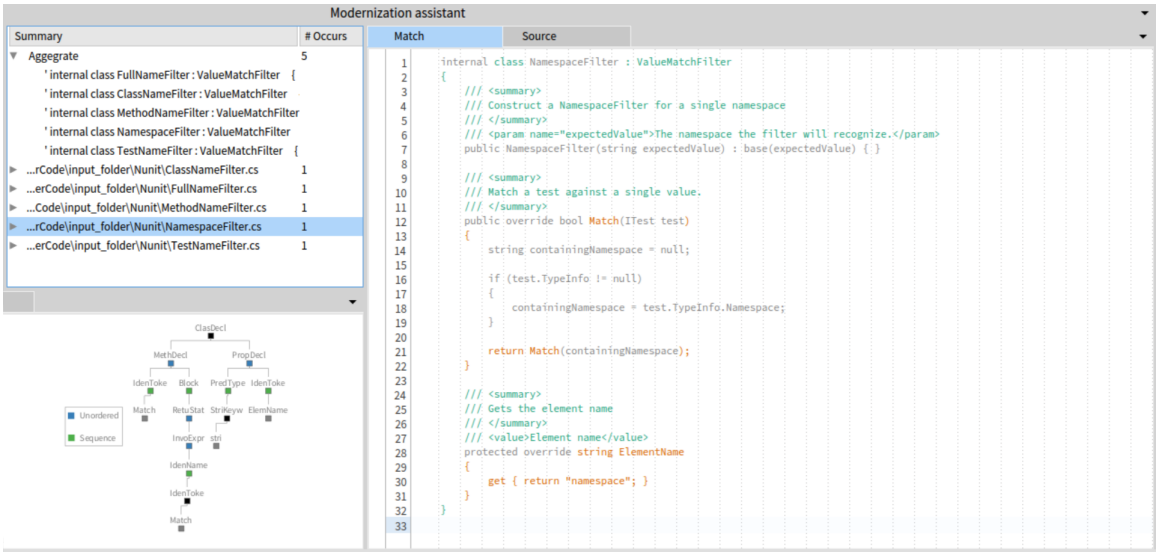


FIGURE 4.9 – **Pattern 58** : Implémentation d'une stratégie

de la partie cliente. Le diagramme UML de ce pattern se trouve sur la figure 4.10. L'interface "Strategy" est implémentée par les différents algorithmes qui sont ici des "ConcreteStrategy". Les clients peuvent alors se coupler à une interface et ne pas avoir à subir les bouleversements associés au changement, lorsque le nombre de classes dérivées change, et lorsque l'implémentation d'une classe dérivée change.

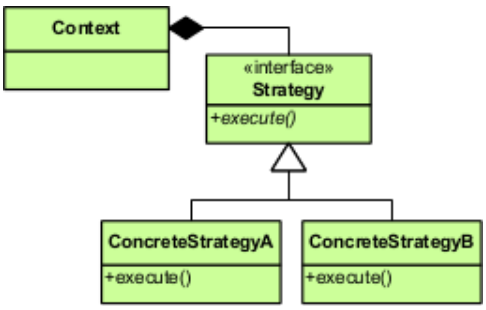


FIGURE 4.10 – Diagramme UML du pattern stratégie [17]

De la même façon que précédemment, nous pouvons lier les composantes de ce patron de conception avec les classes du pattern 58. Les classes, *FullNameFilter, ClassNameFilter, MethodNameFilter, NameSpaceFilter, TestNameFilter*, correspondent à la liste des **ConcreteStrategies**. Elles implémentent la méthode *public override bool Match(ITest test)* de la classe abstraite *ValueMatchFilter* référant à l'interface **Strategy**.

Par la suite, si un dévelopeur désire mettre en place un filtre au comportement différent des stratégies citées ci-dessus, il lui suffira de créer une nouvelle classe qui étend la classe *ValueMatchFilter*, définissant son propre comportement dans la méthode *Match(ITest test)*.

Une autre partie de ce pattern est remarquable. Un "*getter*" est créé sur l'attribut "*ElementName*". Il est notable que la création d'un "*getter*" de cette manière est un idiome propre au langage de programmation C#. Il s'agit en fait d'une propriété. Une **propriété**, selon la documentation Microsoft, "*est un membre qui fournit un mécanisme flexible pour lire, écrire ou calculer la valeur d'un champ privé. Les propriétés peuvent être utilisées comme si elles étaient des membres de données publiques, mais ce sont en fait des méthodes spéciales appelées accesseurs. Cela permet d'accéder facilement aux données tout en favorisant la sécurité et la flexibilité des méthodes.*" [20].
Avec l'existence de ce pattern, le développeur découvre l'existence de cet idiome propre au langage de programmation C#, s'il ne le connaissait pas.

Une deuxième expérience a été menée sur un plus petit projet, Notepads, analysé dans la section suivante.

4.2.2 Notepads

Notepads [5] est un éditeur de texte open-source. Le but de cet éditeur de texte est d'être aussi léger que possible, ce qui explique le petit nombre de fichiers dont le projet est composé, 82 fichiers. Notepads est écrit en C#.

Analyse quantitative

Lors de l'exploration de patterns, les fichiers de configurations, *listRootLabel, listWhiteLabel, nonMandatoryLabel*, utilisés sont les mêmes que ceux de l'expérience précédente et sont visibles sur la figure 4.1. Quant aux contraintes données à l'algorithme FreqT, celles-ci sont également similaires à celles utilisées lors de l'expérience précédente, mis à part la contrainte "minNode" qui va être discutée durant cette analyse. Elles sont observables sur la figure 4.11.
Grâce à l'utilisation de ces fichiers, **48 patterns** ont été retrouvés en **1h30** d'exploration.

Dans ce cas, les patterns ont également été analysés, les résultats de la présence des patterns intéressants sont démontrés sur la figure 4.12. Remarquons que la

```
minNode=10
minLeaf=2
maxLeaf=5
buildGrammar=true
minSupport=3
```

FIGURE 4.11 – Contraintes générales données lors de l'exploration du projet Notepads

précision de cette expérience (48%) est moins grande que celle de la précédente. Cependant, comme évoqué dans la section 1.1.4, il est normal de ne pas avoir une précision proche de 100% si l'on souhaite avoir un rappel acceptable.

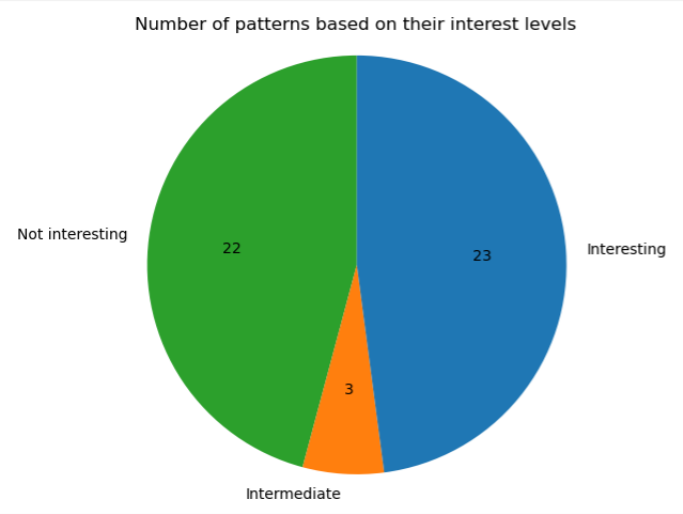


FIGURE 4.12 – Présence des patterns en fonction de leur niveau d'intérêt

De la même manière que l'expérience précédente, nous trouvons des régularités de toutes sortes dans les patterns trouvés. Il est normal que la simple déclaration de variable ou la structure d'une condition ressortent également, ces patterns n'étant pas considérés comme intéressants.

L'histogramme sur la figure 4.13 présente le nombre de patterns retournés en fonction du nombre de nœuds dont sont composés leur AST, appelé *taille du pattern*. Le taille moyenne d'un pattern est de 39 et d'un pattern intéressant, 56.
Dans cette expérience, le "***minNode***" n'a été affecté qu'à une valeur de 10. Or l'histogramme démontre que très peu de patterns ont une taille de 10.

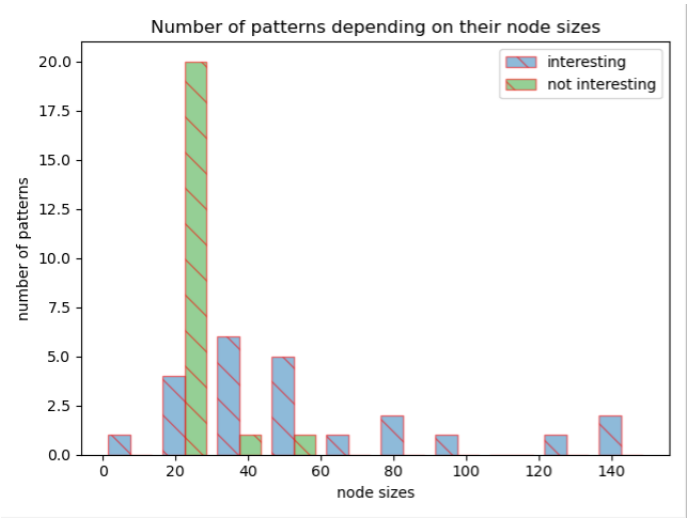


FIGURE 4.13 – Nombre des patterns en fonction du nombre de nœuds dont ils sont composés

C'est pourquoi, dans la suite, nous allons nommer trois expériences avec lesquelles nous allons jouer sur la contrainte "*minNode*" :

— Expérience A : L'expérience sur laquelle se base chacun des graphes ci-dessus, dont la contrainte "*minNode*" a une valeur de 10. Ce sera notre expérience témoin dans le calcul du rappel.

— Expérience B : Expérience sur laquelle nous augmentons légèrement la contrainte "*minNode*", une valeur de 20 lui est attribuée.

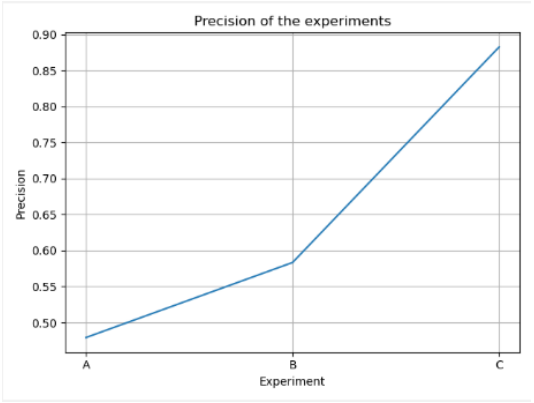
— Expérience C : Expérience sur laquelle nous augmentons plus fortement la contrainte "*minNode*" avec une valeur de 35.

Pour rappel, "l'expérience A" a une précision de 48%. L'exécution de "l'expérience B" entraîne une augmentation de la précision par rapport à cette dernière, en effet, 21 des 36 patterns retournés sont intéressants, la précision est donc de 58%. L'expérience C est encore plus remarquable, la précision est de 88% avec 15 patterns intéressants sur 17 patterns retournés.
Cependant, même si une augmentation de notre "*minNode*" entraîne une élévation importante de la précision, nous pouvons calculer un rappel relatif à "l'expérience A", comportant 23 patterns intéressants. Ce rappel relatif est le ratio du nombre de patterns intéressants dans une certaine expérience sur le nombre de patterns

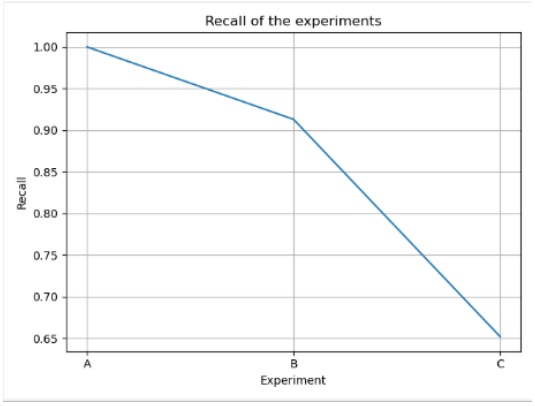
intéressants de "l'expérience A". Nous pouvons voir sur la figure 4.14, l'évolution de la précision et du rappel relatif à "l'expérience A". Est également visible sur la table 4.1 le condensé des résultats énoncés ci-dessus. "L'expérience C" est donc une expérience très précise, mais au rappel relatif très faible.

	Expérience A	Expérience B	Expérience C
Précision	48%	58%	88%
Rappel relatif à l'expérience A	100%	91%	65%

TABLE 4.1 – Tableau des valeurs de la précision et du rappel relatif à "l'expérience A" pour chacune des expériences



(a) Précision des expériences



(b) Rappel des expériences relatif à "l'expérience A"

FIGURE 4.14 – Evolution de la précision et du rappel en fonction de la valeur de *minNode*

Ceci signifie donc que nous sommes plus précis en augmentant la contrainte "*minNode*" mais sur un échantillon beaucoup plus restreint, excluant par ce choix des patterns intéressants de notre résultat. C'est une constatation importante, surtout que, ici, le but est d'avoir un nombre significatif des patterns.

Le choix de ce "*minNode*" dépend, par conséquent, de ce que l'on recherche. Dans notre cas, il est assez important d'avoir un certain nombre de patterns sans pour autant avoir une précision trop faible. Avoir un grand nombre de patterns dont la majorité n'est pas intéressante n'est pas ce que l'on recherche ici.
C'est pour cette raison que le "*minNode*" de 20 serait la valeur la plus pertinente à choisir pour cette expérience au vu des différentes comparaisons expliquées ci-dessus.

Analyse qualitative

L'analyse des patterns trouvés lors l'exploration du projet Notepads amène à un regroupement par "***tags***" que nous pouvons voir sur la figure 4.15, ce regroupement n'est montré que pour les patterns considérés comme intéressants.
La répartition de ces patterns en "*tags*" est proche de celle observée lors de l'expérience du projet Nunit en section 4.2.1, la présence de patrons de conception domine dans ces deux projets.

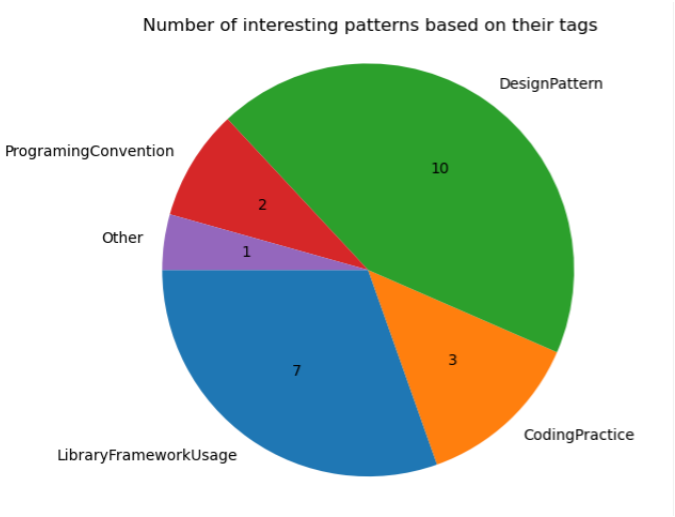


FIGURE 4.15 – Présence des patterns intéressants en fonction du "*tag*" auquel ils ont été assignés

Un tableau regroupant les 48 patterns retournés et leurs propriétés est disponible en annexe dans le dossier "Notepads\analysis". Dans la section suivante, nous allons nous pencher sur un pattern qui a retenu notre attention.

Patterns intéressants

Pattern numéro 12

Le pattern numéro 12 du projet Notepads décrit l'utilisation d'un *try catch*. Il est composé de 62 nœuds dont la racine est un nœud "*Block*", il apparaît 3 fois dans le projet. Ce bout de code se compose d'un bloc *try* suivi d'une ou plusieurs clauses *catch*, qui spécifient comment sont gérées chacune des différentes exceptions qui peuvent arriver dans le bloc *try*.

Ce pattern, visible sur la figure 4.16, a été associé au "*tag*" "Coding practice". En effet, nous pouvons y voir l'utilisation d'un *try catch* qui est une convention de programmation. De plus, celle-ci est accompagnée de l'utilisation d'une librairie de "*logging*" permettant la journalisation des évènements, comme des erreurs, et aidant la compréhension de celles-ci.

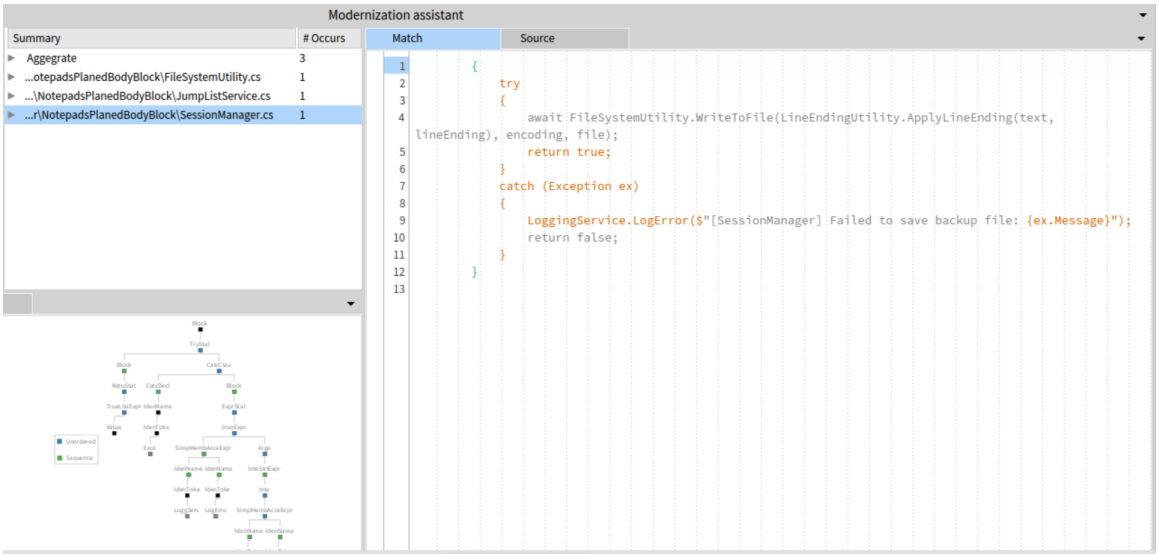


FIGURE 4.16 – **Pattern 12** : Utilisation d'un "*try catch*"

Ceci est un exemple de pattern apportant des informations quant à la pratique utilisée pour la gestion des erreurs. Elles sont capturées afin de ne pas être apparentes aux yeux du client et un message est sauvegardé dans le but de pouvoir retrouver les raisons de ces erreurs.

La section suivante contient l'analyse de l'impact du fichier de configuration *nonMandatoryLabel* sur un projet Java dont l'adaptabilité a déjà été prouvée.

4.2.3 JHotDraw

JHotDraw est un framework graphique en deux dimensions utilisé par les éditeurs de dessins structurés. Ce projet est écrit en Java.

Pour ce projet, notre analyse se concentre principalement sur l'impact de l'application du nouveau fichier de configuration, *"nonMandatoryLabel"* sur un projet déjà analysé sans ce fichier, JHotDraw [14].

Pour analyser cet impact, nous allons mener deux expériences, tout d'abord sans l'utilisation du fichier de configuration *"nonMandatoryLabel"* et ensuite en exploitant ce dernier. Nous nous pencherons avant tout sur les différences que présentent ces deux expériences, sans entrer en détail de l'analyse commune puisque celle-ci a déjà été effectuée dans les recherches précédentes [14].

Définissons les deux expériences que nous allons mener :

— Expérience A : L'expérience ayant été faite durant les précédentes recherches. Dans le fichier *listRootLabel*, nous nous intéressons, de la même façon que pour les expériences précédentes, au contenu des méthodes (i.e. nœud "Block") ainsi qu'à la déclaration de la classe (i.e. nœud "TypeDeclaration") pour favoriser la chance de trouver des régularités intéressantes. Quant au fichier de configuration *listWhiteLabel*, il est modifié pour répondre aux spécificités du langage de programmation Java lors de l'exploration de celui-ci. Les principales contraintes données à l'algorithme FreqT sont très proches de celles utilisées en C#.
L'ensemble de ces fichiers sont visibles sur la figure 4.17.

```
minNode=20
minLeaf=2
maxLeaf=4
buildGrammar=true
minSupport=5
```

(a) Configuration

```
TypeDeclaration
Block
```

(b) *listRootLabel*

```
TypeDeclaration bodyDeclarations
bodyDeclarations MethodDeclaration
MethodDeclaration name body
SimpleName identifier
```

(c) *listWhiteLabel*

FIGURE 4.17 – Aperçu des contraintes générales et des fichiers de configuration *listRootLabel* et *listWhiteLabel* pour "l'expérience A et B"

— Expérience B : Cette expérience utilise les mêmes contraintes qu'expliquées précédemment pour "l'expérience A". Cependant, l'utilisation du fichier de configuration *nonMandatoryLabel* est ajoutée. Celui-ci est complété comme sur la figure 4.18. A l'aide de ce fichier, les noms de variables ne sont plus des nœuds obligatoires lors de la déclaration de celles-ci, il s'agit du

problème expliqué lors du chapitre 2. De plus, les méthodes invoquées sont plus propices à faire partie d'un pattern car nous avons l'assurance que leurs arguments peuvent différer entre les ASTs, ces derniers n'étant pas obligatoires.

```
VariableDeclarationFragment name
MethodInvocation arguments name expression
```

FIGURE 4.18 – Contenu du fichier *nonMandatoryLabel* pour "l'expérience B"

Analyse quantitative

Le "*pattern miner*" trouve **103** patterns en **30** minutes avec l'utilisation du fichier *"nonMandatoryLabel"* et **70** patterns en **30** minutes, sans l'utilisation de ce dernier. En effet, l'utilisation de *nonMandatoryLabel* permet au "*pattern miner*" d'étendre davantage de patterns.

Grâce à la nouvelle configuration, nous avons 33 nouveaux patterns et 18 patterns correspondant chacun à un pattern de "l'expérience A" mais auxquels certains nœuds ont été ajoutés, ces patterns seront appelés par la suite "patterns agrandis". Les 52 derniers patterns de "l'expérience B" peuvent être retrouvés dans "l'expérience A" de manière identique.
Nous pouvons donc remarquer que l'utilisation du fichier de configuration *nonMandatoryLabel* crée ou augmente la taille de certains patterns.

En ce qui concerne les nouveaux patterns, leur taille moyenne est de 28 nœuds avec un support moyen de 5 fichiers.
En effet, ces nouveaux patterns ont une taille moyenne proche de la contrainte "*minNode*" de cette expérience. Ceci s'explique par le fait qu'il est difficilement possible de trouver des patterns complètement différents de ceux retournés par "l'expérience A" sans entrer dans le groupe des "patterns agrandis".

Quant aux "patterns agrandis", le nombre de nœuds supplémentaires est entre 4 et 20 nœuds, pour une moyenne de 10 nœuds ajoutés à ces patterns.
Effectivement, l'ajout du fichier de configuration *nonMandatoryLabel* n'a, au final, que partiellement modifié l'exploration faite par FreqT.

Pour "l'expérience A", nous pouvons calculer une précision 57%.
Nous remarquons une baisse de cette précision pour "l'expérience B" qui contient plus de patterns, elle est de 34%. De plus, nous pouvons calculer le rappel relatif

de "l'expérience A" par rapport à "l'expérience B" qui est de 83%. En effet, seuls 6 des 33 nouveaux patterns retournés par "l'expérience B" sont considérés comme intéressants, ce qui explique que le rappel de "l'expérience A" relatif à "l'expérience B" est élevé.

Suite à cette analyse quantitative, nous pouvons nous poser la question de l'intérêt de l'utilisation du fichier *nonMandatoryLabel* dans le cas de cette expérience. Effectivement, le nombre de patterns évolue à l'ajout de ce fichier. Cependant peu de nouveaux patterns retrouvés grâce à son apport sont considérés comme intéressants. C'est pourquoi il est utile de mener une analyse qualitative de ces patterns dans le but de se rendre compte de l'impact du fichier de configuration *nonMandatoryLabel*.

Analyse qualitative

L'ensemble des patterns intéressants ressortant de *JHotDraw* ayant déjà été analysés dans une recherche précédente [14], cette analyse ne sera pas représentée dans ce document. Nous pouvons donc trouver la correspondance entre ces patterns analysés de "l'expérience B" et les patterns de "l'expérience A", toutes deux expliquées ci-dessus. De plus, un tableau regroupant tous les "nouveaux" patterns trouvés lors de cette expérience avec l'ensemble de leurs propriétés est disponible. Le tout se trouve dans le dossier "JHotDraw\analysis" de l'annexe.

Dans la section suivante, nous allons simplement montrer un exemple de nouveau pattern, c'est-à-dire un pattern retourné par "l'expérience B" et non par "l'expérience A", et un exemple de "pattern agrandi", montrant l'apport du fichier de configuration *nonMandatoryLabel*.

Pattern 53

Le pattern visible sur la figure 4.19 est un exemple de pattern retourné grâce à l'ajout du fichier de configuration *nonMandatoryLabel*. Il est composé de 33 nœuds, a un support de 9 et a un nœud racine *Block*.

Ce pattern montre l'utilisation de la classe "*AffineTransform*", définie comme suit par la documentation de Java : "*La classe AffineTransform représente une transformation affine 2D qui effectue un mappage linéaire des coordonnées 2D aux autres coordonnées 2D qui préserve la «rectitude» et la «parallélisme» des lignes.*" [21].

Ce pattern n'est pas considéré comme étant intéressant. Il informe sur l'utilisa-

tion de l'instance d'un objet de type "*AffineTransform*".

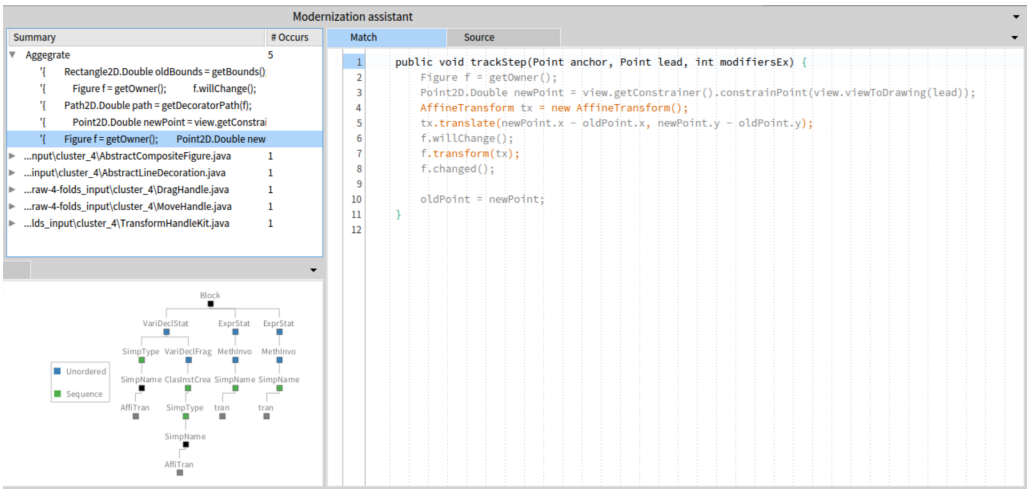


FIGURE 4.19 – **Pattern 53** : méthodes pairées

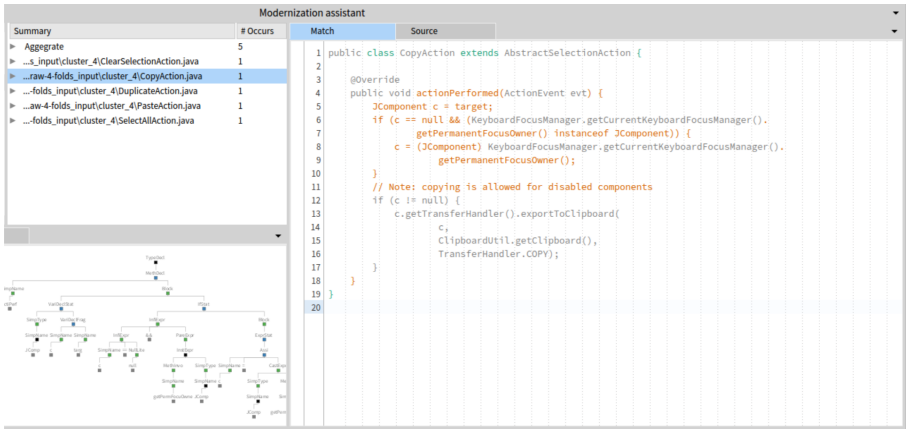
Pattern 46

Sur la figure 4.20, est illustré le pattern 46 qui a été agrandi grâce à l'ajout du fichier de configuration *"nonMandatoryLabel"*. Il avait 86 nœuds et a augmenté de 20 nœuds.

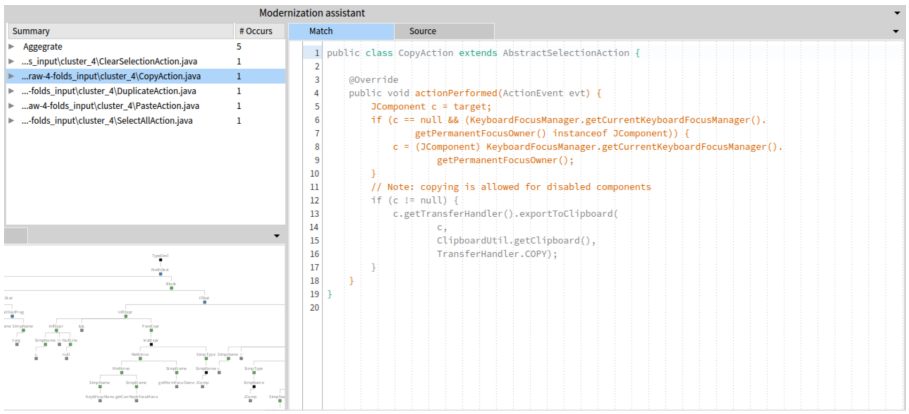
On voit que les nœuds ajoutés sur le pattern ne couvrent pas d'idiomes intéressants mais pourraient éventuellement fournir au développeur des informations utiles pour son analyse.

4.3 Conclusion de la validation

Dans ce chapitre, nous avons montré les résultats que nous obtenons avec notre solution. Pour ce qui est des projets C#, notre solution est l'utilisation d'un "*source code importer*" patché, une certaine complétion des fichiers de configuration "*listRootLabel*" et "*listWhiteLabel*", et enfin l'ajout du fichier de configuration "*nonMandatoryLabel*". Pour ce qui est du projet Java, nous n'avons regardé que l'impact de l'ajout de ce dernier.
Lors de nos expériences C#, le fichier *nonMandatoryLabel* a eu un intérêt important, les problèmes que nous avons soulevés dans le chapitre 2 nous ont poussés à le créer. La quantité ainsi que la qualité des patterns trouvés lors de nos expériences C# sont un gage de l'intérêt de celui-ci.



(a) *Avant*



(b) *Après*

FIGURE 4.20 – Aperçu du **pattern 46** avant et après l'utilisation du fichier de configuration *nonMandatoryLabel*

Cependant, il est à remarquer que dans le cas du projet Java, "JHotDaw", l'expérience n'a pas été concluante. Cet ajout augmentant le nombre de patterns dont très peu sont considérés comme intéressants. En effet, nous avons vu que la perte en précision était importante, pour un trop petit gain en ce qui concerne le rappel. Le prochain chapitre conclura ce mémoire en mettant en avant le travail pouvant encore être fourni autour de ce sujet.

Chapitre 5

Conclusion

La finalité de ce mémoire était d'exploiter l'adaptabilité du framework INTiMALS [1] pour l'exploration de patterns sur du code source C#.
Notre travail s'est inscrit dans le contexte de ce projet de recherche INTiMALS, dont le but est de construire un assistant de modernisation intelligent pour les systèmes hérités. Leur approche est d'explorer les idiomes et les régularités d'un code source à l'aide d'un framework paramétrable utilisant une personnalisation de l'algorithme FreqT. L'ensemble des composants de ce framework est indépendant du langage du projet dans lequel les régularités sont recherchées, mis à part le point d'entrée de ce framework. Afin de contraindre l'algorithme d'exploration, le fonctionnement de ce point d'entrée, le *source code importer*, et celui du composant utilisant l'algorithme d'exploration, le *pattern miner*, ont été étudiés.

Premièrement, nous avons analysé l'apport des fichiers de configuration mis à notre disposition par le framework INTiMALS. Ces fichiers ont été utilisés pour l'exploration de patterns sur du code source C#. Nous avons remarqué que, malgré des résultats intéressants, certains patterns manquaient de complétion, la construction de la grammaire et l'utilisation de celle-ci par le *pattern miner* en étant la cause principale.

Ensuite, nous avons créé un fichier de configuration supplémentaire, *nonMandatoryLabel*. L'ajout de ce fichier permet une approche plus généraliste que celle existante, à l'aide des fichiers de configuration du framework INTiMALS, celui-ci se concentrant sur le comportement spécifique d'un langage plutôt que celui dépendant de l'état du code source en entrée.
Cependant, l'utilisation de ce fichier de configuration exige une connaissance du langage de programmation et surtout de la composition des éléments exportés dans le méta-modèle utilisé par ce framework.

Enfin, nous avons validé ces solutions en accomplissant différentes expériences. Durant celles-ci, nous avons calculé la précision et un rappel relatif pour chacune d'entre elles. Les expériences effectuées sur du code C# ont été concluantes, de nombreux patterns riches sont ressortis de l'utilisation du framework INTiMALS auquel nous avons ajouté le fichier de configuration *nonMandatoryLabel*.
De plus, nous avons étudié l'impact de l'ajout du fichier de configuration *nonMandatoryLabel* sur un autre langage dont l'adaptabilité au framework INTiMALS avait déjà été prouvée, Java. Dans ce cas, nous avons comparé les résultats d'un projet sans l'utilisation du fichier de configuration *nonMandatoryLabel* et les résultats de ce même projet avec l'utilisation de ce dernier. Ce fichier n'a pas apporté un grand nombre de patterns supplémentaires, mais une complétion d'autres patterns est remarquable.

Les contraintes données à l'algorithme d'exploration ont une grande importance. Elles sont utiles pour limiter le temps d'exécution de cette exploration, mais aussi pour diriger l'algorithme sur les nœuds importants du méta-modèle représentant le code, l'influençant lors de la création de patterns qui peuvent, par la suite, aider à la compréhension du code. Cette deuxième utilité est le point central de notre recherche, sans minimiser l'importance de la première.
Il est donc important de bien définir ces contraintes, ce qui implique d'avoir une connaissance du méta-modèle utilisé pour représenter le code.
C'est pourquoi il pourrait être intéressant d'utiliser une grammaire générale à chaque langage contenant les liens de parenté et d'obligation entre chaque nœud, grammaire qui serait complétée à chaque utilisation de l'outil par les nouveaux liens trouvés lors de cette exécution. De cette manière, la grammaire deviendrait de plus en plus complète à force d'exécutions sur de nouveaux projets. De surcroît, elle continuerait à être maintenue malgré l'évolution du langage par l'ajout de nouveaux nœuds et de nouveaux liens n'existant pas encore aujourd'hui.
Les fichiers de configuration dont nous avons parlé tout au long de ce travail, garderaient leur intérêt mais l'oubli de certains nœuds aurait moins d'effet négatif sur les résultats retournés qu'à l'heure actuelle.
Enfin, une correction plus complète du *source code importer*, qui transforme le code source C# en des ASTs utilisés dans l'exploration de patterns, apporterait également une complétion de la solution proposée. Comme le modèle qu'utilise l'algorithme d'exploration sont ces ASTs, un manquement à la structure dictée par le méta-modèle du framework INTiMALS pourrait faire perdre des informations.

Pour conclure ce mémoire, soulignons qu'un travail de recherche demande une connaissance approfondie de plusieurs sujets. En effet, lors de ce travail, plusieurs concepts statistiques, algorithmiques et informatiques ont dû être utilisés pour

mener à bien nos différentes expériences. Ces concepts ont majoritairement été étudiés lors de notre cursus et nous avons pu nous appuyer sur ces notions théoriques apprises et les mettre en œuvre en nous confrontant à la pratique. De plus, la participation à un travail de recherche collaboratif et les échanges hebdomadaires avec nos promoteurs ont été une expérience très enrichissante qui nous a poussés à la réflexion et qui nous a permis d'avoir un avant-goût de ce que pourra être notre vie professionnelle. Et enfin, apporter notre pierre à un travail de recherche informatique et savoir que notre participation peut être utile à INTiMALS a créé en nous un esprit de challenge et de dépassement de soi.

Bibliographie

[1] INTiMALS - Intelligent Modernisation Assistance for Legacy Software, url : `http://soft.vub.ac.be/intimals/`, 03/12/19.

[2] Raincode Labs, url : `https://www.raincodelabs.com/`, 04/12/19.

[3] Dario Di Nucci, Hoang Son Pham, Johan Fabry, Coen De Roover, Kim Mens, Tim Molderez, Siegfried Nijssen, Vadim Zaytsev. *A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns.* In Proceedings of the 12th Seminar Series on Advanced Techniques & Tools for Software Evolution (SATTOSE), Bolzano, Italy, 2019. url : `http://ceur-ws.org/Vol-2510/sattose2019_paper_3.pdf`.

[4] Nunit : A unit-testing framework for all .Net languages, url : `https://github.com/nunit/nunit`, 25/02/2020.

[5] Notepads : A modern, lightweight text editor with a minimalist design, url : `https://github.com/JasonStein/Notepads`, 14/02/20.

[6] Precision and recall, Wikipedia. url : `https://en.wikipedia.org/wiki/Precision_and_recall`, 15/02/20.

[7] Tatsuya Asai, Kenji Abe, Shinji Kawasoe, Hiroki Arimura, Hiroshi Sakamoto and Setsuo Arikawa.
"*Efficient Substructure Discovery from Large SemiStructured Data*". In : Proceedings of the 2002 SIAM International Conference on Data Mining. Ed. by Robert Grossman et al. Philadelphia, PA : Society for Industrial and Applied Mathematics, Apr. 11, 2002, pp. 158– 174. doi: 10.1137/1.9781611972726.10.

[8] Gitlab project INTiMALS, repository metamodel.
url : https://gitlab.soft.vub.ac.be/intimals/metamodel, 14/03/20.

[9] Quelle est la différence entre un arbre de syntaxe abstraite et un arbre de syntaxe concrète ?, url : https://www.ipgirl.com/38892/quelle-est-la-difference-entre-un-arbre-de-syntaxe-abstraite-et-un-arbre-de-sy
html, 05/03/20.

[10] A two-dimensional graphics framework for structured drawing editors that is written in Java, url : https://sourceforge.net/projects/jhotdraw/, 06/03/2020.

[11] The .NET Compiler Platform SDK (Roslyn APIs), url : https://docs.microsoft.com/en-us/dotnet/csharp/roslyn-sdk/, 23/03/20.

[12] Système hérité, Wikipedia. url : https://en.wikipedia.org/wiki/Legacy_system, 23/03/20.

[13] Wee Keong Ng, Masaru Kitsuregawa, Jianzhong Li, Kuiyu Chang.
Advances in Knowledge Discovery and Data Mining : 10th Pacific-Asia Conference, PAKDD 2006, Singapore, April 9-12, 2006, Proceedings. doi: 10.1007/11731139.

[14] Hoang Son Pham, Kim Mens, Dario Di Nucci, Coen De Roover, Vadim Zaytsev, Siegfried Nijssen, Johan Fabry, Tim Molderez.
Mining for Code Regularities in Modern and Legacy Codebases. Submitted to "<Programming> 2021", Cambridge, United Kingdom, 2021.

[15] P. E. MacKenzie and R. Stallman "*What Comparison Means*". Section 1 of book : Comparing and Merging Files with GNU Diff and Patch. Network Theory Ltd., 2003. url : https://www.gnu.org/software/diffutils/manual/diffutils.pdf.

[16] Source Making : Uses of Design Patterns, url : https://sourcemaking.com/design_patterns, 13/04/2020.

[17] Design patterns quick reference, url : http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf, 13/04/2020.

[18] J. Bisbal, D. Lawless, B. Wu, and J. Grimson, "*Legacy Information Systems : Issues and Directions,*" IEEE Software, vol. 16, no. 5, 1999.
J. Bisbal, D. Lawless, Bing Wu and J. Grimson, "Legacy information systems : issues and directions," in IEEE Software, vol. 16, no. 5, pp. 103-111, Sept.-Oct. 1999, doi: 10.1109/52.795108.

[19] C. J. Kapser, M. W. Godfrey, "*"Cloning considered harmful" considered harmful : patterns of cloning in software*", Empirical Software Engineering, vol. 13, pp. 645, 2008, doi: 10.1007/s10664-008-9076-6.

[20] Documentation Microsoft, Properties in C#, url : `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/properties`, 10/05/2020.

[21] Documentation Java, Class AffineTransform, url : `https://docs.oracle.com/javase/7/docs/api/java/awt/geom/AffineTransform.html`, 10/05/2020.

[22] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2014. *Learning natural coding conventions*. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014). Association for Computing Machinery, New York, NY, USA, 281–293. doi: 10.1145/2635868.2635883.

[23] Innoviris Brussels, url : `https://innoviris.brussels/`, 23/05/2020.

Annexes

— **Annexe 1 :** Rwibutso_65051800_Skubiszewski_79551800_2020_Appendix1.zip

Annexe contenant plusieurs dossiers :

- VentilatorState
- Nunit
- Notepads
- JHotDraw

Chacun des dossiers décrit une expérience que nous avons menée et dans lequel nous pouvons trouver les fichiers sources, la ou les configurations utilisées et, si elle existe, l'analyse de cette expérience.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**